# A Prototyping Environment for Specifying, Executing and Checking Communicating Real-time State Machines*

Sitaram C. V. Raju and Alan C. Shaw
Department of Computer Science and Engineering
University of Washington
Seattle WA 98195
{sitaram,shaw}@cs.washington.edu

**Abstract**

We describe a toolset, consisting of a graphical editor, a simulator, and an assertion checker, for prototyping distributed real-time systems that are specified as Communicating Real-time State Machines (CRSMs). CRSMs are timed state machines that communicate synchronously over uni-directional channels. The system behavior of CRSMs is characterized by a time-stamped trace of communication events. Safety and timing assertions on the trace of communication events are expressed in a notation based on Real-Time Logic. We illustrate the novel aspects of the simulator and assertion checker by specifying a traffic-light controller and other real-time systems.

## 1 Introduction

Executable specifications have been advocated as a promising method for understanding the requirements specification of systems, and for developing systems incrementally [11, 18]. An executable specification serves as a prototype of the final implementation. The main advantage of prototyping systems is that it gives the designer feedback that what is being specified is indeed what is desired. In addition, the prototype can be checked for functional and timing correctness before moving to design and implementation.

*This research was supported in part by the Office of Naval Research under grant number N00014-89-J-1040 and by the National Science Foundation under grant number CCR-9200858.

In this paper, we describe an environment for prototyping real-time systems using Communicating Real-time State Machines (CRSMs). CRSMs, introduced in [14], are an executable scheme for specifying the requirements of both a real-time system and its physical environment. They are timed state machines that communicate synchronously over unidirectional channels. Every CRSM has a partner clock machine that provides a timeout mechanism and can be queried for the value of current time. System behavior is characterized by the time-stamped trace or history of communication events between the machines. Desired properties of the system behavior, including safety and timing constraints, are expressed as assertions on the trace of communication events. The prototyping environment consists of a graphical editor for describing CRSMs, a simulator to observe the execution of the prototype, and an assertion checker for monitoring assertions during simulation.

This paper makes two contributions: First, the prototyping environment serves as a validation of the execution algorithm and paper design of CRSMs described in [14]. Second, the paper presents a novel and useful method of specifying safety and timing properties, and checking them during simulation.

A substantial amount of research has been done on executable specifications and prototyping of real-time systems. We give a brief survey and comparison of specification methods that are state machine based, because these are most closely related to our work. Statemate [10] is a tool for executing Statecharts [9]. Statecharts use broadcast for event communication, whereas CRSMs use message passing. Modechart is similar to Statecharts, but emphasizes the timing properties of systems; a simulator for Modechart is presented in [15]. Unlike CRSMs, Modechart does not allow events to have message components. Both Statecharts and Modechart use a shared memory model, whereas CRSMs present a distributed model. Reference [13] describes the ROOM methodology and its associated toolset. ROOM combines object-oriented methods (including inheritance) and Statecharts. Hierarchical Multi-State Machines (HMS) [4] is an executable notation that blends ideas from Petri-nets, Statecharts and temporal logic. Finally, the new model of time in CRSMs (both for describing the passage of time and for specifying timeouts) also distinguishes our work from the ones listed above.

Traces are used for reasoning about the behavior of general systems in [8], and for real-time systems in [1, 14]. Our method of reasoning with traces is noteworthy because it is based on Real-Time Logic (RTL) [5]. RTL is well suited for real-time systems because it deals with event times and can differentiate multiple occurrences of the same event. We believe that such use of timing assertions for analysis of executable specifications is novel.

The rest of the paper is organized as follows. In the next section, we present our system model and the general prototyping approach. Sections 3, 4 and 5 give the details of our two specification languages and prototyping environment, and some experiments. Section 6 discusses some of the limitations of our approach and describes current and future work.

# 2 Prototyping Approach and Model

A real-time specification language must have features for describing the passage of time and for specifying timing constraints. Since real-time systems operate in parallel with their physical environment and are often distributed, the language must also have support for concurrency. A graphical representation is desirable as it is natural for input and is highly readable, thus serving as the documentation.

An important reason for specifying the requirements formally is that it permits analysis of the specification. It must be possible to check the specification for functional and timing correctness. An important class of functional properties are *safety* properties, which assert that the system is never in a hazardous state. Timing properties that must be verifiable include delays, deadlines, and periodic constraints. Finally, the methods for checking or verifying the above properties must be computationally efficient.

As an example to illustrate our specification languages and prototyping environment we will describe a real-time system for controlling traffic lights. It was inspired by the traffic-light controller presented in [3], but is completely different in detail from that example:

> A computer system controls the traffic lights at an intersection of an avenue and street. The avenue and street traffic lights have inputs *avetored, avetoyellow, avetogreen* and *strtored, strtoyellow, strtogreen,* respectively. These inputs are used by the computer controller to set the state of the lights. The controller sequences the avenue and street lights. The sequence for each light is red→green→yellow and back to red, with the light being green for 45 seconds and yellow for 5 seconds. An ambulance can signal (*approaching*) the controller about its impending arrival and its direction. The controller then turns both the avenue and street lights to red. When the ambulance nears the intersection (signal *before*), the controller turns either the avenue or street light to green based on the direction of the ambulance. After the ambulance leaves the intersection (signal *after*), the controller turns the green light to red and returns to its normal sequence. (Note: It is assumed that there is a single ambulance and that the ambulance does not make a turn at the intersection).
>
> The controller must satisfy certain safety constraints so that the traffic lights do not pose a hazard to the ambulance and to other vehicles on the road. One obvious safety constraint is that vehicles on both roads must not be allowed to enter the intersection simultaneously. Also, when the ambulance is crossing the intersection, the lights on that road must be green and the other light must be red.

We use two specification languages. One language describes the behavior of the system including the physical interfaces. The other specifies the safety and timing constraints that the system must satisfy.

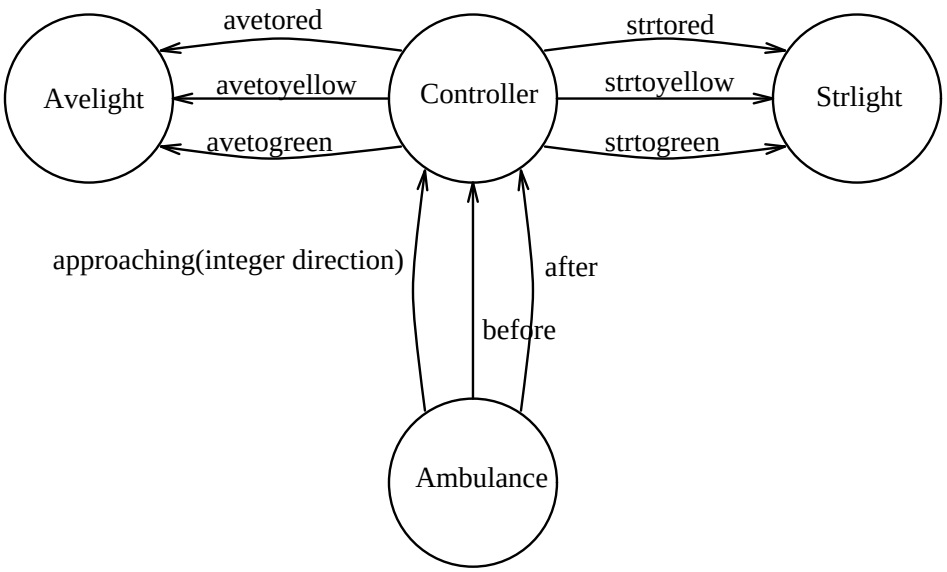



Figure 1: Global view of system

The system specification language is based on CRSMs [14], which are timed state machines that run concurrently except when they need to communicate. Machines communicate synchronously (as in CSP [7, 8]) and instantaneously through messages over uni-directional channels. The set of machines and channels for the traffic-light controller and its environment is shown in Figure 1. There are four machines: Controller (the real-time system), and Avelight, Strlight, and Ambulance that describe the behavior of the physical avenue light, street light, and ambulance respectively; and nine channels: *avetored, avetoyellow, avetogreen* and so on. The CRSM model is a distributed one, except for a single shared variable representing the current time. A channel can have message components, as in,

approaching(integer direction)

where *direction* is of type integer. Each communication instance (i.e. a message) on channel *approaching* will have an integer data value associated with *direction*. The behavior of a system is characterized by the history of time-stamped communication events on its channels, i.e. a history of the messages transmitted.

The assertion language is based on Real-Time Logic (RTL) [5, 6], which views a computation as a sequence of event occurrences. In our context, the only events of interest are instances of communication on channels. Both safety and timing properties are expressed as assertions about the relations between channel events. For example, there must be a delay of at least 5 seconds between the ambulance signals *approaching* and *before*, to give the Controller enough time to clear the intersection. This requires that corresponding events on the *approaching* and *before* channels be at least 5 seconds apart.

RTL uses the occurrence function @ to denote the time of events. For example,

$$@(\text{approaching, i})$$

denotes the time of the $i^{th}$ communication on channel *approaching* ($i > 0$). RTL formulas are composed from occurrence functions, equality/inequality relations, first order logic connectives, and the universal ($\forall$) and existential ($\exists$) quantifiers. The above delay constraint can be specified by the formula,

$$\forall \text{i} \;\; @(\text{before, i}) \geq @(\text{approaching, i}) + 5$$

The reason for choosing RTL is that its occurrence function can be used to distinguish different occurrences of the same event. In addition, earlier work on using RTL for run-time monitoring [2, 12] motivated us to use similar ideas for checking executable specifications.

Our general approach is:

- to build a system of CRSMs and RTL assertions that represent the prototype and its desired behavior,
- to simulate the prototype to observe system behavior, and
- to monitor the execution to check that no safety and timing assertions are being violated.

# 3 Specification Language Constructs

## 3.1 CRSMs

Figures 2 and 3 show the CRSMs for Avelight, Strlight, Ambulance and Controller. A machine has a finite number of states, one of which is designated as the start state. State transitions are guarded commands and consist of a guard, a command, and a time interval. The guard is a Boolean expression constructed from constants and local variables of the machine. A transition is ready for execution if its guard evaluates to *true*. If more than one transition is ready to fire, then a selection is made non-deterministically. A command can be one of input, output, or internal command.

The Controller can get input from the Ambulance by the input command,

$$\text{approaching(dir) ?}$$

and the Ambulance can use the output command

approaching(direction) !

to interact with the Controller. The communication occurs when both Controller and Ambulance are ready. The semantics of IO is the assignment,

dir = direction

at the receiving machine (Controller). IO on a channel with no message components (e.g. *strtored*) denotes a pure synchronization signal. State $n_{11}$ of the Controller has two exiting transitions with guards (dir==0) and (dir==1). For example, the transition from $n_{11}$ to $n_1$ is only executed if (dir==1).

The time interval associated with IO transitions denotes the earliest and latest time IO can occur after entering the state. Suppose Ambulance enters the state *bef* at time $t_{bef}$. Then Controller must accept input on channel *before* somewhere in the time interval

$$[t_{bef} + 8, t_{bef} + 10]$$

Otherwise, the Ambulance machine will be deadlocked and can make no further progress. The common cases of a *true* guard and the time interval [0,$\infty$] are omitted for the sake of brevity. Also, the interval [t,t] is represented simply as [t].

An internal command can be a sequential program[1] that changes the values of local variables, and/or it can denote a physical activity. The time interval of an internal command represents the best and worst case execution time of the command. All internal commands in the two lights take 1 second to execute. The commands set local variables whose values designate the intended state of the light. Internal commands have a label, such as *turn_red*, which serves to identify the sequential code.

Every CRSM has a real-time clock machine that can be queried for the value of current time over the *timer* channel. The Controller uses

timer ? [45]

to obtain a timeout after spending 45 seconds in states where a light is green. Current time can be retrieved, say in the variable $x$, by a call: timer(x) ?

The machines for the two lights accept a signal to change the light state, and then carry out an internal command that sets the new state. The Ambulance machine uses the internal command, *getapptime*, to get the time of next approach (*nexttime*) of the ambulance and the direction of approach from the user. If *direction* is 0, then the direction of travel is along the avenue; if 1, it is along the street. *Nexttime* is used as the time interval on the succeeding transition to schedule the *approaching* signal.

[1]The current implementation uses a C/C++ procedure.

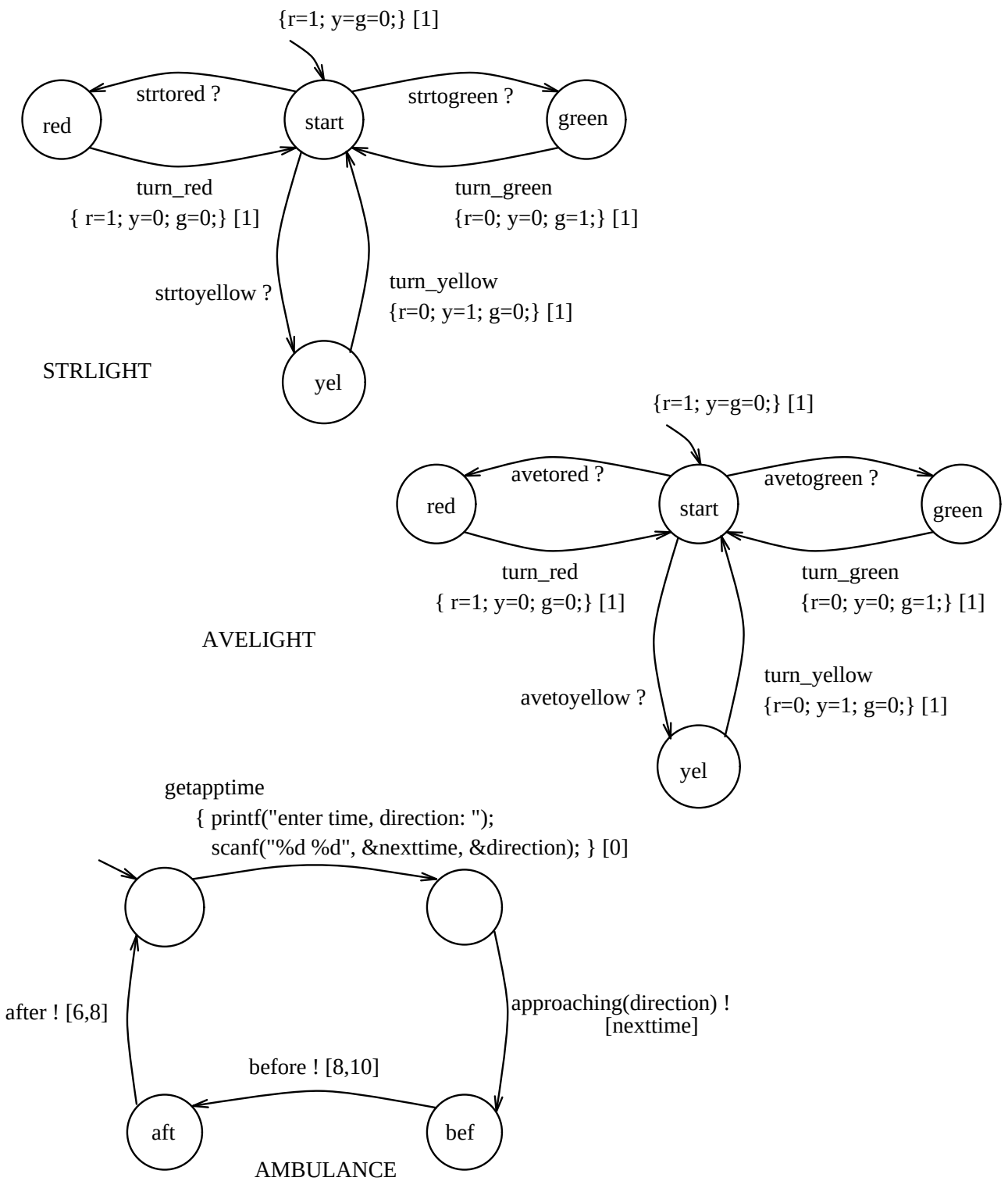
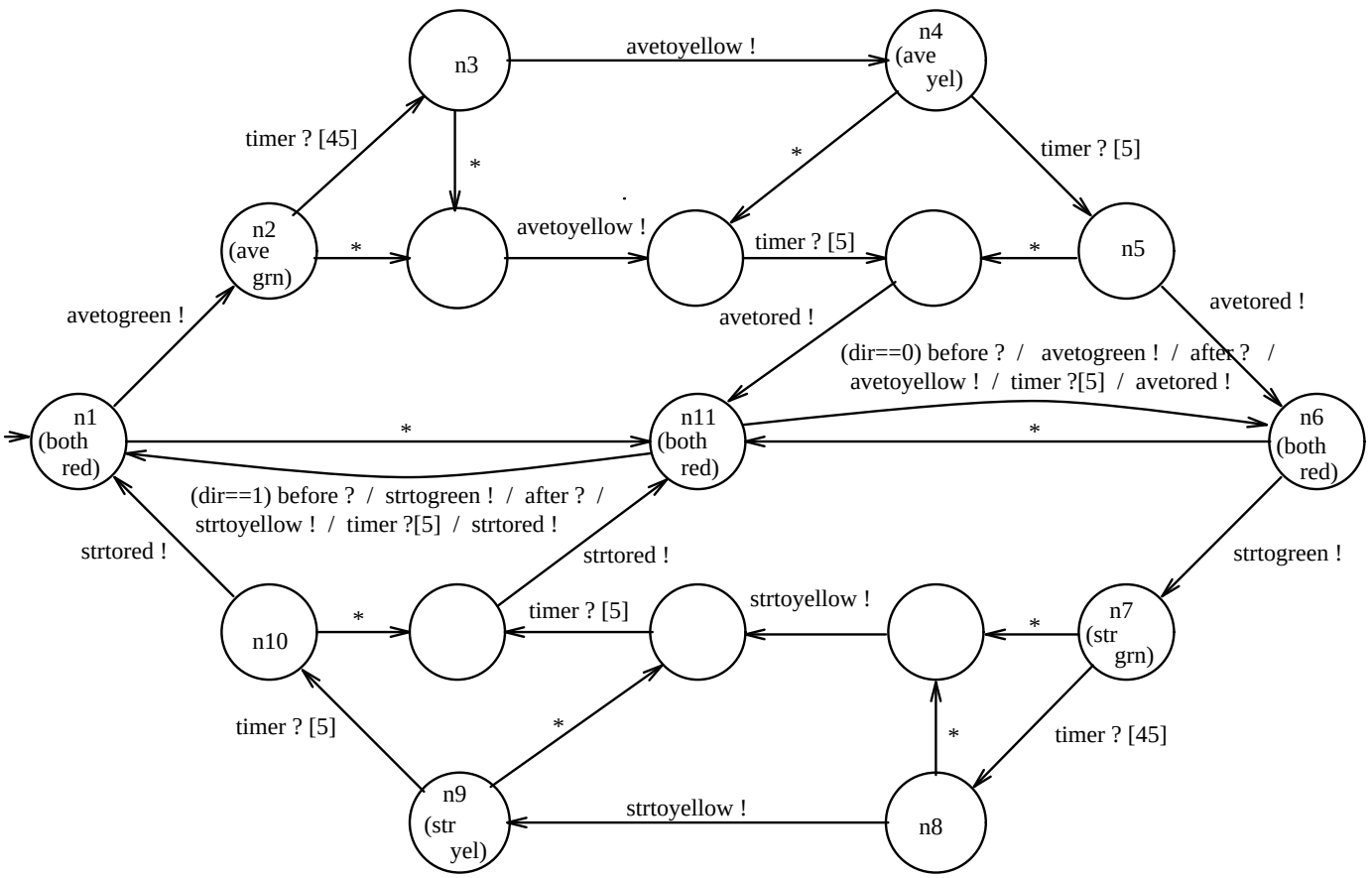



Figure 2: CRSMs for the physical environment




*: approaching (dir) ?

CONTROLLER

Figure 3: CRSM for real-time system

The Controller can be thought of as being in two modes: normal and interrupted. States $n_1$ through $n_{10}$ comprise the normal mode. In this mode, Controller sequences the lights through red→green→yellow and back to red, with the light being green for 45 seconds and yellow for 5 seconds. The Controller moves to the interrupted mode when it receives the signal *approaching*. If the avenue light is green, it is turned to yellow and then to red; if it is yellow it is turned to red (similarly for the street light). When the ambulance signals that it is before the crossing, the Controller uses the value of *dir* in the guard to turn the appropriate light to green. After the ambulance leaves the crossing (signal *after*), the Controller reverts to its normal mode. The slashes ("/") on the two transitions from $n_{11}$ are used as an abbreviation to separate elements in a linear chain of transitions.

## 3.2 Assertions

In our implementation, the assertion for a minimum delay of 5 seconds between the events *approaching* and *before* is specified as:

```
when before
{
   int time_before, time_app;

   time("before", -1, &time_before);
   time("approaching", -1, &time_app);
   assert(time_before >= time_app + 5);
}
```

An assertion consists of two parts: a *when* clause and a C/C++ procedure. The *when* clause specifies when the assertion is to hold, i.e., be checked. An assertion can be checked when a channel event occurs, or at a given time offset from a channel event occurrence. The above assertion will be checked after every communication on channel *before*. (An example of assertion checking at a time offset from a channel event is given in Section 5.1). The second part of the assertion, the user-defined C/C++ procedure, gives the constraint. The constraint checking procedure in the above assertion uses the predefined function *time* to retrieve times of channel events from the channel history, and then tests the values (using function *assert*) to see if the desired relation holds. *Assert* checks if its parameter evaluates to *true*.

To check assertions, it may be necessary to record the times and data values (if any) of the previous occurrences of a channel event. For example to check a minimum delay constraint between two consecutive events on a channel, the times of the current and previous channel event must be known. We maintain a fixed length history for every channel that records the last $n$ occurrences of channel events, where $n$ is the history size. A fixed history size (user-defined) was chosen because keeping the entire history is too large to be practical. The

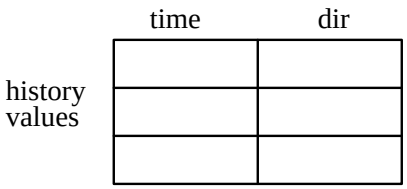



Figure 4: Channel history data structure

history data structure is a circular queue; the data structure for *approaching* is shown in Figure 4. As IO occurs on channels, the time and data values of the IO are stored in the history data structure.

Histories can be read by the following functions:

```
int time(char *channel, int ind, int *result);
int value(char *channel, char *field, int ind, int *result);
int value(char *channel, char *field, int ind, float *result);
int value(char *channel, char *field, int ind, char *result);
int index(char *channel, int ind);
```

A similar history data structure and interface was used in [2]. Function *time* retrieves the communication time and function *value* retrieves the data value. The first parameter *channel* is the channel name. Parameter *field* in the value function specifies the desired message field. *Ind* is the event occurrence index. When *ind* is positive, it refers to the $ind^{th}$ occurrence of the event; when negative, it refers to the $ind^{th}$ most recent occurrence of the event. Thus, *ind* value of -1 refers to the last occurrence of an event. *Result* contains the returned result; its type can be: integer, floating point or character. Function *index*[2] returns the absolute index of an event occurrence corresponding to *ind*. The functions return a value *err* if the desired event has not yet happened or if it has expired (dropped from history). The return code plays a useful role in specifying constraints (see Section 5).

The assertion code presented at the beginning of this section reads the times of the last occurrences of the signals *approaching* and *before*. Since the assertion checking is invoked when communication on *before* occurs, the value of *time_before* is the current simulator time. The assertion code uses *assert* to check that the two time values are at least 5 seconds apart.

[2]The relationship between RTL and these functions is:

- for $i > 0$: time(e, i, v) = @(e, i)
- for $i < 0$: time(e, i, v) = @(e, index(e,i))
- for $i = 0$: time(e, i, v) returns *err*, @(e, i) is undefined

When the parameter to *assert* evaluates to *false*, it prints an error message saying that the assertion has been violated. No recovery action is taken. Our *assert* is different from the C language's *assert* macro because the C language's *assert* macro terminates the program when its parameter evaluates to *false*.

As a measure of the expressiveness of our assertion specification language, note that the constraint checking code can be *any* computable function of history. In this sense, the assertion specification language is universal. It is also important to note that histories are maintained for channel events only. However, the user can ensure history maintenance for states and local variables of a machine by making them the message components of a channel.

Since CRSMs are a universal specification methodology, the user can also define CRSMs that implement the assertion checking. There are two advantages with keeping the normal computation and assertion checking separate. First, it improves the readability as the assertion checking does not clutter the normal computation with extra states and transitions. Second, CRSMs offer a distributed model and hence checking global properties, such as a timing constraint that spans two CRSMs, is not easy.

# 4 Design of the Prototyping Environment

The prototyping environment consists of three integrated tools: graphical editor, simulator, and assertion checker. The tools have been implemented in C/C++ on a Decstation 5000 running UNIX. The relation between the tools is shown in Figure 5.

The *graphical editor* is an extension of an existing graph editor called Xsim [17]. Xsim runs on the X-window system and uses Athena widgets [16]. The graphical editor has two modes: global and local. The global mode is used for creating the global system of CRSMs and the interconnecting channels. The local mode is used for creating the state diagram of a machine. The graphical representation is translated into a text form that is in turn translated and compiled into a simulator. The text form is a straightforward enumeration of the states and transitions of the various machines in the system. The text form can also be used for entering the description of CRSMs and assertions directly.

There are two issues in the design of the *simulator*. One is whether to use discrete ticks for time or some approximation to continuous time. The CRSM scheme as described in [14] uses continuous time. We chose to use discrete time as it simplifies some of the implementation details. The second issue is whether to have a compiled or an interpreted simulator. We implemented a compiled simulator because it permits the use of a general-purpose language (C/C++ in this case) for specifying the guards, lower/upper bounds of time intervals, internal commands, and assertion checking code.

For example, the use of arbitrary expressions[3] in time intervals allows for a simple, straightforward specification of a periodic process (see Section 5.2). The use of C statements

[3]Expressions must be free from side effects.

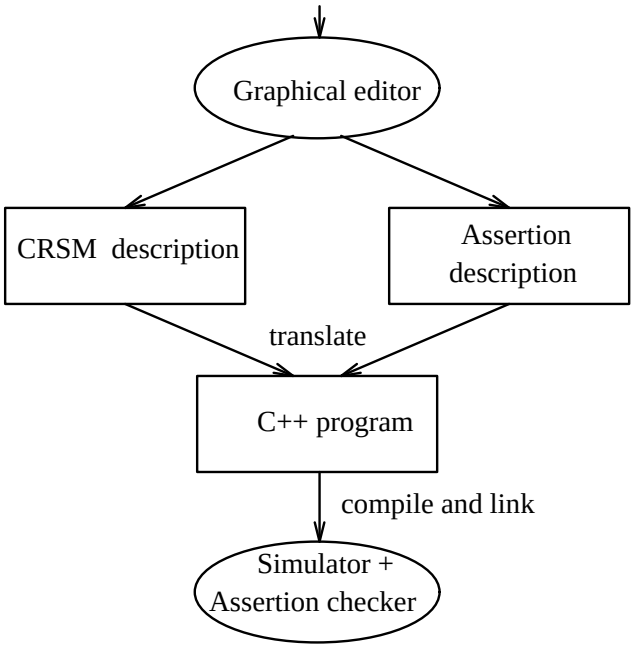



Figure 5: Relation between the tools

for internal commands allows interactive parameterization. This feature has been exploited in the specification of the Ambulance machine (Figure 2). In Ambulance, *getapptime* uses C input/output routines for getting the arrival time and direction of the ambulance. This time is used as the time interval on the succeeding transition to generate the *approaching* signal. The user can easily experiment with a variety of input times and directions.

The ability to use C in internal commands has also been capitalized on to provide a graphical display of traffic lights (Figure 6). The code for internal commands of lights (*turn_red, turn_yellow* and so on in Figure 2) use the X-window system to draw/clear the appropriate filled circle. The state of traffic lights is reflected visually on the screen. The advantages of such graphic output for understanding and debugging prototypes is obvious.

The text form is translated into a C++ program that is then compiled and linked with the simulator/assertion checking library to form an executable file. The major classes in the C++ program correspond naturally to CRSM, channel, and assertion. The simulator uses an event-driven algorithm as described in [14]. The simulator steps through the execution under user control (either a single step or a sequence of steps). The output of the simulator is a trace of the channel events and internal commands along with their occurrence times, plus any special interaction, such as the *printf* statement in the *getapptime* command (Figure 2).

Figure 7 shows a skeleton of the C++ class that is generated for the Strlight machine. *Strlight* is a derived class of a predefined *crsm* class. The local variables of the Strlight machine (integers $r$, $y$ and $g$) are defined in the private part of the class. The list of states

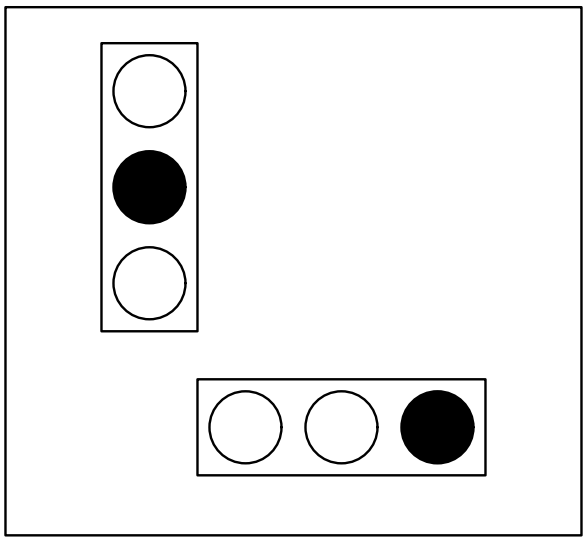


Figure 6: Graphical display of traffic lights

is bundled into an enumerated type.

Each exiting transition from a state is compiled into four functions/procedures, one each for guard, lower bound of time interval, upper bound of time interval, and the command. The procedures for *strtored* transition are: *start_0_guard, start_0_lbound, start_0_ubound*, and *start_0_input*. The code for *start_0_guard* returns 1, because it is always true (default case). To evaluate the guard on a transition at runtime, the function corresponding to the guard is called and the returned value is checked. For the functions corresponding to lower and upper time bounds (*start_0_lbound* and *start_0_ubound*) the code returns the default time values. More complex expressions for guards and time bounds are simply copied into the function body after syntax checking.

The code for input command (*start_0_input*) first inserts the occurrence time of the event into the channel history. Then it updates the state of the Strlight machine. If a channel has message components, then code to assign the data values to the appropriate local variables is generated. The next four procedures in Figure 7 correspond to the transition *turn_red*. The internal code that is enclosed within curly braces is copied verbatim into the function body of *red_0_internal*. The procedures for an output transition are not shown. A procedure for an output command takes a parameter that is a pointer to the corresponding input procedure of its communicant machine (machine with which it is going to do IO). The code for the output command coerces the pointer to a procedure type[4], and then calls the procedure.

Class *crsm* has a data-structure called *transition_table* that stores pointers to the various

[4]The pointer is coerced to type PROC, where,
typedef int (crsm::*PROC)(...);

```
class strlight : public crsm {
    ....
    int r, y, g;
    enum strlight_states {start, red, yel, green};
    strlight_states curstate;
public:
    int start_0_guard()    { return 1; }
    int start_0_lbound()   { return 0; }
    int start_0_ubound()   { return infinity;}
    int start_0_input()
       {
          channel *chan = chlist->getchannel("strtored");
          int ind = chan->get_next_index();
          chan->set_value("time", simulator_time, ind);
          curstate = red;
       }
    ....
    int red_0_guard()       { return 1; }
    int red_0_lbound()      { return 1; }
    int red_0_ubound()      { return 1; }
    int red_0_internal()
       {
          r = 1; y = 0; g = 0;
          curstate = start;
        }
    ....
};
```

Figure 7: Compiled code for *strlight* machine

procedures of the translated transitions. Class *crsm* exports a high level abstraction of *transition_table*. For example, it exports a method called *get_nel()* that returns the next event list, i.e., the list of transitions that are ready for execution in the current state (the corresponding guards evaluate to *true*).

The *assertion checker* maintains a list of currently enabled assertions. Assertions are assigned unique id's. The user can enable/disable checking of any assertion by specifying the particular id. To help the user with debugging violations of assertions, a trace feature has been provided. When the tracing is enabled, each time a function that reads the history is called, the result and the returned value are printed. As mentioned earlier, violation of an assertion simply prints an error message; no recovery action is taken.

The graphical editor only generates the input for the simulator tool. There is no provision for incorporating the output of the simulator into the graphical front-end; i.e. there is no animated simulation. The simulator and assertion checker have been integrated into a single program. The simulator updates the history when executing the channel events. After all channel events in a particular time instant (or after the specified time offset) have occurred, the assertion checker is invoked. If any assertions are triggered, the corresponding constraint checking code is invoked.

We have not analyzed the performance of the simulator and assertion checker in detail yet. For the specification examples described in this paper and in [14], the simulator/assertion checker responds to single-step commands from the user in real-time.

# 5 Experiments

We used the simulator and assertion checker described in the previous section to experimentally validate most of the example specifications described in [14]. These included a real-time bounded buffer; a mouse clicker recognizer that distinguishes among single clicks, double clicks and selections; and a train crossing gate controller. In each case, we validated the examples by observing simulator output and by checking relevant assertions during execution.

In this section we present more example assertions for the traffic-light controller, and give partial specifications of a computer calendar and a real-time dining philosophers problem. We also show how the histories can be used for monitoring performance.

## 5.1 Traffic-light Controller

Suppose that it takes as little as 4 seconds for the ambulance to reach the intersection after it signals *before*. This imposes a deadline of 3 seconds from the signal *before* to the signal *avetogreen* or *strtogreen* in order that the light be green when the ambulance arrives at the

intersection (the internal action to change the light takes one second). An RTL formula for specifying a deadline of 3 seconds between the $i^{th}$ occurrences of two events $e_1$ and $e_2$ is:

$$\forall i \;\; @(e_2, i) \leq @(e_1, i) + 3$$

The deadline assertion for the traffic-light controller is more complicated as the direction of the ambulance must be taken into account. The assertion is

```
when before + 3
{
   int time_before, dir_value, time_green;

   time("before", -1, &time_before);
   value("approaching", "dir", -1, &dir_value);
   if (dir_value == 0) {
      assert(time("avetogreen", -1, &time_green) != err);
   } else {
      assert(time("strtogreen", -1, &time_green) != err);
   }
   assert(time_green >= time_before && time_green <= time_before + 3);
}
```

The constraint checking code is invoked 3 time units after an event on channel *before*. The value of direction (*dir_value*) is read from history. The check that the return value is not *err* is to make sure that the event has indeed occurred in the past. Otherwise, the assertion has been violated. A stricter assertion is to check that the light remains green until the *after* signal is given by the ambulance.

Another safety constraint that the specification must satisfy is that vehicles on both roads must not be allowed to enter the intersection simultaneously, i.e, the green/yellow lights on the two roads must be mutually exclusive. The safety requirement is specified with two assertions:

```
when strtogreen
{
   int t0, t1, t2;
   time("strtogreen", -1, &t0);
   if (time("avetogreen", -1, &t1) != err) {
      assert(time("avetored", -1, &t2) != err);
      assert(t1 < t2 && t2 <= t0);
   }
}
```

```
when avetogreen
{
   int t0, t1, t2;
   time("avetogreen", -1, &t0);
   if (time("strtogreen", -1, &t1) != err) {
        assert(time("strtored", -1, &t2) != err);
        assert(t1 < t2 && t2 <= t0);
   }
}
```

The above assertion relies on the implicit assumption that the lights sequence is red→green→yellow and back to red. This is true from the specification, but the cautious user can check the assumption by another assertion[5].

## 5.2 Computer Calendar

Figure 8 shows a Calendar machine that maintains time and date. A user machine (not shown) can access the current time and date by IO on channel *curtime*. *Ctime* contains the message components for time and date. The Calendar machine receives ticks from Periodic Ticker and updates the time and date through an internal command. Periodic Ticker is a periodic process that sends ticks every 10 seconds. The periodic process makes use of its real-time clock machine (section 3.1). *Timer(y)* ? assigns the current global time to $y$. Periodic Ticker uses the expression $[x - y]$ to achieve a delay until the start of the next period. The periodic timing constraint can be described and checked by,

```
when tick
{
   int t1, t2;

   if (index("tick", -1) >= 2) {
     time("tick", -1, &t1);
     time("tick", -2, &t2);
     assert(t1 - t2 == 10);
   }
}
```

The function index returns the absolute index corresponding to the most recent *tick*, i.e., the number of ticks since startup. So, the periodic check is done when at least 2 ticks have occurred. The last two occurrences of tick event need to be kept in history in order to check

[5] We spotted an error in an earlier specification of the Controller because it had violated this assertion.

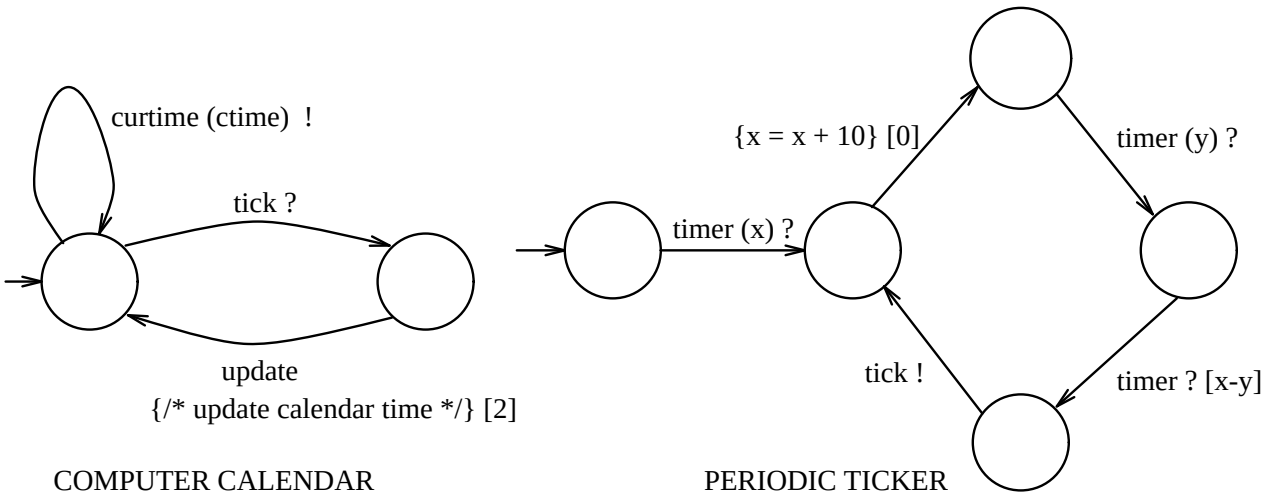



Figure 8: Computer calendar

the periodic timing constraint. Note that this assertion would not catch the violation if the Ticker halted and stopped generating ticks. A stronger/additional assertion would be to check for the existence of a *tick* event at 10 time units offset from every *tick* event.

If Calendar is inundated with requests on channel *curtime*, then Calendar may not receive ticks on time. One can impose a maximum capacity check on *curtime* of say 3 requests per tick interval as follows,

```
when tick
{
   extern int count;
   count = 0;
}

when curtime
{
   extern int count;
   count++;
   assert(count <= 3);
}
```

where *count* is defined in a separate C file that is compiled and then linked with the other simulator files.

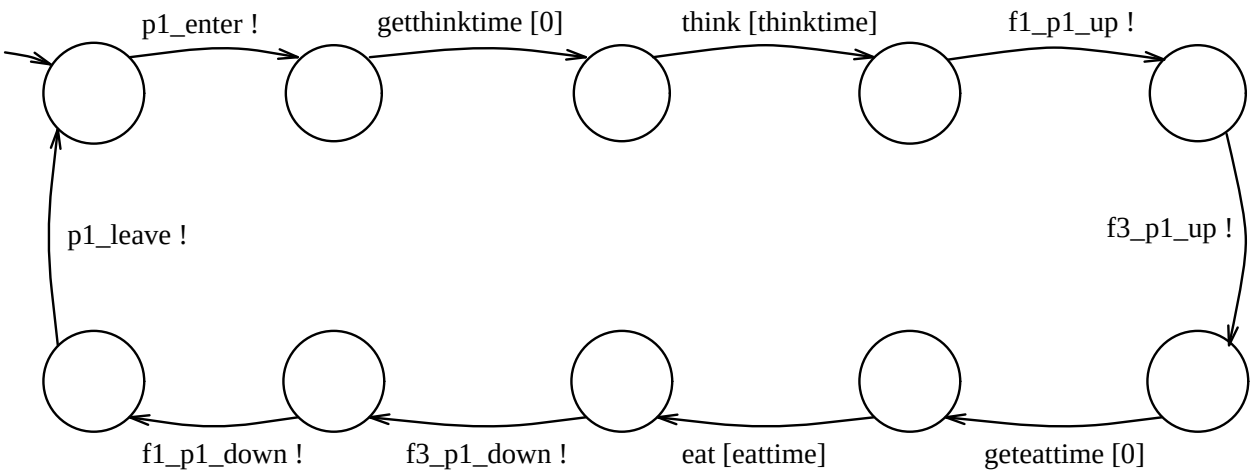



Figure 9: CRSM for philosopher 1

## 5.3 Real-time Dining Philosophers

The following example shows how the information stored in histories can be used to measure resource usage. The basic idea is that code that is invoked in response to channel events gathers statistics. We use a real-time version of the well-known dining philosophers problem [7] to illustrate the idea by measuring the fraction of total time that each philosopher spends on eating.

The example has 3 philosophers, 3 forks and a room. The CRSM for philosopher 1 is shown in Figure 9. First the philosopher requests to enter the room by signaling *p1_enter*. Only 2 philosophers are allowed inside the room at any time to prevent deadlock. *Getthinktime* generates a random think time from the interval 0 to 11 seconds (random() is a UNIX library call). After thinking, the philosopher tries to acquire forks *f1* (signal *f1_p1_up*) and *f3* (signal *f3_p1_up*). The forks are shared with other philosophers and the philosopher waits until they are available. After both forks have been acquired, the philosopher spends a random amount of time eating. Then the philosopher puts down the two forks and leaves the room. This cycle is repeated. The fraction of time philosopher 1 spends eating can be computed as

```
when f3_p1_down
{
        extern int sum_eattime1;
```

```
        int starteat, endeat;

        time("f3_p1_up", -1, &starteat);
        time("f3_p1_down", -1, &endeat);
        sum_eattime1 += endeat - starteat;
        printf("    philosopher 1 eat_fraction = %g\n",
             ((float)sum_eattime1)/ ((float)simulator_time));
}
```

Simulator_time is a predefined variable that denotes current global time. In our simulations of the entire specification, we observed that the *eat_fraction* for philosopher 3 is 0, which shows that the philosopher is being starved. The specification can be modified to avoid this unfortunate situation. Note: As shown here an *eat_fraction* of 0 would not be printed as philosopher 3 would never put down a fork. In the actual specification, the printing of the resource usage is done at a more convenient time.

# 6 Discussion

### Checking Consistency between Specification and Implementation

Assuming that a CRSM specification has been implemented in software, the question arises: how does one check the consistency between the specification and the implementation ? One approach is to execute the specification and implementation using the same test cases and to compare the results. Our assertion methodology permits an alternate and perhaps simpler approach: monitor the same assertions in both the specification and implementation.

If certain assumptions about the coding of the specification are made, then it is possible to compile the assertions into a run-time monitor that checks the assertions in the implementation. This is analogous to automatic code generation from the specification, except that, the code is being generated for the assertions only. The assumptions are:

- the external interface of CRSMs is preserved in the implementation
- the message components of channels and the interface between the corresponding modules in the implementation are the same.

A run-time monitor for checking timing constraints in distributed real-time systems is described in [12]. The run-time monitor is based on RTL formalism, and histories are maintained for events. The assertions that can be checked are delays and deadlines only, unlike the arbitrary constraints in our assertion language. Our assertions can be translated, in a straightforward manner, into a similar monitor. The main change to the monitor described in [12] will be to have it execute user-defined assertion code in response to events. Many

issues have not been addressed, such as the potential interference caused by monitoring, and more research is needed on this topic.

### Limits of Assertion Checking

The major drawback with our assertion checking method is that the assertion may not be checked for all possible executions of the specification, i.e., it is not a verification method. There is always the possibility that further simulations will uncover an unknown bug in the specification. However, automatic verification of models as powerful as CRSMs (equivalent to Turing machines) is usually intractable. We are currently developing verification methods for a restricted model of CRSM — in which the timed reachability graph is finite. Exhaustive testing is a possibility for small but critical parts of the system. For example, in the traffic-light controller, the *getapptime* command can be replaced with a procedure that systematically selects all possible values of arrival time (assume a maximum value) and direction, and runs the simulation (perhaps for days) with the assertion checking enabled. Clearly, such testing can only increase one's confidence in the specification.

A second drawback is that the assertion language cannot express liveness requirements, such as a requirement that states *eventually a light will* turn green. This is because checking of liveness properties requires infinite histories, and our assertions only check finite simulation histories. If there is a deadline on a liveness requirement, for example, a requirement that a light should turn green *within* 2 minutes, then the liveness requirement becomes a safety requirement, and hence it can be checked in our assertion language. In real-time systems safety, not liveness, is of paramount concern; the system should not enter a state where damage can be caused to people or environment. A partial solution for checking liveness is to monitor the usage of resources as described in the real-time dining philosophers example (Section 5.3). In that example, an *eat_fraction* of 0 suggests an absence of liveness.

### Current and Future Work

The simulator and assertion checker have proved useful in understanding and debugging specifications. Based on our experience with the current environment, we plan to extend the above work in several directions.

A short term task is to integrate the graphical front end and the simulator, so that during execution the state changes of CRSMs and channel events are reflected visually in the front end.

We are currently developing automatic verification methods for a restricted model of CRSMs. The restrictions, which enable the construction of a finite timed reachability graph, are: limiting the values of variables to a finite set, and allowing only constants (including 0 and $\infty$) for time bounds of intervals. Most of the specification of the traffic-light example belongs to the restricted category of CRSM, except for the *getapptime* command and the

arrival time (Figure 2). A reachability graph allows verification of properties that can be cast as invariants, such as the mutual exclusion of green/yellow lights and absence from deadlock.

We also intend to refine the CRSM scheme to allow sequential and parallel composition of individual CRSMs into higher-level machines, and to permit the definition of a variety of higher-level paradigms for "specification-in-the-large", such as abstract data types and more general objects. For example, the Controller machine (Figure 3) can be decomposed into two sequential machines: one for normal mode, and another for interrupted mode. Issues in this topic include specifying the communication between component machines, and providing facilities for handling faults or interrupts.

Finally, we hope to use the environment for larger and more practical examples.

# 7 Conclusions

We have described a prototyping environment for specifying, executing and checking CRSMs. The environment serves as a validation of the simulator algorithm and the CRSM examples described in [14]. The environment has also proved useful in understanding and debugging specifications. We defined a novel assertion language for specifying timing assertions, and illustrated its use by specifying common timing constraints and some safety properties. It has also been shown that the assertions can be checked in a practical and efficient manner. Directions for future work include improving the prototyping environment and refining the CRSM scheme.

# Acknowledgements

We thank Farnam Jahanian and Raj Rajkumar for helpful initial discussions. We also thank Becky Callison, Ricardo Pincheira and Travis Craig for valuable discussions and for their detailed comments on the manuscript.

# References

[1] H. Attiya and N. A. Lynch, "Time Bounds for Real-time Process Control in the Presence of Timing Uncertainty", *Proc. IEEE Real-time Systems Symp.*, pp. 268-284, Dec. 1989.

[2] S. Chodrow, F. Jahanian and M. Donner, "Run-time Monitoring of Real-time Systems", *Proc. IEEE Real-time Systems Symp.*, pages 74-83, Dec. 1991.

[3] D. Drusinsky and D. Harel, "Using Statecharts for Hardware Description and Synthesis", *IEEE Trans. on CAD 8,7*, pages 798-807, July 1989.

[4] A. Gabrielian and M. Franklin, "Multilevel Specification of Real-time Systems", *Comm. of the ACM 34,5*, pages 50-60, May 1991.

[5] F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-time Systems", *IEEE Trans. on Software Eng. 12, 9*, pages 890-904, Sept. 1986.

[6] F. Jahanian and A. Goyal, "A Formalism for Monitoring Real-time Constraints at Run-time", *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 148-155, June 1990.

[7] C. Hoare, "Communicating Sequential Processes", *Comm. of the ACM 21,8*, pages 666-677, Aug. 1978.

[8] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[9] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8,3*, pages 231-274, 1987.

[10] D. Harel et al., "Statemate: A Working Environment for the Development of Complex Reactive Systems", *IEEE Trans. on Software Eng. 16,4*,pages 403-414, Apr. 1990.

[11] Luqi, V. Berzins and R. T. Yeh, "A Prototyping Language for Real-time Software", *IEEE Trans. on Software Eng. 14,10*, pages 1409-1423, Oct. 1988.

[12] S. Raju, R. Rajkumar and F. Jahanian, "Monitoring Timing Constraints in Distributed Real-time Systems", *TR 92-09-03*, Dept. of Computer Science and Eng., University of Washington, Sept. 1992. (To appear in *Proc. IEEE Real-time Systems Symp.*, Dec. 1992.)

[13] B. Selic et al., "ROOM: An Object-Oriented Methodology for Developing Real-time Systems", *Proc. International Workshop on Computer-Aided Software Eng.*, July 1992.

[14] A. Shaw, "Communicating Real-time State Machines", *IEEE Trans. on Software Eng.*, Sept. 1992. (An earlier version was published as *TR 91-08-09*, Dept. of Computer Science and Eng., University of Washington, Aug. 1991.)

[15] D. A. Stuart and P. C. Clements, "Clairvoyance, Capricious Timing Faults, Causality and Real-time Specifications", *Proc. IEEE Real-time Systems Symp.*, pages 254-263, Dec. 1991.

[16] R. Swick and M. S. Ackerman, "The X Toolkit: More Bricks for Building User-Interfaces or Widgets for Hire", *Proc. Winter Usenix Conference*, pages 221-228, Feb. 1988.

[17] G. Thomas, "Xsim 2.0 User's Guide", Dept. of Computer Science and Eng., University of Washington, Apr. 1990.

[18] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Trans. on Software Eng. 8,3*, pages 250-269, May 1982.