An Automatic Verification Technique for Communicating Real-Time State Machines*

Sitaram C. V. Raju
Department of Computer Science and Engineering
University of Washington
Seattle WA 98195
sitaram@cs.washington.edu

Abstract

We describe an automatic verification technique for 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 proposed approach is to model the behavior of the system of (an expressive subclass of) CRSMs by a timed reachability graph. The system behavior of CRSMs is characterized by a time-stamped trace of communication events. We provide a decision procedure for verifying timing and safety properties (specified in a notation based on Real-Time Logic) of the reachability graph, and hence of the corresponding system of CRSMs. We also present a condition for the existence of deadlock in a system of CRSMs. Finally, we briefly describe an implementation of a verifier program based on the above algorithms.

Keywords — automatic verification, deadlock, model checking, reachability, real-time, requirements specification.

1 Introduction

The distinguishing characteristic of a hard real-time system is that it must be temporally as well as functionally correct. Subtle timing or functional errors in a real-time system can potentially cause loss of life or can be hazardous to the physical environment. To cope with such problems, formal methods [1, 3, 5, 7, 9] have been developed to specify real-time systems and to verify their properties.

*This research was supported in part by NSF under grant number CCR-9200858.

In this paper, we describe an automatic verification technique for distributed real-time systems using Communicating Real-Time State Machines (CRSMs). CRSMs, introduced in [11], are an executable scheme for specifying the requirements and design of both a real-time system and its physical environment. CRSMs are timed state machines that communicate synchronously over uni-directional channels. A CRSM can have data variables that are local to the machine (i.e., not shared), and do arbitrary computations on the variables. 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 properties on the trace of communication events.

In an earlier work [10] we described a toolset consisting of a graphical editor, a simulator and an assertion checker for prototyping and testing CRSMs. The assertion checker tests for violations over the simulation trace only, i.e., it is not an exhaustive verification technique[1]. This work addresses the shortcomings of the simulator/testing approach by developing an automatic verification method for a *restricted* model of CRSMs. The restrictions of the model are:

1. using discrete time instead of continuous, or dense, time[2].
2. restricting the values of data variables of CRSMs to a finite set (as opposed to unbounded integers or floating point variables for example).

The verification approach is based on model-checking: We represent all possible behaviors of the system of CRSMs by a finite timed reachability graph. By proving timing or safety properties of the reachability graph we prove the same property of the corresponding system of CRSMs.

The main difference between this work and other related work on automatic verification of finite state real-time systems is that our novel specification model, namely CRSMs, is more expressive. For example, both the structure of transitions and the use of clocks in CRSMs are very general as explained in the section on related work (Section 7). We believe that the expressive power of CRSMs will enable us to specify and verify more complex and interesting real-time systems.

The principal contribution of this paper is a method of generating a finite reachability graph for a system of restricted CRSMs. Generating a reachability graph for real-time systems (and for CRSMs in particular) is not trivial because the global state of the system includes a time component, and hence the finiteness assumption is *not* necessarily true. We

[1] Automatic verification of models as powerful as CRSMs (equivalent to Turing machines) is undecidable.

[2] The original model of CRSMs [11] uses continuous time, but the simulator for CRSMs [10] uses discrete time because it simplifies some of the implementation details. The choice of discrete time was fortuitous because the use of continuous time will make automatic verification impossible. Also, the original CRSM model assumes that every machine spends a minimum time δ in every state.

also provide a decision procedure for verifying timing and safety properties of the reachability graph. The properties are specified in an assertion language that is a programming extension of Real-Time Logic (RTL) [6]. The assertion language was first presented in [10], and was shown to be useful for checking properties of a trace of simulation events. By using a common assertion language for both simulation and verification we gain an important advantage: The entire system, which can be too complex to verify (because it is not finite state), can be simulated and the simulation trace tested for timing/safety properties (see [10]); and by focusing on a small but critical portion of the system, which is finite state, the same properties that were tested on the simulation trace can now be verified.

Since the deadlock problem is critical in real-time applications, we present a condition for the existence of deadlock in a system of CRSMs. The condition can be checked in every node of the reachability graph to identify the subset of CRSMs that are deadlocked forever.

The rest of the paper is organized as follows. In the next section, we present the CRSM model. Section 3 describes the construction of reachability graphs. In section 4 we present the decision procedure. The condition for the existence of deadlock is given in section 5. Section 6 describes current and future work. Section 7 discusses related work.

2 CRSM: An Informal Introduction

As an example to describe the CRSM model we use a variation of the mouse clicker system from [11]. The main variation is that the mouse CRSM is more elaborate. The description of the system is:

> A mouse sends down (D) and up (U) clicks to a recognizer. The mouse clicks correspond to a user depressing the mouse button and releasing the mouse button respectively. The recognizer uses the times between D and U signals to decide if it should send a single click (SC), a double click (DC) or neither signal to a handler. If the time between the D and U events is $\leq$ 10ms then it is a single click. A double click is two single clicks separated by an interval $\leq$ 5ms. Double clicks do not overlap and they override single clicks.

The CRSM model is a distributed one, except for a single shared variable representing the current time. CRSMs are timed state machines that run concurrently except when they need to communicate. Machines communicate synchronously (as in CSP [8]) and instantaneously through messages over uni-directional channels. The set of machines for the mouse clicker and its physical environment is shown in Figure 1. The Mouse and Handler machines describe the behavior of the physical environment and the Recognizer machine describes the real-time system. There are four channels: D, U, SC and DC.

Figure 2 shows the CRSMs for the Mouse, Handler and Recognizer. The CRSM for the Handler is shown as a simple state machine; an alternative is a more elaborate machine that

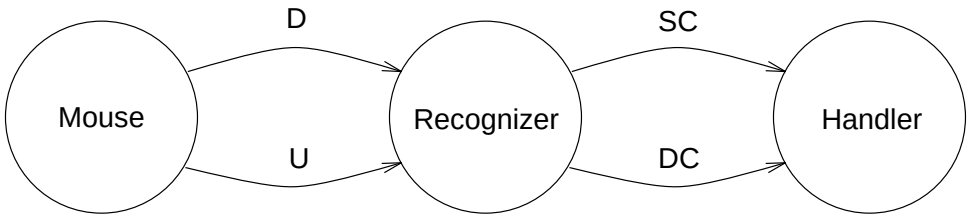

Figure 1: Global view of mouse clicker system

processes the single and double click events. 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 Mouse uses the output command $U!$ and the Recognizer uses the input command $U?$ to communicate with each other. The communication occurs when both the Mouse and the Recognizer are ready, i.e., it is a synchronization signal. A channel can have message components. For example,

temperature(integer centigrade)

When input (say, temperature(deg)?) and output (say, temperature(32)!) occur on the channel, the effect of the IO is the assignment

$$\text{deg} = 32$$

at the receiving machine.

The time interval associated with IO transitions denotes the earliest and latest time IO can occur after entering the state. Suppose Mouse enters the state *up* at time t_{up}. Then the Recognizer must accept the input on channel D somewhere in the time interval $[t_{up}+5, t_{up}+9]$ Otherwise, the Mouse 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 (C language syntax is used in this paper) that changes the values of local variables, and/or it can denote a physical activity. The time of an internal command represents the best and worst case execution time of the command. In the Mouse machine, the internal command *get_x* toggles the value of local data variable x between 5 and 10 (Note: All variables are initialized to 0). This value is used as the time interval $([x, \infty])$ on the succeeding transition to generate the U event. This trick, which is a form of parameterization of the system, generates U event at different times.

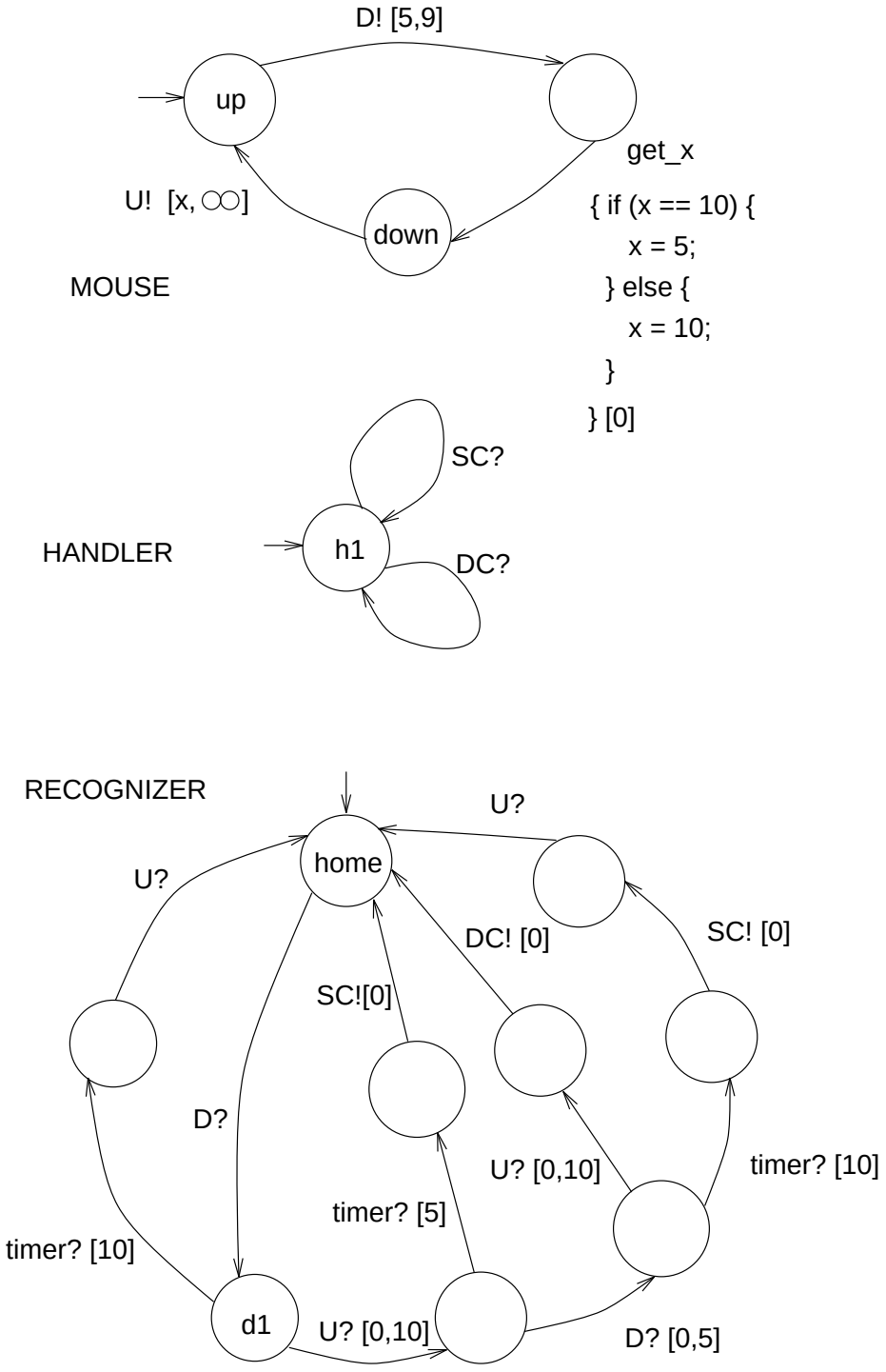

Figure 2: CRSMs for Mouse, Handler and Recognizer

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

timer? [10]

to obtain a timeout after spending 10ms in state *d1* and no IO has occurred on channel U. If the timeout and U event occur simultaneously then one of them will be chosen non-deterministically[3]. Current time can be retrieved, say in the variable t, by a call: timer(t)?

2.1 Formal Model of CRSMs

We now present a more formal description of CRSMs with our restrictions. In the rest of the paper by CRSMs, we mean our restricted CRSMs, and not the CRSMs as defined in [11]. A CRSM is a tuple $(S, S_0, V, I, O, R, C, T)$, where

- S is a finite set of states
- $S_0 \in S$ is the start state
- V is a finite set of integer valued variables. Each variable $v \in V$ has a finite range of values. Also, each variable v is initialized to 0.
- I is a finite set of input channels, including the *timer* channel
- O is a finite set of output channels
- R is the discrete-time clock machine that is associated with the CRSM. R is always ready to do IO on the *timer* channel.
- C is the union of all input, output and internal commands of the CRSM.
- $T \subseteq S \times S \times B(V) \times A(V) \times A(V) \times C$ is the set of transitions. $B(V)$ is the set of Boolean expressions composed from V and constants. $A(V)$ is the set of non-negative integer valued expressions composed from V and constants.

All CRSMs start in their respective start states at time 0. At time t_k a CRSM m_i changes state from s to s' using a transition of the form (s, s', b, a_1, a_2, c). The guard b, which is the enabling condition, must evaluate to *true* in state s. The effect of command c, which can be one of input, output or internal, is as described in the previous section. If c is an internal command then its execution time must be an integer in the interval $[a_1, a_2]$. If c is an IO command then its partner CRSM (say m_j), with whom m_i is doing IO, must also do an IO

[3]When generating the reachability graph (Section 3), however, all possible execution paths, or behaviors, of the system must be considered. In this case the corresponding node in the graph will have two successors: one for the timeout event and another for the U event.

transition at time t_k. Let the corresponding transition of CRSM m_j be $(s_j, s_j', b_j, a_{1_j}, a_{2_j}, c_j)$. Let t_s be the time CRSM m_i entered state s and let t_{s_j} be the time CRSM m_j entered state s_j. Then the two intervals

$$[t_s + a_1, t_s + a_2] \text{ and } [t_{s_j} + a_{1_j}, t_{s_j} + a_{2_j}]$$

must overlap so that IO can happen, and t_k is the earliest possible communication time given by the formula:

$$t_k = \max(t_s + a_1, t_{s_j} + a_{1_j}) \tag{1}$$

The external behavior of a system of CRSMs is given by a trace of IO events:

$$(I_0, t_0), (I_1, t_1) \ldots (I_i, t_i)$$

where, for each (I_i, t_i), I_i is an IO command that has completed at time t_i, and $t_i \leq t_{i+1}$ for all $i \geq 0$.

We avoid the case where time stops advancing by disallowing a cycle of transitions that can be traversed in zero time. A simple way of avoiding this pathological case is by ensuring that the time bounds of all commands is non-zero. Note that in Figure 2 even though state *h1* has self-loops with the default time intervals $[0, \infty]$, the synchronization requirement with the Recognizer machine (Formula 1) ensures that the time spent in state *h1* is always non-zero.

3 Constructing a Finite Reachability Graph

A reachability graph contains all possible states that can be reached starting from the start state of the system. The nodes of the graph represent the global state of the system and the edges represent transitions from one global state to another global state.

We first present the algorithm for constructing a finite reachability graph for the case where the lower and upper time bounds of a transition are constants (including 0 and ∞). In Section 3.2 we extend the algorithm to the case where the time bounds on transitions can be arbitrary expressions.

To account for timing constraints on transitions, we augment a node of a reachability graph with the time spent in each machine state. A node of a reachability graph consists of :

1. current state of each machine
2. current values of local variables
3. time spent by each CRSM in its current state

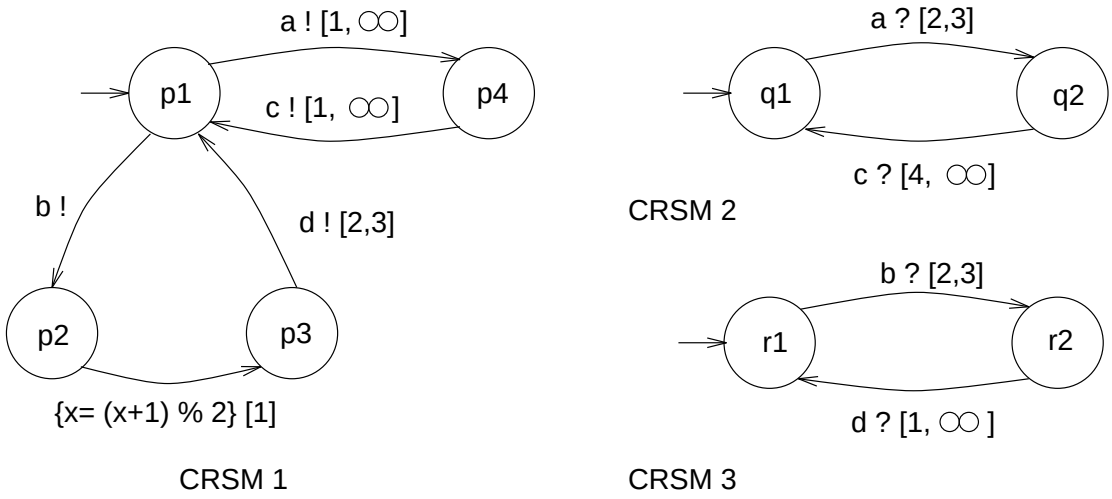

Figure 3: CRSMs example

For the system of CRSMs shown in Figure 3, a node of the reachability graph is the tuple

(s1, t1, x, s2, t2, s3, t3)

where,

s1: current state of CRSM 1
t1: time spent in current state of CRSM 1
x: value of variable x
s2: current state of CRSM 2
t2: time spent in current state of CRSM 2
s3: current state of CRSM 3
t3: time spent in current state of CRSM 3

The start node is (p1, 0, 0, q1, 0, r1, 0). Figure 4 shows the reachability graph for the system of CRSMs.

We use the semantics of CRSMs to generate the successors of a node. Reference [11] describes a simulation algorithm that serves as the operational semantics of CRSMs; the algorithm chooses a single successor nondeterministically. We now present an extension of the simulation algorithm. The extension is that instead of making a single nondeterministic choice, all possible successors of a node are generated, i.e., all possible simulation scenarios are considered. The new algorithm will prune away nodes that cannot be reached. Note that some nodes cannot be reached because of timing constraints imposed on transitions. This fact differentiates reachability algorithms for real-time systems from reachability algorithms for concurrent (but untimed) systems.

The algorithm for generating all possible successors from a node is:

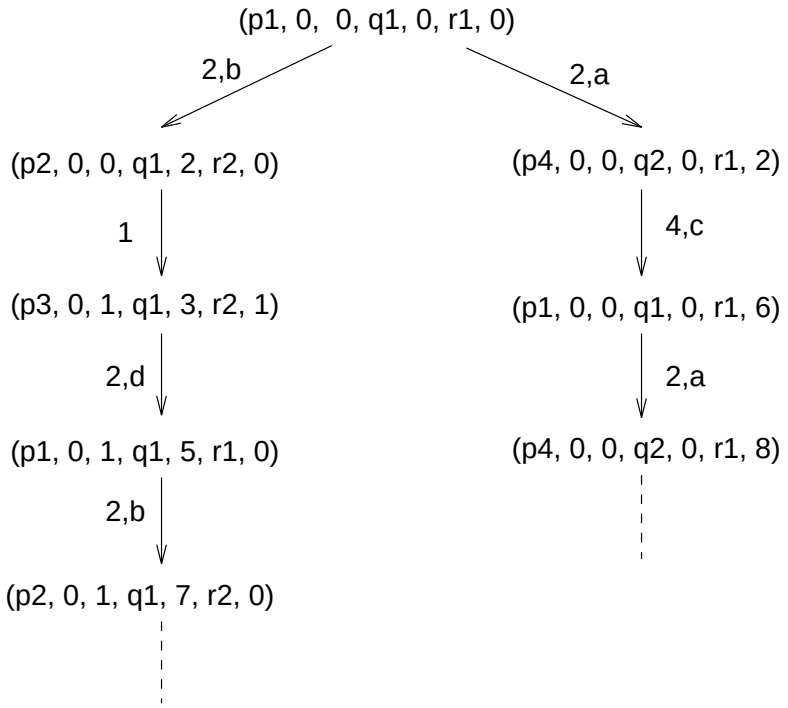

Figure 4: CRSMs reachability graph

Algorithm 1 (All Successors)

1. For each CRSM, find all possible next events.
2. Calculate the times of next events. If the next event is the completion of an internal action with time bounds $[t_1,\ t_2]$, then the actual execution time of the action can be one of $t_1, t_1 + 1, t_1 + 2, \ldots t_2$. If the next event is an IO then it is necessary to check if it is an actual event, i.e., its IO partner must also be ready to engage in IO. Calculate the communication time of the IO event according to Formula 1 (Section 2.1).
3. Compute the global earliest time among all possible next events.
4. From the possible next events find the set of events that can happen at the global earliest time.
5. Determine all maximal matchings of the above set of events as follows:
   (a) Construct a graph $G = (V, E)$ from the set of events where the vertices correspond to the CRSMs in the system and the edges correspond to the events. If there is an IO between CRSMs c_i and c_j on channel *chan*, then there is an edge between c_i and c_j labeled *chan*. If CRSM c_i has an internal action *inta*, then there is a self-edge on c_i labeled *inta*.

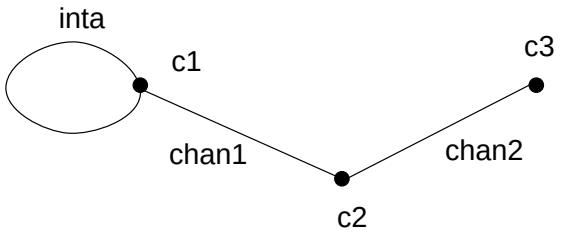

Figure 5: Graph for finding all maximal matchings

(b) Find all maximal matchings of graph G. (A maximal matching is $E^{'} \subseteq E$ such that no two edges in $E^{'}$ share a common endpoint and every edge in $E - E^{'}$ shares a common endpoint with some edge in $E^{'}$.)

6. The events in each maximal matching are executed, and a successor of the current node obtained.

To illustrate Step 5 of the algorithm consider a system of three CRSMs and let the possible next events (at the global earliest time) in the current state be:

- CRSM c_1 can start executing internal action *inta*.
- CRSMs c_1 and c_2 can do IO on channel $chan1$.
- CRSMs c_2 and c_3 can do IO on channel $chan2$.

Figure 5 shows the graph; it has two maximal matchings {chan1} and {inta, chan2}.

In the above algorithm, finding all maximal matchings is intractable. This is established by the following theorem.

Theorem 1 *Finding all maximal matchings in a graph is NP-hard.*

Proof: By restricting to the Minimum Maximal Matching (MMM) problem [4]. The MMM problem is: Given a graph $G = (V, E)$ and a positive integer K, is there a subset $E^{'} \subseteq E$ with $|E^{'}| \leq K$ such that $E^{'}$ is a maximal matching ? To decide if an instance of MMM has a solution, we simply run the all maximal matchings algorithm on graph G, and check if there is any maximal matching with size $\leq K$. ■

Each edge in the reachability graph is assigned a weight that is the relative time of the next event. Also, the edges are labeled with the channel names on which IO events have occurred. This information is used by the decision procedure in Section 4.

When generating successor nodes, a node may turn out to be an instance of an earlier (i.e., already generated) node. If the corresponding components of two nodes (i.e., states, time values and data values) are the same then the nodes are instances of each other. Putting the ideas in this section together, we obtain the following algorithm for constructing a reachability graph.

Algorithm 2 (Reachability Graph Construction)

```
Construct N_start the initial node of the graph, mark N_start as an unexpanded node
While there are unexpanded nodes do {
   Choose an unexpanded node, say N
   Generate all successors of node N (Algorithm 1)
   For each successor N_suc of N {
      If there exists a previously generated node (say N_prev) such
      that the corresponding components of N_suc and N_prev are the same
         then
            Add an edge from N to N_prev
         else
            Create a new node N_suc, add an edge from N to N_suc
            Mark N_suc as an unexpanded node
   }
   Mark N as an expanded node
}
```

If a CRSM queries a discrete-time clock machine for the value of current time by a call *timer(x) ?* (x is set to the value of current time), then the reachability graph is not finite because clock time is unbounded. In this case, only a partial graph can be generated; this corresponds to finding all behaviors of the system from start time until a fixed limit on clock time. Even if the discrete-time clock machine is used only for the purpose of achieving timeouts (e.g., the mouse clicker system), it is still not clear if Algorithm 2 will terminate, because the set of unexpanded nodes may never become empty. Figure 4 illustrates this point; the time in state $q1$ or state $r1$ increases monotonically with no end in sight.

In the sequel we assume that the discrete-time clock machine is used for doing timeouts only. Note that this assumption follows from the earlier assumption that the range of values of a data variable is finite.

In Figure 3, after CRSM 2 has spent 4 time units in state $q1$, the exiting transition from $q1$ cannot be taken (i.e., the machine is deadlocked). Also, after CRSM 2 has spent 4 time units in state $q2$, it is ready to engage in IO on channel c whenever its partner (CRSM 1) is ready. These examples suggest that there is a time (say $threshold$) associated with each state of a machine, such that, after a machine spends $threshold$ time units in the state, all primed[4] IO transitions from the state either:

1. can perform IO whenever their partner is ready to engage in IO, or
2. can never perform IO because the upper time bound on the transition has been exceeded.

[4]A transition is primed for execution if its guard evaluates to *true*.

We use this property to ensure that the timed reachability graph is finite.

For a given transition with time bounds $[l, u]$, define the *bound_val* of the transition to be

$$\text{bound\_val} = \begin{cases} l & \text{if } u = \infty \\ u+1 & \text{if } u \neq \infty \end{cases}$$

For all exiting transitions i from a state, let *threshold* be

$$\text{threshold} = \max(\text{bound\_val}_i) \tag{2}$$

For example, the *threshold* value for state $p3$ in Figure 3 is 4. We use the value '*' to denote the case where the time spent in a state is $\geq$ the *threshold* value for the state. Hence, the possible values of time spent in a state are 0, 1, 2, ...,threshold-1, *.

Two nodes are defined to be *instances* of each other if the corresponding components have the same value or if the corresponding times are both '*'. Nodes that are instances of each other have exactly the same successors. This property is proved in Section 3.1.

Algorithm 3 (Reachability Graph Construction)

This algorithm is the same as Algorithm 2, except that the criteria for judging when a node is an instance of another node is extended to include the value: *.

Figure 6 is the reachability graph for CRSMs of Figure 3 generated by Algorithm 3; it is finite.

3.1 Proof of Finiteness of Reachability Graph

In this section, we prove that the reachability graph generated by Algorithm 3 is finite. We first prove the following lemma.

Lemma 1 *If two nodes, n and m of the reachability graph have the same values for corresponding components (including '*'), then the nodes are instances of each other, i.e., they have exactly the same successors.*

Proof. The proof consists of showing that the trees rooted at nodes n and m in the reachability graph are identical to each other. Let the components of n and m be

$$\text{n} : (\text{s}_{\text{n1}}, \text{d}_{\text{n1}}, \text{t}_{\text{n1}}, \text{s}_{\text{n2}}, \text{d}_{\text{n2}}, \text{t}_{\text{n2}}, \ldots \text{s}_{\text{nk}}, \text{d}_{\text{nk}}, \text{t}_{\text{nk}})$$

$$\text{m} : (\text{s}_{\text{m1}}, \text{d}_{\text{m1}}, \text{t}_{\text{m1}}, \text{s}_{\text{m2}}, \text{d}_{\text{m2}}, \text{t}_{\text{m2}}, \ldots \text{s}_{\text{mk}}, \text{d}_{\text{mk}}, \text{t}_{\text{mk}})$$

where s_{n1}, s_{m1} are the CRSM states, d_{n1}, d_{m1} are the data variables (the proof easily extends to the case of an arbitrary number of data variables per machine), and t_{n1}, t_{m1} are the times

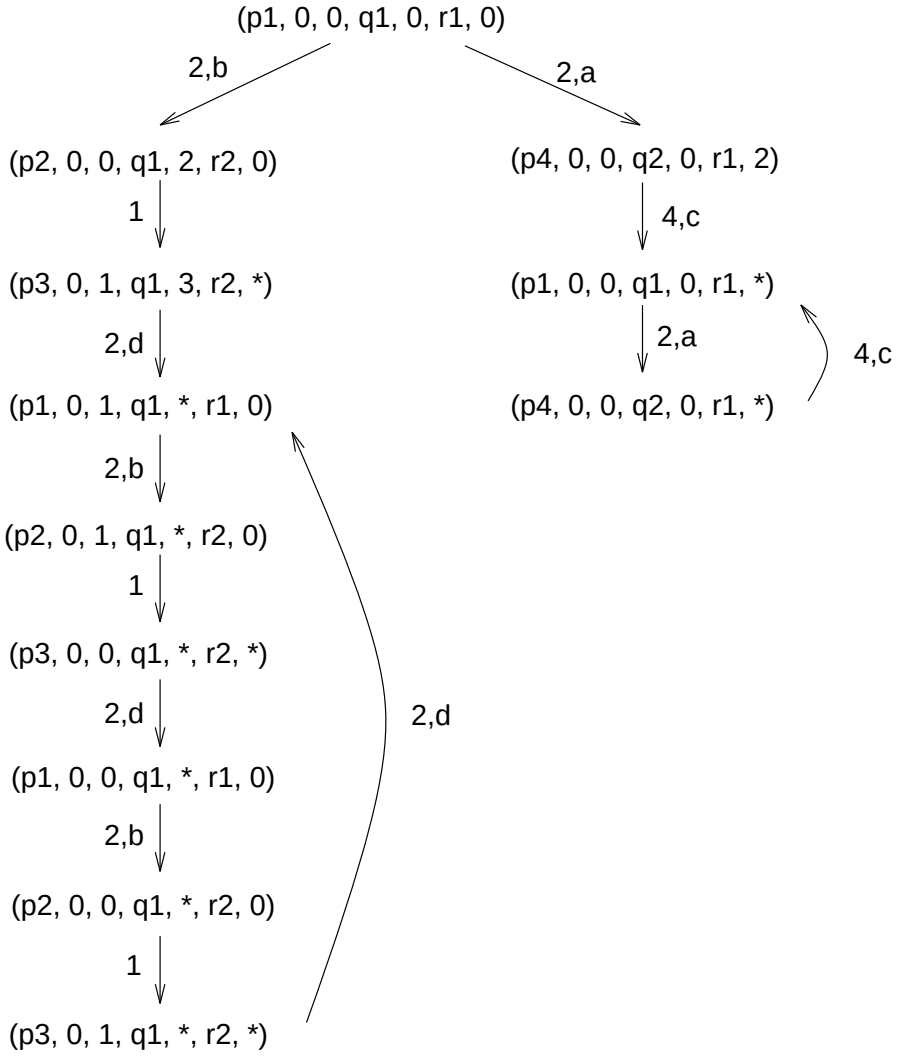

Figure 6: CRSMs reachability graph (finite)

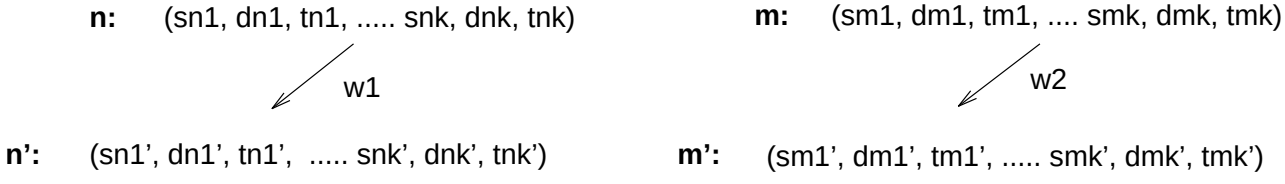

Figure 7:

spent in the current state. Note that $s_{n1} = s_{m1}$, $d_{n1} = d_{m1}$ and $t_{n1} = t_{m1}$. Clearly, the interesting case is when $t_{n1} = t_{m1} = *$.

We first consider the case of a successor node of n and m being generated by an IO (say on channel a) between the machines whose current states are s_{n1} and s_{n2} respectively. In node m the corresponding states are s_{m1} and s_{m2}. Figure 7 shows the nodes n, m and their successors nodes. Figure 8 shows the relevant portions of the state diagrams of the two machines. There may be other transitions from (and into) states s_{n1} and s_{n2}, but these are not of interest and they are not shown. Figure 9 shows the corresponding machines of node m. The values of $t_{n1}, t_{n2}, t_{m1}, t_{m2}$ can be categorized into four cases :

1. t_{n1}, t_{m1} are between 0 and the *threshold* value for s_{n1}, and t_{n2}, t_{m2} are between 0 and the *threshold* value for s_{n2}. The time of IO on channel a is given by Formula 1 (Section 2.1). Since $t_{n1} = t_{m1}$ and $t_{n2} = t_{m2}$ it follows that the time of IO in Figures 8 and 9 will be the same.

2. t_{n1}, t_{m1} are between 0 and the *threshold* value for s_{n1}, and t_{n2}, t_{m2} are $\geq$ the *threshold* value for s_{n2}, i.e., $t_{n2} = t_{m2} = *$. Here, if $l2 = \infty$ then machine 2 is ready to do IO whenever machine 1 is ready to do IO. If $l2$ is finite then since the *threshold* value is greater than $l2$ (by definition), there can be no IO on a as long as machine 2 is in state s_{n2}. That is if IO can occur on channel a the IO time will be the same in Figures 8 and 9.

3. t_{n1}, t_{m1} are $\geq$ the *threshold* value for s_{n1}, and t_{n2}, t_{m2} are between 0 and the *threshold* value for s_{n2}. The reasoning here is similar to case 2 above.

4. The values of t_{n1}, t_{m1}, t_{n2} and t_{m2} are all equal to $*$. There are again four cases to consider based on the values of k_2 and l_2. The cases are:

   (a) Both k_2 and l_2 are finite. Since the *threshold* value of state s_{n1} is greater than k_2, and the *threshold* value of state s_{n2} is greater than l_2, there can be no IO on channel a.

   (b) k_2 is finite and $l_2 = \infty$. Again, since the *threshold* value of state s_{n1} is greater than k_2, there can be no IO on channel a.

   (c) $k_2 = \infty$ and l_2 is finite. The reasoning here is similar to case (b) above.

   (d) Both $k_2 = \infty$ and $l_2 = \infty$. This case cannot occur because it means that both machines are ready to do IO on channel a. Hence the IO would have already occurred from that state at an earlier time.

Based on the above we have shown that if IO between CRSMs 1 and 2 is the only possible event to occur in nodes n and m, then $s'_{n1} = s'_{m1}$, $t'_{n1} = t'_{m1} = t'_{n2} = t'_{m2} = 0$, and the weight of the edges $w1$ and $w2$ (which is the time of next event) will be the same. Now, $d'_{n1} = d'_{m1}$ because the corresponding machines are performing outputs. Also, $d'_{n2} = d'_{m2}$ because the

Figure 8:

Figure 9:

value of the message component (if any) will be the same in Figures 8 and 9. Since the remaining components of nodes n and m do not take part in a state transition, they will remain in the same state and their times will be incremented by the weight $w1$, and converted to * if the times are above a threshold.

We now consider the case of a successor node of n and m being generated by an internal action, say *intact*. The start of an internal action is modeled by a transition (which takes 0 time) to a brand new state. The brand new state (say s_{start}) signifies that an internal action is in progress. In state s_{start} the completion of *intact* is the only possible event. It is straightforward to show that in this case also, the successor nodes of n and m have the same corresponding components.

It is possible for two or more events to occur simultaneously in a system. Step 5 of Algorithm 1 determines the sets of events that can be executed to yield the successors of the current node in graph. The reasoning presented above for the cases of IO and internal events can be composed to show that if the same set of events happens in nodes n and m then the successors of n and m will have the same corresponding components, and that the weight on the edges will be the same.

We have shown that the successor nodes of n and m , say $n^{'}$ and $m^{'}$, have the same corresponding components. Hence, the above argument can be applied to $n^{'}$ and $m^{'}$ to show that the successor nodes of $n^{'}$ and $m^{'}$ have the same corresponding components. Thus, by induction on the level of subtree below n and m, it can be seen that the trees rooted at n and m are identical to each other. ■

Theorem 2 *The timed reachability graph is finite.*

Proof: The node of a reachability graph is a tuple and each element of the tuple is either a state, or a value of a data variable, or time spent in a state. Each CRSM has a finite number of states and each variable has a finite range of values. Lemma 1 shows that the time values also have a finite range, namely 0, 1, ...$threshold - 1$, *, where $threshold$ is a fixed maximum value for a given state of a CRSM. Therefore the number of possible tuples (or equivalently nodes) is finite. Hence the timed reachability graph is finite. ■

3.2 Extension to Expressions on Time Bounds of Transitions

Notice that the proof of finiteness of reachability graphs does not rely on transitions having constant time bounds. The calculation of $threshold$ value, however, depends on the transitions having constant time bounds. We can permit an expression in a transition time bound if the user can provide the maximum value of the expression. The maximum value can be used to find the $threshold$ value for the state. The maximum value can be checked during graph construction and if the expression value exceeds the user given bound, an error can be flagged.

For example, in the mouse clicker specification (Figure 2) the user can state that the maximum value of lower time bound (x) from state *down* is 10. Hence, the $threshold$ value for state *down* is 10 (Formula 2). In this case since the expression is simply a variable the maximum value of the expression can be deduced from the range of values of x and no user intervention is required.

4 Decision Procedure

A key advantage of a finite reachability graph is that algorithms can be developed for proving properties of the graph and this implies proving properties of the original system of CRSMs. An example of such a procedure is determining if all nodes in a reachability graph satisfy a certain property, i.e., invariant checking.

Many logics have been proposed to express properties of real-time systems (for example RTL[7], RTTL[9]). Decision procedures developed for these logics (or for a subset of these logics) can be used to determine if our reachability graphs satisfy a given property. In this section we specify properties using an assertion language that is a programming extension of Real-Time Logic (RTL) [6]. The assertion language was introduced in [10], and was shown to be useful for checking properties of a trace of simulation events. As mentioned earlier (Section 1) the use of a common assertion language for simulation and verification is advantageous.

In the mouse clicker system a double-click must be emitted only when there are two single-clicks that are separated by less than 5ms. The property is stated as an *assertion* that must hold true over all possible traces of the system. In our syntax, the assertion is stated:

```
when DC
{
  int tu1, tu2, td1, td2;

  time("down", -1, &td1);
  time("down", -2, &td2);
  time("up", -1, &tu1);
  time("up" -2, &tu2);
  assert(td2-tu2 <= 10);
  assert(td1-tu1 <= 10);
  assert(tu2-td1 <= 5);
}
```

An assertion consists of two parts: a *when* clause and a 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 whenever there is a communication on channel *DC*. The second part of the assertion, the user-defined C procedure, gives the constraint. The constraint checking procedure in the above assertion uses the predefined function *time* to get the relative times[5] (from the occurrence of the event on channel *DC*) of channel events, and then tests the values (using function *assert*) to see if the desired relation holds. *Assert* checks if its parameter evaluates to *true*. The assertion is true if all executed *assert* statements evaluate to true.

The relative times and values of message components of previous channel events can be obtained by the following functions:

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

Function *time* returns the communication time and function *value* returns the values of message components. 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. *Ind* is always negative, and it refers to the ind^{th} most recent occurrence of the event. For example, an *ind* value of -1 refers to the last occurrence of an event. *Result* contains the returned result. The functions return a value *err* if the desired event has not yet happened.

In the double-click assertion presented earlier, the times of the two previous down and up events are obtained. Then the code checks there are two single-clicks (D and U separated by 10ms or less), and that the single-clicks are separated by $\leq$ 5ms. For example, consider the following simulation trace,

$$\ldots(D,12),(U,19),(D,21),(U,25),(DC,25)\ldots$$

[5]There are some minor differences between the above assertion language and the one described in [10]. For example, the *time* function in [10] returns absolute time and not relative time.

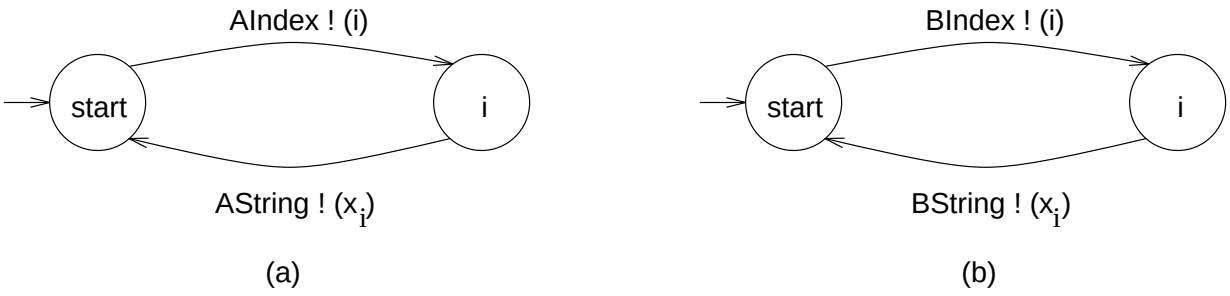

Figure 10: Transitions for strings A and B

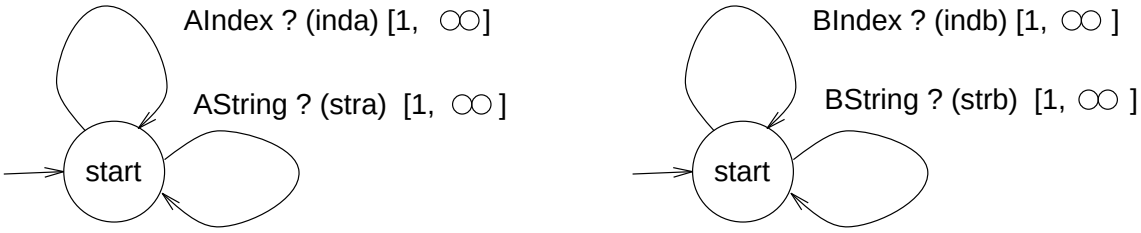

Figure 11: CRSMs 3 and 4

where the times are absolute times. The values of $td1, td2, tu1$ and $tu2$ are 4, 13, 0 and 6 respectively. For these values all three *assert* statements evaluate to *true*. Hence the assertion is true for this particular trace.

The same assertion can be checked on a reachability graph. The reason is that the graph is a compact finite representation of all possible simulation traces of the system. A path in the graph starting from the start state represents one possible simulation trace, or behavior, of the system. If the assertion evaluates to *true* for all possible simulation traces of the graph then the assertion is true for the graph, and hence for the system of CRSMs.

Finding all possible traces from the reachability graph can be done as follows. First every edge (say from node n_1 to node n_2) that is labeled with the channel event that triggered the checking must be found. Alternatively, this information can be computed once and for all when the graph is being created. From node n_2 start searching backwards and recreate all possible simulation traces that lead up the to node n_2. The simulation trace needs to be recreated only so far into the past that the values of all time/value statements in the assertion can be computed. If the assertion is to be checked at a time offset from node n_2, then one needs to search forward from node n_2 and use the weights on edges to find every state that can be reached at the time offset. The simulation traces must be recreated from each such offset state.

Unfortunately, the presence of cycles in the reachability graph makes the verification problem undecidable. This is shown below.

Theorem 3 *Verifying arbitrary assertions for a system of CRSMs is undecidable.*

Proof Sketch: By reduction from Post's Correspondence Problem (PCP), which is known to be undecidable. An instance of PCP consists of two lists : $A = x_1, x_2, \ldots, x_m$ and $B = y_1, y_2, \ldots, y_m$ on an alphabet $\sum$. An instance of PCP has a solution if and only if there is a finite sequence of integers $i_1, i_2, \ldots, i_l$ such that $x_{i_1} x_{i_2} \ldots x_{i_l} = y_{i_1} y_{i_2} \ldots y_{i_l}$.

Given an arbitrary instance of PCP, we construct four CRSMs. CRSM 1 has a start state, and for every string x_i in A (note that a string can be encoded into an integer), add a state i and two transitions shown in Figure 10a. Similarly, CRSM 2 has a start state and for every string y_i in B, add a state i and two transitions shown in Figure 10b. CRSMs 3 and 4 are shown in Figure 11. The following property checks for the absence of a simulation trace where the concatenation of strings sent on channels AString and BString match.

```
when AString
{
   int i, j, indexa, indexb, indicesmatch = true;
   for (;;) {
      i = -1;
      for (j = -1; j>=i; j- -) {
         value("Aindex","inda", j, &indexa);
         value("Bindex","indb", j, &indexb);
         if (indexa != indexb)
            indicesmatch = false;
      }
      if (indicesmatch)
         if (stringsmatch(j)) { /* does concatenation of strings on channels
                                   AString and BString match ? */
            assert(0);          /* assert(false) */
         }
      i = i -1;
   }
}
```

Thus, from an arbitrary PCP we have constructed a system of CRSMs and a property. It is a straightforward exercise to show that the property is false if and only if the instance of PCP has a solution. ■

Since the assertion language is in general undecidable, we identified an expressive subclass (which includes the double-click property of the mouse clicker) whose satisfiability is decidable. One restriction is to do away with loops in the assertion, i.e., the user defined C procedure cannot contain loops. Also, we bound the number of times we traverse a cycle in the reachability graph as follows: Let ind be the smallest index among all time/value statements in the assertion, and let $absind = |ind|$. We restrict the number of traversals of a cycle in the reachability graph to a maximum of $absind$ times. If a cycle in the graph

has been traversed *absind* times and not all time/value statements have been accounted for then the property cannot be checked. With these restrictions the algorithm for checking properties is as outlined earlier: from every trigger node search backwards and recreate all possible simulation traces that lead up to the trigger node, and then evaluate the assertion on all the traces.

5 Deadlock Detection

In this section we present a condition for detecting deadlock in CRSMs. As each new node in the reachability graph is created, the condition can be used to identify the CRSMs that are deadlocked in the node.

If the entire system is deadlocked then there is no next event, or successor state. This condition is detected easily by the all successors algorithm (Algorithm 1, Section 3). It is also possible for a subset of CRSMs in a system to be deadlocked. A CRSM is deadlocked if none of the transitions in its current state can be executed. This can happen if for each transition from the current state of the CRSM either:

1. the guard evaluates to *false*, or
2. the guard evaluates to *true*, the command is an IO, and the CRSM is unable to do IO because the time it has spent in the current state exceeds the upper time bound on the transition.

A more interesting case in shown in Figure 12a, where CRSMs $m1$ (in state $s1$) and $m2$ (in state $s3$) are deadlocked waiting for IO on different channels (a and b). To find such deadlocked CRSMs we first represent the state of the entire system by a wait-for graph $G = (V, E)$. (Recall that a node of a reachability graph represents a particular system state.) For each CRSM there is a vertex $v \in V$. Each edge in the graph is defined as : if CRSM v_i is waiting indefinitely (i.e., its partner is not ready) to do IO on a channel c with CRSM v_j, then there is an edge, labeled c, from v_i to v_j. We show that a knot in the wait-for graph is a sufficient condition for deadlock. (A knot K is a subset of graph G such that every vertex in K is reachable from every other vertex in K, and no vertex outside K is reachable from within K.) The wait-for graph for CRSMs in Figure 12a is shown in Figure 12b.

Theorem 4 *A knot K in a wait-for graph is a sufficient condition for deadlock.*

Proof. Consider the set of channels C on which the CRSMs in knot K are waiting to do IO. By the definition of a knot, there are no edges out of K. Therefore all channels in C belong to the CRSMs in K. So, no CRSM outside K can do IO on the channels in C. Hence all CRSMs in K cannot progress further, i.e., they are deadlocked. ■

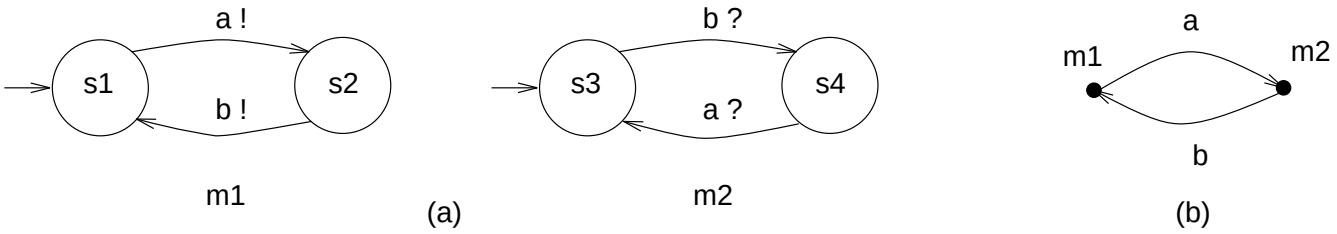

Figure 12: CRSMs and their wait-for graph

Note that a knot is not a necessary condition for deadlock. For example, consider a system of two CRSMs, and, say that the guards on all transitions are not enabled. In this case there is no knot in the wait-for graph, but the system is deadlocked nevertheless.

6 Current and Future Work

We have implemented a prototype verifier program (in C++ on a Decstation 5000/125) based on the theory described in the previous sections. A graphics editor has been developed for creating CRSM specifications. The graphical representation is translated into a text form that serves as the input to the verifier. We have used the verifier to check the mouse clicker system (Section 2) and the traffic-light controller described in [10]. For the mouse clicker system the verifier generates a reachability graph with 29 nodes in about 50ms. The double-click property of the system (Section 4) and another property relating to single-clicks were verified. For the traffic-light controller we checked properties relating to mutual exclusion, delays and deadlines. Additional experimentation with more complex systems is planned to determine the usefulness of our method and to understand the expressive power of our assertion language. The details of the verifier program will be described in another paper.

A concern with our verification method (as with many other mechanical verification techniques) is that the reachability graphs may be too big for realistic systems. We are considering techniques such as partial graph generation and minimal model checking [2] to ameliorate the problem.

7 Related Work

We give a brief survey and comparison of verification techniques for specification methods that are state machine based, because these methods are most closely related to our work. The main difference between this work and other related work is that the CRSM specification scheme is different; CRSMs cover some needs (of real-time systems) that other schemes do not, as discussed below.

Modechart is a specification language that partitions the state space of a real-time sys-

tem into modes. A verification method for Modechart is presented in [7]. Unlike CRSMs, Modechart does not allow IO events to have message components. Also, Modechart uses a shared memory model (with broadcast for event communication), whereas CRSMs present a distributed model. In addition, the time bounds on transitions in Modechart are constants, whereas CRSMs permit arbitrary expressions.

Communicating Shared Resources (CSR) is a formalism that is also based on CSP. An interesting aspect of CSR is that it permits the specification of resource constraints (e.g. CPU constraints) in a real-time system. A reachability analyzer for CSP is described in [5]. CSR appears to have no provision for performing computations in commands. Also, time is not associated with IO.

Timed IO automata [3] have some similarities with CRSMs in that their transitions can be one of input, output or internal actions, but the detailed definitions of transitions are different. For example, timed I/O automata are input enabled, which means that they are unable to block inputs. This is in contrast with CRSMs, where inputs can be blocked until the machine reaches an appropriate state. Also, we are not aware of a mechanical procedure for verification of timed I/O automata.

Timed Automata [1] is a specification method that uses continuous, or dense, time instead of discrete time. Another major difference between timed automata and CRSMs is that timed automata do not permit data variables, and arbitrary expressions on time bounds of transitions and assignment statements.

Finally, the new model of time in CRSMs (both for describing the passage of time and for specifying timeouts) also distinguishes the model from the ones listed above.

Traces are used for reasoning about the behavior of general systems in [8], and for real-time systems in [3, 10, 11]. Our method of reasoning with traces is noteworthy because it is based on RTL. RTL is particularly well suited for real-time systems because it deals directly with event times and can differentiate multiple occurrences of the same event.

8 Conclusions

We have described an automatic verification technique for CRSMs. The approach is to create a reachability graph that represents all possible behaviors of the system, and to use the graph to verify timing and safety properties of the system. The graph is also used to detect deadlock in CRSMs.

We have a prototype implementation of a verifier program. The verifier along with the simulator program described in [10] can be used to prototype, simulate/test and verify distributed real-time systems as follows: The entire real-time system, which may not be finite state (and therefore is difficult to verify), is simulated and tested. Next small (but critical) portions of the system, which are finite state, are verified.

Acknowledgements

I thank Prof. Alan Shaw, my advisor, for numerous fruitful discussions. I also thank Becky Callison, Travis Craig, Ricardo Pincheira, Alan Shaw and Rakesh Sinha for their helpful comments on the paper.

References

[1] R. Alur and D. Dill, "The Theory of Timed Automata", *Real-Time: Theory in Practice, Proc. REX Workshop, LNCS 600*, pages 45-73, Springer-Verlag, June 1991.

[2] R. Alur et al., "An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness", *Proc. IEEE Real-Time Systems Symp.*, pages 157-166, IEEE Computer Soc. Press, Dec 1992.

[3] 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.*, pages 268-284, IEEE Computer Soc. Press, Dec. 1989.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[5] R. Gerber and I. Lee, "A Layered Approach to Automating the Verification of Real-Time Systems", *IEEE Trans. on Software Eng. 18,9*, pages 768-784, Sept. 1992.

[6] 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.

[7] F. Jahanian and D. A. Stuart, "A Method for Verifying Properties of Modechart Specifications", *Proc. IEEE Real-Time Systems Symp.*, pages 12-21, IEEE Computer Soc. Press, Dec 1988.

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

[9] J. S. Ostroff, "Real-Time Temporal Logic Decision Procedures", *Proc. IEEE Real-Time Systems Symp.*, pages 92-101, IEEE Computer Soc. Press, Dec 1989.

[10] S. C. V. Raju and A. C. Shaw, "A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines", *TR 92-10-03*, Dept. of Computer Science and Eng., University of Washington (submitted for publication).

[11] A. C. Shaw, "Communicating Real-Time State Machines", *IEEE Trans. on Software Eng. 18,9*, pages 805-816, Sept. 1992.