Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty*

HAGIT ATTIYA[†]

Department of Computer Science, The Technion,
Haifa 32000, Israel

AND

NANCY A. LYNCH

Laboratory for Computer Science, Massachusetts Institute of Technology,
Cambridge, Massachusetts 02139

A timing-based variant of the *mutual exclusion* problem is considered. In this variant, only an upper bound, m, on the time it takes to release the resource is known, and no explicit signal is sent when the resource is released; furthermore, the only mechanism to measure real time is an inaccurate clock, whose tick intervals take time between two constants, $c_1 \leqslant c_2$. When control is centralized it is proved that

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1)] + l$$

is an exact bound on the worst case response time for any such algorithm, where n is the number of contenders for the resource and l is an upper bound on process step time. On the other hand, when control is distributed among processes connected via communication lines with an upper bound, d, for message delivery time, it is proved that

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1) + d + c_2 + 2l]$$

is an upper bound. A new technique involving *shifting* and *shrinking* executions is combined with a careful analysis of the best allocation policy to prove a corresponding lower bound of

$$n \cdot c_2(m/c_1) + (n-1)\, d.$$

These combinatorial results shed some light on modeling and verification issues related to real-time systems. © 1994 Academic Press, Inc.

* A preliminary version of this paper appeared in "Proceedings of the 10th IEEE Real-Time Systems Symposium, Santa Monica, December 1989," pp. 268–284. This work was supported by ONR Contract N00014-85-K-0168, by NSF Contract CCR-8611442, and by DARPA Contracts N00014-83-K-0125 and N00014-89-J-1988.

[†] Most of this work was performed while this author was visiting the Laboratory for Computer Science, MIT.

0890-5401/94 $6.00

1. Introduction

An important area of computer applications is real-time process control, in which a computer system interacts with a real-world system in order to guarantee certain desirable real-world behavior. In most interesting cases, the real-world requirements involve timing properties, and so the behavior of the computer system is required to satisfy certain timing constraints. In order to be able to guarantee timing constraints, the computer system must satisfy some assumptions about time—for example, its various components should operate at known speeds.

It is clear that good theoretical work in the area of real-time systems is necessary. In the past few years, several researchers have proposed new frameworks for specifying requirements of such systems, describing implementations, and proving that the implementation satisfy the requirements. These frameworks are based on, among others, state machines [7, 29], weakest precondition methods [12], first-order logic [14, 15], temporal logic [5], Petri nets [6, 18, 30], and process algebra [4, 10, 13, 16, 28, 33]. Work is still needed in evaluating and comparing the various models for their usefulness in reasoning about important problems in this area and perhaps in developing new models if these prove to be inadequate.

Work is also needed in developing the complexity theory of such systems; very little work has so far been done in this area. An example of the kind of work needed is provided by the theory of asynchronous concurrent systems. That theory contains many combinatorial results that show what can an cannot be accomplished by asynchronous systems; for tasks that can be accomplished, other combinatorial results determine the inherent costs. In addition to their individual importance, these results also provide a testbed for evaluating modeling decisions and a stimulus for the development of algorithm verification techniques. Similar results should be possible for real-time systems. Some examples of complexity results that have already been obtained for real-time systems are the many results on clock synchronization, including [8, 11, 17, 20, 32] (see [31] for a survey).

In this paper, we embark on a study of complexity results for real-time systems. We begin this study by considering timing-based variations of certain problems that have previously been studied in asynchronous concurrent systems. In particular, we study a variant of the *mutual exclusion problem*. This problem is one of the fundamental problems in distributed computing; it serves as an abstraction of a large class of *hazard avoidance* problems. We note that this particular problem appears in the real-time computing literature (cf. [15]) as the "nuclear reactor problem." There, operators push different buttons to request the motion of different control rods in the same nuclear reactor. It is undesirable to have more than one control rod moving at the same time, presumably since in that case the nuclear reaction might be slowed down too much.

More specifically, we consider a system consisting of some number, n, of identical moving parts (e.g., control rods), no two of which are supposed to move at the same time. An operator associated with each moving part can request permission for the associated part to move by pushing a button that sends a *REQUEST* signal to the computer system. The system responds with *GRANT* signals; each *GRANT* signal gives permission to the designated moving part to move, but such motion is expected to be finished no more than a fixed time, m, later. The system is only supposed to issue a *GRANT* signal when it knows that it is safe to move the corresponding moving part, i.e., at least real time m has elapsed since the last *GRANT* signal. We assume, for simplicity, that a *REQUEST* signal is only issued by a particular operator if any preceding *REQUEST* by that operator has already been satisfied (by a corresponding *GRANT* signal). Our goal is to minimize the worst-case time between a *REQUEST* signal and the corresponding *GRANT* signal, i.e., the *worst-case response time.*

The computer system might consist of a single process running on a dedicated processor or might be a distributed system running on separate processors communicating over a message system. Solving the problem efficiently requires the computer system to make accurate estimates of the elapsed time since the las *GRANT* signal; the difficulty, however, is that the computer system only has inaccurate information about time, as given by inaccurate clock components within the system and by estimates of the time required for certain events. Specifically, the only information about time that the computer system has is the following:

1. the knowledge that a moving part will stop moving within time m after a *GRANT* signal,

2. the knowledge that the time between successive ticks of any clock is always in the interval $[c_1, c_2]$, for known constants c_1 and c_2, where $0 < c_1 \leqslant c_2$,

3. the knowledge that the time between successive steps of any process within the computer system is always in the interval $[0, l]$, for a known constant l, $0 \leqslant l$, and

4. (if the system is distributed) the knowledge that the time to deliver the oldest message in each channel is no greater than a known constant d, $0 \leqslant d$.

In the cases we have in mind, we suppose that $l \ll c_1 < c_2 \ll d \ll m$, but we state explicitly any assumptions that we require about relative sizes of the various constants.

One way in which our problem differs from the mutual exclusion problem usually studied in asynchronous systems is that we do not assume that an explicit signal is conveyed to the computer system when a moving part stops moving; the only information the system has about the completion of the critical activity is based on its estimates of the elapsed time. It is fairly typical for real-time systems to use time estimates in order to make deductions about real-world behavior. The results of this paper indicate some of the costs that result from using such estimates.

We obtain the following results. First, we consider a centralized computer system, consisting of just a single process with a local clock. For that case, we show that

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1)] + l$$

is an *exact* bound on the worst-case response time for the timing-based mutual exclusion problem. The upper bound result arises from a careful analysis of a simple FIFO queue algorithm, while the matching lower bound result arises from explicitly constructing and "retiming" executions to obtain a contradiction.

We then consider the distributed case, which is substantially more complicated. For that case, we obtain very close (but not exact) bounds: an upper bound of

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1) + d + c_2 + 2l]$$

and a lower bound of

$$n \cdot c_2(m/c_1) + (n-1)\, d.$$

Assuming that the parameters have the relative sizes described earlier, e.g., that d is much larger then l, c_1, and c_2, the gap between these two bounds is just slightly more than a single message delay time. The upper bound arises from a simple token-passing algorithm, while the lower bound proof employs a new technique of shifting some of the events happening at a process while carefully retiming other events.

The model that we use for proving our results is the I/O automaton model [23], which has been extended recently to include timing [25]. As noted earlier, many people are working on the development of other models and frameworks for reasoning about real-time systems. The most popular way of evaluating such frameworks involves their application to the specification and verification of substantial examples of practical utility. This paper, however, suggests a complemetary approach. Since a framework for real-time processing should allow proof of combinatorial upper and lower bounds and impossibility results, in addition to allowing specification and verification of systems, careful proofs of combinatorial results such as those in this paper should teach us a good deal about the appropriateness of a model for real-time processing.

The rest of this paper is organized as follows. Section 2 presents the timed I/O automaton model. Section 3 contains the general statement of the problem to be solved. Section 4 contains our results for the centralized case, Section 5 contains our results for the distributed case, and Section 6 contains some discussion and open problems.

2. Model and Definitions

2.1. *I/O Automata*

An *I/O automaton* consists of the following components: a set of *actions*, classified as *output*, *input*, and *internal*, a set of *states*, including a distinguished subset called the *start states*, a set of (*state*, *action*, *state*) triples called *steps*, and a *partition* of the *locally controllel* (output and internal) actions into equivalence classes. An action π is said to be *enabled* in a state s' provided that there is a step of the form (s', π, s). An automaton is required to be *input enabled*, which means that every input action must be enabled in every state. The partition groups actions together that are to be thought of as under the control of the same underlying process.

Concurrent systems are modeled by compositions of I/O automata, as defined in [23]. In order to be composed, automata must be *strongly compatible*; this means that no action can be an output of more than one component, that internal actions of one component are not shared by any other component, and that no action is shared by infinitely many components. The result of such a composition is another I/O automaton. The *hiding* operator can be applied to reclassify output actions as *internal* actions.

We refer the reader to [23, 24] for a complete presentation of the model and its properties.

2.2. *Timed Automata*

We augment the I/O automaton model as in [25] to allow discussion of timing properties. Namely, a *timed I/O automaton* is an I/O automaton with an additional component called a *boundmap*. The boundmap associates a closed subinterval of $[0, \infty]$ with each class in the automaton's partition; to avoid certain boundary cases we assume that the lower bound of each interval is not ∞ and the upper bound is nonzero. This interval represents the range of possible differences between successive times at which the given clas gets a chance to perform an action. We sometimes use the notation $b_l(C)$ to denote the lower bound assigned by boundmap b to class C, and $b_u(C)$ for the corresponding upper bound.

A *timed sequence* is a sequence of alternating states and (action, time) pairs:

$$s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots$$

Define $t_0 = 0$. The times are required to be nondecreasing; i.e., for any $i \geqslant 1$ for which t_i is defined, $t_i \geqslant t_{i-1}$, and if the sequence is infinite then the times are also required to be unbounded. For any finite timed sequence α define $t_{\text{end}}(\alpha)$ to be the time of the last event in α, if α is nonempty, or 0, if α is empty; for an infinite timed sequence α, $t_{\text{end}}(\alpha) = \infty$.

If i is a nonnegative integer and $C \in part(A)$, we say that i is an *initial index* for C in α if $s_i \in enabled(A, C)$ and either $i = 0$ or $s_{i-1} \in disabled(A, C)$ or $\pi_i \in C$. Thus, an initial index for class C is the index of a step at which C becomes enabled; it indicates a point in α from which we will begin measuring upper and lower time bounds.

A timed sequence is said to be a *timed execution* of a timed automaton A with boundmap b provided that when the time components are removed, the resulting sequence is an execution of the I/O automaton underlying A, and it satisfies the following conditions, for each class C of the partition of A and every initial index i for C:

1. If $b_u(C) < \infty$, then there exists $j > i$ with $t_j \leqslant t_i + b_u(C)$ such that either π_j is in C or no action of C is enabled in s_j.

2. There does not exist $j > i$ with $t_j < t_i + b_l(C)$ and π_j in C.

The first condition says that, starting from when an action in C occurs or first becomes enabled, within time $b_u(C)$ either some action in C occurs or there is a point at which no such action is enabled. The second condition says that, again starting from when an action in C occurs or first becomes enabled, no action in C can occur before time $b_l(C)$ has elapsed.

Note that the definition of a timed execution includes a liveness condition (in 1) in addition to safety conditions (in both 1 and 2). For finite timed sequences, it is sometimes interesting to consider only the safety properties. Thus, we define a weaker notion, as follows. A finite timed sequence is said to be a *timed semi-execution* provided that when the time components are removed, the resulting sequence is an execution of the I/O automaton underlying A, and it satisfies the following conditions, for every class C and every initial index i for C.

1. If $b_u(C) < \infty$, then either $t_{\text{end}}(\alpha) \leqslant t_i + b_u(C)$ or there exists $j > i$ with $t_j \leqslant t_i + b_u(C)$ such that either π_j is in C or no action of C is enabled in s_j.

2. There does not exist $j > i$ with $t_j < t_i + b_l(C)$ and π_j in C.

Intuitively, timed semi-executions represent sequences in which the safety conditions described by the boundmap are not violated. The following lemmas say that such a sequence can be extended to a timed execution in which the liveness conditions described by the boundmap are also satisfied.

LEMMA 2.1. *If α is a timed semi-execution of a timed automaton A and no locally controlled action of A is enabled in the final state of α, then α is a timed execution of A.*

Proof. Straightforward. ▌

LEMMA 2.2. *Let $\{\alpha_i\}_{i=1}^{\infty}$ be a sequence of timed semi-executions of a timed automaton A such that*

1. *for any $i \geqslant 1$, α_i is a prefix of α_{i+1}, and*
2. $\lim_{i \to \infty} t_{\text{end}}(\alpha_i) = \infty$.

Then there exists an infinite timed execution α of A such that for any $i \geqslant 1$, α_i is a prefix of α.

Proof. Straighforward. ▌

LEMMA 2.3. *Let A be a timed automaton having finitely many classes in its partition, and let α be a timed semi-execution of A. Then there is a timed execution α' of A that extends α, such that only events from classes with finite upper bound occur in α' after α.*

Proof. First, for each class C and each finite timed semi-execution β, we define a time *deadline*(β, C) to represent the latest time by which an action of C must occur in order to satisfy the liveness requirements. The definition is by inductin on the number of events in β. In the base case β consists of a single start state, s_0, and we define, for any class C such that some action in C is enabled in s_0, *deadline*$(\beta, C) = b_u(C)$. Otherwise, let *deadline*$(\beta, C) = \infty$. Let

$$\beta = s_0, (\pi_1, t_1), s_1, ..., (\pi_j, t_j), s_j$$

and assume we have defined *deadline* for all finite timed semi-executions with $j-1$ events. Denote

$$\beta' = s_0, (\pi_1, t_1), s_1, ..., (\pi_{j-1}, t_{j-1}), s_{j-1}.$$

Let $\pi_j \in C$; then *deadline*$(\beta, C) = t_j + b_u(C)$ if some action in C is enabled in s_j, and *deadline*$(\beta, C) = \infty$, otherwise. For any class $D \neq C$, *deadline*$(\beta, D) = t_j + b_u(D)$ if some action in D is enabled in s_j and no action in D is enabled in s_{j-1}; if some action in D is enabled in s_j and also some action in D is enabled in s_{j-1}, then $deadline(\beta, D) = deadline(\beta', D)$; if no action in D is enabled in s_j, then $deadline(\beta, D) = \infty$.[1]

We construct α' as the limit of a sequence $\{\alpha_i\}_{i=1}^{\infty}$ of timed semi-executions, where $\alpha_1 = \alpha$. Starting from α_i, we define α_{i+1} as follows. Let C be a class that has an action enabled in the final state of α_i, for which the value of $deadline(\alpha_i, C)$ is minimum among all such classes. Then α_{i+1} is obtained from α_i by appending a single enabled action from C, occurring at time $deadline(\alpha_i, C)$. If there is no such class, then we define $\alpha_{i+1} = \alpha_i$. Clearly, α_i is a timed semi-execution.

It remains to verify that α', the limit of the α_i, is a timed execution. There are three cases.

1. α' is a finite sequence. Then $\alpha' = \alpha_i$ for some i such that no action in any class is enabled in the final state of α_i. Then Lemma 2.1 implies that α' is a timed execution.

2. α' is an infinite execution in which the time component is unbounded. Then Lemma 2.2 implies that α' is a timed execution.

3. α' is an infinite execution in which the time component is bounded. Let k be the number of classes in A's partition. Since the values of $b_u(C)$ are nonzero, there is some bound $\varepsilon > 0$ such that $t_{\text{end}}(\alpha_{i+k}) \geqslant t_{\text{end}}(\alpha_i) + \varepsilon$ for all i. This implies that this case cannot occur. ▌

For any timed execution or semi-execution α we define $sched(\alpha)$ to be the sequence of (action, time) pairs occurring in α, i.e., α with the states removed. We say that a sequence of (action, time) pairs is a *timed schedule* of A if it is $sched(\alpha)$, where α is a timed execution of A. We also define $beh(\alpha)$ to be subsequence of $sched(\alpha)$ consisting of external (input and output) actions and associated times, and say that a sequence of (action, time) pairs is a *timed behavior* of A if it is $beh(\alpha)$, where α is a timed execution of A.

Definitions for composing timed automata to yield another timed automaton, analogous to those for I/O automata, are developed in [25]. We model real-time systems as compositions of timed automata. (Real-time systems were also modeled in this way in [21].)

2.3. *Adding Time Information to the States*

We would like to use standard proof techniques such as invariant assertions to reason about timed automata. In order to do this, we find it convenient to define an ordinary I/O automaton $time(A)$ corresponding

[1] These rules are similar to the rules given for maintaining the variable $last(C)$ in the $time(A)$ definition in the following subsection.

to a given timed automaton A. This new automaton has the timing restrictions of A built into its state, in the form of predictions about when the next event in each class will occur. Thus, given any timed I/O automaton A having boundmap b, the *ordinary* I/O automaton *time*(A) is defined as follows.

The automaton *time*(A) has actions of the form (π, t), where π is an action of A and t is a nonnegative real number. Each of its states consists of a state of A, augmented with a time called *current* and, for each class C of the partition, two times, *first*(C) and *last*(C). *current* (the "current time") represents the time of the last preceding event, initially 0. The *first*(C) and *last*(C) components represent, respectively, the first and last times at which an action in class C is scheduled to be performed (assuming some action in C stays enabled). (We use record notation to denote the various components of the state of *time*(A); for instance, $s.basic$ denotes the state of A included in state s of *time*(A).) More precisely, each initial state of *time*(A) consists of an initial state s of A, plus $current = 0$, plus values of *first*(C) and *last*(C) with the following properties. If there is an action in C enabled in s, then $first(C) = b_l(C)$ and $last(C) = b_u(C)$. Otherwise, $first(C) = 0$ and $\text{last}(C) = \infty$.

If (π, t) is action of *time*(A), then $(s', (\pi, t), s)$ is a step of *time*(A) exactly if the following conditions hold.

1. $(s'.basic, \pi, s.basic)$ is a step of A.

2. $s'.current \leqslant t = s.current$.

3. If π is locally controlled action of A in class C, then

   (a) $s'.first(C) \leqslant t \leqslant s'.last(C)$,

   (b) if some action in C is enabled in $s.basic$, then $s.first(C) = t + b_l(C)$ and $s.last(C) = t + b_u(C)$, and

   (c) If no action in C is enabled in $s.basic$, then $s.first(C) = 0$ and $s.last(C) = \infty$.

4. For all classes D such that π is not in class D,

   (a) $t \leqslant s'.last(D)$,

   (b) if some action in D is enabled in $s.basic$ and some action in D is enabled in $s'.basic$ then $s.first(D) = s'.first(D)$ and $s.last(D) = s'.last(D)$,

   (c) if some action in D is enabled in $s.basic$ and no action in D is enabled in $s'.basic$ then $s.first(D) = t + b_l(D)$ and $s.last(D) = t + b_u(D)$, and

   (d) if no action in D is enabled in $s.basic$, then $s.first(D) = 0$ and $s.last(D) = \infty$.

Note that property 4(a) ensures that an action does occur if any other class has an action that must be scheduled first. The partition classes of *time*(A) are derived one-for-one from the classes of A (although we will not need them in this paper).

The finite executions of *time*(A), when the states are projected onto their *basic* components, are exactly the same as the *finite* prefixes of the timed executions of A. This implies that safety properties of a timed automaton A can be proved by proving them for *time*(A), e.g., using invariant assertions.

3. Problem Statement

For either the centralized or distributed case, we assume that there are n modules called *moving parts*, n modules called *operators*, and some modules composing the *computer system*. The actions of the complete system, exclusive of any internal actions of the computer system, are *REQUEST*(i), *GRANT*(i) and *FINISH*(i), for $0 \leqslant i \leqslant n-1$. Each *operator*($i$) has input action *GRANT*(i) and output action *REQUEST*(i). Each *movingpart*(i) has input action *GRANT*(i) and output action *FINISH*(i). The computer system has input action *REQUEST*(i) for all i and output actions *GRANT*(i) for all i. See Fig. 1.

Let *movingpart*(i) be a particular timed automaton with the given signature, having a state consisting of one component, *GRANTED*, a Boolean variable, initially *false*. The state transitions of *movingpart*(i) are as follows:

```
GRANT(i)
Effect:
    GRANTED := true
FINISH(i)
Precondition:
    GRANTED = true
Effect:
    GRANTED := false
```

There is only one class in the partition for *movingpart*(i), a singleton containing the one action *FINISH*(i). The boundmap associates the interval $[0, m]$ with this class. As desribed in the Introduction, the timed executions of this timed automaton, have the property that, within time m after a *GRANT*(i) occurs, a *FINISH*(i) must also occur—that is, *movingpart*(i) "stops moving."

Now consider *operator*(i). It is described as an automaton with the maximum amount of freedom we want to allow to the operator. Let *operator*(i) be the timed automaton with the appropriate signature, having a state

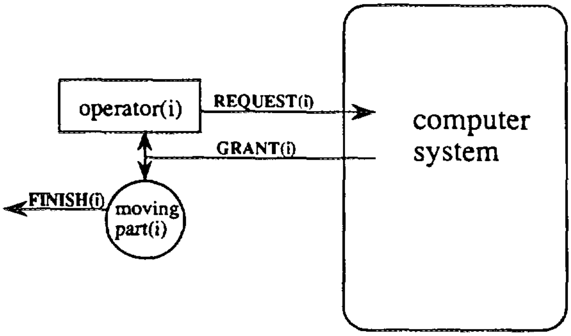

FIG. 1. The system architecture.

consisting of one component, *PUSHED*, a Boolean variable, initially *false*. The state transitions of *operator*(*i*) are as follows:

GRANT(*i*)
Effect:
    PUSHED := *false*

REQUEST(*i*)
Precondition:
    PUSHED = *false*
Effect:
    PUSHED := *true*

Again, there is only one (singleton) class in the partition for *operator*(*i*). We define the boundmap to assign the interval $[0, \infty]$ to this class. The upper bound is ∞ because we do not want to insist that the operator push the button within a particular amount of time after a *GRANT*. (It may never do so, in fact.) The lower bound is 0 because we do no want to assume any a priori lower bound on how quickly *operator*(*i*) might request after a previous grant.

A computer system *solves the timed mutual exclusion problem* if when it is composed with the given operators and moving parts, all the behaviors of the resulting system satisfy the following conditions:

1. *Request well-formedness*: For any $0 \leqslant i \leqslant n-1$, *REQUEST*(*i*) and *GRANT*(*i*) actions alternate, starting with a *REQUEST*(*i*).

2. *Moving part well-formedness*: For any $0 \leqslant i \leqslant n-1$, *GRANT*(*i*) and *FINISH*(*i*) actions alternate, starting with *GRANT*(*i*).

3. *Mutual exclusion*: There are never two consecutive *GRANT* events without an intervening *FINISH* event.

4. *Eventual granting*: Any *REQUEST*(*i*) event has a following *GRANT*(*i*) event.

We measure the *performance* of the system by the *worst case response time*, i.e., the longest time between *REQUEST*(i) and the next subsequent *GRANT*(i) in any timed behavior.

4. A Centralized System

We first consider the case of a "centralized" computer system to solve this exclusion problem. In this case, the architecture is as follows. There are two modules (timed I/O automata), the *manager* and the *clock*. The *clock* has only one action, the output *TICK*, which is always enabled, and has no effect on the clock's state. It can be described as the particular one-state automaton with the following steps.

```
TICK
Precondition:
    true
Effect:
    none
```

The boundmap associates the interval $[c_1, c_2]$ with the single class of the partition. This means that successive *TICK* events will occur with intervening times in the given interval.

The *manager* has input action *TICK* and *REQUEST*(i) for all i, and output actions *GRANT*(i). It is an arbitrary automaton, subject to the restriction that it has only a single class in its partition. (This says that is is really a sequential process—it cannot be running several processes in parallel.) We associate the boundmap $[0, l]$ with the single class of locally controlled actions. This means that successive locally controlled steps of the manager are done within the given intervals (if there are any enabled).

The computer system is the composition of the manager and the clock, (with the I/O automaton hiding operating applied to hide the *TICK* actions). See Fig. 2.

Note that the timed automaton model forces us to model the step time of the manager process explicitly. Other models (e.g., the one used for clock sychronization in [32]) might avoid this level of detail by hypothesizing that the manager's steps are triggered only by input events such as clock ticks or requests. We regard such a model (informally) as a limiting case of our model, as the upper bound on manager step time approaches zero.

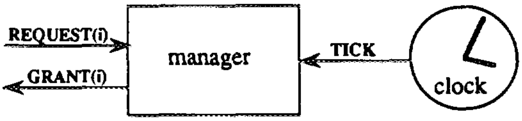

FIG. 2. The architecture of the centralized computer system.

4.1. *Upper Bound*

4.1.1. *The Algorithm*

The following simple algorithm for the manager for the manager process solves the problem. The manager simply puts requests on a FIFO queue. If there is a pending request, the manager issues a *GRANT* signal to the node whose request is first on the queue, and sets a timer to measure the time until the moving part stops moving. When the timer goes off, the manager repeats.

There is some subtlety in determining the minimum number of clock ticks that guarantee that time m has elapsed since the *GRANT*. At first glance, one might be tempted to count $\lfloor m/c_1 \rfloor + 1$ ticks, but a careful examination shows that this might cause a violation of the exclusion property, if a *TICK* happens immediately after the *GRANT*, and the next *GRANT* happens immdiately after the last *TICK*. Waiting for $\lfloor m/c_1 \rfloor + 2$ suffices to overcome this difficulty, but the lower bound presented in Subsection 4.2 suggests that this might not be optimal. In order to achieve the best possible timing performance, the algorithm only grants immediately after a clock tick, and the timer is set to $\lfloor (m+l)/c_1 \rfloor + 1$ clock ticks. The reason for only granting immediately after a tick is that this allows more precise knowledge of the time at which the grant occurs, which can sometimes permit fewer ticks to be counted. To implement this, the manager maintains a flag (*TICKED*) whose value is *true* if and only if no locally controlled operation has been performed since the last *TICK*.

In addition to the *REQUEST* and *TICK* inputs and *GRANT* outputs already specified, the manager has an internal action *ELSE*. This action is enabled exactly when no output action is enabled; this has the effect of ensuring that locally controlled steps of the manager occur at (approximately) regular intervals, as determined by the manager's boundmap.

The manager's state is divided into components:

TICKED	holding a boolean value, initially *true*;
QUEUE	holding a queue of indices $i \in [0 \ldots n-1]$, initially empty;
TIMER	holding an integer , initially 0;

The manager's algorithm is as follows:

```
REQUEST(i), $0 \leqslant i \leqslant n-1$
Effect:
    add i to QUEUE

TICK
Effect:
    TIMER := TIMER − 1
    TICKED := true
```

```
$GRANT(i)$, $0 \leqslant i \leqslant n-1$
Precondition:
    $i$ is first on QUEUE
    TIMER $\leqslant 0$
    TICKED $=$ $true$
Effect:
    remove $i$ from fron of QUEUE
    TIMER $:= \lfloor (m+l)/c_1 \rfloor + 1$
    TICKED $:= false$

$ELSE$
Precondition:
    QUEUE is empty or TIMER $> 0$ or TICKED $= false$
Effect:
    TICKED $:= false$
```

4.1.2. *Correctness Proof*

Let A be the composition of the four given kinds of timed automata–operators, moving parts, manager, and clock. This subsection is devoted to provind the following theorem.

THEOREM 4.1. *Algorithm A solves the timed mutual exclusion problem.*

We prove this theorem using automaton $time(A)$, as defined above. In this case, the system state is augmented with the variable *current*, plus the variables *first* and *last*, for the following partition classes:

1. $REQUEST(i)$ for each i, which contains the single action $REQUEST(i)$,
2. $FINISH(i)$ for each i, which contains the single action $FINISH(i)$,
3. $TICK$, which contains the single action $TICK$, and
4. $LOCAL$, the locally controlled actions, which contains all the actions $GRANT(i)$, $0 \leqslant i \leqslant n-1$ and the $ELSE$ action.

Initially, we have $first(REQUEST(i)) = 0$, $last(REQUEST(i)) = \infty$, $first(FINISH(i)) = 0$, and $last(FINISH(i)) = \infty$, $first(TICK) = c_1$, $last(TICK) = c_2$, $first(LOCAL) = 0$, and $last(LOCAL) = l$.

The proof of mutual exclusion rests on the following invariant for $time(A)$.

LEMMA 4.2. *Let s be a reachable state of $time(A)$. Then the following all hold:*

1. *If $FINISH(i)$ is enabled in $s.basic$, then*
   (a) $s.TIMER > 0$,
   (b) $s.first(TICK) + (s.TIMER - 1)c_1 > s.last(FINISH(i))$, *and*
   (c) *$FINISH(j)$ is not enabled in $s.basic$, for any $j \neq i$.*
2. *If $s.TICKED = true$ then $s.first(TICK) \geqslant s.last(LOCAL) + c_1 - l$.*

Thus, if a part is moving, the manager's TIMER is positive. Moreover, the TIMER is large enough so that waiting that number of ticks would cause enough time to elapse so that the part would be guaranteed to have stopped moving. Property 1(c) implies mutual exclusion, while property 2 guarantees a lower bound on the time till the next *TICK*, if no *LOCAL* step has occurred since the previous *TICK*.

The proof of correctness is done in careful detail; since it is quite straightforward, we relegate it to Appendix A.1.

Proof (of Theorem 4.1). Lemma 4.2 implies mutual exclusion. Moving part well-formedness follows easily from the same lemma and the definition of the moving part. Request well-formedness follows from the definitions of the operators and the manager. The remaining condition, eventual granting, can be argued from the queue-like behavior of the manager and the fact that the clock keeps ticking. (This latter property also follows from the formal proof of the upper bound on response time in the following subsection.) ▌

4.1.3. *Response Time*

Now we prove our upper bound on response time for the given algorithm A.

THEOREM 4.3. *Assume that* $l < c_1$. *The worst case response time for algorithm* A *is at most*

$$n[c_2(\lfloor m+l)/c_1\rfloor + 1)] + l.$$

The proof of this theorem requires several lemmas.

LEMMA 4.4. *In any reachable state there are at most* n *entries in QUEUE.*

Proof. We have already argued that all timed executions of the system are request well-formed; i.e., *REQUEST*(i) and *GRANT*(i) alternate for any $0 \leqslant i \leqslant n-1$, starting with *REQUEST*(i). The preconditions for *REQUEST*(i) and the operation of the manager imply that when *REQUEST*(i) happens, i is not in the queue. A simple induction implies that in any reachable state of the system, i appears only once in QUEUE. ▌

LEMMA 4.5. *In any reachable state* s, $s.TIMER \leqslant \lfloor(m+l)/c_1\rfloor + 1$.

Proof. By an easy induction. ▌

LEMMA 4.6. *Let s be any state occurring in a timed execution, in which* $s.TIMER \leqslant k$, *for* $k \geqslant 1$. *Then (at least) one of the following two conditions holds*:

1. $s.TIMER \leqslant 0$ *and* $s.TICKED = true$, *or*

2. *the time from the given occurrence of s until a later TICK event resulting in* $TIMER \leqslant 0$ *is bounded above by* $c_2 \cdot k$.

Proof. Suppose that it is not the case that s.TIMER $\leqslant 0$ and s.TICKED = *true*. Then a *GRANT* cannot occur until a state is reached in which TIMER $\leqslant 0$ and TICKED = *true*, and this condition requires at least one *TICK* to occur after the given occurrence of s. The bound follows from the upper bound on clock time, the way the *TICK* actions manipulate the TIMER, and the way the variable TICKED gets set. ▮

Proof (of Theorem 4.3). When a request arrives, it is at worst in position n on the QUEUE, by Lemma 4.4. By Lemmas 4.5 and 4.6, either TIMER $\leqslant 0$ and TICKED = *true* at the time when the request arrives, or else within time $c_2(l(m+l)/c_1\rfloor + 1)$ a *TICK* event (call it π_1) occurs which sets TIMER to 0. In the former case, there must be a *TICK* event that sets TIMER $\leqslant 0$, occurring prior to the request with no intervening local events; let π_1 denote this *TICK* event. In either case, within time l after π_1 (but after the request) the first entry gets its request granted, it is removed from the QUEUE, and TIMER is set to

$$\lfloor (m+l)/c_1 \rfloor + 1.$$

Since $l < c_1$, within time c_2 after π_1, another *TICK* evert φ_1 occurs, this one decreasing TIMER to $(\lfloor (m+l)/c_1 \rfloor)$.

Immediately after φ_1, either TIMER = 0, or $\lfloor (m+l)/c_1 \rfloor \geqslant 1$; in this latter case, by Lemma 4.6, within at most time c_2 $(\lfloor (m+l)/c_1 \rfloor)$ after φ_1, a *TICK* event occurs that sets TIMER $\leqslant 0$. Thus, in either case, from event π_1 until another *TICK* event π_2 that sets TIMER $\leqslant 0$, at most

$$c_2(\lfloor (m+l)/c_1 \rfloor + 1)$$

time elapses. The next entry in the queue is enabled immediately after π_2. In this manner, we can construct a sequence of *TICK* events, $\pi_1, ..., \pi_n$, such that the time between π_i and π_{i+1}, for each i, $1 \leqslant i < n$, is at most

$$c_2(\lfloor (m+l)/c_1 \rfloor + 1),$$

and for any $1 \leqslant i \leqslant n$, the ith entry on the original queue (if there is any) is enabled after π_i. Hence, within time

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1)],$$

the enabling condition is satisfied for the given request. Then within time at most l afterwards, the request is granted. This completes the proof of the upper bound on response time. ∎

Note that this proof requires the assumption that $l < c_1$; in case this assumption is not made, an analysis similar to the one in the proof above yields a slightly higher upper bound of

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1) + l].$$

Also, note that the limit of the given upper bound, as l approaches 0, is $n \cdot c_2(\lfloor m/c_1 \rfloor + 1)$. We think of this as an upper bound for this algorithm when it is run on an interrupt-driven model. It follows from the lower bound in Section 4.2 that algorithm A has optimal response time.

Although this proof is currently written in terms of executions, it could also be done in an assertional style similar to that used to prove Lemma 4.2. The invariant assertion techniques used in this paper are extended to handle response time analysis in [22]. One of the examples presented in that paper is a simplified version of the centralized algorithm above.

4.2. *Lower Bound*

Now we turn to proving lower bounds. We begin with a fairly simple lower bound result that is quite close to the upper bound proved in the preceding subsection, but does not match exactly. The gap between this lower bound and the upper bounds depends on the manager's step time and the roundoffs. This lower bound proof is presented here, although we later prove a better (higher) lower bound, because we believe it gives insight into a basic lower bound technique, called *shrinking*, which is used later.

THEOREM 4.7. *The worst case response time of any centralized computer system that solves the timed mutual exclusion problem is strictly greater than*

$$n \cdot m(c_2/c_1).$$

In order to see why this is so, define a timed executin or timed semi-execution to be *slow* if the times between successive *TICK* events (and the time of the first *TICK* event) are exactly c_2. We have:

LEMMA 4.8. *Let α be a slow timed execution of a centralized computer system that solves the timed mutual exclusion problem. Then the time between any two consecutive GRANT events in α is strictly greater than*

$$m(c_2/c_1).$$

Proof. If this were not so, then we could "retime" the whole timed execution by multiplying the time at which each event occurs by c_1/c_2 (without changing the ordering of events), resulting in a new timed execution in which the time between the two *GRANT* events is at most m. The time between clock ticks is now c_1, so the resulting sequence is a timed execution. Then moving the *FINISH* event corresponding to the first *GRANT* event to the point just after the second *GRANT* event (to occur at the same time) yields another timed execution, this one violating mutual exclusion. ▮

Intuitively, we have used clock uncertainty to take a slow execution and "schrink" it (by retiming) to force the manager to use pessimistic estimates.

Note that the contradiction in the proof of Lemma 4.8 involves a *FINISH* and *GRANT* event occurring at the same time. Although this violates our formulation of the mutual exclusion property, one might argue that the instantaneous overlap of two intervals during which rods move is not a real problem. We could modify the proof to obtain a contradiction involving a positive (but arbitrarily small) amount of overlap, but then the bound in Lemma 4.8 would be slightly smaller.

Proof (of Theorem 4.7). We create a slow timd semi-execution in which a *REQUEST*(0) event occurs, and immediately after the corresponding *GRANT*(0) event (and at the same time) a sequence of

$$REQUEST(0), ..., REQUEST(n-1)$$

events occur. Now extend this timed semi-execution (keeping it slow) until all these requests are fulfilled. By Lemma 4.8 the time between any two of these *GRANT* events is strictly greater than

$$m(c_2/c_1).$$

Let *GRANT*(j) be the last *GRANT*. The time from *REQUEST*(j) until the corresponding *GRANT*(j) is strictly greater than

$$n \cdot m(c_2/c_1). \quad ▮$$

Now we present the more delicate arguments needed to prove a lower bound that matches the upper bound given in Section 4.1. Note that the only differences between the lower bound to be proved and the one already proved in Theorem 4.7 are the presence of the l terms describing bounds on the manager's step time and the careful treatment of roundoff. Since we consider these to be very small, for practical purposes one might be satisfied with the simpler lower bound. However, it is interesting theoretically to note that in this case, we can obtain a tight bound by a related but somewhat more difficult argument.

THEOREM 4.9. *Assume that $l \leqslant c_1$.*[2] *Then the worst case response time of any centralized computer system that solves the timed mutual exclusion problem is at least*

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1)] + l.$$

An I/O automaton is called *active* if in every state there is a locally controlled action enabled. (Recall, for example, that the manager in the algorithm of the preceding subsection was made active by the inclusion of the *ELSE* action.) Before proceeding with the proof of the theorem, it is useful to prove the following lemma, which claims that there is no loss of generality in assuming that the manager is active. As in the previous subsection, denote by *LOCAL* the class of *all* the actions that are locally controlled by the manager (including *GRANT*(i), for all i).

LEMMA 4.10. *Suppose that A is a centralized computer system that solves the timed mutual exclusion problem with response time $\leqslant b$, for a real number b. Then there is another such algorithm A', with response time $\leqslant b$, in which the manager is active.*

Proof. Given A, we produce A' by adding a new internal action *NULL* to the manager. The steps associated with this action are exactly those triples of the form $(s', NULL, s)$, where $s' = s$ and no other locally controlled action of the manager is enabled in s'. Clearly, the manager is active in A'. We claim that A' solves the problem and has response time $\leqslant b$. In order to see this, it suffices to show that every timed behavior of A' is also a timed behavior of A. This is done by showing that for every timed execution of A' there is a timed execution of A with the same behavior.

Let

$$\alpha' = s'_0, (\pi'_1, t'_1), s'_1, ..., s'_{i-1}, (\pi'_i, t'_i), s'_i, ...,$$

be any timed execution of A'. Construct α, a new timed sequence, by removing all *NULL* steps from α'. Assume that

$$\alpha = s_0, (\pi_1, t_1), s_1, ..., s_{i-1}, (\pi_i, t_i), s_i, ...,$$

let Π be the mapping from the indices of events in α to the indices of the corresponding events in α', and set $\Pi(0) = 0$. Note that, for $i \geqslant 1$, if $j = \Pi(i)$, then $s'_j = s_i$, $t'_j = t_i$, and $\pi'_j = \pi_i$. We claim that α is a timed execution of A. Then it follows that every timed behavior of A' is a timed behavior of A.

[2] Note that a nonstrict inequality is used in this assumption, whereas a corresponding assumption for Theorem 4.3 uses a strict inequality. This reflects the difference in the kinds of reasoning needed for lower and upper bound results.

All we have to show is that α satisfies the boundmap of A. The only interesting case is the class *LOCAL*, and since the lower bound for this class is 0, we have to check only the upper bound, l.

Fix some i such that in s_i some locally controlled action of the manager is enabled, and either $i=0$ or no locally controlled action of the manager is enabled in s_{i-1}, or π_i is a locally controlled action of the manager. We must show that within time l after t_i either a locally controlled action of the manager occurs, or there is a state in which no such action is enabled. Let $j=\Pi(i)$. It must be that some locally controlled action of the manager is enabled in s'_j, since some such action is enabled in all states of the manager in A'. We first show that a locally controlled event π of the manager must occur in α' within at most l time after t'_j. There are two cases:

Case 1: $i=0$ or π_i is a locally controlled action of the manager in A. If $i=0$, then it must be that $j=0$. If π_i is a locally controlled action of the manager in A, then it must be that $\pi'_j=\pi_i$. In either case, as the manager in A' is active, a locally controlled event π of the manager must occur in α' within time at most l after t'_j, by the fact that α' is a timed execution of A' and satisfies the boundmap.

Case 2: $i \geqslant 1$ and no locally controlled action of the manager is enabled in s_{i-1}. Then $\pi_i \notin LOCAL$, and hence $\pi'_j \notin LOCAL$. Let k be the largest index of a locally controlled event in α' that has an index $\leqslant j$ (0 if there is no such event). The fact that the class *LOCAL* is always enabled in α' implies that within time l from t'_k a locally controlled event of the manager must occur in α'. By the way k was selected this event must happen after s'_j, so the fact that $t'_j \geqslant t'_k$ implies that a locally controlled event π of the manager must occur in α' within time at most l after t'_j.

In both cases, if $\pi \neq NULL$, then π, with the same time, appears in α, which suffices. If $\pi = NULL$, then the definition of A' implies that in the state just prior to π in α', no non-null locally controlled action of the manager A is enabled. Then no locally controlled action of the manager is enabled in the corresponding state in α, which suffices. ∎

Now we return to the task of proving Theorem 4.9. As in the proof of Theorem 4.7, the proof proceeds by iterative construction of a particular slow timed execution, in segments corresponding to the intervals between two successive *GRANT* events. One again, a lower bound is provided for the time taken by each segment. This time, however, the lower bound for each segment is $c_2(\lfloor (m+l)/c_1 \rfloor + 1)$, which is slightly larger than the bound of $m(c_2/c_1)$ shown previously; the difference involves an additive term of l and a roundoff. This time bound in turn rests on a lower bound of $\lfloor (m+l)/c_1 \rfloor + 1$ on the number of *TICK* events occurring in each segment.

The lower bound on the number of *TICK* events in each segment is achieved by forcing the *GRANT* event at the beginning of the segment to occur after *LOCAL* and *TICK* events occurring at the same time. The segment is then retimed as before to yield a violation of the mutual exclusion propertu; now, however, retiming involves not only shrinking, as it did before, but also moving the *GRANT* event at the beginning of the segment to a time point l later. The fact that the *GRANT* follows *LOCAL* and *TICK* events occurring at the same time implies that this can be done without violating the time constraints.

The following technical lemma gives the inductive step for the construction of the slow timed execution. The proof of this lemma uses the fact that the manager is active.

If i is an index with $0 \leqslant i \leqslant n-1$, we say that i is *unfulfilled* in a timed semi-execution α if the number of $REQUEST_i$ events in α is strictly greater than the number of $GRANT_i$ events in α. We say that a timed execution or timed semi-execution α is *heavily loaded starting from time* t if for all times t', $t \leqslant t' < t_{\text{end}}(\alpha)$, all indices in $\{0, ..., n-1\}$ are unfulfilled in the prefix of α consisting of all the events occurring up to and including time t'. We say that an action is an *ELSE* action if it is a locally controlled action of the manager other than a *GRANT*; *ELSE* events and steps are defined similarly.

LEMMA 4.11. *Let A be a centralized computer system with an active manager, that solves the timed mutual exclusion problem, and let* α *be a slow timed semi-execution of A. Assume that there exists an unfulfilled index in* α, *any GRANT event in* α *is followed by a corresponding FINISH event, and LOCAL and TICK events occur in* α *at time* $t_{\text{end}}(\alpha)$. *Then there exists a slow timed semi-execution* β *extending* α, *such that for some* i, $0 \leqslant i \leqslant n-1$,

$$shed(\beta) = sched(\alpha\sigma)(GRANT(i), t)(REQUEST(i), t)(FINISH(i), t),$$

where $t = t_{\text{end}}(\alpha\sigma)$, σ *consists entirely of TICK and ELSE events, and LOCAL and TICK events occur in* $\alpha\sigma$ *at time* t.

Note that if α is heavily loaded starting from an arbitrary time t' then β is also heavily loaded starting from time t'.

Proof. Assume by way of contradiction that there does not exist a timed semi-execution with the desired properties. We extend α to an infinite timed execution in which no *GRANT* events occur. As there exists an unfulfilled index in α this contradicts the eventual granting property.

This is done by constructing, inductively starting from $j = 0$, successive slow timed semi-executions, $\alpha\sigma_j$, each extending the previous one, such that for every j:

1. σ_j consists entirely of *TICK* and *ELSE* events.
2. *LOCAL* and *TICK* events occur in $\alpha\sigma_j$ at time $t_{\text{end}}(\alpha\sigma_j)$.
3. If $j > 0$ then $t_{\text{end}}(\alpha\sigma_j) \geqslant t_{\text{end}}(\alpha\sigma_{j-1}) + c_2$.

We start with σ_0 being the empty sequence. Clearly, 1 and 3 hold, and the assumptions of the lemma imply that 2 holds. Now, assume we have constructed σ_j, and let s_j be the system state resulting after $\alpha\sigma_j$. There are two cases:

Case 1: There is an execution fragment of the manager along, σ', starting from state s_j, which consists of a sequence of zero or more *ELSE* events followed by some *GRANT*(i) event. (Note that this is an execution fragment of the underlying I/O automaton, without any timing constraints.) Then let β be any timed semi-execution that extends $\alpha\sigma_j$ such that

$$sched(\beta) = sched(\alpha\, \sigma_j\, \sigma')(REQUEST(i),\, t_{\text{end}}(\alpha\sigma_j))(FINISH(i),\, t_{\text{end}}(\alpha\sigma_j)),$$

where the events of σ' are all timed to occur exactly at time $t_{\text{end}}(\alpha\sigma_j)$. Then β has the properties required by the lemma: it ends with *GRANT*(i), *REQUEST*(i), and *FINISH*(i) events, there are only *TICK* and *ELSE* events in the prefix of $\sigma_j\sigma'$ preceding the final *GRANT*(i) event, and *LOCAL* and *TICK* events occur in β at time $t_{\text{end}}(\alpha\sigma_j) = t_{\text{end}}(\beta)$. This is a contradiction to the assumed nonexistence of such a timed semi-execution.

Case 2: There is no such execution fragment. In this case, we can extend $\alpha\sigma_j$ by allowing *ELSE* events to occur, at arbitrary allowable times, ending with an *ELSE* event and a *TICK* event, (occurring in that order) at time $t_{\text{end}}(\alpha\sigma_j) + c_2$. This is possible since the manager is active. Let $\alpha\sigma_{j+1}$ be an execution extending $\alpha\sigma_j$ such that

$$sched(\alpha\sigma_{j+1}) = sched(\alpha\sigma_j\, \delta)(\pi,\, t_{\text{end}}(\alpha\sigma_j) + c_2)(TICK,\, t_{\text{end}}(\alpha\sigma_j) + c_2),$$

where all events (if any) or δ are *ELSE* events, and π is an *ELSE* event.

From the way σ_{j+1} was constructed, it follows that $\alpha\sigma_{j+1}$ is slow, and that it has the following properties:

1. σ_{j+1} consists entirely of *TICK* and *ELSE* events.
2. *LOCAL* and *TICK* events occur in σ_{j+1} at time $t_{\text{end}}(\alpha\sigma_{j+1})$.
3. $t_{\text{end}}(\alpha\sigma_{j+1}) \geqslant t_{\text{end}}(\alpha\sigma_j) + c_2$.

This completes the construction of the timed semi-executions $\alpha\sigma_j$, $0 \leqslant j < \infty$.

Now Lemma 2.2 implies that there exists an infinite timed execution $\alpha\sigma$ extending all the $\alpha\sigma_j$. Since there are no *GRANT* events in σ and there exists an unfulfilled index in α, this contradicts the eventual granting property. ▌

Now we are ready to present the main proof. This proof uses the inductive step described in Lemma 4.11 to construct the needed slow timed execution.

Proof (of Theorem 4.9). Assume that we have a particular centralized computer system that solves the timed mutual exclusion problem. By Lemma 4.10, we may assume without loss of generality that the manager is active. We explicitly construct a (slow) timed execution in which the response time for a particular grant is at least

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1)] + l.$$

We first construct an initial section, β_0. We begin by allowing some *LOCAL* events to occur (at arbitrary allowable times), ending with both a *LOCAL* event and a *TICK* event occurring at exactly time c_2, in that order. Note that by the *request well-formedness* property these *LOCAL* events must be *ELSE* events. We let

$$REQUEST(0), REQUEST(1), ..., REQUEST(n-1)$$

events happen immediately after these *ELSE* and *TICK* events, also at time c_2. Formally, let β_0 be a timed semi-execution that extends another timed semi-execution δ containing only *ELSE* events, such that

$$sched(\beta_0) = sched(\delta)(\pi, c_2)(TICK, c_2)(REQUEST(0), c_2)$$
$$\cdots (REQUEST(n-1), c_2),$$

where π is an *ELSE* event. Note that $0, ..., n-1$ are unfulfilled indices in β_0, that there are no *GRANT* events in β_0, and *LOCAL* and *TICK* events occur in β_0 at time $c_2 = t_{\text{end}}(\beta_0)$; furthermore, note that β_0 is heavily loaded starting from time $t_0 = t_{\text{end}}(\beta_0) = c_2$.

Starting from β_0, we construct successive proper extensions $\beta_1, ..., \beta_k, ...$, such that for each $k \geqslant 1$, β_k is a slow timed semi-execution of the form $\beta_{k-1}\gamma_k$ that ends at time $t_k = t_{\text{end}}(\beta_k)$, that is heavily loaded starting from time t_0, and that has the following properties:

1. β_k ends with $GRANT(j_k)$, $REQUEST(j_k)$, and $FINISH(j_k)$ events, occurring in that order at time t_k.

2. Except for these last three events, γ_k consists entirely of *TICK* and *ELSE* events.

3. Every *GRANT* event in β_k is followed by a corresponding *FINISH* event.

4. A *LOCAL* event (other than the *GRANT*(j_k)) and a *TICK* event occur in β_k at time t_k.

The construction is done inductively; the base case is the construction of β_1. Since there are unfulfilled indices in β_0, there are no *GRANT* events in β_0, and *LOCAL* and *TICK* events occur in β_0 at time $t_{\text{end}}(\beta_0)$, we can apply Lemma 4.11 to get a timed semi-execution β_1 with the properties above.

For the inductive step, assume we have constructed a slow timed semi-execution β_{k-1}, for $k > 1$, with the above properties; we show how to construct β_k. Since β_{k-1} is heavily loaded starting at time t_0, every *GRANT* event in β_{k-1} is followed by a corresponding *FINISH* event, and *LOCAL* and *TICK* events occur in β_{k-1} at time t_{k-1}, we can apply Lemma 4.11 to β_{k-1}, and get a slow timed semi-execution β_k that extends β_{k-1} such that

$$\begin{aligned} sched(\beta_k) = {} & sched(\beta_{k-1}\sigma_k)(GRANT(j_k), t_k) \\ & \times (REQUEST(j_k), t_k)(FINISH(j_k), t_k), \end{aligned}$$

where $t_k = t_{\text{end}}(\beta_{k-1}\sigma_k)$, σ_k consists entirely of *TICK* and *ELSE* events, and *LOCAL* and *TICK* events occur in $\beta_{k-1}\sigma_k$ at time t_k. Let γ_k be such that

$$\beta_k = \beta_{k-1}\gamma_k.$$

Clearly, β_k has the required properties.

The timed execution β_k is depicted in Fig. 3.

CLAIM 4.12. *For any $k > 1$, there are at least $\lfloor (m+l)/c_1 \rfloor + 1$ ticks in segment γ_k of β_k.*

Proof. Suppose this is not the case, for some fixed k. Then we modify β_k to get a new timed semi-execution β'_k, in which the mutual exclusion property is violated.

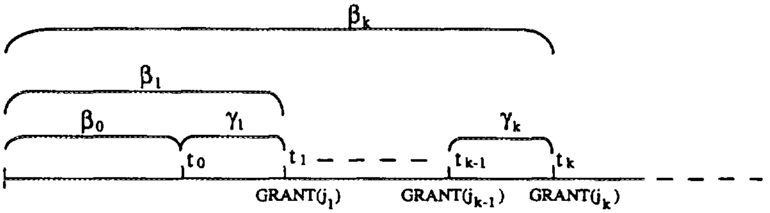

FIG. 3. The timed execution β_k.

First, we do some retiming without changing the order of any of the events. Segment γ_k of β_k is "shrunk" in β'_k so that all ticks contained within segment γ_k take time exactly c_1 (rather than c_2 as in β_k). Moreover, the *GRANT*(j_{k-1}), *REQUEST*(j_{k-1}) and the *FINISH*(j_{k-1}) events occurring at time t_{k-1} are timed to occur at time $t_{k-1}+l$; some *ELSE* steps after *FINISH*(j_{k-1}) and before the next *TICK* may need also to have their times increased slightly to maintain monotonicity. By the fact that $l \leqslant c_1$, and the fact that there is a *LOCAL* event preceding *GRANT*(j_{k-1}), with the same time assignment, it follows that the resulting sequence is a timed execution.

We now obtain β'_k by moving *FINISH*(j_{k-1}) from time $t_{k-1}+l$ to time t_k, after *GRANT*(j_k). We show that β'_k is a timed semi-execution, by showing that moving the *FINISH* event to a later time does not violate the m upper bound on the time between *GRANT*(j_{k-1}) and the corresponding *FINISH*(j_{k-1}). By the assumption, there are at most $\lfloor (m+l)/c_1 \rfloor$ ticks in section γ_k. As *GRANT*(j_{k-1}) occurs at time $t_{k-1}+l$, while *FINISH*(j_{k-1}) occurs at time t_k, the total time between these two events is at most

$$(c_1 - l) + c_1(\lfloor (m+l)/c_1 \rfloor - 1) \leqslant m.$$

So we have obtained a timed semi-execution in which the mutual exclusion properly is violated. By Lemma 2.3, β'_k can be extended to a timed execution; this contradicts the correctness of the algorithm, thus proving the claim. ▌

The claim implies that

$$t_{k+1} - t_k \geqslant c_2(\lfloor (m+l)/c_1 \rfloor + 1),$$

for any $k \geqslant 1$, because β_{k+1} is slow.

We continue the proof of Theorem 4.9. Since for every $k \geqslant 0$, β_k is heavily loaded starting from time t_0 and the algorithm satisfies the eventual granting property, there exists k' such that for every i, $0 \leqslant i \leqslant n-1$, at least one *GRANT*$(i)$ event appears in $\beta_{k'}$ at or after time t_1. By the same reasoning, there exists $k'' > k'$ such that for every i, $0 \leqslant i \leqslant n-1$ at least one *GRANT*(i) event appears in $\beta_{k''}$ after time $t_{k'}$. It follows that there is some i, $0 \leqslant i \leqslant n-1$ for which there are two consecutive *GRANT*(i) events in $\beta_{k''}$ having at least $n-1$ intervening *GRANT*(j) events for $j \neq i$. Suppose that the first of these *GRANT*(i) events occurs at time t_{k_1}, and the second at time t_{k_2}; it must be that $k_2 - k_1 \geqslant n$. Note that the *REQUEST*(i) event corresponding to the second of these *GRANT*(i) events occurs at time t_{k_1}. By the remark after Claim 4.12 the total amount of time from time t_{k_1} in β_{k_2}, when *REQUEST*(i) occurs, until the corresponding *GRANT*(i) occurs at time t_{k_2}, is at least

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1)].$$

We now construct from β_{k_2} a timed semi-execution δ in which the $GRANT(j_{k_2})$ event occurs at time $t_{k_2}+l$, retiming later events as necessary to maintain monotonicity. The timed sequence δ is a timed semi-execution since $l \leqslant c_2$, and since there is a $LOCAL$ event preceding $GRANT(j_{k_2})$ at time t_{k_2} in β_{k_2}. It follows that the total amount of time from time t_{k_1} in δ, when $REQUEST(i)$ occurs, until the corresponding $GRANT(i)$ occurs at time $t_{k_2}+l$, is at least

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1)] + l.$$

Since δ can be extended to a timed execution (by Lemma 2.3) the theorem follows. ▌

We note that Theorem 4.7 seems quite robust in that it can be extended to any reasonable model, including those in which the manager takes steps only in response to inputs. However, the better lower bound in Theorem 4.9 depends more heavily on the features of the timed automaton model. Note that the limiting case of the lower bound in Theorem 4.9 is

$$n[\lfloor m/c_1 \rfloor + 1]\, c_2,$$

which is slightly better than the lower bound given by Theorem 4.7.

5. A Distributed System

Now we consider the case where the computer system is distributed. We assume that the events concerning the different moving parts occur at separate manager processes p_i, $0 \leqslant i \leqslant n-1$, which communicate over unidirectional channels. More precisely, for each ordered pair (i, j), $i \neq j$, we assume that there is a channel automaton $channel(i, j)$ representing a channel from p_i to p_j, having $SEND$ events as inputs and $RECEIVE$ events as outputs. The channel operates as a FIFO queue; when the queue is nonempty, the channel is always enabled to deliver the first item. All $RECEIVE$ actions are in the same partition class, with associated bounds $[0, d]$; this means that the channel will deliver the first item on the queue within time d. Also, we assume that there is a separate clock, $clock(i)$, for each process p_i. It is similar to the centralized clock described earlier, with output action $TICK(i)$ that is an input to p_i, and with associated bounds $[c_1, c_2]$. See Fig. 4.

If the clocks are perfectly accurate, i.e., $c_1 = c_2$, then since all processes start at the same time, there is a very simple algorithm that assigns to each process a periodic predetermined "time slice" and whose worst case

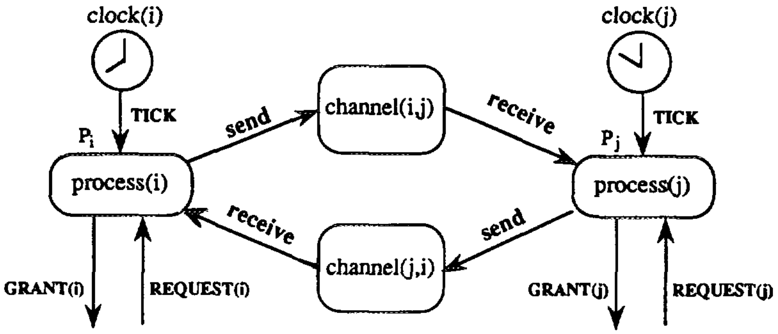

FIG. 4. The architecture of the distributed computer system.

response time is $n \cdot m$ (plus some terms involving and c_2 and l). This is optimal.[3] So, for our lower bound, we assume that $c_1 < c_2$.

5.1. *The Upper Bound*

5.1.1. *The Algorithm*

The following algorithm implements a round-robin granting policy: the processes issue grants when they are in possession of a token that circulates on a ring.

Assume processes are numbered 0, ..., $n-1$ in clockwise order, and interpret $i+1$ to be $i+1 \bmod n$. Each process p_i has input action *REQUEST*(i), *TICK*(i), and *RECEIVE-TOKEN*(i), output actions *GRANT*(i) and *SEND-TOKEN*(i), and internal action *ELSE*(i). The state of process i is divided into components:

REQUESTED	holding a Boolean value, initially *false*;
TIMER	holding an integer, initially 0;
TICKED	holding a Boolean value, initially *true*;
TOKEN	holding a value in {*not_here*, *available*, *used*}, initially *used* for p_0, *not_here* for the other processes.

Process p_i executes the following algorithm:

[3] In fact, even if we deviate from the model by allowing accurate clocks with non-synchronized starts, there is an algorithm which selects synchronization points so that its worst case response time is at most $n \cdot (m + (d/2))$ (plus some terms involving c_2 and l). A corresponding lower bound can also be proved. A formal treatment of these results requires several changes to our model, and we prefer not to present it here. The clock synchronization algorithm of [20] yields synchronization points that can be used by a distributed allocation algorithm whose response time is at most $n \cdot m + (n-1)d$. Since the lower bound of [20] implies that this clock synchronization algorithm is optimal, it does not appear that a naive use of clock synchronization produces optimal mutual exclusion algorithms.

```
REQUEST(i)
Effect:
    REQUESTED := true

TICK(i)
Effect:
    TIMER := TIMER − 1
    TICKED := true

GRANT(i)
Precondition:
    REQUESTED = true
    TOKEN = available
    TICKED = true
Effect:
    REQUESTED := false
    TOKEN := used
    TIMER := $\lfloor (m+l)/c_1 \rfloor + 1$
    TICKED := false

SEND-TOKEN(i)/* to process $p_{i+1}$*/
Precondition:
    TOKEN = used
    TIMER $\leqslant 0$
Effect:
    TOKEN := not_here
    TICKED := false

ELSE(i)
Precondition:
    neither GRANT(i) nor SEND-TOKEN(i) is enabled
Effect:
    TICKED := false

RECEIVE-TOKEN(i)
Effect:
    if REQUESTED then TOKEN := available else TOKEN := used
```

5.1.2. *Correctness Proof*

Now let B be the composition of all the given timed automata: operators, moving parts, processes, channels, and clocks. This subsection is devoted to proving the following theorem.

THEOREM 5.1. *Algorithm B is a distributed computer system that solves the timed mutual exclusion problem.*

As in the proof of the centralized algorithm, we construct the I/O automaton *time*(B). This time, the new state components are *current*, plus, for each i, *first*, and *last* for the following partition classes:

1. *REQUEST*(i), which contains the single action *REQUEST*(i),
2. *FINISH*(i), which contains the single action *FINISH*(i),
3. *TICK*(i), which contains the single action *TICK*(i), and
4. *LOCAL*(i), the class of locally controlled actions of process i, which contains all the actions *GRANT*(i), *SEND-TOKEN*(i), and *ELSE*(i).

Initially, we have $first(REQUEST(i)) = 0$, $last(REQUEST(i)) = \infty$, $first(FINISH(i)) = 0$, $last(FINISH(i)) = \infty$, $first(TICK(i)) = c_1$, $last(TICK(i)) = c_2$, $first(LOCAL(i)) = 0$, and $last(LOCAL(i)) = l$.

Let $\# tokens(i)$ be the length of the queue in $channel(i, i+1)$. We first prove a lemma giving an invariant for *time*(B); this invariant happens not to involve any of the state components that encode time information. The proof appears in Appendix A.2.

Lemma 5.2. *Let s be a reachable state of time*(B). *Then* $|\{i \mid s.TOKEN(i) \neq not\_here\}| + \sum_{i=0}^{n-1} s.\# tokens(i) = 1$.

We now prove another invariant, this one involving the timing information. The result is similar to Lemma 4.2. The proof appears in Appendix A.3.

Lemma 5.3. *Let s be a reachable state of time*(B), *and let* $0 \leqslant i \leqslant n-1$. *Then the following all hold*:

1. *If FINISH*(i) *is enabled in s.basic, then*
   (a) $s.TIMER(i) > 0$,
   (b) $s.first(TICK(i)) + (s.TIMER(i) - 1)\, c_1 > s.last(FINISH(i))$, *and*
   (c) $s.TOKEN(i) = used$.
2. *If* $s.TICKED(i) = true$ *then* $s.first(TICK(i)) \geqslant s.last(LOCAL(i)) + c_1 - l$.

The following corollary implies that mutual exclusion is maintained by the algorithm.

Corollary 5.4. *In any reachable state s of B, if FINISH*(i) *is enabled, for some i, then FINISH*(j) *is not enabled for all* $j \neq i$.

Proof. Assume to the contrary that *FINISH*(j) is enabled in s, for $j \neq i$, Since *FINISH*(i) and *FINISH*(j) are both enabled in s, invariant 1(c) (proved in Lemma 5.3) implies that

$$s.\mathrm{TOKEN}(i) = s.\mathrm{TOKEN}(j) = used.$$

But this implies that the number of processes for which TOKEN $\neq$ *not_here* is at least 2, contradicting Lemma 5.2. Therefore, this case cannot occur. ∎

Proof (of Theorem 5.1). Corollary 5.4 implies mutual exclusion. Moving part well-formedness follows from the same corollary and the definition of the moving part. Request well-formedness follows from the definitions of the operators and the processes. Eventual granting can be argued from the round-robin behavior of the processes; it also follows from the upper bound on response time proved formally in the following subsection. ∎

5.2. *Response Time*

Now we prove the upper bound on response time for the given distributed algorithm *B*.

THEOREM 5.5. *The worst case response time for algorithm B is at most*

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1) + d + c_2 + 2l].$$

We use the following lemmas.

LEMMA 5.6. *In any reachable state s, and for any i,*

$$s.TIMER(i) \leqslant \lfloor (m+l)/c_1 \rfloor + 1.$$

Proof. By an easy induction. ∎

LEMMA 5.7. *Let s be any state occurring in a timed execution, in which $s.TIMER(i) \leqslant k$, for $k \geqslant 1$. Then (at least) one of the following two conditions holds:*

1. *$s.TIMER(i) \leqslant 0$ and $s.TICKED(i) = true$, or*

2. *the time from the given occurrence of s until a later TICK(i) event resulting in $TIMER(i) \leqslant 0$ is bounded above by $c_2 \cdot k$.*

Proof. As for Lemma 4.6. ∎

Say that process p_i is *operative* in state s if s.TOKEN$(i) =$ *used.* By Lemma 5.2 at any time there is at most one operative process.

Lemma 5.8. *If process p_i is operative, then the time until process p_{i+1} becomes operative is at most*

$$c_2(\lfloor (m+l)/c_1 \rfloor + 1) + d + c_2 + 2l.$$

Proof. By Lemmas 5.6 and 5.7, either TIMER$(i) < 0$ and TICKED$(i) =$ *true*, or else within time

$$c_2(\lfloor (m+l)/c_1 \rfloor + 1),$$

a *TICK*(i) event occurs setting TIMER(i)$\leqslant 0$; in either case, *SEND-TOKEN*(i) is enabled within time

$$c_2(\lfloor (m+l)/c_1 \rfloor + 1).$$

Within time l after that, *SEND-TOKEN*(i) occurs and *RECEIVE-TOKEN*($i+1$) is enabled (since it is the only message in the channel), and within an additional time d, it is executed. If there is a pending request at process p_{i+1} when this *RECEIVE-TOKEN*($i+1$) occurs, i.e., REQUESTED($i+1$)$=$*true* at this point, then this *RECEIVE-TOKEN*($i+1$) sets TOKEN($i+1$)$=$*available*. Then within time c_2, *GRANT*($i+1$) is enabled and within time l is is executed, causing process p_{i+1} to become operative. On the other hand, if there is no pending request, i.e., REQUESTED($i+1$)$=$*false*, then the *RECEIVE-TOKEN*($i+1$) sets TOKEN($i+1$)$=$*used* and thereby causes process p_{i+1} to become operative. ▌

Define the *distance* from process p_i to process p_j to be the distance between them along the ring (in the clockwise direction); if $i=j$ we define the distance to be n.

Proof (of Theorem 5.5). Consider the point in the timed execution at which a request arrives, say at process p_j. We consider cases (one of which must hold, by Lemma 5.2).

1. There is some operative process, p_i, when the request arrives (where it is possible that $i=j$). Then the distance from p_i to p_j is at most n. Applying Lemma 5.8 repeatedly (at most n times) yields the claimed bound.

2. The value of TOKEN(i)$=$*available* for some i. If $i=j$, then the request will be granted within time c_2+l. If $i\neq j$, then within time c_2+l, process p_i becomes operative. Applying Lemma 5.8 repeatedly (at most $n-1$ times) yields the claimed bound.

3. There is a message in one of the channels, say *channel*($i-1, i$). If $i=j$, then te request will be granted within time $d+c_2+l$. If $i\neq j$, then within time $d+c_2+l$, process p_i becomes operative. Applying Lemma 5.8 repeatedly (at most $n-1$ times) yields the claimed bound. ▌

Again, we note the limiting case of the upper bound as l approaches 0 is

$$n[c_2(\lfloor m/c_1 \rfloor + 1) + d + c_2].$$

5.3. *Lower Bound*

Now we prove our lower bound on worst case response time for an arbitrary distributed solution to the timed mutual exclusion problem. This proof is similar to that of the simple lower bound for centralized algorithms (Theorem 4.7) rather than the more complicated tight bound (Theorem 4.9) in that we do not concern ourselves with process step time or with roundoffs. As a result, this proof is sufficiently robust to extend to other reasonable models for timing-based computation.

Note that the gap between our upper and lower bounds for the distributed case involves not only process step times and roundoffs, but also involves additive terms of d and of $n \cdot c_2$.

In order to prove this lower bound we must make the assumption that the moving time is much larger than the message delivery time; more precisely, that $(n-1) \cdot d \leqslant m(c_2/c_1)$.

THEOREM 5.9. *Assume that* $c_1 < c_2$ *and that* $(n-1) \cdot d \leqslant m \cdot (c_2/c_1)$. *Then the worst case response time of any distributed computer system that solves the timed mutual exclusion problem is strictly greater than*

$$n \cdot c_2(m/c_1) + (n-1) \cdot d.$$

The lower bound is proved under the assumption that every message is delivered within time d. This is a stronger assumption than the one used for the upper bound; there, we only insist that this upper bound hold for the *first* message on any link. Since the present assumption is stronger, it only serves to strengthen the lower bound.

We start with an informal overview of the proof.

The basic technique used in the proof is called *shifting* and works as follows. Assume that process p_j grants right after p_i; furthermore, assume messages from p_i to p_j are delayed d time, while the messages from p_j to p_i are delayed 0 time. Assume, by way of contradiction, that the difference between the time p_i grants and the time p_j grants is less than or equal to $c_2(m/c_1) + d$. Then we can "shift" events at p_j back in time, by retiming them to occur d time earlier, and then "shrink" them (as in Lemma 4.8) to obtain a violation of mutual exclusion.

If it were possible to apply this argument n times, we could have obtained a lower bound of $n \cdot c_2(m/c_1) + n \cdot d$. Unfortunately, there are two major dificulties with applying this technique. Handling these difficulties requires some assumptions and significantly complicates the proof.

First, applying this technique requires knowing the granting order in advance, so we can fix an appropriate message delay policy. However, it is possible that the algorithm changes the granting order depending on the message delays that occur in the execution. To address this difficulty, we show that the round-robin granting policy used by the algorithm of Section 5.1 is optimal in the following sense: for any "efficient" algorithm, in any heavily loaded execution, the order in which requests are first granted must be repeated in a round-robin fashion. This is done in Lemma 5.11, and requires the assumption that $(n-1) \cdot d \leqslant m \cdot (c_2/c_1)$. Once such an order has been established, we extend the execution while fixing a particular pattern of message delays (according to the *fixed* granting order).

Second, since all processes start operating simultaneously at time 0, we cannot simply shift events at p_j to occur d time earlier. (For example, this might force some events at p_j to occur in the interval $[-d, 0]$.) Indeed, as mentioned earlier, if the clocks are perfectly accurate then the lower bound does not hold. However, if $c_1 < c_2$ then clocks drift apart after a sufficiently long time. We exploit this fact in a technique which we call *shifting while shrinking*: we retime parts of the execution by carefully "shifting" certain events earlier, while appropriately retiming certain preceding events so the events all occur at times $\geqslant 0$. (This is done in Lemma 5.12.)

We now present the details of the proof.

Recall the definition of a *heavily loaded* timed execution or timed semi-execution from Section 4.2. In a manner similar to the centralized case, we define a timed execution or timed semi-execution to be *slow* if, for each i, the times between successive $TICK(i)$ events (and the time of the first $TICK(i)$ event) are exactly c_2. The following lemma is the distributed version of Lemma 4.8.

LEMMA 5.10. *Let α be a slow timed execution of a distributed computer system that solves the timed mutual exclusion problem. Then the time between any two consecutive GRANT events in α is strictly greater then $c_2(m/c_1)$.*

The next lemma shows that if an execution is heavily loaded, the best policy (for an "efficient" algorithm) is to grant the resource in a round robin manner, because changing the granting order will cause the response time to exceed a bound higher than the one we are attempting to prove as a lower bound.

LEMMA 5.11. *Let B be a distributed computer system that solves the timed mutual exclusion problem with response time at most $(n+1) \cdot c_2(m/c_1)$. Let α be a slow timed execution of B that is heavily loaded starting from time t. Then there exists some permutation, ρ, of $\{0, ..., n-1\}$ such that the subsequence of all GRANT events that occur in α after time t is of the form*

$$GRANT(\rho_0), ..., GRANT(\rho_{n-1}), GRANT(\rho_0), ..., GRANT(\rho_{n-1}), ...$$

Proof. Suppose by way of contradiction that there is no such permutation ρ. Then there is some index, for which two *GRANT*(i) events π_1 and π_2 occur (at times t_1 and t_2 respectively) after time t, where there are at least n *GRANT*(j) events, $j \neq i$, intervening between π_1 and π_2.

By Lemma 5.10, the time between any two consecutive *GRANT* events from among this set of $n+1$ *GRANT* events is strictly greater than $c_2(m/c_1)$. Therefore, the time between π_1 and π_2 is strictly greater than

$$(n+1) \cdot c_2(m/c_1).$$

Since α is heavily loaded, a *REQUEST*(i) event must follow π_1 and occur at time t_1. Since that *REQUEST*(i) is fulfilled by π_2 at time t_2, the response time for that *REQUEST*(i) is strictly greater than $(n+1) \cdot c_2(m/c_1)$, which contradicts the assumed bound on the response time of the algorithm. ▌

Proof (of Theorem 5.9). Assume by way of contradiction that there is some algorithm that always responds within time

$$n \cdot c_2(m/c_1) + (n-1)\, d.$$

By assumption

$$(n-1)\, d \leqslant m(c_2/c_1),$$

which implies that

$$n \cdot c_2(m/c_1) + (n-1)\, d \leqslant (n+1) \cdot c_2(m/c_1).$$

Thus, the response time for the algorithm is at most

$$(n+1) \cdot c_2(m/c_1).$$

We construct a slow timed execution of the algorithm that either exceeds the claimed bound on response time or violates the mutual exclusion property. We begin by considering a slow timed execution α' that is heavily loaded starting from some time t, and letting α be the shortest prefix of this timed execution that ends just after exactly n *GRANT* events have occurred after time t. Lemma 5.11 implies that there is some permutation ρ, such that all *GRANT* events that appear in α' after time t occur in the order $\rho_0, ..., \rho_{n-1}, \rho_0, ...$ In fact, Lemma 5.11 implies that *GRANT* events that occur after time t in any timed semi-execution that extends α and is heavily loaded starting from time t, appear in the order $\rho_0, ..., \rho_{n-1}$. We sometimes abuse notation and write $p_{\rho_i} < p_{\rho_j}$ when $i < j$; that is, p_{ρ_i} precedes p_{ρ_j} in the order established by ρ.

We now consider the "ring" of processes formed by the round-robin order defined above. We extend the execution in such a way that messages are delivered with maximum delay when sent from lower numbered processes to higher numbered processes (in the order established by ρ), while messages going the other way are delivered immediately. Intuitively, this "postpones" notification of granting as long as possible and will enable us later to "shift" back in time events that occur at processes that grant later in the granting order.

More formally, we extend α to slow timed execution $\alpha\beta'$ which is heavily loaded starting from time t and such that the message delivery times for messages sent in β' are as follows:

- If $i<j$, then a message from p_{ρ_i} to p_{ρ_j} takes exactly time d.
- If $i>j$, then a message from p_{ρ_i} to p_{ρ_j} takes exactly time 0.

Let $\alpha\beta$ be a "sufficiently long" prefix of $\alpha\beta'$; specifically, one for which

$$\frac{c_1}{c_2} \leqslant \frac{t_{\text{end}}(\alpha\beta) - t_{\text{end}}(\alpha) - d}{t_{\text{end}}(\alpha\beta) - t_{\text{end}}(\alpha)}.$$

This can be done since, by assumption, $c_1/c_2 < 1$. Let $r_1 = t_{\text{end}}(\alpha)$ and $r_2 = t_{\text{end}}(\alpha\beta)$.

Let γ be such that $\alpha\beta\gamma = \alpha\beta'$. We know that γ contains a subsequence of $n+1$ consecutive *GRANT* events, in the order

$$GRANT(\rho_0), GRANT(\rho_1), \ldots, GRANT(\rho_{n-1}), GRANT(\rho_0).$$

Now divide γ into $n+2$ *segments*, $\gamma_0, \ldots, \gamma_{n+1}$, where

1. γ_0 ends with the first of these $GRANT(\rho_0)$ events,
2. for each i, $1 \leqslant i \leqslant n-1$, γ_i starts just after $GRANT(\rho_{i-1})$ and ends with $GRANT(\rho_i)$,
3. γ_n starts just after $GRANT(\rho_{n-1})$ and ends with the second $GRANT(\rho_0)$, and
4. γ_{n+1} includes the rest of γ.

For each i, $0 \leqslant i \leqslant n+1$, let $t_i = t_{\text{end}}(\alpha\beta\gamma_0 \cdots \gamma_i)$. For any $1 \leqslant i \leqslant n$, define the *length* of any segment, γ_i, to be $l_i = t_i - t_{i-1}$. Intuitively, l_i is the amount of time that passes during γ_i.

Figure 5 depicts the timed execution $\alpha\beta\gamma$. Each horizontal line represents events happening at one process, the arrows show delay times between pairs of processes (after time r_0), while dashed vertical lines mark time point that are used in the proof.

We now prove a key lemma that provides a lower bound for the length of each segment $\gamma_1, \ldots, \gamma_{n-1}$.

LEMMA 5.12. *For any i, $1 \leqslant i \leqslant n-1$,*

$$l_i > c_2(m/c_1) + d.$$

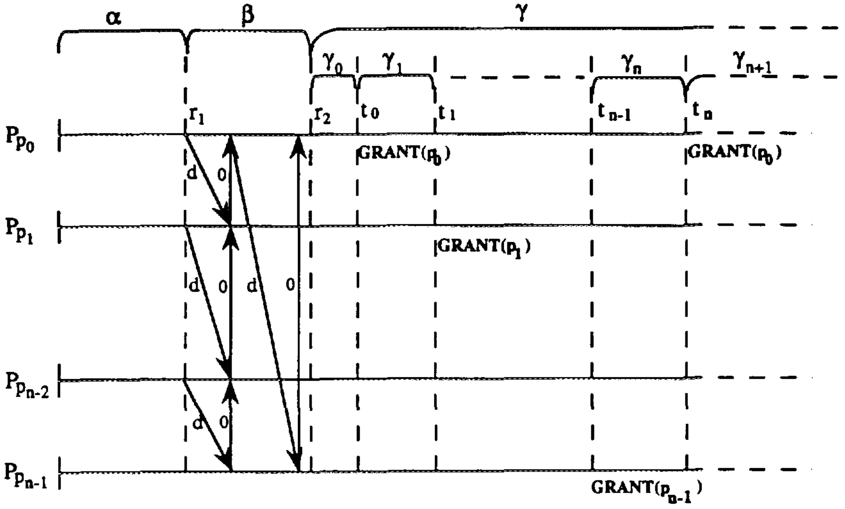

FIG. 5. The timed execution $\alpha\beta\gamma$.

Proof. Asssume by way of contradiction that

$$l_i \leqslant c_2(m/c_1) + d$$

for some particular i, $1 \leqslant i \leqslant n-1$.

From $\alpha\beta\gamma$ we construct a new timed execution, $\alpha\delta$, in which the mutual exclusion property is violated. We first construct an intermediate timed execution $\alpha\delta'$ in which we "shift" back in time the events occurring at processes $p_{\rho_i}, ..., p_{\rho_{n-1}}$, in the following way:

1. Each event occurring at any of the processes $p_{\rho_0}, ..., p_{\rho_{i-1}}$ that occurs in $\beta\gamma$ at time u also occurs in δ' at time u.

2. Each event occurring at any of the processes $p_{\rho_i}, ..., p_{\rho_{n-1}}$ that occurs in $\beta\gamma$ at time u occurs in δ' at time u', where:

(a) If $u > r_2$ then $u' = u - d$.

(b) If $r_1 \leqslant u \leqslant r_2$ then

$$u' = r_1 + \frac{r_2 - r_1 - d}{r_2 - r_1} \cdot (u - r_1).$$

I.e.,

$$\frac{u' - r_1}{u - r_1} = \frac{r_2 - r_1 - d}{r_2 - r_1}.$$

That is, the events occuring at processes $\geqslant p_{\rho_i}$ at times $> r_2$ are moved d earlier; note that events occurring in α (at times $\leqslant r_1$) are not moved. All the intermediate events are shifted back in proportion.

The resulting sequences of timed events must be merged into a single sequence consistenly with the order of the times; events occurring at different processes at the time can be merged in arbitrary order, except that a *SEND* event that corresponds to a *RECEIVE* event in $\alpha\beta\gamma$ must precede it in $\alpha\delta'$.

Claim 5.13. *$\alpha\delta'$ is a timed execution of the system.*

Proof. The key things that need to be shown are that

- No message is received before it is sent.
- No message takes more than time d to be delivered.
- No clock tick takes time less than c_1.

For the first two conditions, note that in $\beta\gamma$ we have that messages take time

- d from all processes $\leqslant p_{\rho_{i-1}}$ to all processes $\geqslant p_{\rho_i}$, and
- 0 in the reverse direction.

We are only shifting events of processes $\geqslant p_{\rho_i}$ earlier by at most d, so message delivery time is kept $\leqslant d$, and no message is received before it is sent.

For the third condition, note that all clock tick intervals are of length c_2 in $\alpha\beta\gamma$, and no portion of this timed execution is shrunk by more than the ratio

$$\frac{r_2 - r_1 - d}{r_2 - r_1}.$$

As the original length of the tick interval was c_2, the new length of a clock tick interval is at least

$$c_2 \cdot \frac{r_2 - r_1 - d}{r_2 - r_1} \geqslant c_1,$$

by the way β was selected. This completes the proof of Claim 5.13. ▮

Now we resume the proof of Lemma 5.12. Note the following additional properties of $\alpha\delta'$:

- Any clock tick interval at a process $\leqslant p_{\rho_{i-1}}$ takes time exactly c_2.
- Any clock tick interval at a process $\geqslant p_{\rho_i}$ that begins at a time $\geqslant r_2 - d$ takes time exactly c_2.

- Any clock tick interval at a process $\geqslant p_{\rho_i}$ that begins at a time $\leqslant r_2 - d$ and ends at a time $u > r_2$ takes time at least $u - r_2 + (c_2 - (u - r_2))(c_1/c_2)$.

- The length of the new segment corresponding to γ_i is at most $c_2(m/c_1)$.

Now to get $\alpha\delta$ from $\alpha\delta'$, we "shrink" the portion of $\alpha\delta'$ after time r_2 by the ratio (c_1/c_2) and move the *FINISH*(ρ_{i-1}) event (of segment γ_i) after the *GRANT*(ρ_i) event (at the end of segment γ_i), thus creating a violation of the *mutual exclusion* property. More precisely, if an event happens at time u' in $\alpha\delta'$, then the corresponding event happens at time u in $\alpha\delta$, where:

1. If $u < r_2$, then $u' = u$.
2. If $u \geqslant r_2$, then $u' = r_2 + (c_1/c_2)(u - r_2)$.

CLAIM 5.14. *$\alpha\delta$ is a timed execution of the system.*

Proof. The key things that need to be shown are that

- No clock tick interval is smaller than c_1.

- The *FINISH*(ρ_{i-1}) event occurs within time m after the corresponding *GRANT*(ρ_{i-1}) event.

For the first condition, if a tick interval happens at process $p_j \leqslant p_{\rho_{i-1}}$ or a tick interval starts no sooner than time $r_2 - d$ in $\alpha\delta'$, then this clearly holds, since the properties of $\alpha\delta'$ stated above implies that those intervals are of length c_2.

The only case left is that of a tick interval that occurs at a process $\geqslant p_{\rho_i}$ and starts before $r_2 - d$ in $\alpha\delta'$. Let u be the time at which the interval ends in $\alpha\delta'$. If $u \leqslant r_2$, then the interval is not shrunk at all, so we can assume that $u > r_2$. Then by the properties of $\alpha\delta'$ stated above, the length of this interval in $\alpha\delta'$ is at least $u - r_2 + (c_2 - (u - r_2))(c_1/c_2)$. But in going from $\alpha\delta'$ to $\alpha\delta$, only the portion of the interval after time r_2 gets shrunk; therefore, the length of the new interval is at least

$$(u - r_2)(c_1/c_2) + (c_2 - (u - r_2))(c_1/c_2) = c_1,$$

as needed for the first condition.

For the second condition, the time between the *GRANT*(ρ_{i-1}) and the *GRANT*(ρ_i) in $\alpha\delta$, i.e., the length of the segment corresponding to γ_i in $\alpha\delta$, is at most m; hence moving *FINISH*(ρ_{i-1}) after *GRANT*(ρ_i) does not violate the m upper bound.

This completes the proof of Claim 5.14. ▌

To complete the proof of Lemma 5.12, wse need only observe that $\alpha\delta$ is a timed execution of the system in which the mutual exclusion property is violated, a contradiction. ▌

Finally, to complete the proof of Theorem 5.9, consider the execution $\alpha\beta\gamma$ and consider the *REQUEST*(ρ_0) that occurs just after the first of the designated *GRANT*(ρ_0) events in γ. From Lemma 5.10 it follows that

$$l_n > c_2(m/c_1).$$

Together with Lemma 5.12 this implies that the total time from that *REQUEST*(ρ_0) event until the corresponding *GRANT*(ρ_0) event is strictly greater than

$$(n-1)(c_2(m/c_1)+d)+c_2(m/c_1)=n\cdot c_2(m/c_1)+(n-1)\,d,$$

as claimed. ▌

6. Discussion and Open Problems

In this paper, we have defined a timing-based variant of the mutual exclusion problem, and have considered both centralized and distributed solutions to this problem. We have proved upper bounds for both cases, based on simple algorithms; these bounds are fairly complicated functions of clock time, manager or process step time, moving time for the moving parts, and (in the distributed case) message delivery time.

We have proved corresponding lower bounds for both cases. In the centralized case, the lower bound exactly matches the upper bound, even when the manager step time and the roundoffs are considered. In the more complicated distributed setting, the lower bound is very close to the upper bound, but does not match it exactly.

The bounds are all proved using the *timed automaton* model for timing-based concurrent systems. It is interesting to ask how dependent the results are on this choice of model. The timed automaton model differs from some others in modeling process steps explicitly (rather than assuming that the algorithms are interrupt-driven); thus, our results involving this process step time would not be expected to extend immediately to such interrupt-driven models (except possibly in the limit, as this step time approaches 0). However, some of our results—most notably, the lower bound for the distributed case—do not involve process step times and thus appear to be quite model-independent. An alternative approach would be to use a general model that describes interrupt-driven computation, but we do not yet know (in general) how to define such a model.

There are several open questions directly related to the work presented in this paper. First, there is a gap remaining between the upper and lower bound results for the distributed timed mutual exclusion problem. Even neglecting process step time, there is a difference of an additive term of d, the upper bound on message delivery time, plus a term of $n \cdot c_2$, the number of processes times the upper bound on the clock tick time. Preliminary results suggest that under certain assumptions about the relative sizes of the parameters, the upper bound can be reduced by approximately d. However, we do not yet have a general result about this.

Our lower bound for the distributed timed mutual exclusion problem assumes that $(n-1) \cdot d \leqslant m \cdot (c_2/c_1)$. It would be interesting to see if this assumption can be removed.

It would also be interesting to consider the same problem in a model in whick there are nontrivial lower bounds on the time for message delivery (and perhaps for process steps). While our upper bound proofs still work in this situation, the same is not true for our lower bound proofs. The strategy of shrinking and shifting timed executions to produce other timed executions becomes much more delicate when lower bounds on these various kinds of events must also be respected.

Our results imply that the ratio c_2/c_1 has a significant impact on the response time of the system. It would also be interesting to consider the case where a process has more than one clock, say an additional clock with bounds $[c'_1, c'_2]$. We would like to understand how the results depend on the four parameters c_1, c_2, c'_1, and c'_2.

A variant of the problem studied in this paper includes explicit notification from the control rods to the computer system when the rods have finished moving. For this variant, it is possible to obtain upper bounds that are smaller than ours; in particular, the timing uncertainty c_2/c_1 does not appear in the bounds. It appears that for many problems, explicit notifications will permit similar improvements in time complexity. On the other hand, such notifications may not always be available, or may incur significant costs in hardware or communication, or may themselves introduce overhead in time complexity. Liskov [19] discusses several practical situations in which it seems better to use time estimates than to rely on explicit notifications.

Other related problems can also be studied using the models and techniques of this paper. One could define timing-based analogs of other problems besides mutual exclusion that have been studied in the asynchronous setting (for example, other exclusion problems such as the *dining philosophers* problem, distributed consensus problem, or synchronization problems such as the *session problem* of [1]); some results in this direction appear in [2, 3, 26, 27]. In addition to defining variants of asynchronous problems, one can also extract prototypical problems from practical real-time systems research and use them as a basis for combinatorial work.

In another direction, the algorithm proofs presented here suggests general approaches to verification of real-time systems. As mentioned in Section 4.1.3, we believe that there is a unified method for treating *correctness* and *performance analysis* of timing-based algorithms; this is explored in [22].

Work of the sort presented here (and the extensions proposed above) should provide an excellent basis for evaluating the timed automaton model as a general model for reasoning about timing-based systems (and comparing it with alternative models for timing-based computation).

A. Proofs of Lemmas

A.1. *Proof of Lemma* 4.2

The proof is by induction on the length of a finite execution, α, that ends in state s. The base, length 0, is trivial since $FINISH(i)$ is not enabled in any initial state. So suppose that $\alpha = \alpha'(s', (\pi, t), s)$ and the result holds for α' and s'. We show that it holds for α and s. We consider cases.

Case 1: $\pi = REQUEST(j)$, for some j, $0 \leqslant j \leqslant n-1$, or $\pi = ELSE$. First suppose that $FINISH(i)$ is enabled in $s.basic$, for some i, $0 \leqslant i \leqslant n-1$ (where i might or might not be equal to j). Then it is also enabled in $s'.basic$. The inductive hypothesis implies that

1. (a) $s'.\mathrm{TIMER} > 0$,
   (b) $s'.first(TICK) + (s'.\mathrm{TIMER} - 1)\, c_1 > s'.last(FINISH(i))$, and
   (c) $FINISH(k)$ is not enabled in $s'.basic$, for any $k \neq i$.

Since $s.\mathrm{TIMER} = s'.\mathrm{TIMER}$, we have $s.\mathrm{TIMER} > 0$. Since

$$s.first(TICK) = s'.first(TICK),$$

and

$$s.last(FINISH(i)) = s'.last(FINISH(i)),$$

we have that

$$s.first(TICK) + (s.\mathrm{TIMER} - 1)\, c_1 > s.last(FINISH(i)).$$

Also, $FINISH(k)$ is not enabled in $s.basic$, for any $k \neq i$.

Now suppose that s.TICKED $=$ *true*. Then it must be that π is *REQUEST*(j) and s'.TICKED $=$ *true*. Then

$$s'.\mathit{first}(\mathit{TICK}) \geqslant s'.\mathit{last}(\mathit{LOCAL}) + c_1 - l.$$

Since

$$s.\mathit{first}(\mathit{TICK}) = s'.\mathit{first}(\mathit{TICK}),$$

and

$$s.\mathit{last}(\mathit{LOCAL}) = s'.\mathit{last}(\mathit{LOCAL}),$$

we have that

$$s.\mathit{first}(\mathit{TICK}) \geqslant s.\mathit{last}(\mathit{LOCAL}) + c_1 - l.$$

Case 2: $\pi = \mathit{FINISH}(j)$, for some j, $0 \leqslant j \leqslant n-1$. First, suppose that *FINISH*(i) is enabled in $s.\mathit{basic}$, for some i, $0 \leqslant i \leqslant n-1$. It cannot be that $i=j$, so $j \neq i$. But then both *FINISH*(i) and *FINISH*(j) are enabled in $s'.\mathit{basic}$, which contradicts the inductive hypothesis. Therefore, this case cannot occur.

Second, suppose that s.TICKED $=$ *true*. Then the same argument as in Case 1 shows that

$$s.\mathit{first}(\mathit{TICK}) \geqslant s.\mathit{last}(\mathit{LOCAL}) + c_1 - l.$$

Case 3: $\pi = \mathit{TICK}$. First suppose that *FINISH*(i) is enabled in $s.\mathit{basic}$ for some i, $0 \leqslant i \leqslant n-1$. Then it is also enabled in $s'.\mathit{basic}$, so the inductive hypothesis implies that

1. (a) s'.TIMER > 0,

   (b) $s'.\mathit{first}(\mathit{TICK}) + (s'.\mathrm{TIMER} - 1)\, c_1 > s'.\mathit{last}(\mathit{FINISH}(i))$, and

   (c) *FINISH*(k) is not enabled in $s'.\mathit{basic}$, for any $k \neq i$.

We first prove that s.TIMER > 0. If not, then it must be that s'.TIMER $= 1$. Then the inductive hypothesis implies that

$$s'.\mathit{first}(\mathit{TICK}) > s'.\mathit{last}(\mathit{FINISH}(i)).$$

But then the definition of *time*(A) implies that (TICK, t) is not enabled in s', since a *FINISH*(i) must happen first. This is a contradiction.

For invariant 1(b), we see that

$$\begin{aligned}
&s.\mathit{first}(\mathit{TICK}) + (s.\mathrm{TIMER} - 1)\, c_1 \\
&\quad = t + c_1 + (s'.\mathrm{TIMER} - 1 - 1)\, c_1 \\
&\quad = t + (s'.\mathrm{TIMER} - 1)\, c_1, \\
&\quad > t + s'.\mathit{last}(\mathit{FINISH}(i)) - s'.\mathit{first}(\mathit{TICK}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by the inductive hypothesis)} \\
&\quad \geqslant s'.\mathit{last}(\mathit{FINISH}(i)) \qquad \text{(by the definition of } \mathit{time}(A)\text{)} \\
&\quad = s.\mathit{last}(\mathit{FINISH}(i)).
\end{aligned}$$

Thus,

$$s.\mathit{first}(\mathit{TICK}) + (s.\mathrm{TIMER} - 1)\, c_1 > s.\mathit{last}(\mathit{FINISH}(i)).$$

The third clause carries over easily.

Now suppose (actually, it must happen) that $s.\mathrm{TICKED} = \mathit{true}$. By the definition of $\mathit{time}(A)$, $s.\mathit{first}(\mathit{TICK}) = t + c_1$ and $s.\mathit{last}(\mathit{LOCAL}) \leqslant t + l$, so

$$s.\mathit{first}(\mathit{TICK}) \geqslant s.\mathit{last}(\mathit{LOCAL}) + c_1 - l.$$

Case 4: $\pi = \mathit{GRANT}(j)$, for some j, $0 \leqslant j \leqslant n-1$. First suppose that $\mathit{FINISH}(i)$ is enabled in $s.\mathit{basic}$, for some i, $0 \leqslant i \leqslant n-1$. If $i \neq j$, then $\mathit{FINISH}(i)$ is also enabled in $s'.\mathit{basic}$, so by the inductive hypothesis, $s'.\mathrm{TIMER} > 0$. But this contradicts the preconditions of $\mathit{GRANT}(j)$. Therefore, it must be that $i = j$.

The effects of $\mathit{GRANT}(i)$ imply that $s.\mathrm{TIMER} > 0$. Note that

$$s'.\mathit{last}(\mathit{LOCAL}) \geqslant t$$

(since *GRANT* is a locally controlled action) and that

$$s'.\mathit{first}(\mathit{TICK}) = s.\mathit{first}(\mathit{TICK}).$$

Then

$$\begin{aligned}
&s.\mathit{first}(\mathit{TICK}) + (s.\mathrm{TIMER} - 1)\, c_1 \\
&\quad = s.\mathit{first}(\mathit{TICK}) + (s.\mathrm{TIMER} - 1)\, c_1 \\
&\quad \geqslant s'.\mathit{last}(\mathit{LOCAL}) + c_1 - l + (s.\mathrm{TIMER} - 1)\, c_1 \\
&\qquad\qquad \text{(by the inductive hypothesis, since } s'.\mathrm{TICKED} = \mathit{true}\text{)} \\
&\quad \geqslant t + c_1 - l + (s.\mathrm{TIMER} - 1)\, c_1 \qquad \text{(by the inequality above)} \\
&\quad = t + c_1 - l + (\lfloor (m+l)/c_1 \rfloor)\, c_1 \\
&\quad > t + m = s.\mathit{first}(\mathit{FINISH}(i)).
\end{aligned}$$

Thus,

$$s.\mathit{first}(\mathit{TICK}) + (s.\mathrm{TIMER} - 1)\, c_1 > s.\mathit{last}(\mathit{FINISH}(i))$$

as needed.

The mutual exclusion condition has already been shown.

It is not possible that TICKED $= \mathit{true}$ in s, by the effects of the *GRANT*.

A.2. *Proof of Lemma* 5.2

The proof is by induction on the length of a finite execution, α, that ends in state s. The base, length 0, is trivial. So suppose that $\alpha = \alpha'(s', (\pi, t), s)$ and the result holds for α' and s'. We show that it holds for α and s, by considering cases.

Case 1: *π is a REQUEST, ELSE, FINISH, TICK,* or *GRANT* action. These steps do not change the contents of any channel or the number of processes for which $s.\mathrm{TOKEN}(i) \neq \mathit{not\_here}$.

Case 2: $\pi = \mathit{RECEIVE\text{-}TOKEN}(j)$, for some j, $0 \leqslant j \leqslant n-1$. Since *RECEIVE-TOKEN*$(j)$ is enabled in $s'.\mathit{basic}$ we have that $\#\mathit{tokens}(j-1) \geqslant 1$. By the induction hypothesis, this implies that for all processes i, $s'.\mathrm{TOKEN}(i) = \mathit{not\_here}$. The length of one channel queue is decreased by one, while one token state (of j) is changed from *not_here* to *available*; thus, the total number of tokens on channels plus the number of processes holding the token (i.e., having TOKEN $\neq \mathit{not\_here}$) is preserved.

Case 3: $\pi = \mathit{SEND\text{-}TOKEN}(j)$, for some j, $0 \leqslant j \leqslant n-1$. The number of processes for which $s.\mathrm{TOKEN}(j) = \mathit{not\_here}$ is decreased by one relative to s', while the number of messages on the channels is increased by one. This implies that the sum we are interested in remains the same.

A.3. *Proof of Lemma* 5.3

The proof is by induction on the length of a finite execution, α, that ends in state s. The base, length 0, is trivial. So suppose that $\alpha = \alpha'(s', (\pi, t), s)$ and the result holds for α' and s'. We show that it holds for α and s, by considering cases.

Case 1: $\pi = \mathit{REQUEST}(j)$ or $\pi = \mathit{ELSE}(j)$, for some j, $0 \leqslant j \leqslant n-1$. First suppose that *FINISH*(i) is enabled in $s.\mathit{basic}$, for some i, $0 \leqslant i \leqslant n-1$ (where i might or might not be equal to j). Then it is also enabled in $s'.\mathit{basic}$. The inductive hypothesis implies that:

1. (a) $s'.\mathrm{TIMER}(i) > 0$,

   (b) $s'.\mathit{first}(\mathit{TICK}(i)) + (s'.\mathrm{TIMER}(i) - 1)\, c_1 > s'.\mathit{last}(\mathit{FINISH}(i))$,

and

   (c) $s'.\mathrm{TOKEN}(i) = \mathit{used}$.

Since $s.\mathrm{TIMER}(i) = s'.\mathrm{TIMER}(i)$ we have $s.\mathrm{TIMER}(i) > 0$, showing 1(a). Since

$$s.\mathit{first}(\mathit{TICK}(i)) = s'.\mathit{first}(\mathit{TICK}(i)),$$

and

$$s.\mathit{last}(\mathit{FINISH}(i)) = s'.\mathit{last}(\mathit{FINISH}(i)),$$

we have that

$$s.\mathit{first}(\mathit{TICK}(i)) + (s.\mathrm{TIMER}(i) - 1)\, c_1 > s.\mathit{last}(\mathit{FINISH}(i)).$$

So we have invariant 1(b). Invariant 1(c) carries over as this step does not change token states.

Now suppose that $s.\mathrm{TICKED}(i) = \mathit{true}$.

Then $s'.\mathrm{TICKED}(i) = \mathit{true}$, and

$$s'.\mathit{first}(\mathit{TICK}(i)) \geqslant s'.\mathit{last}(\mathit{LOCAL}(i)) + c_1 - l.$$

Since

$$s.\mathit{first}(\mathit{TICK}(i)) = s'.\mathit{first}(\mathit{TICK}(i))$$

and

$$s.\mathit{last}(\mathit{LOCAL}(i)) = s'.\mathit{last}(\mathit{LOCAL}(i))$$

we have that

$$s.\mathit{first}(\mathit{TICK}(i)) \geqslant s.\mathit{last}(\mathit{LOCAL}(i)) + c_1 - l.$$

So we have invariant 2.

Case 2: $\pi = \mathit{FINISH}(j)$, for some j, $0 \leqslant j \leqslant n-1$. First suppose that $\mathit{FINISH}(i)$ is enabled in $s.\mathit{basic}$, for some i, $0 \leqslant i \leqslant n-1$. It cannot be that $i = j$ so $j \neq i$. Then $\mathit{FINISH}(i)$ is also enabled in s'. As $\mathit{FINISH}(j)$ is also enabled in s', we have, by invariant 1(c), that $s'.\mathrm{TOKEN}(j) = \mathit{used}$. Similarly, as $\mathit{FINISH}(i)$ is enabled in s', we have, by invariant 1(c), that $s'.\mathrm{TOKEN}(i) = \mathit{used}$. But this implies that the number of processes for which $\mathrm{TOKEN} \neq \mathit{not\_here}$ is at least 2, contradicting Lemma 5.2. Therefore, this case cannot occur, and we have invariant 1.

For invariant 2, suppose that $s.\mathrm{TICKED}(i) = true$. Then the same argument as in Case 1 shows that, for all i,

$$s.\mathit{first}(\mathit{TICK}(i)) \geqslant s.\mathit{last}(\mathit{LOCAL}(i)) + c_1 - l.$$

Case 3: $\pi = \mathit{TICK}(j)$, for some j, $0 \leqslant j \leqslant n-1$. First suppose that $\mathit{FINISH}(i)$ is enabled in $s.\mathit{basic}$. Then it is also enabled in $s'.\mathit{basic}$, so the inductive hypothesis implies that

1. (a) $s'.\mathrm{TIMER}(i) > 0$,

   (b) $s'.\mathit{first}(\mathit{TICK}(i)) + (s'.\mathrm{TIMER}(i) - 1)\, c_1 > s'.\mathit{last}(\mathit{FINISH}(i))$,

and

   (c) $s'.\mathrm{TOKEN}(i) = used$.

We first prove that $s.\mathrm{TIMER}(i) > 0$. If not, then since only $\mathit{TICK}(i)$ can decrease $\mathrm{TIMER}(i)$, it must be that $j = i$ and $s'.\mathrm{TIMER}(i) = 1$. Then the inductive hypothesis implies that

$$s'.\mathit{first}(\mathit{TICK}(i)) > s'.\mathit{last}(\mathit{FINISH}(i)).$$

But then the definition of $\mathit{time}(B)$ implies that $\mathit{TICK}(i)$ is not enabled in s' (since $\mathit{FINISH}(i)$ must happen first). This is a contradiction, so we have invariant 1(a).

For invariant 1(b), if $i = j$, then

$$s.\mathrm{TIMER}(i) = s'.\mathrm{TIMER}(i) - 1$$

and we see that

$$\begin{aligned}
&s.\mathit{first}(\mathit{TICK}(i)) + (s.\mathrm{TIMER}(i) - 1)\, c_1\\
&\quad = t + c_1 + (s'.\mathrm{TIMER}(i) - 1 - 1)\, c_1\\
&\quad = t + (s'.\mathrm{TIMER}(i) - 1)\, c_1\\
&\quad > t + s'.\mathit{last}(\mathit{FINISH}(i)) - s'.\mathit{first}(\mathit{TICK}(i))\\
&\qquad\qquad\qquad \text{(by the inductive hypothesis)}\\
&\quad \geqslant s'.\mathit{last}(\mathit{FINISH}(i))\\
&\quad = s.\mathit{last}(\mathit{FINISH}(i)).
\end{aligned}$$

Therefore,

$$s.\mathit{first}(\mathit{TICK}(i)) + (s.\mathrm{TIMER}(i) - 1)\, c_1 > s.\mathit{last}(\mathit{FINISH}(i)),$$

and we have invariant 1(b). If $i \neq j$ then invariant 1(b) follows as in Case 1. Invariant 1(c) carries over as this step does not change token states.

Now suppose that $s.\mathrm{TICKED}(i) = \mathit{true}$. If $i = j$, then $s.\mathit{first}(\mathit{TICK}(i)) = t + c_1$ and $s.\mathit{last}(\mathit{LOCAL}(i)) \leqslant t + l$, so

$$s.\mathit{first}(\mathit{TICK}(i)) \geqslant s.\mathit{last}(\mathit{LOCAL}(i)) + c_1 - l,$$

as needed for invariant 2. On the other hand, if $i \neq j$, then $s'.\mathrm{TICKED}(i) = \mathit{true}$ and the induction hypothesis on invariant 2 implies that

$$s'.\mathit{first}(\mathit{TICK}(i)) \geqslant s'.\mathit{last}(\mathit{LOCAL}(i)) + c_1 - l.$$

Then invariant 2 for s follows as in Case 1.

Case 4: $\pi = \mathit{GRANT}(j)$, for some j, $0 \leqslant j \leqslant n-1$. Then $s'.\mathrm{TOKEN} = \mathit{available}$. First suppose that $\mathit{FINISH}(i)$ is enabled in $s.\mathit{basic}$, for some i, $0 \leqslant i \leqslant n-1$. If $i \neq j$ then $\mathit{FINISH}(i)$ is also enabled in $s'.\mathit{basic}$, so by inductive hypothesis (invariant 1(c)), $s'.\mathrm{TOKEN}(i) = \mathit{used}$. But this contradicts Lemma 5.2, so $i = j$.

Then the effects of $\mathit{GRANT}(j)$ imply that $s.\mathrm{TIMER}(j) > 0$, so we have invariant 1(a). Note that

$$s'.\mathit{last}(\mathit{LOCAL}(j)) \geqslant t$$

and that

$$s'.\mathit{first}(\mathit{TICK}(j)) = s.\mathit{first}(\mathit{TICK}(j)).$$

Then

$$\begin{aligned}
&s.\mathit{first}(\mathit{TICK}(j)) + (s.\mathrm{TIMER}(j) - 1)\, c_1 \\
&\quad = s'.\mathit{first}(\mathit{TICK}(j)) + (s.\mathrm{TIMER}(j) - 1)\, c_1 \\
&\quad \geqslant s'.\mathit{last}(\mathit{LOCAL}(j)) + c_1 - l + (s.\mathrm{TIMER}(j) - 1)\, c_1 \\
&\qquad\qquad\qquad\qquad \text{(by the inductive hypothesis)} \\
&\quad \geqslant t + c_1 - l + (s.\mathrm{TIMER}(j) - 1)\, c_1 \\
&\quad = t + c_1 - l + (\lfloor (m + l)/c_1 \rfloor)\, c_1 \\
&\quad > t + m = s.\mathit{last}(\mathit{FINISH}(j)).
\end{aligned}$$

Thus,

$$s.\mathit{first}(\mathit{TICK}(j)) + (s.\mathrm{TIMER}(j) - 1)\, c_1 > s.\mathit{last}(\mathit{FINISH}(j))$$

and we have invariant 1(b).

Invariant 1(c) follows from the effects of the *GRANT*.

Now suppose that $s.\mathrm{TICKED}(i) = \mathit{true}$. Then the effects of $\mathit{GRANT}(j)$ imply that $j \neq i$. Then invariant 2 follows as in Case 3.

Case 5: $\pi = \mathit{RECEIVE\text{-}TOKEN}(j)$, for some j, $0 \leqslant j \leqslant n-1$. From the inductive hypothesis on invariant 1(c) and Lemma 5.2 it follows that $\mathit{FINISH}(i)$ is not enabled in s', hence it is not enabled in s. So we have invariant 1.

Invariant 2 follows as in Case 1.

Case 6: $\pi = \mathit{SEND\text{-}TOKEN}(j)$, for some j, $0 \leqslant j \leqslant n-1$. If $\mathit{FINISH}(i)$ is enabled in s, then it is also enabled in s'. Then from the induction hypothesis on invariant 1(a) it follows that $s'.\mathrm{TIMER}(i) > 0$, and $s'.\mathrm{TOKEN}(i) = \mathit{used}$. This implies that $\mathit{SEND\text{-}TOKEN}(i)$ is not enabled in s'. Hence, $j \neq i$. However, by Lemma 5.2, $s'.\mathrm{TOKEN}(j) = \mathit{not\_here}$, hence $\mathit{SEND\text{-}TOKEN}(j)$ is not enabled in s'. This is a contradiction, so invariant 1 holds.

Invariant 2 follows as in Case 1.

Acknowledgments

We thank Nancy Leveson for providing us with background information on real time systems, and for suggestions and encouragement in the early stages of this work. Thanks are also due to Jennifer Welch for discussions about clock synchronization and for reading the paper and providing us with valuable comments. We also thank Michael Merrit and Mark Tuttle for discussions about modeling time and John Keen, Marios Mavronicolas, and Steve Ponzio for comments on earlier versions of this paper.

Received July 17, 1989; final manuscript received December 10, 1991

References

1. Arjomandi, E., Fischer, M. J., and Lynch, N. (1983), Efficiency of synchronous versus asynchronous distributed systems, *J. Assoc. Comput. Mach.* **30**, 449–456.
2. Attiya, H., Dwork, C., Lynch, N. A., and Stockmeyer, L. J. (1991), Bounds on the time to reach agreement in the presence of timing uncertainty, *in* "Proceedings, 23rd ACM Symposium on Theory of Computing," pp. 359–369; also (1990), Technical Memo MIT/LCS/TM-435, Laboratory for Computer Science, MIT.
3. Attiya, H., and Mavronicolas, M. (1990), Efficiency of semi-synchronous vs. asynchronous networks, *in* "Proceedings, 28th annual Allerton Conference on Communication, Control and Computing," pp. 578–587; Also (1990), Technical Report 21-90, Department of Computer Science, Harvard University.
4. Baeten, J. C. M., and Bergstra, J. A. (1990), "Real Time Process Algebra," Technical Report P8916b, University of Amsterdam.
5. Bernstein, A., and Harter, P. Jr. (1981), Proving real-time properties of programs with temporal logic, *in* "Proceedings, 8th Symposium on Operating System Principles," *Operating Systems Rev.* **15**, 1–11.
6. Coolahan, J. E., and Roussopoulus, N. (1983), Timing requirements for time-driven systems using augmented Petri nets, *IEEE Trans. Software Engrg.* **SE-9**, 603–616.
7. Dasarathy, B. (1983), Timing constraints of real-time systems: Constructs for expressing them, methods for validating them, *IEEE Trans. Software Engrg.* **SE-11**, 80–86.

8. Dolev, D., Halpern, J., and Strong, H.R. (1986), On the possibility and impossibility of achieving clock synchronization, *J. Comput. System Sci.* **32**, 230–250.

9. Dwork, C., and Stockmeyer, L. (1991), "Bounds on the Time to Reach Agreement as a Function of Message Delay," IBM Research Report RJ8181, Almaden Research Center, San Jose, CA.

10. Gerber, R., and Lee, I. (1989), The formal treatment of priorities in real-time computation, *in* "Proceedings, 6th IEEE Workshop on Real-Time Software and Operating Systems."

11. Halpern, J., Megiddo, N., and Munshi, A. A. (1985), Optimal precision in the presence of uncertainty, *J. Complexity* **1**, 170–196.

12. Hasse, V. H. (1981), Real-time behavior of programs, *IEEE Trans. Software Engrg.* **SE-7**, 494–501.

13. Huizing, C., Gerth, R., and deRoever, W. P. (1987), Full abstraction of a real-time denotational semantics for an OCCAM-like language, *in* "Proceedings, 14th ACM Symposium on Principles of Programming Languages," pp. 223–237.

14. Jahanian, F., and Mok, A. (1986), Safety analysis of timing properties in real-time systems, *IEEE Trans. Software Engrg* **SE-12**, 890–904.

15. Jahanian, F., and Mok, A. (1987), A graph-theoretic approach for timing analysis and its emplementation, *IEEE Trans. Comput.* **C-36**, 961–975.

16. Koymans, R., Shyamasundar, R. K., deRoever, W. P., Gerth, R., and Arun-Kumar, S. (1988), Compositional semantics for real-time distributed computing, *Inform. and Comput.* **79**, 210–256.

17. Lamport, L. (1978), Time, clocks and the ordering of events in distributed systems, *Comm. ACM* **21**, 558–565.

18. Leveson, N., and Stolzy, J. (1989), Safety analysis using Petri Nets, *IEEE Trans. Software Engrg.* **SE-13**, 386–397.

19. Liskov, B. (1990), Practical uses of synchronized clocks, invited talk at the 9th Annual ACM Symposium on Principles of Distributed Computing.

20. Lundelius, J., and Lynch, N. (1988), A new fault-tolerant algorithm for clock synchronization, *Inform. and Comput.* **77**, 1–36.

21. Lynch, N. (1988), Modelling real-time systems, *in* "Foundations of Real-Time Computing Research Initiative," ONR Kickoff Workshop, 1–16.

22. Lynch, N. A., and Attiya, H. (1990), Using mappings to prove timing properties, *in* "Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC)," pp. 265–280; Also (1991), Technical Memo MIT/LCS/TM-412.c, Laboratory for Computer Science, MIT.

23. Lynch, N., and Tuttle, M. (1987), Hierarchical correctness proofs for distributed algorithms, *in* "Proceedings, 7th ACM Symposium on Principles of Distributed Computing," pp. 137–151; Also (1987), Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT.

24. Lynch, N., and Tuttle, M. (1989), An introduction to input/output automata, *CWI-Quarterly* **2**; also (1988), Technical Memo, MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.

25. Merrit, M., Modugno, F., and Tuttle, M. (1991), Time constrained automata, *CONCUR*, to appear.

26. Ponzio, S. (1991), Consensus in the presence of timing uncertainty: Omission and byzantine failures, *in* "Proceedings, 10th Annual ACM Symposium on Principles of Distributed Computing (PODC)," to appear.

27. Ponzio, S. (1991), "The Real-Time Cost of Timing Uncertainty: Consensus and Failure Detection," S. M. Thesis, Laboratory for Computer Science, MIT.

28. Reed, G. M., and Roscoe, A. W. (1986), A timed model for communicating sequential processes, *in* "ICALP '86".

29. Shankar, A. U., and Lam, S. (1989), Time-dependent distributed systems: Proving safety, liveness and timing properties, *Distrib. Comput.* **2** 61–79.

30. Sifakis, J. (1975), Petri nets for performance evaluation, *in* Measuring, Modeling and Evaluating Computer Systems, *in* "Proceedings 3rd Symposium IFIP Working Group 7.3," H. Beilner and E. Gelenbe, (Eds.) Amsterdam, North-Holland.

31. Simons, B., Welch, J. L., and Lynch, N. (1988), "An Overview of Clock Synchronization," IBM Technical Report RJ 6505.

32. Welch, J. L., and Lynch, N. (1984), An upper and lower bound for clock synchronization, *Inform. and Control* **62**, Nos. 2/3 (August/September 1984), pp. 190–204.

33. Zwarico, A., Lee, I., and Gerber, R. A complete axiomatization of real-time processes, submitted for publication.