A Practical Technique for Bounding the Time Between Events in Concurrent Real-Time Systems

James C. Corbett
Information and Computer Science Department
University of Hawaii
Honolulu, HI 96822

George S. Avrunin
Department of Mathematics and Statistics
University of Massachusetts at Amherst
Amherst, MA 01003

Abstract

Showing that concurrent systems satisfy timing constraints on their behavior is difficult, but may be essential for critical applications. Most methods are based on some form of reachability analysis and require construction of a state space of size that is, in general, exponential in the number of components in the concurrent system. In an earlier paper with L. K. Dillon and J. C. Wileden, we described a technique for finding bounds on the time between events without enumerating the state space, but the technique applies chiefly to the case of logically concurrent systems executing on a uniprocessor, in which events do not overlap in time. In this paper, we extend that technique to obtain upper bounds on the time between events in maximally parallel concurrent systems. Our method does not require construction of the state space and the results of preliminary experiments show that, for at least some systems with large state spaces, it is quite tractable. We also briefly describe the application of our method to the case in which there are multiple processors, but several processes run on each processor.

1 Introduction

As the use of real-time software systems in safety-critical applications becomes widespread, the verification of their correctness has become an important concern. A real-time system is correct if it produces the correct result *and* produces it within specific deadlines. If a system does not guarantee that every result will be produced within its deadline, then either the system's performance or its requirements must be changed before it can be considered correct. One standard way to try improve the performance of the system is by dividing computations into segments that can be executed concurrently. Unfortunately, this makes analysis of the timing properties of the system more complicated and, consequently, the verification of the presumably faster system more difficult.

In this paper, we consider the case of a concurrent system utilizing synchronous communication between its processes and executing in a maximally parallel fashion in a multiprocessor environment. This means that each process proceeds with its computation unless it is blocked because it is unable to communicate with another process or gain access to some resource (usually this will require that each process have its own processor). We describe a method for deriving an upper bound on the time that can elapse between any two given events in an execution of such a system. This method does not require construction of the state space of the system and can be fully automated. A prototype implementation has been built and application of this prototype to several families of examples has shown that the method can be quite tractable on systems with very large state spaces. We also briefly describe the extension of our technique to the case in which multiple processes are statically assigned to each processor.

ACM-ISSTA'93-6/93/Cambridge, MA, USA

2 Previous Work

Various methods have been proposed for showing that concurrent systems satisfy timing constraints. Most have relied on either proving theorems in some logical setting (e.g., [7]) or some form of reachability analysis (e.g., [5], [6]). The theorem-proving techniques have been hard to automate and the reachability-based techniques require construction of the state space of the concurrent system. Since the size of this state space is, in general, exponential in the number of processes in the system, these techniques are computationally infeasible except in certain special cases.

In an earlier paper with Laura K. Dillon and Jack C. Wileden [3], we described a technique for finding upper and lower bounds on the time between events in the execution of a concurrent system. This technique does not require construction of the state space of the system and experiments have shown that it can be used with systems having state spaces that are quite

large (e.g., with more than 2^{200} states). However, the technique assumes that events do not overlap in time, so the upper bounds determined by this method, though valid, are likely to be useful only in the case of a logically concurrent system executing on a single processor and the lower bounds are valid only in that case. The work described here extends that work to the maximally parallel multiprocessor setting. Since the uniprocessor technique is an important component of the new work, we review it briefly here.

We model each task, or process, in a concurrent program with a finite automaton whose alphabet contains symbols representing internal computation or synchronous communication with another task. We derive the bounds from necessary conditions, in the form of linear equations, for a sequence of events to be a trace of the concurrent system. These equations are derived by considering the automata as flow networks and finding a flow from the start state to some accepting state. More precisely, to each transition in an automaton we assign an integer variable that represents the number of times that transition is made. We then write an equation for each state equating the number of times the state is entered (the sum of the variables labeling incoming transitions) with the number of times the state is exited (the sum of variables labeling outgoing transitions). In addition, we generate an equation for each synchronous communication channel requiring that the two tasks the channel connects agree on the number of times they have communicated along that channel. This enforces some consistency between the actions of the tasks, but does not guarantee that the communications take place in a consistent order (e.g., these conditions allow the impossibility that one task could synchronously communicate with another on channel A and then on channel B, while the other task performs the communications in the reverse order). Primarily for this reason, these equations are not sufficient conditions for a sequence of events to be a trace.

To derive bounds on the execution time of the whole program, we find an integer solution to the linear system that maximizes (or minimizes) an objective function consisting of the sum of all the variables, each weighted by the duration of the event labeling the corresponding transition. Since the linear system represents necessary conditions for a set of events to be a trace (i.e., every trace has a corresponding solution), the value of the objective function at a maximal (minimal) solution gives an upper (lower) bound on the execution time of the program. To derive bounds on the time between two specific events, we force the flow to start before the beginning event in the task(s) containing that event, and allow the flow to start anywhere in a task if it contains no beginning events. Similarly, we force the flow to stop after the ending event in the task(s) containing that event, and allow the flow to stop anywhere in a task if it contains no ending events.

This approach is described in detail in [3]. It is an extension of the constrained expression analysis technique for concurrent software [2] which uses essentially the same necessary conditions to overcome, at least in some cases, the state explosion problem inherent in reachability-based analysis techniques.

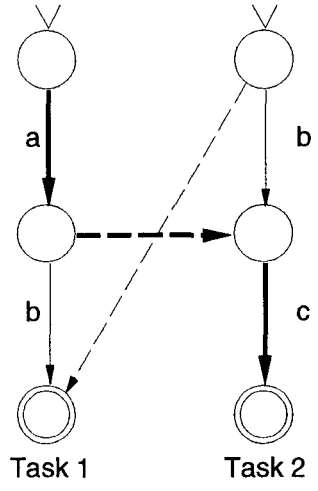


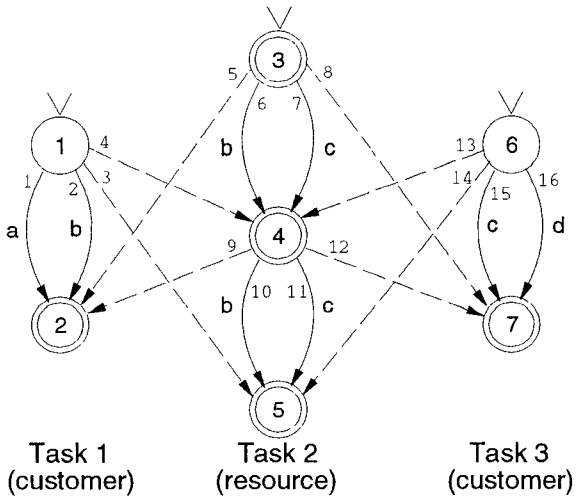
Figure 1: Wait Graph

3 Extension to the Multiprocessor Setting

The chief difficulty in extending the above technique to the multiprocessor setting is calculating the parallel execution time of a set of events. On a uniprocessor, the execution time is simply the sum of the durations of the events that occur. On a multiprocessor, however, some events may happen concurrently and the execution time depends not only on the choices that the tasks make but on the synchronization structure created by those choices.

By restricting attention to the case where each task has its own processor, we can use a well-known technique from scheduling theory to calculate the parallel execution time. Given a set of actions to be completed, each with a duration, and a partial order ($\leq$) on these actions representing a *wait* relation (i.e., if $A \leq B$ then action A must complete before action B can start), the following is well known: The minimum time to complete all the actions, assuming an action can proceed once ready, is equal to the length of the longest path, called the *critical path*, through the graph of the wait relation. Here the length of a path is the sum of the durations of the actions along the path. If the durations of the actions are known, the critical path can be found in time linear in the size of the wait relation by successively marking actions with the earliest time they could commence, starting with actions that do not wait for other actions and proceeding up the partial order.

Given a specific execution of a concurrent program in a maximally parallel setting (which can then be viewed as a set of "straight-line" tasks that allow no choices), we can calculate the parallel execution time using this technique. Events in the concurrent program model correspond to actions in the technique. Event A immediately precedes event B in the partial order if either A and B are in the same task and A immediately precedes B in that task, or A is a communication event and B is the event following the matching communication event in the



$$1 - x_1 - x_2 = 0 \quad (1)$$

$$x_6 + x_7 - x_{10} - x_{11} - h_4 = 0 \quad (2)$$

$$x_2 - x_6 - x_{10} = 0 \quad (3)$$

$$b_1 - y_1 - y_2 - y_3 - y_4 = 0 \quad (4)$$

$$y_1 + y_2 + y_5 + y_9 - e_2 = 0 \quad (5)$$

$$b_1 + b_3 + b_6 = 1 \quad (6)$$

$$y_1 - x_1 \leq 0 \quad (7)$$

$$y_3 - x_2 \leq 0 \quad (8)$$

$$y_3 - x_{10} \leq 0 \quad (9)$$

Figure 2: A simple system and some of the inequalities generated for it

other task. We graph this partial order for a trivial example in Figure 1, using arcs to represent actions and nodes to connect the actions. This graph is the dual of the usual graph of a partial order, in which actions are represented by nodes, however, it allows us to use the finite automata representing the tasks directly in the wait graph. Arcs in the wait graph that come from the matching of communication events rather than from the task automata are called *cross arcs* and are shown as dashed arrows. For simplicity, we will assume that a symbol representing an internal computation event occurs only in the alphabet of a single automaton, and symbols representing synchronous communication between two tasks occur only in the alphabets of the automata corresponding to those tasks. This can always be achieved by suitable encoding of task and channel names. Thus, the arcs labelled a and c in the figure represent events internal to the two tasks and the arcs labelled b represent a synchronous communication between the two tasks. The critical path, assuming all events take one time unit, is shown by the bold arrows in the figure.

Unfortunately, tasks in real concurrent systems are more complicated than those in the example of Figure 1. Allowing branches in the task automata makes static determination of matching communication events impossible since the matching may depend on what branches the tasks execute. Loops in the tasks introduce cycles in the task automata and the wait graph, making the above algorithm, which only applies to partial orders, inapplicable. Scheduling theorists, to our knowledge, have not addressed the problem of finding the minimum time to complete a set of actions when the structure of the wait relation varies. Rather, their work has focussed on the more common problem of finding expected completion times when the wait relation is fixed, but the durations of the actions are given by random variables having distributions of a specific form. Our technique assumes fixed upper bounds on action durations and derives an upper bound on the minimum time to complete the actions given a variable wait relation that reflects the different possible executions of the concurrent program.

An overview of the technique follows. Just as in our uniprocessor technique, we derive a set of linear inequalities from the task automata and the semantics of the model. The optimal integer solution to this system is a bound on the execution time. The inequality system we derive for the multiprocessor technique has three parts. The *execution part* is exactly the set of linear equations derived for the uniprocessor technique that was described in Section 2. These equations "find" an execution of the concurrent program (i.e., choose what branches the tasks take, how many times each loop is traversed, etc.). The *critical path part* of the inequality system is a set of equations derived from the *potential wait graph*, a kind of wait graph described below. These equations "find" a path through the potential wait graph. The *bounding part* of the inequality system is a set of inequalities that bounds the variables from the critical path part with variables from the execution part, forcing the path found by the critical path part to pass through only those events that actually occurred in the execution found by the execution part. Maximizing the length of the path found by the critical path part will give an upper bound on the length of the critical path over all possible executions. This is an upper bound on the parallel execution time.

The potential wait graph is formed by assuming each communication event could be paired with any syntactically possible match and adding a pair of cross arcs to the task automata for each possible match (as was done for the single possible match in Figure 1). This graph represents all possible wait relations and also some that are not possible. If this graph contains cycles (as it will if the tasks contain loops), a path through the graph may cross certain arcs multiple times; each traversal of an arc represents a different instance of the corresponding event. The number of traversals will usually be bounded by the execution part.

Due to space limitations, we omit a formal description and simply illustrate the technique with the example shown in Fig-

ure 2. (A complete description of the technique can be found in [4].) Two tasks representing customers each choose whether to rendezvous with a common task that represents a resource. The resource task would most naturally be modeled with a loop, but the resulting cycle in the task's automaton degrades the bound derived by our technique. For this reason we have unrolled the loop as many times as it could possibly execute. We seek a bound on the time for all tasks to complete.

Some of the inequalities generated for the example are also shown in Figure 2. The variables are numbered according to the states and arcs they represent as follows: x_i represents the number of times arc i is traversed in the execution; y_i represents the number of times arc i is traversed as part of the critical path; b_i and e_i are 1 if the critical path starts or ends, respectively, at state i and 0 otherwise; h_i is 1 if the execution of a task halts at state i and 0 otherwise. All of these variables represent nonnegative integers and the b_i, e_i, and h_i variables can only have the values 0 or 1.

The first three equations are from the execution part, which finds a flow from a start state to an accepting state in each task. Since we are bounding the time of the whole execution rather than an interval between events, we can omit the start variables for all tasks (as described in [3]) and simply add an implicit flow in of one for the starting states of the tasks. For the same reason, we can omit the halt variables for tasks one and three, which have only one accepting state, and simply add an implicit flow out of one for their accepting states. Task 2, however, has three accepting states and so we add a halt variable h_i to each accepting state (counted as flow out) which allows the flow through that task to stop at any of these states. Equations 1 and 2 are the flow equations for states 1 and 4 respectively. Note the implicit flow in of one for state 1 since it is the starting state of task 1. Similar equations would be generated for the remaining states. Equation 3 requires that the number of b events in Task 1 match the number of b events in Task 2. A similar equation is generated for the c communication events.

The next three equations belong to the critical path part, which finds a flow through the potential wait graph. The critical path must begin at a start state and end at an accepting state, thus there are only b_i's (counted as flow in) for states 1, 3, and 6, and there are only e_i's (counted as flow out) for states 2, 3, 4, 5, and 7. Equations (4) and (5) are the flow equations for states 1 and 2 respectively. A similar equation would be generated for each state. In addition, we force the critical path to begin at exactly one state by summing the b_i's to 1, as done in equation (6).

The last three inequalities belong to the bounding part, which restricts the critical path found to use only events that occur in the execution found. Inequality (7) is the bounding inequality for arc 1. A similar inequality would be generated for each arc in the task automata. Inequalities (8) and (9) are the bounding inequalities for cross arc 3, which requires a matching between the events on arcs 2 and 10. A similar pair of inequalities would be generated for each cross arc.

Let d_i be the duration of the event labeling arc i, where we regard cross arcs as labeled by the corresponding communication event. Then the value of $\Sigma_i y_i d_i$ for a solution that maximizes this function is an upper bound on the parallel execution time of the concurrent program when run on a multiprocessor in a maximally parallel fashion. If all events take unit time, then the bound will be 2, produced by a solution in which the tasks both try to use the resource and must contend. If the b event takes 10 units instead, then the bound will be 11, produced by the same solution. Note that the critical path part alone would yield a bound of 20 by selecting a path that does not correspond to a feasible execution (i.e., one passing through two b events in the resource task). If event a takes 10 units and the other events take only one, then the bound will be 10, produced by a solution in which task 1 does not use the resource.

As shown in [4], this technique can easily be generalized to find an upper bound on the time between two specific events by generating an execution part of the inequality system that finds segments of an execution beginning and ending with specific events, as described in [3]. Additional b_i and e_i variables would allow the path found by the critical path part to begin anywhere a task could be after the starting event and end anywhere a task could be after the ending event.

The bound obtained by this technique need not be the least upper bound for several reasons. As mentioned in Section 2, the conditions used in the execution part are only necessary and not sufficient, so the solution to this part of the inequality system might not correspond to any possible execution. Also, cycles in the wait graph may allow circular flow in the solution that is not part of the critical path found. Finally, the execution part might find an execution in which a task chooses to wait to communicate with a task that is busy rather than communicating with a third task that is ready, thereby violating the assumption of maximal parallelism. Refinements to the method that partially address the first two problems are described in [3]. In the next section, we show how the third problem can be eliminated if the DFAs are acyclic.

4 Enforcing the maximal parallelism assumption

This failure to enforce the maximal parallelism assumption can be seen in the example of Figure 3. This is a modification of the system of Figure 2 in which Task 1 can no longer perform the internal computation a and Task 2 performs an additional internal computation e before communicating with the resource or performing the computation d. Suppose that each event in the system has duration 1. The upper bound on the duration of an execution obtained from the system of inequalities is 3 and a critical path achieving this bound runs from state 6 to state 5 along the arcs 12, 13, and 9. This corresponds to an execution in which Task 3 performs the internal computation represented by e, then uses the resource (as represented by the communication event c), and finally Task 1 uses the resource (as represented by the communication event b). In this execution,

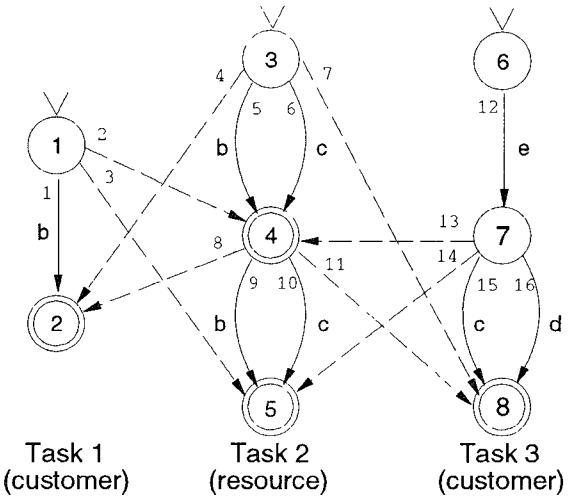


Figure 3: Modified example

Task 2 waits to communicate with Task 3 first, even though it could communicate with Task 1 immediately. This violates the assumption that processes proceed with their computations unless blocked. In order to exclude solutions to the system of inequalities corresponding to such executions, we need to be able to determine when a process is waiting for a communication in a particular state and then add additional inequalities that prevent such waiting when the process can proceed. In the remainder of this section, we outline a method for achieving this when the automata representing the tasks are acyclic. Complete details of the method can be found in [1].

Suppose that Task 2 enters state 3 at time s_3 and leaves it at time t_3. Then it is waiting in state 3 during the interval from s_3 to t_3. Let s_1 and t_1 be the corresponding times for state 1. To ensure that Task 2 does not wait in state 3 when it could communicate with Task 1, we need to ensure that either Task 2 enters state 3 after Task 1 leaves state 1 or Task 1 enters state 1 after Task 2 leaves state 3. Equivalently, we want to avoid cases in which the open intervals (s_1, t_1) and (s_3, t_3) overlap. (We allow the cases in which one task's arrival in a state occurs at the same instant as the other's departure from the corresponding state. Thus, for example, Task 1 could enter state 1 at the same time as Task 2 leaves state 3.)

This can be achieved with the quadratic inequality

$$(s_1 - t_3)(s_3 - t_1) \leq 0$$

In general, however, integer programming with nonlinear constraints is much more difficult than when all constraints are linear. If we impose an upper bound B on the times at which the tasks could enter or leave the states, we can achieve the same results using linear inequalities and additional variables, as described below. A safe upper bound can be calculated easily for acyclic DFAs by summing the durations of all possible events.

Note that it is impossible for both factors of the left side of this inequality to be positive. If $s_1 - t_3$ is positive, Task 1 entered state 1 after Task 2 left state 3. It follows that Task 2 entered state 3 before Task 1 left state 1, so that $s_3 - t_1$ is negative. The case we need to exclude is the one in which both factors are negative. We therefore introduce new variables that indicate when each of the factors is negative, and use these variables to enforce the quadratic inequality. Let $w_{1,3}$ and $w_{3,1}$, be 0-1 variables. The inequalities

$$\begin{aligned} t_3 - s_1 &\leq B w_{1,3} \\ s_1 - t_3 &< (B+1)(1 - w_{1,3}) \\ t_1 - s_3 &\leq B w_{3,1} \\ s_3 - t_1 &< (B+1)(1 - w_{3,1}) \end{aligned}$$

force $w_{i,j}$ to be 1 exactly when $s_i - t_j$ is negative. The linear inequality

$$w_{1,3} + w_{3,1} \leq 1$$

then has the same effect as the quadratic inequality above.

Of course, we have to make sure that the times at which states are entered and exited are consistent, given the durations of events and the synchronous nature of communication. The first set of consistency requirements simply requires that, if a task changes from state i to state j along an arc labeled by an event α with duration d_α, it must enter state j exactly d_α units of time after it leaves state i. These conditions are easily expressed by inequalities involving the entry times for states i and j and the variable corresponding to the given arc in the execution part. The consistency requirements reflecting the synchronization between tasks due to communication are somewhat more complicated.

To express these conditions, we must be able to determine which pairs of communication events actually match up in an execution. The communication between Tasks 1 and 2 can occur when Task 2 is in state 3 or state 4. In the first case, we want to set the times the tasks leave states 1 and 3 equal, while

n	Tasks	Time			Number of	
		Gen	Solve	Total	Ineqs	Vars
20	20	78	7	85	627	555
40	40	157	27	184	1287	1135
60	60	226	53	279	1947	1715
80	80	330	82	412	2607	2295
100	100	452	143	595	3267	2875

TABLE 1. Performance results for divide-and-conquer example

n	Tasks	Time			Number of	
		Gen	Solve	Total	Ineqs	Vars
20	42	147	26	173	1225	1114
40	82	354	95	449	2465	2234
60	122	573	255	828	3705	3354
80	162	832	393	1225	4945	4474
100	202	1177	855	2032	6185	5594

TABLE 2. Performance results for network example

in the second, we need to synchronize the time Task 1 leaves state 1 with the time Task 2 leaves state 4. To do this, we introduce another variable for each pair of cross arcs, and generate inequalities to force this variable to be 0 unless the communication matching the events corresponding to those cross arcs takes place. We can then use these variables to enforce the appropriate synchronization conditions.

Full details of this method, and the extensions needed to bound the time between two given events, rather than the duration of a complete execution, are given in [1].

5 Experiments

We have demonstrated the feasibility of our approach by conducting experiments on several concurrent systems. The constrained expression toolset has been modified to implement our new techniques. We first report on the application of the technique described in Section 3 to two example systems with very large state spaces.

The constrained expression toolset, described fully in [2], takes as input the specification of a concurrent system in an Ada-like specification language as well as a query that describes the property to be verified and other information needed for the analysis (e.g., the durations of atomic events). The toolset translates this specification into a set of communicating finite state machines and from these produces an inequality system. This system is solved by a simplex-based branch-and-bound integer programming package and the solution interpreted for the analyst in the context of the analysis. The technique described in Section 3 was implemented by changing the component of the toolset that generates the inequality system.

The first concurrent system to which we applied our technique models a divide-and-conquer computation using the fork/join concurrency construct. Each task $i = 1, \ldots, n$, once "activated", nondeterministically chooses to divide the problem by forking (modeled by activating task $i + 1$) and performing a "small" computation, or conquer the problem by performing a "big" computation. Task 1 is activated when the program begins; task n cannot fork (the problem size, n, is a limit on the number of tasks that can simultaneously exist). We sought a bound on the execution time of the program in a maximally parallel setting given durations for the fork and computation times. Table 1 shows the performance of the toolset on several sizes of this example. The columns of the table give the size of the example, the number of tasks, the time required to generate the system of inequalities from the specification, the time to solve the system, the total time, and the size of the system of inequalities (all times are in seconds on a DECstation 5000/125). In each case, the correct critical path was identified and the exact bound obtained.

The second example models the sending of a packet through a network. The geometry of the network is a grid of nodes with 2 rows and n columns, plus a sender node on the right of the grid and a receiver node on the left. Each node is connected to the four nodes in adjacent columns, while the sender and receiver are connected to the two nodes in the column at their end of the grid. Upon reception of a packet, a node nondeterministically chooses a node in the column to its left and sends the packet there. We sought an upper bound on the time to transmit a packet given durations on the transmission times between nodes. Table 2 shows the performance of the toolset on several sizes of this example. In each case, the correct critical path was identified and the exact bound obtained.

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 analysis times for these systems are clearly subexponential.

A first implementation of the method for enforcing maximal parallelism outlined in Section 4 has been completed and we have some preliminary experience with it. However, this implementation does not take advantage of special structure of the automata used to represent processes to reduce the size of the system of inequalities. As a result, the systems of inequalities are larger than necessary, and considerable degeneracy arises in solving the associated integer programming problems. The performance of our toolset is thus relatively poor with these systems. For instance, for the version of the divide-and-conquer example with 20 processes, our preliminary implementation produces 1571 inequalities in 784 variables. This is 2.5 times the number of inequalities and more than 1.4 times the number of variables required for the method of Section 3, and our integer programming package takes approximately 6 times longer to solve this larger system. We are currently modifying our code to make better use of the structure of the automata representing processes to reduce the size of the systems of inequalities, and we expect this to lead to substantial improvements in performance.

6 The General Multiprocessor Case

In a general multiprocessor setting, more than one task can share a processor. When tasks are statically assigned to processors, our technique can be extended to analyze such a system by reducing it to a maximally parallel system. This reduction is accomplished by the parallel composition, in a process algebraic sense, of the tasks that share a processor into a single task that represents the actions of all of the component tasks on that processor. The system can then be regarded as a maximally parallel one in which each task has its own processor, and the preceding techniques can be applied. From this standpoint, a communication between tasks on the same processor, which can be treated as a single event in the composite task corresponding to that processor, is an internal event in one task of the maximally parallel system. Since the composite represents all possible interleavings of the component tasks, its size is, in general, exponential in the number of those tasks. We believe this technique will often be feasible, however, for two reasons. First, load-balancing the concurrent system will tend to assign relatively few tasks to each processor. Second, tasks on a single processor are likely to be scheduled (e.g., using priorities) and this information can be used to reduce the size of the composition.

7 Conclusion

We have presented a technique for automatically deriving an upper bound on the time that can elapse between two events in a concurrent real-time program run in a maximally parallel multiprocessor setting. Our technique uses necessary conditions in the form of linear inequalities to obtain this bound and does not require the enumeration of the system's states. A prototype implementation of the technique has demonstrated its feasibility on several sample systems with over 2^{100} reachable states.

Acknowledgement

This research was partially supported by grants from the Office of Naval Research and the National Science Foundation.

References

[1] G. S. Avrunin. Sharpening bounds on the time between events in maximally parallel systems. Technical Report 92-69, Department of Computer Science, University of Massachusetts at Amherst, 1992. Available for anonymous ftp on ext.math.umass.edu.

[2] 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. Softw. Eng.*, 17(11):1204–1222, Nov. 1991.

[3] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden. A method for deriving bounds on the time between events in concurrent systems executing on a single processor. Submitted for publication. Available for anonymous ftp on ext.math.umass.edu.

[4] J. C. Corbett. *Automated Formal Analysis Methods for Concurrent and Real-Time Software*. PhD thesis, University of Massachusetts at Amherst, 1992.

[5] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 399–409, Aalborg, Denmark, July 1991. Springer-Verlag.

[6] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Trans. Softw. Eng.*, 18(9):768–784, Sept. 1992.

[7] F. Jahanian and A. K.-L. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, 12(5):890–904, 1986.