Automated Derivation of Time Bounds in Uniprocessor Concurrent Systems

George S. Avrunin, James C. Corbett, Laura K. Dillon, *Member, IEEE,* and Jack C. Wileden, *Senior Member, IEEE*

Abstract— **The successful development of complex real-time systems depends on analysis techniques that can accurately assess the timing properties of those systems. This paper describes a technique for deriving upper and lower bounds on the time that can elapse between two given events in an execution of a concurrent software system running on a single processor under arbitrary scheduling. The technique involves generating linear inequalities expressing conditions that must be satisfied by all executions of such a system and using integer programming methods to find appropriate solutions to the inequalities. The technique does not require construction of the state space of the system and its feasibility has been demonstrated by using an extended version of the constrained expression toolset to analyze the timing properties of some concurrent systems with very large state spaces.**

Index Terms— **Concurrent systems, real-time systems, automated analysis, timing analysis, linear inequalities, integer programming, finite state systems.**

I. Introduction

As the use of real-time software systems grows, there is increasing need for analysis methods to accurately assess the timing properties of such systems. This is true for both "hard" real-time systems, in which inability to produce results within specified deadlines is tantamount to failure, and "soft" real-time systems, in which inability to produce results within specified deadlines is tantamount to failure, and "soft" real-time systems, in which the utility of the results produced declines as the time required to produce those results increases [25]. For instance, even real-time systems relying on sophisticated scheduling mechanisms require accurate predictions (produced either *a priori* or at run-time) of the timing properties of the units of computation that they are attempting to schedule. The problem of determining timing properties is, however, already a difficult one for sequential software, and the introduction of concurrency increases that difficulty significantly by greatly increasing the number of possible executions that must be considered.

Manuscript received September 10, 1993; revised July 1994. This research has been supported by grants from the National Science Foundation, the Office of Naval Research, and the Defense Advanced Research Projects Agency. Recommended by J. Gannon.

G. S. Avrunin and J. C. Wileden are with the University of Massachusetts at Amherst, MA 01003 USA; e-mail: avrunin@math.umass.edu, wileden@cs.umass.edu.

J. C. Corbett is with the University of Hawaii at Manoa, Manoa, HI 96822 USA; e-mail: corbett@hawaii.edu.

L. K. Dillon is with the University of California at Santa Barbara, CA 93106 USA; e-mail: dillon@cs.ucsb.edu.

IEEE Log Number 9404462.

One approach has been to study particular algorithms for scheduling the execution of different processes, and to try to determine whether a given concurrent system can be scheduled in a way that meets its timing requirements. If the component processes of a concurrent system are *periodic* (i.e., request execution repeatedly at fixed intervals) and do not interact, the effects of various pre-emptive scheduling algorithms are well understood. If dynamic assignment of priorities to processes is possible, then the earliest deadline algorithm is optimal, in the sense of providing the best processor utilization [18]. For static priority assignments, rate monotonic scheduling, in which the processes with shorter periods get higher priorities, is an optimal scheduling algorithm in the sense that any set of processes that can be scheduled successfully with a static priority assignment can be scheduled with the rate monotonic algorithm. For both earliest deadline and rate monotonic scheduling, there are simple sufficient conditions to determine whether a set of processes can be successfully scheduled [18, Theorems 5 and 7]. When the processes are not periodic and interact, the situation is more complex, and the problem of determining schedulability is *NP*-hard [19]. For instance, rate monotonic scheduling methods can still be applied, but it is necessary to recast aperiodic processes as periodic ones (using periodic server processes, for example) and to use special scheduling to avoid priority inversion problems [22], [23].

This paper is concerned with the case in which processes in a concurrent system are aperiodic and interact in complex ways, and in which the system developer has (or wishes to exercise) little control over scheduling beyond that provided by the semantics of interprocess communication. Such a situation, for instance, would frequently be the case for systems using Ada tasking constructs without special runtime support for rate monotonic scheduling. In this setting, it may be important to show that the system satisfies certain critical timing requirements without establishing full schedulability under a particular scheduling algorithm.

Various approaches to analyzing timing properties have been described in the literature. Some (e.g., [1], [8]) have relied upon testing or simulation to obtain timing information. Of course, testing and simulation can, at best, only provide representative samples of timing behavior and hence cannot be expected to accurately determine timing properties of concurrent systems.

Most of the other analysis methods that have been proposed have involved the introduction of special logics and proof techniques (e.g., [13], [16]) or construction and analysis of the state space of the system (e.g., [7], [11], [14], [20]). The

techniques based on proving theorems in special logics have typically been difficult to automate, and therefore have limited potential for practical application by developers of concurrent real-time software. Automation of logic-based techniques, such as Modechart [17], SCR [3], and Trio [12], is possible if restrictions can be made on the kinds of properties to be verified or on the domains of the variables in the logic (i.e., all domains, including time, must be finite). In such cases, these techniques are similar to those based on state enumeration. Although techniques based on analyzing the state space of the system are relatively straightforward to automate, the size of the state space is, in general, exponential in the number of processes in the concurrent system [21], [26]. Hence these techniques are computationally infeasible except for special classes of systems, and so their potential for practical use is also limited.

In this paper, we describe a technique for assessing timing properties of a concurrent software system executing on a single processor by deriving upper and lower bounds on the time that can elapse between two given events in any execution of the system. The technique finds worst-case bounds, under arbitrary scheduling, without requiring enumeration of the state space of the system. Experiments with the technique, some of which are described later in this paper, show that the technique can be used efficiently with some examples having more than 2^{500} reachable states, indicating that this technique might provide a foundation of practical automated tools for developers of real-time software.

Our technique is based on a formal model in which the execution of a concurrent system is treated as a totally ordered set of event occurrences, representing the activities in which the system engages and the order in which those activities occur. Example events might include the synchronous exchange of messages involving two processes, a process asynchronously sending (or receiving) a message to (or from) another process, a process entering its critical section, a process incrementing the value of some variable, etc. Our analysis method involves generating a set of linear inequalities expressing necessary conditions that must be satisfied by any execution of the concurrent system being analyzed. Integer programming techniques can then be used to find solutions to these inequalities that maximize or minimize an objective function reflecting the durations of events in the system. The minimum and maximum values of the objective function are lower and upper bounds, respectively, on the time that can elapse between occurrences of the events of interest.

Although integer programming is an NP-complete problem, the systems of inequalities produced by our method are network flows with side constraints and can frequently be solved very quickly. We have demonstrated the feasibility of our method by implementing it in an extended version of our constrained expression toolset [4] and analyzing the timing properties of some concurrent systems with large state spaces. These experiments provide encouraging evidence for the potential usefulness of the technique in real-time software development.

In the next section of this paper, we briefly outline the formal model on which our technique is based and then discuss our method for generating inequalities. The third section explains why the bounds may not be tight and describes two extensions to our technique that can be used to sharpen the bounds. Some of our experimental results are described in the fourth section, and the final section summarizes the paper and discusses some key issues concerning our method and results.

II. Deriving Bounds

A. The Model

Our inequality-based technique for deriving bounds on the time that can elapse between occurrences of a specified pair of events in behaviors of a concurrent system requires that the system be described as a collection of deterministic finite state automata (DFA's), such as those produced by the front-end of our constrained expression toolset. For simplicity, we will describe our technique for the case in which processes of the concurrent system communicate by synchronous message-passing over named channels and only briefly indicate the modifications required for other communication mechanisms. Each process is represented by a DFA in which each arc is labeled with a symbol representing either the communication of a message over a particular channel or a computation internal to the process. We will assume that each symbol representing a communication event belongs to the alphabets of exactly two DFA's and that each symbol representing a computation belongs to the alphabet of exactly one DFA. This can easily be achieved by encoding such things as process and channel names in the symbols and imposes no additional restrictions on the systems that can be modeled (e.g., we can still model multiple callers of an Ada entry by using distinct symbols for each caller). An example appears in Fig. 1, where communication events are represented by capital letters (e.g., A) and computation symbols have been omitted. A string over the union of the alphabets of the DFA's corresponds to a trace of an execution of the concurrent system if its projection on the alphabet of each DFA lies in the language accepted by the automaton.

The representation of a concurrent system as a collection of DFA's suffices for analysis of logical properties and is, in fact, essentially the internal representation used by our constrained expression toolset, which was originally developed for such logical analysis. To perform timing analysis, however, this representation must first be extended to account for time. This can be done straightforwardly by assigning a duration to each event[1] and regarding the time required for a sequence of events to be the sum of the durations of the individual events in the sequence. Of course, such an interpretation only makes sense when the events are non-overlapping, as would be the case if the concurrent system being analyzed were to be run on a single processor. We adopt this straightforward extension to the DFA representation of a concurrent system, and the corresponding limitation on the class of concurrent

[1] Our technique could be used equally well with a model that viewed events as instantaneous and assigned durations to the intervals between events, simply by mapping events to the nodes of appropriate DFA's and intervals to the arcs connecting nodes. However, the approach taken in this paper was more convenient with our existing toolset.

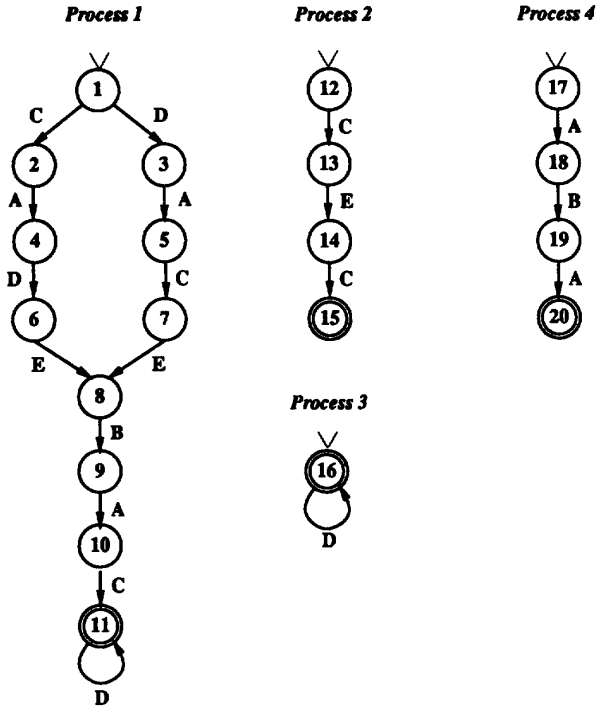


Fig. 1. A simple example.

systems whose timing properties we can analyze, as a first step toward applying our inequality-based techniques to the analysis of real-time systems. (It should be noted that upper bounds obtained under the assumption that events do not overlap remain valid in general. Of course, such bounds will not be very tight in cases where execution on a multiprocessor results in significant performance improvements.)

Other communication mechanisms, in addition to the synchronous message-passing discussed above, can be described by augmenting the DFA's representing sequential processes with a set of languages over subsets of the union of the DFA alphabets, as in the constrained expression formalism [4], [9] where these languages are generated using regular expression operators and a single additional operator. (The constrained expression formalism is capable of describing all recursively enumerable languages.) The additional languages, called *constraints,* express restrictions on traces that are imposed by the communication (or other synchronization) primitives. For systems in which communication is by asynchronous buffered message-passing, for example, constraints over symbols that denote matching send and receive events require that each receive event is preceded by a coresponding send event so that no messages are received before they are sent but several messages can be sent before any are received. A detailed description of one such case can be found in [9], for example. The inequality-based analysis method described below is easily adapted for use with this more general representation for a concurrent system.

Clearly our model cannot represent all real-time systems. The most important restrictions are the following:

Requires Bounds on Execution Time of Sequential Code: We assume that a technique like that of Shaw [24] can be used to derive upper and lower bounds on the execution time of sequential code regions between communications. This is already a difficult problem for sequential programs since obtaining good bounds may require knowledge of the program's input as well as its structure. We require such bounds in order to assign a duration to each event.

Finite State Processes: With DFA's, we cannot model variables with infinite ranges, unbounded recursion, or unbounded heap allocation. We can model the effect of local variables on control flow by encoding the values of variables into the states of the DFA's, although we note that encoding the values of variables with large ranges may lead to a state explosion that makes our analysis infeasible in practice.

No Dynamic Task Creation: We cannot directly model dynamic task creation, although the creation of a bounded number of tasks can be modeled using communication, as will be shown in Section IV.

All Events are Busy: We cannot model events, such as timeouts or delays, that take some amount of real time to occur, but do not consume CPU time.

No Scheduling Policy: Our model does not incorporate any specific scheduling policy. This makes the model more widely applicable, but may render it incapable of verifying systems in which the scheduling policy is essential to the corectness.

B. Inequality Generation

Whereas analysis of logical properties of concurrent systems primarily involves answering questions such as "Does any behavior of the system, starting from its initial state, produce a deadlock?", analysis of timing properties requires answering questions such as "What is the longest (or shortest) time that can elapse between an occurrence of event A and an occurrence of event B?" It is sometimes desirable to further restrict attention to only those parts of behaviors that do, or do not, contain certain other events, that is, to ask questions such as "What is the longest (or shortest) time that can elapse between an occurrence of event A and an occurrence of event B, when there are at least two intervening C events and no intervening D events?" In fact, we always implicitly impose the restriction that no other A or B events occur in the part of the behavior starting with A and ending with B. This restriction, while not necessary, simplifies the statement of timing questions and improves the results obtained from the marking algorithm described in Section III-A.

In general, therefore, analysis of timing properties focuses on various subsequences of events, which we refer to as partial behaviors, that might occur within the full sequences that correspond to complete system behaviors. Of course, sometimes the subsequence of interest is the full sequence.) Our approach is to generate a system of inequalities representing necessary conditions that must be satisfied by all such subsequences.

Suppose that we wish to generate the set of inequalities needed to find an upper or lower bound on the time that can elapse between two events corresponding to arc labels in a DFA representation of a concurrent program, such as the ex-

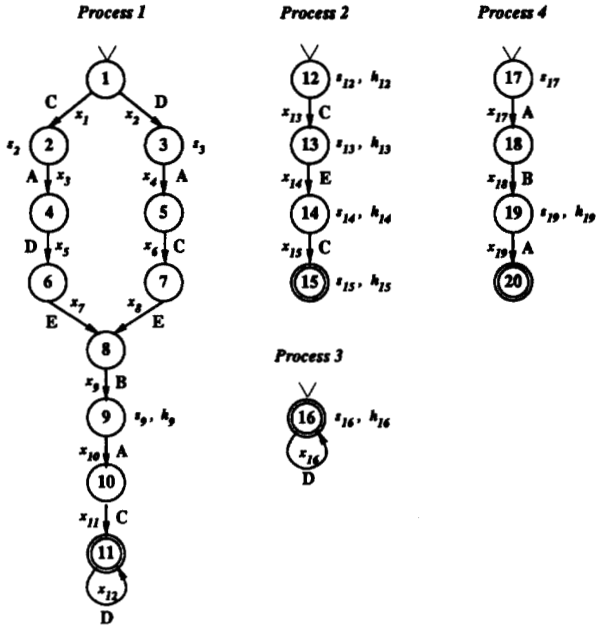


Fig. 2. Example of Fig. 1 with start, halt, and transition variables.

ample appearing in Fig. 1. We first assign a *transition variable* x_a to each arc a, which will represent the number of times that arc a is crossed in a partial behavior. Any partial behavior will correspond to some assignment of nonnegative integer values to these variables. We then assign a *start variable* s_i to each state i, which will be one if the process containing state i is in state i at the beginning of the partial behavior and zero otherwise. Similarly, we assign a *halt variable* h_j to each state j, which will be one if the process containing state j is in state j at the end of the partial behavior and zero otherwise. When seeking partial behaviors from A to B (inclusive), we can, in the processes in which A appears, omit start variables from any states not having an outbound arc labeled A. Similarly, in the processes in which B appears, we can omit halt variables from any states not having an inbound arc labeled B. Fig. 2 illustrates the result of applying this procedure to the DFA's of Fig. 1 when we are interested in partial behaviors starting with event A and ending with event B.

We next write a set of flow equations over these variables. The flow equations require that the flow into each state equal the flow out (i.e., the number of times a state is entered during a partial behavior equals the number of times it is exited). Starting is counted as flow in and halting is counted as flow out. For the example in Fig. 2 the following equations can be generated in this way (the equation number is the number of the state from which the equation comes):

$$0 = x_1 + x_2 \tag{1}$$
$$x_1 + s_2 = x_3 \tag{2}$$
$$x_2 + s_3 = x_4 \tag{3}$$
$$x_3 = x_5 \tag{4}$$
$$x_4 = x_6 \tag{5}$$
$$x_5 = x_7 \tag{6}$$
$$x_6 = x_8 \tag{7}$$
$$x_7 + x_8 = x_9 \tag{8}$$
$$x_9 + s_9 = x_{10} + h_9 \tag{9}$$
$$x_{10} = x_{11} \tag{10}$$
$$x_{11} + x_{12} = x_{12} \tag{11}$$
$$s_{12} = x_{13} + h_{12} \tag{12}$$
$$x_{13} + s_{13} = x_{14} + h_{13} \tag{13}$$
$$x_{14} + s_{14} = x_{15} + h_{14} \tag{14}$$
$$x_{15} + s_{15} = h_{15} \tag{15}$$
$$x_{16} + s_{16} = x_{16} + h_{16} \tag{16}$$
$$s_{17} = x_{17} \tag{17}$$
$$x_{17} = x_{18} \tag{18}$$
$$x_{18} + s_{19} = x_{19} + h_{19} \tag{19}$$
$$x_{19} = 0. \tag{20}$$

We also want the partial behavior to start in exactly one place in each process and halt in exactly one place in each process, so we generate an equation for each process stating that the sum of the start variables equals one. Given the flow equations, this is sufficient to force exactly one of the halt variables in each process to equal one.

$$s_2 + s_3 + s_9 = 1 \tag{21}$$
$$s_{12} + s_{13} + s_{14} + s_{15} = 1 \tag{22}$$
$$s_{16} = 1 \tag{23}$$
$$s_{17} + s_{19} = 1. \tag{24}$$

Finally, we equate the sums of transition variables on arcs representing the same communication event in different processes.

$$x_3 + x_4 + x_{10} = x_{17} + x_{19} \tag{25}$$
$$x_9 = x_{18} \tag{26}$$
$$x_1 + x_6 + x_{11} = x_{13} + x_{15} \tag{27}$$
$$x_2 + x_5 + x_{12} = x_{16} \tag{28}$$
$$x_7 + x_8 = x_{14}. \tag{29}$$

The system of equations produced by the above steps represents a set of conditions that must be satisfied by a partial behavior beginning with a designated event (e.g., a communication corresponding to symbol A in our example) and ending with another designated event (e.g., a communication corresponding to symbol B in our example). To further restrict attention to partial behaviors that do, or do not contain certain other symbols, additional inequalities expressing those restrictions would be added. For example, to restrict attention in our example to only those partial behaviors beginning with A and ending with B, having atleast two intervening C events and no intervening D events, we could add the following:

$$x_1 + x_6 + x_{11} \geq 2 \tag{30}$$
$$x_{13} + x_{15} \geq 2 \tag{31}$$
$$x_2 + x_5 + x_{12} = 0 \tag{32}$$
$$x_{16} = 0. \tag{33}$$

Note that only (30) and (32) would actually need to be added since the other two are redundant given equations (27) and (28). Note further that the additional restrictions on C and D events make it impossible to find any conforming partial behaviors since there are no executions in which C occurs twice between an A and a B. Therefore, in the remainder of our discussion of this example we will ignore these restrictions and the corresponding inequalities (30)–(33).

Given a system of inequalities representing the set of conditions that must be satisfied by partial behaviors beginning and ending with some designated events, a nonnegative integral solution to this system indicates that these conditions are simultaneously satisfiable, and hence that a partial behaviour having the specified properties could exist. (Reasons that such a behavior might not exist, despite the existence of a solution to the set of inequalities, are discussed in Section II). If such a partial behavior did exist, in each process it would start at the state whose start variable is one and end at the state whose halt variable is one in the solution to the inequalities. The flow equations guarantee that, in each process, it is possible to traverse some set of arcs connecting the start and halt states for the process, while (25)–(29) guarantee that the arcs in different DFA's that represent the same communication event are traversed the same number of times. Let t_i be the duration associated with the communication or computation symbol on the arc with transition variable x_i (for simplicity, we divide the time required for synchronous communication evenly between the participating processes). The duration t_i is either the upper or lower bound for executing the corresponding code, depending on whether we seek an upper or lower bound in the analysis. Then we define the integer programming objective function to be

$$\sum_i t_i x_i.$$

Standard integer programming techniques can then be used to find a solution to the system of inequalities that gives the maximum or minimum value for the objective function. (The maximum value of the objective function could, of course, be unbounded. In practice, for reasons discussed below, we usually impose some relatively large upper bound on the variables, thus ensuring that there is an upper bound on the total duration of the subsequences we are considering. Since all our variables represent counts, and are therefore assumed to be nonnegative, the minimum value is always bounded by zero).

To apply the method to systems with other communication primitives, we need only replace the equations enforcing synchronous communication. For systems communicating by buffered asynchronous message-passing, for example, instead of equating the sums of transition variables on arcs representing the same communication event in different processes, we would introduce inequalities on the numbers of send and receive events reflecting the fact that messages must be sent before they can be received. (The method discussed in Section III-B can be used to deal with the fact that some message might have been sent before the beginning of the subsequence of interest).

III. Tightening the Bounds

The procedure described above produces upper and lower bounds on the duration of partial behaviors starting and ending with designated events. As noted above, however, in some cases solutions to the generated system of inequalities may not actually correspond to subsequences of behaviors. Hence, while the bound found by the integer programming system is an upper or lower bound, depending on whether we maximized or minimized the total duration, it is not necessarily the least upper bound or greatest lower bound for the duration of the partial behaviors in question. There are two reasons that solutions to the system of inequalities may not correspond to actual partial behaviors.

The first reason is that these inequalities reflect most, but not all, of the semantics of the synchronously communicating DFA's. In particular, they do not guarantee that communication events in different processes will occur in a consistent order. In the example of Fig. 1, for instance, there is a solution to the system of (1)–(29) that corresponds to the partial behavior denoted by the set of state sequences

$$\{(3,5,7,8,9),(12,13,14),(16),(17,18,19)\}.$$

That is, in this solution the start variables associated with states 3, 12, 16, and 17, the halt variables associated with states 9, 14, 16, and 19, and the transition variables associated with arcs between states 3 and 5, 5 and 7, etc., all have the value one, and all other variables have the value zero. There is also another solution to the system (1)–(29) that corresponds to the "partial behavior" denoted by the set of state sequences

$$\{(3,5,7,8,9),(13,14,15),(16),(17,18,19)\}.$$

The former in fact represents a possible behavior subsequence, but the latter does not, since it involves an inconsistent ordering of the occurrences of events C and E in the first and second processes (C occurring before E in the first process and E occurring before C in the second).

Another way in which the generated system of inequalities does not reflect all the semantics of the DFA's is that it does not adequately restrict the numbers of occurrences of events labeling arcs that form a cycle in a process DFA. As a result, cycles in the DFA's can lead to solutions to the system of inequalities corresponding to "partial behaviors" that contain "extra" events not actually in the behavior subsequence. An example of this latter problem can be seen by examining (11), (16), and (28) above, which are the only equations containing the transition variables x_{12} and x_{16} associated with the arcs in the cycles of the example in Fig. 2. These equations can be satisfied by choosing any value of, say, x_{12}, and taking $x_{16} = x_2 + x_5 + x_{12}$ as required by equation (28). These variables can thus be assigned arbitrarily large values in solutions found by the integer programming package. The communication equations add enough additional restrictions to eliminate such solutions in many, but not all, cases.

The nonzero variables in an optimal solution to the system of inequalities determine a path in each DFA from a starting state to a halting state, possibly together with some cycles not connected to that path and having transition variables

with nonzero values. This problem with unconstrained cycles affects the sharpness of an upper bound on duration much more severely than that of a lower bound, since a solution minimizing the total duration will tend to set unconstrained variables to zero. If a large upper bound for all variables is introduced into the integer programming problem, the variables corresponding to such "disconnected" cycles will all take values near that upper bound in a solution maximizing the total duration and can be easily detected by inspection of the solution. Since such "disconnected" cycles cannot actually occur in an execution, it might seem that a valid upper bound could be obtained by simply subtracting those variables from the solution. This is not the case, however, since the cycle may contain an event that occurs in the subsequence attaining the true maximum duration and eliminating the events in the cycle would eliminate this subsequence, possibly leading to an incorrect bound. For instance, suppose that in our example the communication corresponding to symbol D takes more time than the communication corresponding to symbol C, and that both take a nonzero amount of time. Let u be the (analyst-specified) maximum value for all variables in the integer programming problem, and let t_R denote the time taken by a communcation corresponding to symbol R. (For simplicity, we will assume that all instances of the communication corresponding to R take the same amount of time). Then the partial behavior having the maximum duration is

$$\{(2,4,6,8,9),(13,14),(16,16),(17,18,19)\},$$

but the solution to our system of equations that maximizes the durations corresponds to the set of state sequences

$$\{(3,5,7,8,9),(11^u),(12,13,14),(16^u),(17,18,19)\}.$$

(The state sequence (11^u) in this represents the fact that the variables associated with the arc forming a cycle at state 11 has the value u in the solution. Note, however, that there are no start or halt variables associated with state 11 and that the variable associated with the arc between states 10 and 11 has the value zero in this solution.) The maximum value of the objective function is thus $t_A + t_B + t_C + ut_D + t_E$. We see that the cycles at states 11 and 16 are taken an "extra" u times here. If we simply subtract ut_D from this value, we have $t_A + t_B + t_C + t_E$. However, the partial behavior with maximum duration has duration $t_A + t_B + t_D + t_E$, which is larger since $t_D > t_C$. Thus, subtracting the events in the cycle from a solution maximizing the objective function leads to an incorrect bound.

As we have just described, one reason that a solution to the inequalities may not correspond to an actual partial behavior is that the inequalities do not reflect all of the semantics of the DFA's from which they are generated. The second reason is that is may not be possible to reach all of the start states corresponding to that solution at the same time in an actual behavior of the system. For instance, in the example of Fig. 1 it is possible to reach all the start states of the partial behavior

$$\{(3,5,7,8,9),(12,13,14),(16),(17,18,19)\}$$

simultaneously in an actual behavior (via the initial partial behavior $\{1,3),(12),(16,16),(17)\}$). On the other hand, it is not possible to simultaneously reach all the start states of the "partial behavior"

$$\{(3,5,7,8,9),(13,14,15),(16),(17,18,19)\}$$

in any actual behavior.

In some cases it is possible to tighten the bound obtained by integer programming through procedures that overcome some of the problems introduced by cycles and eliminate some "partial behaviors" whose initial states are not all simultaneously reachable. We now present a marking algorithm that reduces the problems arising due to cycles in the DFA's, and then describe an approach that reduces the problem of spurious solutions due to unreachable initial states.

A. Eliminating Unreachable Cycles

In order to prevent cycles from intefering with the integer programming system's computation of the upper bound, we wish to remove as many of them as possible from the DFA's before inequalities are generated. We use a straightforward marking algorithm to identify arcs that can be removed from DFA's without leading to an incorect bound, thereby breaking or removing cycles that cannot be reached. The marking algorithm accepts as input a set of start events, a set of stop events (but not more than one start and one stop event per DFA), a set of required events, and a set of forbidden events. The last two inputs are optional, but may be used by the analyst for various purposes such as to remove cycles, block consideration of "partial behaviors" that are known to be infeasible or further restrict the set of partial behaviors being considered. For instance, an analyst interested in the time that can elapse between events A and B when C also occurs and D does not could specify C as a required event and D as a forbidden event. In our prototype implementation, the analyst may specify required and forbidden events when giving the interval for which bounds are sought.

The marking algorithm is based on the observation that there are two conditions under which it is clear that an arc cannot actually be reached in a partial behavior starting at A and ending at B, and hence can be eliminated:

1) The arc is in one of the DFA's containing A or B and does not lie along any path starting with A, ending with B, or both. For example, the cycle labeled D in Process 1 of Fig. 1 does not lie along any path ending with B, and so cannot be reached along any path that is part of a partial behavior starting at A and ending at B.
2) The arc is in a DFA with a required event and every path from the required event to the arc passes through a forbidden event. The start and stop events A and B are required, in addition to those events the analyst specifies as required. Initially, the forbidden events are those specified as forbidden by the analyst. A communication event in a DFA can also become forbidden if all of its matching events in other DFA's are removed.

An example of the use of these conditions is given in Fig. 3. The marking algorithm would remove the arc labeled C in Process 1 due to the first condition (recall that we are not allowing a partial behavior starting at A and ending at B to

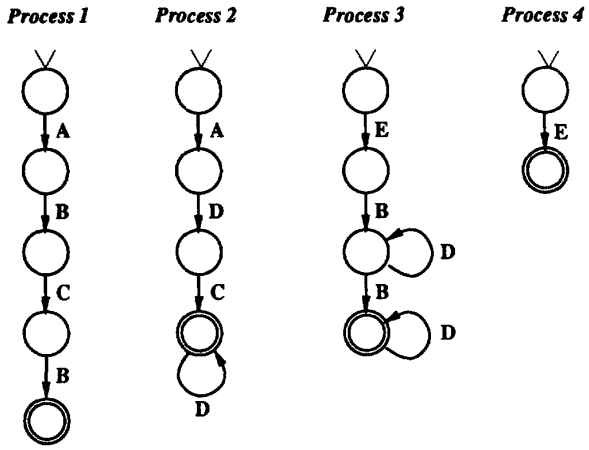


Fig. 3. Example of a cycle that can be removed.

Input: A DFA M_j
A set of start events S such that $|S \cap alphabet(M_j)| \leq 1$
A set of halt events H such that $|H \cap alphabet(M_j)| \leq 1$
A set of required events R such that $R \supseteq S \cup H$
A set of forbidden events F
Output: DFA $M_j' = M_j$ with unreachable arcs removed
An updated set of forbidden events F

Mark all transition arcs $k \in trans(M_j)$ with the empty set $\emptyset$.
For each required event $e \in R$:
  If $e \notin H$ then
    For each transition $k \in trans(M_j)$ where $label(k) = e$:
      Search forward from k to all arcs reachable without crossing an arc labeled with a forbidden event, or continuing past an arc labeled with a halt event.
      Add e to the set marking each arc crossed.
  If $e \notin S$ then
    For each transition $k \in trans(M_j)$ where $label(k) = e$:
      Search backward from k to all arcs reachable without crossing an arc labeled with a forbidden event, or continuing past an arc labeled with a start event.
      Add e to the set marking each arc crossed.
Set $M_j' = M_j$ less all transition arcs not marked with the set $alphabet(M_j) \cap R$.
For each communication symbol c in $alphabet(M_j)$:
  If $c \notin alphabet(M_j')$, then $F := F \cup \{c\}$

Fig. 4. Algorithm MARK-DFA.

contain any intervening A's or B's). This makes the event C in Process 2 forbidden. We then know that the D cycle in Process 2 cannot be reached since there is no path from the A event in that process, which must occur, to the cycle that does not pass through a forbidden event.

The marking algorithm uses conditions 1 and 2 to remove all arcs meeting either of these criteria from the DFA's, thereby eliminating solutions with positive flow through cycles not connected to the path from the starting event to the ending event. The algorithm to mark one DFA, embodied in the procedure MARK-DFA, is shown in Fig. 4, where $trans(M_j)$ is the set of transitions in the DFA M_j, $label(k)$ is the symbol labeling transition k, and $alphabet(M_j) = \{label(k) \mid k \in trans(M_j)\}$.

Essentially, we mark only those arcs in the DFA that can be reached from all required events without crossing any forbidden events. Any remaining unmarked arcs cannot appear in any partial behavior containing all required events and can thus be removed. The complete marking algorithm proceeds

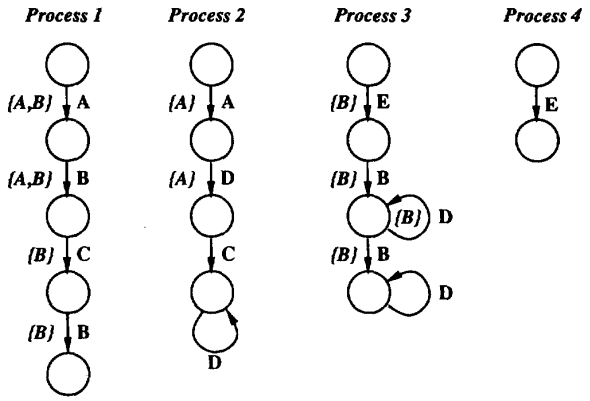


Fig. 5. Example of DFA marking.

as follows. First the DFA's are placed in a list of DFA's to process. Then the MARK-DFA procedure is applied to each of the DFA's from this list in turn. If an event that is still present in some DFA that has already been processed is subsequently marked fobidden by MARK-DFA, then more of the events in the already-processed DFA may need to be removed, and the DFA is placed back on the list of DFA's to process. The processing of DFA's continues until the list of DFA's to process is empty. To minimize the number of times each DFA is processed, it is advantageous to place the DFA's with the most start and stop events first on the list, since these events have the greatest potential to mark parts of the DFA unreachable and cause other events occurring only in those parts to be marked forbidden. Given a set of DFA's with a total of n arcs, the time complexity of MARK-DFA is clearly $O(n)$. Since each arc may cause MARK-DFA to be called at most one additional time, the overall time complexity of the marking procedure is $O(n^2)$.

Applying this algorithm to the example of Fig. 3, we first place the DFA's on the list in the order 1, 2, 3, 4. Process 1 comes first because it has two distinct start/stop events (A and B). Processes 2 and 3 come next since they have one start/stop event each. Finally, Process 4 has no start/stop events. Applying MARK-DFA to Process 1 will produce the marked DFA shown in Fig. 5. Removing the two arcs not marked $\{A, B\}$ removes all communication events corresponding to the C in Process 2, thereby making that event forbidden. In Process 2, MARK-DFA now stops at C, causing the D cycle to be removed. (Note that the presence of the first D in Process 2 prevents D from being forbidden.) Next, in Process 3, MARK-DFA removes the lower D cycle, but it cannot remove the upper one, since that can be reached backwards from the second B. Since the cycle in Process 2 has been removed, however, the remaining D in Process 3 cannot occur an unbounded number of times. MARK-DFA removes nothing from Process 4. The final DFA's, from which inequalities would be generated, are shown in Fig. 6.

The marking algorithm may not eliminate all cycles from the DFA's. Additional cycles may sometimes be eliminated by "unrolling" loops for which the maximum number of iterations is known.

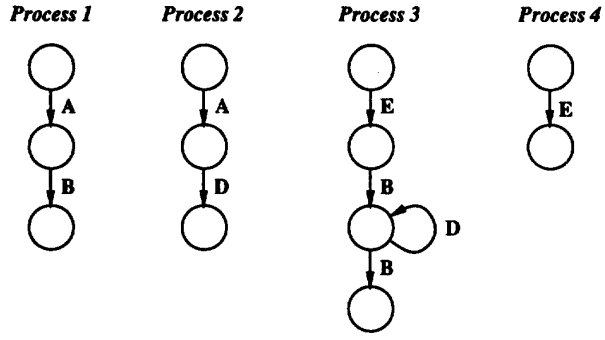


Fig. 6. Example of Fig. 5 after cycle removal.

B. Eliminating Unreachable Start States

As noted earlier, it is possible that the upper or lower bound found by the method described in Section II-B will correspond to a "partial behavior" that cannot actually occur because it is not possible to reach all of the start states of that "partial behavior" at the same time in any actual behavior of the system. We can eliminate many, though not all, of these cases by using a somewhat larger system of inequalities. The larger system includes additional inequalities representing conditions that must be satisfied by the *initial segment* of an actual behavior containing the partial behavior of interest, that is, the event subsequences leading from the initial states of each of the tasks in the concurrent system to the states at which the partial behavior in question begins in each of those tasks. Essentially, we use the system of inequalities to simultaneously describe necessary conditions for a partial behavior starting at A and ending at B and for an initial segment ending at the start of the partial behavior.

To construct this augmented system of inequalities, we generate a new transition variable y_a for each arc a in the DFA's to represent the number of times that arc is crossed in the initial segment. Another set of flow equations is generated for these transition variables just as is done for the partial behavior starting at A and ending at B, with two exceptions:

1) Start variables associated with the partial behavior are counted as flow out in generating the flow equations for the initial segment, since they correspond to states where the initial segment will end. Halt variables associated with the partial behavior are ignored in generating the flow equations for the initial segment.
2) At the initial states of the task DFA's there is an implicit flow in of one, thus forcing the initial segment to begin at those states.

Equations are also generated to require equal numbers of matching communication events from different tasks in the initial segment. (These equations involve only the y_a's.)

The additional equations represent necessary, but not, in general, sufficient conditions for the existence of an initial segment from the initial state of each task to the start states of each task relative to a partial behavior starting at A and ending at B. As with the original system of inequalities, these additional equations impose conditions on numbers of event occurrences, but cannot enforce consistent ordering of event occurrences in different tasks nor accurately control the number of times that cycles can be traversed. Hence the additional equations do not guarantee the existence of an appropriate initial segment, so the bound reported by the integer programming system still need not be the least upper bound or greatest lower bound. We have found, however, that generating the augmented system of inequalities does eliminate many unreachable "partial behaviors," frequently sharpening the bounds that we derive.

IV. Experiments

In order to demonstrate the feasibility of our analysis method, we have modified parts of our constrained expression toolset to support the inequality-based analysis techniques described above. In this section we describe the application of our method for deriving upper bounds to two families of concurrent systems.

The constrained expression toolset was developed to support inequality-based analysis of logical properties of concurrent systems. Our toolset contains a tool for producing *constrained expressions*, or sets of DFA's and constraints like those described in Section II, from designs written in an Ada-like design notation. It also provides tools for generating systems of linear inequalities from constrained expressions, for solving systems of linear inequalities, and for generating sample traces representing behaviors of a system. (A detailed description of the toolset appears in [4].) To perform timing analysis, we adapted the component of the toolset that generates inequalities to implement the techniques described in the previous sections. The integer linear programming (ILP) package provided in the toolset finds an optimal solution to the system of inequalities, assuming that one exists. It requires the specification of an objective function, which is set to reflect the duration of events for timing analysis. In [5], the modified tools are used to derive bounds for a version of Helmbold and Luckham's gas station problem [15], a mutual exclusion protocol based on coteries [10], and a system modeling resource contention [5]. These relatively small and simple systems comprise between four and eight tasks, some of which had DFA's with up to 80 states and many interconnected cycles. The number of reachable (untimed) system states, however, does not exceed a few thousand in any of these systems, so exact techniques requiring the enumeration of the system's states could probably be applied. To demonstrate the utility of our technique on problems of more realistic size, we apply it here to two scalable examples with very large state spaces that are well beyond the limits of state enumeration-based techniques.

The first of these examples models a divide-and-conquer computation using the fork/join concurrency construct. Since our analysis method, like most others, cannot handle dynamically created tasks, we assume an upper bound on the number of tasks that can be created. A fork is modeled by placing a `fork` synchronization at the top of the task being spawned and at the point in the parent at which it spawns the task. Joins are modeled similarly using a `join` synchronization at the bottom of the child task and at the point of the join in

n	Tasks	Time			Number of	
		Generate	Solve	Total	Inequalities	Variables
100	101	59	15	74	891	989
200	201	109	36	145	1791	1989
300	301	174	74	248	2691	2989
400	401	252	134	386	3591	3989
500	501	334	221	555	4491	4989

Fig. 7. Performance results for divide-and-conquer example.

n	Tasks	Time			Number of	
		Generate	Solve	Total	Inequalities	Variables
60	122	45	16	61	957	1012
120	242	85	39	124	1917	2032
180	362	169	74	243	2877	3052
240	482	243	113	356	3837	4072
300	602	319	167	486	4797	5092

Fig. 8. Performance results for network router example.

the parent task. In the n-task version of this example, n is the limit on the number of tasks involved in forks. The forking structure is very simple: task $i+1$ is potentially created by task i. Task 1 is the only task that must exist; task n cannot fork.

Each task, except task n decides whether to fork a child and then performs some amount of computation. If a task forks a child, then it performs "small" computation involving a shared resource, modeled by communication with a separate task representing the resource; otherwise it performs a "big" computation. Task n always performs a "big" computation if it is spawned. We applied our automated analysis tools to various size versions of this example system, seeking an upper bound on the total execution time of the program given analyst-specified bounds on duration of the communcations and computations in each task. For this particular example, the exact upper bound can be determined rather straightforwardly. Assume for simplicity that the fork and join communications are instantaneous. Then, given that $U(i)$ is an upper bound on the total execution times of tasks $i, \cdots n$, the desired bound, $U(1)$, can be computed recursively as follows:

$$U(n) = B_n$$
$$U(i) = \max(B_i, S_i + U(i+1))$$

where B_i and S_i are the durations of the big and small computations, respectively, for task i. This fact allows us to readily determine whether our automated analysis has yielded a sharp bound.

Fig. 7 shows the results of using the automated tools to derive the desired upper bound automatically from a specification of the system. The table shows the size of the system analyzed (n), the number of tasks in the system, the time to generate the inequality system, the time to solve the inequality system, the total analysis time, and the size of the inequality system produced. All times are in seconds on a SPARCstation 10 Model 41. In each case, the toolset found the exact upper bound, i.e, the value of $U(1)$ given by the formula above. For the results shown, the durations used for big and small computations were 5 and 1 time units, respectively, except for tasks 5, 15, 25, $\cdots$ whose big computation took 50 time units. For the values of n in the table, these durations result in all but the last 5 tasks being forked. Results for other experiments with different durations were similar.

The second example models communication through a network. Each task represents a node that can receive and transmit packets. For scalability, we consider a network with a simple and regular structure, where the nodes are arranged in a grid with 2 rows and n columns. (This is, in fact, the structure of a fault tolerant phone network used in urban India.) Each node in the grid is connected to the four nodes in adjacent columns. Two special nodes, the source and the target, lie at opposite ends of the grid and are connected to the two nodes in the column at that end of the grid. We model the transmission of a single packet through the network from the source node to the target node. Each node upon reception of the packet, nondeterministically chooses one of the two nodes in the next column to send the packet to. We model the transmission of a packet from one node to another as a synchronization between the corresponding tasks. The duration assigned to each such synchronization represents the time required for the transmission between those nodes.

As before, we applied our automated analysis tools to various size versions of this example system. In this case, the analysis seeks an upper bound on the time to transmit a packet from the source node to the target node. We have again chosen an example for which the exact upper bound can be determined rather straightforwardly, this time using shortest-path techniques, allowing us to readily determine whether our automated analysis has yielded a sharp bound.

Fig. 8 shows the results of using the automated tools to derive this bound automatically from a specification of the system. The first column of the table shows the number of columns in the network (n), and the remaining columns are as in Fig. 7. In each case the toolset found the exact upper bound. For the results shown, the durations used the transmission times were 10 time units for each transmission, except for the transmissions from nodes, $5, 15, 25, \cdots$ in row 1 to nodes $6, 16, 26, \cdots$ in row 2, and the transmissions from nodes $6, 16, 26, \cdots$ in row 2 to nodes $7, 17, 27, \cdots$ in row 1, which were assigned a duration of 20 time units. These durations cause the packet to zigzag from the top to the bottom row and back again every 10 columns. Results for other experiments with different durations were similar.

Note that, as the network router example demonstrates, the applicability of our technique is not strictly limited to systems executing on a single processor. Rather, the technique's applicability is determined by the degree to which computations overlap, not the architecture *per se*; it is unlikely that the network would be run on a single processor. When there is little overlap in the computations of sequential components, as in this system, our analysis yields good bounds, even in the multiprocessor case. Note also that, in both this and the last example, we could just as easily have sought lower bounds by setting the ILP package to minimize the objective function.

In theory, the technique can produce very loose bounds due to unenforced event orderings and cycles in the DFA's. In practice, however, the technique produced sharp bounds for all the experiments described here as well as those described in [5], demonstrating that the necessary conditions are strong enough to be useful on some nontrivial systems.

Finally, and most importantly, note that the state spaces of both examples grow exponentially in n (the size n problem has at least 2^n reachable states), but the rates of growth of the analysis times for these systems are clearly subexponential. We believe this is evidence that our technique is scalable and can be used on some systems for which other proposed techniques would be prohibitively expensive.

V. Conclusion

A fundamental timing analysis problem is the determination of the maximum, or minimum, amount of time that can elapse between the occurrence of two given events during a system's execution. In this paper we have described an approach to finding upper and lower bounds that can approximate these quantities under arbitrary scheduling of processes. Neither testing nor simulation can establish maximum or minimum values, in general. Some other approaches to analysis, such as model checking applied to a representation of a system's state space or the use of special logics and proof techniques for reasoning about timing properties, can, in principle, produce exact values for the minimum and maximum time between two events. As noted earlier, however, these alternatives either are extremely difficult to automate or, because they require construction of a state space whose size grows exponentially with the number of processes in the concurrent system, are computationally intractable for concurrent systems of nontrivial size. As demonstrated by the examples of the previous section, analysis times for our method can grow much less than exponentially and therefore the method can be applied to some systems with very large state spaces. Thus, by providing bounds rather than exact values, our method trades some accuracy for greater feasibility.

The key questions that must be answered about our method in order to evaluate this trade-off are:

- When can it be applied practically?
- When does it provide useful bounds?

Obviously, the method is of little use if it does not produce results more quickly than the exact approaches and the bounds produced will be of little value if they are far from the exact times. Of course, determining when an application is "practical" and when bounds are "useful" depends very much on the particular situation, including the speed of the machine used to carry out the analysis, the requirements for the system being analyzed, and the precise value for the time that can elapse between the specified events, and all these parameters will vary.

These questions pose very challenging research problems. Given the current state of the art, we believe that insight into the first question can, at least for the foreseeable future, best be obtained empirically. Furthermore, we would argue that software developers will derive relatively little benefit from theoretical answers to the second question until enough experience with the method has accumulated to provide some useful guidelines for answering the first.

The method described in Sections II and III converts a question about a system of communicating processes to an integer linear programming problem by generating a system of inequalities and an objective function to be maximized or minimized over the nonnegative integer solutions to those inequalities. Integer linear programming is *NP*-complete, and is regarded in practice as computationally quite difficult. For special classes of problems (the "totally unimodular" ones), however, integer linear programming reduces to linear programming, for which polynomial algorithms are known and the simplex algorithm is almost always practical even though its worst-case performance is exponential. The integer linear programming problems arising from our method are not totally unimodular, but they do have special properties that seem to make them easier to solve than the general case—in particular large portions of them are network flow problems that are totally unimodular. Although there has been some effort to develop special-purpose solvers for this type of problem (network flows with certain side constraints), we are not aware of any theoretical results that indicate when such problems can be solved much more efficiently than the general integer linear programming problem, or how much more efficiently. Even papers proposing new methods for solving such problems validate those methods by presenting empirical data on their performance on standard test problems [2].

Since a formal characterization of the class of systems for which our method is efficient would depend on a similar characterization of the class of integer linear programming problems that can be solved efficiently, and this second characterization is unavailable, we are unable to say theoretically when our method is more practical than exact methods. Furthermore, the question of practicality in real applications depends not just on the details of a particular integer programming algorithm, but also on the details of a particular implementation that must deal with numerical problems involving such things as instability and the inaccuracy of floating point arithmetic. It therefore seems clear that, until stronger results about the solution of these integer programming problems are available, the best information about the practicality of our method will come from accumulating experience in applying it to a wide range of systems.

Given that reachability-based approaches can provide exact answers to questions about the time that can elapse between events, our method is of interest only when it is more feasible than such approaches. Thus, theoretical results about the quality of the bounds provided by our method, which seem difficult to obtain, will be of most value to system developers only in cases where there is reason to believe that the method is more practical than exact approaches. Although our experiments, such as those described in Section IV, demonstrate that such cases exist, a good deal more experience with the method will be needed to justify the effort required to obtain such results. This experience will also help to guide theorists in proposing appropriate hypotheses under which such theorems can be proved.

Another issue related to the quality of the bounds we obtain is the accuracy of the durations that our method assigns to individual events in the modeled system's behavior. One could generalize the model to treat durations as varying over some probability distribution(s), but for finding maximum and minimum timings one would still only be interested in the

maximum and minimum values that those durations could assume. For certain classes of relatively low-level events, of course, some of this information is available from component specifications, but obtaining such maxima and minima for higher-level events in the face of sophisticated compilers and complex hardware architectures is extremely challenging. Work like that of Shaw [24], who has developed an automated proof rule-based technique for deriving timing properties from programs in high-level languages and has measured the accuracy of the derived times against actual run times, is necessary to provide guidance on the assignment of the durations used in our method. Unlike some other approaches, which attempt to establish that a system satisfies a complete set of timing requirements, our method derives upper and lower bounds on the time that can elapse between a specified pair of events. It is thus particularly suited for analyzing systems with a small number of critical timing bounds. A broad range of systems can be represented using the model on which the method is based. As indicated earlier, the model can be applied to concurrent systems that assume different underlying models of computation and communication [9]. In particular, neither the restriction to two-party communication nor the assumption that the automata are deterministic, both made for purposes of the examples in this paper, is critical to the model or the analysis method.

A good deal of further experience with the method will be required to provide good answers to the questions about its range of practical application and the accuracy of the bounds it provides. Nevertheless, we have demonstrated that the method can be feasibly automated and have successfully applied it to some scalable families of examples. Although these examples are smaller and more restricted than real concurrent systems, the ability of our prototype implementation to analyze systems having more than 2^{500} reachable states is very encouraging. Furthermore, initial work on extending the method to the multiprocessor setting [6] is quite promising. We, therefore, believe that our method has considerable potential as a foundation for practical tools for developers of real-time software.

REFERENCES

[1] M. W. Alford, "SREM at the age of eight; The distributed computing design system," *Computer*, vol. 18, pp. 36–46, Apr. 1985.

[2] A. I. Ali, J. Kennington, and B. Shetty, "The equal flow problem," *European J. Oper. Res.*, vol. 36, pp. 107–115, 1988.

[3] J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements," *IEEE Trans. Software Eng.*, vol. 19, no. 1, pp. 24–40, Jan. 1993.

[4] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden, "Automated analysis of concurrent systems with the constrained expression toolset," *IEEE Trans. Software Eng.*, vol. 17, no. 11, pp. 1204–1222, Nov. 1991.

[5] J. C. Corbett, "Automated Formal Analysis Methods for Concurrent and Real-Time Software," Ph.D. thesis, Univ. of Massachusetts at Amherst, 1992.

[6] J. C. Corbett and G. S. Avrunin, "A practical method for bounding the time between events in concurrent real-time systems," in *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, T. Ostrand and E. Weyuker, Eds. New York: ACM Press, June, 1993 pp. 110–116.

[7] C. Courcoubetis and M. Yannakakis, "Minimum and maximum delay problems in real-time system," in *Computer Aided Verification, 3rd International Workshop Proceedings*, vol. 575 of *Lecture Notes in Computer Science*, K. G. Larsen and A. Skou, Eds., (Aalborg, Denmark, July,1991, pp. 399–409). New York: Springer-Verlag.

[8] B. Dasarathy, "Timing constraints of real-time systems: Constructs for expressing them, methods of validating them," *IEEE Trans. Software Eng.*,vol. 11, no. 1, pp. 80–86, 1985.

[9] L. K. Dillon, G. S. Avrunin and J. C. Wileden, "Constrained expressions: Toward broad applicability of analysis methods for distributed software systems," *ACM Trans. Prog. Lang. Syst.*, vol. 10, no. 3, pp. 374–402, July 1988.

[10] H. Garcia-Molina and D. Barbara, "How to assign votes in a distributed system," *J. ACM*, vol. 32, no. 4, pp. 841–860, Oct. 1985.

[11] R. Gerber and I. Lee, "A Layered approach to automating the verification of real-time systems," *IEEE Trans. Software Eng.*, vol. 18, no. 9, pp. 768–784, Sept. 1992.

[12] C. Ghezzi, D. Mandriolli and A. Morzenti, "Trio: A logic language for executable specifications of real-time systems," *J. of Systems and Software*, vol. 12, no. 2, pp. 107–123, May 1990.

[13] V. H. Haase, "Real-time behavior of programs," *IEEE Trans. Software Eng.*, vol. 7, no. 5, pp. 494–501, 1981.

[14] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The synchronous data flow programming language LUSTRE," in *Proc. IEEE*, vol. 79 no. 9, pp. 1305–1320, Sept. 1991.

[15] D. Helmbold and D. Luckham, "Debugging Ada tasking programs," *IEEE Software*, vol. 2, no. 2, pp. 47–57, Mar. 1985.

[16] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Trans. Software Eng.*, vol. 12, no. 5, pp. 890–904, 1986.

[17] F. Jahanian and D. Stuart, "A method for verifying properties of modechart specifications," in *Proc. Real-Time Syst. Symp.*, 1988, pp. 12–21.

[18] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[19] A. J. Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment," Ph.D. thesis, Massachusetts Inst. Technology, 1983.

[20] J. S. Ostroff, "Deciding properties of timed transition models," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 2, pp. 170–183, 1990.

[21] J. Reif and S. Smolka, "The complexity of reachability in distributed communicating processes," *Acta Informatica*, vol. 25, no. 3, pp. 333–354, 1988.

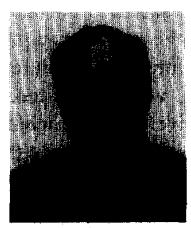
[22] L. Sha and J. B. Goodenough, "Real-time scheduling theory and Ada," *IEEE Comput.*, vol. 23, pp. 53–62, Apr. 1990.

[23] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[24] A. C. Shaw, "Towards a timing semantics for programming languages," in A. M. van Tilborg and G. M. Koob, Eds., *Foundations of Real-Time Computing: Formal Specifications and Methods*. Boston: Kluwer Academic Publishers, 1991, ch. 9, pp. 217–249.

[25] J. A. Stankovic and K. Ramamritham, editors, *Hard Real-Time Systems*. Washington, DC: Computer Society Press, 1988.

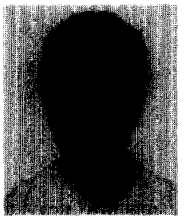
[26] R. N. Taylor, "Complexity of analyzing the synchronization structure of concurrent programs," *Acta Informatica*, vol. 19, pp. 57–84, 1983.



George S. Avrunin received the B.S., M.A., and Ph.D. degrees in mathematics from the University of Michigan.

He is a Professor in the Department of Mathematics and Statistics and Adjunct Professor in the Department of Computer Science at the University of Massachusetts at Amherst. In addition to formal methods and tools for the analysis of concurrent and real-time software systems, his research interests include the cohomology and representation theory of finite groups.

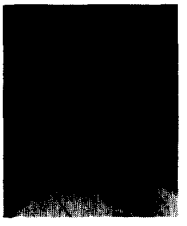
Dr. Avrunin is a member of the American Mathematical Society, the Association for Computing Machinery, the Association for Women in Mathematics, and the IEEE Computer Society.



James C. Corbett received the B.S. degree in computer science from Rensselaer Polytechnic Institute and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts at Amherst.

He is currently an Assistant Professor in the Department of Information and Computer Science at the University of Hawaii at Manoa. His research is directed toward devising practical techniques and building automated tools for the analysis and verification of concurrent and real-time software.

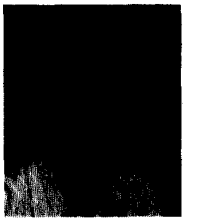
Dr. Corbett is a member of the Association for Computing Machinery.



Laura K. Dillon (S'81–M'83–S'83–M'84) received the B.A. and M.S. degrees in mathematics from the University of Michigan, Ann Arbor, and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts, Amherst.

She is an Associate Professor in the Computer Science Department at the University of California, Santa Barbara. Her research interests include formal methods for analysis of concurrent software systems, software specification and verification, and programming languages. Her research focuses on providing automated support for reasoning about the behavior of software systems.

Dr. Dillon is currently an Editor of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and is serving as an ACM National Lecturer. She is a member of the Association for Computing Machinery, the IEEE Computer Society, the American Association of University Women and Computer Professionals for Social Responsibility.



Jack C. Wileden (S'77–M'78–SM'92) received the A.B. degree in mathemetics and the M.S. and Ph.D. degrees in computer and communications sciences from the University of Michigan, Ann Arbor.

He is a Professor in the Computer Science Department at the University of Massachusetts at Amherst, where he has been a faculty member since 1978. His research interests center on advanced software technology, particularly software system infrastructure and software development analysis tools.

Dr. Wileden is the author or co-author of more than forty papers in journals and refereed conferences. He is a member of the Association for Computing Machinery. He is currently an Editor of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS and has previously served as an ACM National Lecturer and an IEEE Distinguished Visitor.