CHAPTER 5

Theory of Real-Time Systems – Project Survey*

Hagit Attiya†
Nancy A. Lynch
Laboratory for Computer Science
MIT
Cambridge, MA 02139

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 implementations satisfy the

*This work was supported by ONR contract N00014-85-K-0168, by NSF grants CCR-8611442 and CCR-8915206, and by DARPA contracts N00014-89-J-1988 and N00014-87-K-0825.
†Current address: Computer Science Department, The Technion, Haifa 32000, Israel.
requirements. These frameworks are based on, among others, state machines ([18, 78, 92]), weakest precondition methods ([43]), first-order logic ([47, 48]), temporal logic ([11]), Petri nets ([17, 62, 91]), and process algebra ([9, 40, 46, 53, 88, 100]). 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.\textsuperscript{1} That theory contains many combinatorial results that show what can and 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 [24, 42, 56, 66, 99] (see [94] for a survey).

In this project, we have embarked on a study of complexity results for real-time systems. We have formulated several abstract problems that seem to be characteristic of real-time computing, and have obtained upper and lower bounds for the time complexity of those problems. The problems we have defined are all variations of problems that have previously been studied for asynchronous concurrent systems; the major differences are that we impose rather stringent timing requirements on the solutions and that we assume that our systems satisfy certain assumptions about the timing of events. The assumptions we make about the timing of events are not exact; rather, we assume that the time required for various events is known to be within certain bounds. For example, in real-time systems, there can be uncertainty in the time required for real-world tasks to be completed, for processors to take steps, for clocks to advance and for messages to be delivered.

We have obtained bounds on the time complexity of solving three fundamental problems. One is a model of a finite automaton, another is a timed automaton, and the third is a model of [47]. A timed automaton is a model of a system with concurrency and timing constraints; the automaton is timed because the system has at least one clock. A timed automaton is a timed version of the automaton used in the basic model of [47].

In the timed automaton model, we consider two kinds of clocks: clock variables and symbolic clocks. Clock variables are integers; symbolic clocks are real numbers. A timed automaton is a finite automaton with bounded clock variables and symbolic clocks. The clock variables are used to model the real-time aspect of the system; the symbolic clocks are used to model the concurrent aspect of the system. The timed automaton is a model of a real-time system with concurrency.

The timed automaton model is a generalization of the automaton model. The automaton model is a special case of the timed automaton model, in which the clock variables are integers and the symbolic clocks are real numbers. The timed automaton model is a model of a real-time system with concurrency and timing constraints.

The timed automaton model is a generalization of the automaton model. The automaton model is a special case of the timed automaton model, in which the clock variables are integers and the symbolic clocks are real numbers. The timed automaton model is a model of a real-time system with concurrency and timing constraints.

\textsuperscript{1}Asynchronous systems are those in which processes work at completely independent rates and have no way of estimating time.
s, state ma-
, first-order
], and pro-
1 evaluating
oning about
new models

of such sys-
ample of the
s concurrent
s that show
ns; for tasks
mine the in-
hese results
l a stimulus
ilar results
; complexity
ems are the
66, 99] (see

y results for
blems that
obtained up-
blems. The
at have pre-
 the major
irements on
ertain as-
ake about the
time re-
sounds. For
the time re-
take steps,
ng three fun-
etly indepen-
damental problems: mutual exclusion, synchronization and agreement, in
a model with inexact timing assumptions. These results are described, respectively, in Sections 2, 3 and 4.

In the course of our work on these problems, we felt a need for a
systematic method for reasoning about the correctness and performance
of timing based systems. To satisfy this need, we have developed a new
assertional method of reasoning about timing based systems.

Assertional reasoning is a very useful technique for proving safety
properties of sequential and concurrent algorithms. This proof method
involves describing the algorithm of interest as a state machine, and
defining a predicate known as an assertion on the states of the machine.
One proves inductively that the assertion is true of all the states that are
reachable in a computation of the machine, i.e., that it is an invariant
of the machine. The assertion is defined so that it implies the safety
property to be proved. Assertional reasoning is a rigorous, simple and
general proof technique. Furthermore, the assertions usually provide
an intuitively appealing explanation of why the algorithm satisfies the
property. One kind of assertional reasoning uses a mapping to describe a
correspondence between the given algorithm and a higher-level algorithm
used as a specification of correctness. (See, for example, [57, 67, 71].)

We have developed an assertional technique based on mappings, for
proving correctness and timing properties of timing-based systems. This
work is described in Section 5.

The formal model we have used to describe our results is the timed
automaton model, a slight variant of the time constrained automaton
model of [78]. We have used this model to state the requirements to
be satisfied, to define the basic architectural and timing assumptions, to
describe the algorithms, and to prove their correctness and timing prop-
erties. We have also used it for describing our mapping proof technique.

The last work described in this survey considers the time complexity
of wait-free algorithms in shared-memory distributed systems. In such
systems, it is possible for processes to operate at very different speeds,
e.g., because of implementation issues such as communication and mem-
ory latency, priority-based time-sharing of processors, cache misses and
page faults. It is also possible for processes to fail entirely. A wait-free
algorithm guarantees that each nonfaulty process terminates regardless
of the speed and failure of other processes ([44, 58]). Because wait-free algorithms guarantee that fast processes terminate without waiting for slow processes, wait-free algorithms seem to be generally thought of as fast. However, while it is obvious from the definition that wait-free algorithms are highly resilient to failures, we believe that the assumption that such algorithms are fast requires more careful examination.

We have addressed this general problem by studying the time complexity of wait-free algorithms for the approximate agreement problem. Our results are described in Section 6.

A major emphasis in our project has been on the impact of uncertainty in the system on the time complexity of solving problems. This uncertainty might be due to inexact timing assumptions as described above, to unpredictable inputs, or to failures. More specifically, our work on mutual exclusion involves timing uncertainty and unpredictable inputs, our work on synchronization involves only timing uncertainty, and our work on agreement and wait-free algorithms involves timing uncertainty and failures.

The portion of our work that deals with resource allocation problems is related to prior work in scheduling theory (for example, [40, 65, 86]). Our work on resource allocation is distinguished from the scheduling theory work in its emphasis on distributed algorithms and on timing uncertainties and failures. Also, our emphasis has been on upper and lower bounds for solving particular problems, e.g., mutual exclusion, whereas the emphasis in scheduling theory seems to be on general strategies (e.g., first-come-first-serve) for solving large classes of problems. As the subject matter of scheduling theory broadens, however, we believe that the two areas will become more closely related.

Our work has taken advantage of many of the approaches, techniques and results of distributed computing theory. In particular, our selection of problems, our use of automaton-style formal models and assertional reasoning, the design of our algorithms, and the techniques we use to prove lower bounds (e.g., the perturbation of executions and the use of the limitations of local knowledge) have all been heavily influenced by prior work in distributed computing theory.

2 Mutual Exclusion

We have studied mutual exclusion as an abstract problem, and note that in the push-difference model, the same rod moving at the same reaction rate is equivalent to a reaction

More precisely, a system is expected to be safe if, when the system is

The concept of a...
2 Mutual Exclusion

We have studied 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. [48]) 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 have considered 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 control system. The control 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 control 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 control 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 control system to make accurate estimates of the elapsed time since the last GRANT signal; the difficulty, however, is that the control 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 control system has is the following:

1. the knowledge that a moving part will stop moving within time \( m \)
after a \textit{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 \(0 < c_1 \leq c_2\),

3. the knowledge that the time between successive steps of any process within the control system is always in the interval \([0, l]\), for a known constant \(l, 0 \leq l\), 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 \(d, 0 \leq d\).

In the cases we have in mind, we suppose that \(l \ll c_1 < c_2 \ll d \ll m\). We use the notation \(C\) to represent the ratio \(c_2/c_1\); \(C\) is a useful measure of the timing uncertainty.

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

\[ n \cdot c_2 \left(\left\lfloor \frac{m + l}{c_1} \right\rfloor + 1 \right) + l \]

(approximately \(nmC\)) is an \textit{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 "remining" 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

\[ n \left[ c_2 \left(\left\lfloor \frac{m + l}{c_1} \right\rfloor + 1 \right) + d + c_2 + 2l \right] \]

(approximately \(nmC + nd\)) and a lower bound of \(nmC + (n - 1)d\). Assuming that the parameters have the relative sizes described earlier, e.g., that \(d\) is much larger than \(l\), \(c_1\) and \(c_2\), the gap between these two bounds is just slightly more than a single message delay time. The upper bound arises from:

3 Synchro

Some problems in distributed computer systems, especially those occurring in operating systems, have been shown to arise from the synchrony constraint. In this section, we consider the case of a system in which there is a local clock

The following session

Roughly speaking, a session is one step in a distributed computation. A session is a sequence of steps in a session is synchronous.

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

\[ n \left[ c_2 \left(\left\lfloor \frac{m + l}{c_1} \right\rfloor + 1 \right) + d + c_2 + 2l \right] \]

(approximately \(nmC + nd\)) and a lower bound of \(nmC + (n - 1)d\). Assuming that the parameters have the relative sizes described earlier, e.g., that \(d\) is much larger than \(l\), \(c_1\) and \(c_2\), the gap between these two bounds is just slightly more than a single message delay time. The upper bound arises from:

2 Synchro
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 work described in this section appears in [5].)

3 Synchronization

Some problems in real-time computing involve synchronization of several computer system components, in order that they might cooperate in performing a task involving real-world components. For example, multiple robots might cooperate to build a car on an assembly line, with each robot responsible for assembling a small piece of the machinery. Similar synchronization problems arise in distributed computing as well, where the separate components might each be responsible for performing a small part of a computation. We have studied the time complexity of achieving synchronization, in the presence of various assumptions about the timing of basic events.

The particular synchronization problem we have considered is the session problem, first defined by Arjomandi, Fischer and Lynch ([2]). Roughly speaking, a session is a sequence of events that contains at least one step by each process. An algorithm for the s-session problem guarantees that each execution of the algorithm includes at least s separate sessions. It was assumed in [2] that processes communicate via shared variables, and the time complexity of the session problem was studied in synchronous and asynchronous models.\footnote{Synchronous systems are those in which processes operate in lock-step, taking steps simultaneously.} The results of [2] show that the problem can be solved much faster in the synchronous model than in the asynchronous model.

We have studied the time requirements for the session problem in distributed networks rather than shared memory systems. That is, we have considered a collection of n processes located at the nodes of an undirected communication graph G; communication is assumed to be by messages sent over the edges of G. We have considered both the asynchronous model and a model with inexact timing assumptions (which we call here the partially synchronous model).

\footnote{Synchronous systems are those in which processes operate in lock-step, taking steps simultaneously.}
Time is measured under the following assumptions on the system. We assume that the delivery time for every message is in the range \([0, d]\), where \(d\) is a known nonnegative constant. Instead of assuming explicit clocks (as in the previous section), we assume that the times between successive local steps are in the range \([0, c_2]\) for the asynchronous model, and in the range \([c_1, c_2]\) for the partially synchronous model, where \(c_1\) and \(c_2\) are constants, \(0 < c_1 \leq c_2\). As before, we define \(C = c_2/c_1\).

Our upper bound results are as follows. Our first algorithm relies on explicit communication to ensure that the needed steps have occurred, and does not rely on any timing information. In either the asynchronous model or the partially synchronous model, this algorithm has time complexity \((s - 1)diam(G)(d + c_2)\), where the diameter of an undirected graph \(G\), \(diam(G)\), is the maximum distance between any two nodes. Our second algorithm does not use any communication and relies only on timing information; this algorithm works only in the partially synchronous model. Its time complexity is \(c_2 + (s - 2)(\lceil C \rceil + 1)c_2\). These algorithms can be combined to yield a partially synchronous algorithm for the \(s\)-session problem whose time complexity is \(c_2 + (s - 2)\min\{\lceil C \rceil + 1)c_2, diam(G)(d + c_2)\}\).

Our lower bound results are as follows. For the asynchronous model, we prove a lower bound of \((s - 1)diam(G)d\) for the time complexity of any algorithm for the \(s\)-session problem; this almost matches our upper bound for that model. For the partially synchronous model, we prove two lower bounds. We first show a simple lower bound of \((s - 2)\lceil C \rceil c_2\) for the case where communication is not used. We then present our main result: a lower bound of \(c_2 + (s - 2)\min\{\lceil C \rceil c_2, diam(G)d\}\) for the time complexity of any partially synchronous algorithm for the \(s\)-session problem. For appropriate values of the various parameters, these results imply a time separation between partially synchronous and asynchronous networks.

The lower bounds presented in this paper use the same general approach as in [2]. However, since we assume processes communicate by sending messages while [2] assumes processes communicate via shared memory, the precise details differ substantially. The lower bound proof in [2] uses fan-in arguments, while our lower bounds are based on information propagation arguments using long delays of messages, combined with appropriate selection of processes and careful timing arguments.

Awerbuch et al. [2] give a way to translate an asynchronous algorithm into a synchronous algorithm, but this is not the case for partially synchronous algorithms. In the \(s\)-session problem, the \(C\) is constant throughout.

(The writer's note:)

4 Distributed Agreement

We have a well studied agreement problem of the nodes. Two processes have considered running in some round, each receives all information from other processes, or by stopping the clock, the model agreed by scientific sensors, or by reaching agreement.

The time is a key issue. As well studied problems, it proceeds in a well defined way. In round, each process sends and receives all information to the other processes, and by reaching agreement, they reach the agreement.

We have considered the problems of the synchronous model.
Awerbuch ([8]) has introduced the concept of a synchronizer as a way to translate algorithms designed for synchronous networks to asynchronous networks. Although the results of [8] may suggest that any synchronous network algorithm can be translated into an asynchronous algorithm with constant time overhead, our results imply that this is not the case: for some values of the parameters, any translation of a partially synchronous (and in particular, a synchronous) algorithm for the $s$-session problem to an asynchronous algorithm must incur a non-constant time overhead.

(The work described in this section appears in [7].)

## 4 Distributed Agreement

We have also considered the time complexity of the problem of reaching agreement among nodes in a distributed system, in the case where some of the nodes are faulty. In the version of the agreement problem we have considered, each of $n$ processes starts with an input value. Each process that does not fail must choose a decision value such that (i) no two processes decide differently, and (ii) if any process decides $v$ then $v$ was the input value of some process. We assume that processes fail only by stopping (without warning). This abstract problem can be used to model agreement on the value of a sensor reading obtained with multiple sensors, or agreement on a course of action such as whether an airplane landing should be completed or aborted.

The time complexity of the distributed agreement problem has been well studied in the synchronous model. In this model, computation proceeds in a sequence of numbered rounds of communication. In each round, each non-failed process sends out messages to all processes, receives all messages sent to it at that round, and carries out some local computation. (See, for example, [60, 82, 21, 32, 19, 59, 29, 41, 76, 28, 79, 13, 80, 10] for results involving time complexity in this model.) The most basic time bound results in these papers are matching upper and lower bounds of $f + 1$ on the number of rounds of communication required for reaching agreement in the presence of at most $f$ faults.

We have considered how these bounds are affected by using, instead of the synchronous model, one in which there are inexact timing as-
sumptions. In particular, we have used the partially synchronous model described in Section 3, assuming that $d$ is an upper bound on message delivery time, $c_1$ and $c_2$ are lower and upper bounds, respectively, on process step time, and $C = c_2/c_1$.\footnote{Results of [33, 20] imply that if any one of the bounds $c_1, c_2, d$ does not exist, then there is no agreement algorithm tolerant to even one fault.} We have assumed that processes fail only by stopping, so that failures can be detected by “timeouts”: if an expected message from some process is not received within a sufficiently long time, then that process is known to have failed.

Initially, we had hoped to be able to adapt known results about the synchronous model to obtain good bounds for the version with inexact timing. Indeed, an $(f + 1)$-round algorithm can be adapted in a straightforward way to yield an algorithm for the partially synchronous model that requires time at most $(f + 1)dC$ if there are $f$ faults. On the other hand, a simple transformation in the reverse direction, of any partially synchronous algorithm to a synchronous algorithm yields a lower bound of $(f + 1)d$. There is a significant gap between these two bounds, namely, a multiplicative factor equal to the timing uncertainty, $C$. We would like to obtain closer bounds on the time complexity of this problem; in particular, we would like to understand how this complexity depends on $C$.

Our main result is an agreement algorithm in which the uncertainty factor $C$ is only incurred once, i.e., for one “round” of communication, yielding a running time of approximately $2td + dC$ in the worst case. The term of $dC$ arises from the possible need to time out a failed process; if a process fails, then it stops sending messages, and within time approximately $dC$, every other process can determine that the failure has occurred. It seems surprising that the overhead for such a timeout need only occur once in any execution. The algorithm can be viewed as an asynchronous algorithm which uses a fault detection (e.g., timeout) mechanism. That is, the timing bounds $c_1, c_2, d$ are used only in the fault detection mechanism.

Our second result shows that any agreement algorithm must take time at least $(f - 1)d + dC$ in the worst case. The lower bound is unusual in that it combines, in a nontrivial way, three different lower bound techniques: a “chain argument” ([32, 19, 29, 76, 14, 28]), used previously to prove that $f + 1$ rounds are required in the synchronous

rounds model, our lower bound is valid even if faults are not immediate and a “timeouts” are not used. We described an algorithm whose running time is only about a factor of two tight, they may be trivially modified to allow just a single timeout of the message.

Some previous results on timing assumptions for determining whether faults occurred; only recently the bounds were given.\footnote{The work was initially presented in [76].} Consider the latency $C$ of the synchronous model and of ours. The bounds of ours were obtained within some distance from ours. (The work was initially presented in [76].)

5 A model of real-time systems

Assertional models have been developed for sequential and parallel algorithms. These can be used to verify properties of programs. The assumptions of real-time properties (upper and lower bounds) and timing assumptions for the existence of real-time systems are essential in proving systems.
rounds model; a "bivalence" argument ([33, 20]) used previously to prove that fault-tolerant agreement is impossible in an asynchronous system; and a "time stretching" argument developed to prove the lower bounds described in Section 2 ([5]). Although these bounds are not completely tight, they do demonstrate that the inherent dependence on $C$ involves just a single term $dC$; there is no need to multiply $C$ by larger multiples of the message delivery time.

Some prior work on distributed agreement in a model with inexact timing assumptions appears in [27]. The main emphasis in [27] was on determining the maximum fault tolerance possible for various fault models; only rough upper bounds on the time complexity of the algorithms were given, and no lower bounds on time were proved. Related work on the latency\(^4\) of reaching agreement when processors are not completely synchronous appears in [12] and [96], although the results are different from ours. These papers assume that process clocks are synchronized to within some fixed additive error. Unlike our results, these results are not stated in terms of absolute real time.

(The work described in this section appears in [4].)

\section{A new proof technique for timing-based systems}

Assertional reasoning has been used primarily to prove properties of sequential algorithms and synchronous and asynchronous concurrent algorithms. We have developed a way in which assertional reasoning can be used to prove timing properties for algorithms that have timing assumptions of the kind described in the previous sections. Also, the kinds of properties generally proved using assertional reasoning have been "ordinary" safety properties; our method can be used to prove timing properties (upper and lower bounds on time) for algorithms that have timing assumptions. Predictable performance is often a desirable characteristic of real-time systems [95]; assertional techniques could be very helpful in proving such performance properties.

Our method involves constructing a multivalued mapping from an

\footnote{The worst-case elapsed time measured on the clock of any correct process.}
automaton representing the given algorithm to another automaton representing the timing requirements. The key to our method is a way of representing a system with timing constraints as an automaton whose state includes predictive timing information. Timing assumptions and timing requirements for the system are both represented in this way, and the mappings we construct map from the "assumptions automaton" to the "requirements automaton".

The formal model used in our work is the timed automaton model, adapted from the time-constrained automaton model of [78]. A timed automaton is a pair \((A, b)\), consisting of an I/O automaton \(A\) ([71, 72]), together with a boundmap \(b\), which is a formal description of the timing assumptions for the components of the system. We have introduced the notion of a timing condition to state upper and lower bounds on the difference between the times at which certain events or states appear in an execution; the conditions imposed by a boundmap are timing conditions of a particular kind. An automaton and a set of timing conditions (in particular, a timed automaton) generate a set of timed executions and a corresponding set of timed behaviors.

While convenient for specifying timing assumptions and requirements, timed automata are not directly suited for carrying out assertional proofs about timing properties, because timing constraints are described by specially-defined timing conditions and are not part of the automaton itself. We have therefore introduced a way of incorporating timing conditions into an automaton definition. For a given timed automaton \(A\), and a set \(U\) of timing conditions, the automaton \(\text{time}(A, U)\) is defined to be an ordinary I/O automaton (not a timed automaton) whose state includes predictive information describing the first and last times at which various events can next occur; this information is designed to enforce the timing conditions in \(U\).

The timing requirements to be proved for an algorithm described as a timed automaton, \((A, b)\), are described as a set of timing conditions, \(U\), for \(A\). The requirements automaton is defined to be \(\text{time}(A, U)\). Thus, predictive information about the first and last times at which certain events of interest can next occur are built into the state of the requirements automaton.

The problem of showing that a given algorithm \((A, b)\) satisfies the timing requirements is then reduced to that of showing that any behavior of the automaton using invariant is valid and is multivalued.

We have described a system as a simplified timed automaton described in terms of upper and lower bounds on upper and lower bounds on time. We have done this to simplify the concept of a system. The system is modeled as a system with timing constraints, and it enforces timing constraints on the system. The system is modeled as a system with timing constraints on the system, and it enforces timing constraints on the system.

The mapping from the timed automaton to the requirements automaton is not one-to-one, but it is sufficiently close to make the requirement satisfied.

Technically, a timed automaton can only be used to describe the behavior of a system. Timing properties can only be described by lower bounds on the duration of time. If the timing property is regarded as a condition, then the event must occur in a specific amount of time. This event cannot occur in an infinite amount of time. In our work, we are able to prove that any behavior proved in our work satisfies the timing properties.
of the automaton \(time(A, b)\) is also a behavior of \(time(A, U)\). This is done using invariant assertion techniques; in particular, one demonstrates a multivalued mapping from \(time(A, b)\) to \(time(A, U)\).

We have applied our technique to three examples. The first example is a simplified version of the timing-dependent resource granting system described in Section 2. We have given careful proofs of upper and lower bounds on the amount of time prior to the first \(GRANT\) event and in between each successive pair of \(GRANT\) events. The second example is a system consisting of a "line" of processes, in which each process relays a signal received from the process at its left to the process at its right. We have given careful proofs of upper and lower bounds on the time to propagate a signal from the left end to the right end of the line. The third, more complicated example involves one process incrementing a counter until another process modifies a flag, and then decrementing this counter. When the counter reaches 0, a \(DONE\) action occurs. We have given careful proofs of upper and lower bounds on the time until a \(DONE\) occurs.

The mappings we provide for all three of these examples have a particularly interesting and simple form—a set of inequalities relating the time bounds to be proved to those that can be computed from the state. These inequalities contain information about how the bounds are to be satisfied.

Technically, mapping techniques of the sort used in this paper are only capable of proving safety properties, but not liveness properties. Timing properties have aspects of both safety and liveness. A timing lower bound asserts that an event cannot occur before a certain amount of time has elapsed; a violation of this property is detectable after a finite prefix of a timed execution, and so a timing lower bound can be regarded as a safety property. A timing upper bound asserts that an event must occur before a certain amount of time has elapsed. This can be regarded as making two separate claims: that the designated amount of time does in fact elapse (a liveness property), and that that amount of time cannot elapse without the event having occurred (a safety property). In our work, we have assumed the liveness property that time increases without bound, so that all the remaining properties that need to be proved in order to prove either upper or lower time bounds are safety properties. Thus, our mapping technique provides complete proofs for
timing properties without requiring any special techniques (e.g., variant functions or temporal logic methods) for arguing liveness.

We have shown that this method is complete: if every behavior of \((A, b)\) is also a behavior of \(\text{time}(A, \mathcal{U})\) then is there necessarily a strong possibilities mapping (in the form of inequalities) from \(\text{time}(A, b)\) to \(\text{time}(A, \mathcal{U})\). Related completeness results for the usage of refinement mappings to prove properties of non timing-based algorithms were proved in [1] and [77].

There has been some prior work on using assertional reasoning to prove timing properties. In particular, Hasse [43], Shankar and Lam [92], Tel [97], Schneider [90], Lewis [63] and Shaw [93] have all developed models for timing-based systems that incorporate time information into the state, and have used invariant assertions to prove timing properties. In [97] and [63], in fact, the information that is included is similar to ours in that it is also predictive timing information (but not exactly the same information as ours). None of this work has been based on mappings, however. Several other, quite different formal approaches to proving timing properties have also been developed. Some representative papers describing these other methods are [11], [52], [48], [45], [100], [49], and [35].

(The work described in this section appears in [68].)

6 Time complexity of asynchronous resilient algorithms

In shared-memory distributed systems, some number \(n\) of independent asynchronous processes communicate by reading and writing to shared memory. \textit{Wait-free} algorithms have been proposed as a mechanism for computing in the face of variable speeds and failures: a wait-free algorithm guarantees that each nonfaulty process terminates regardless of the speed and failure of other processes ([44, 58]). The design of wait-free shared-memory algorithms has recently been a very active area of research (see, e.g., [3, 22, 44, 58, 64, 83, 84, 89, 98]).

We have studied the \textit{time complexity} of wait-free and non-wait-free algorithms in “normal” executions, where no failures occur and processes operate at a particular speed, which is only relative and where both terminate even if they fail to terminate. A measure of time execution is needed to evaluate the time cost [2, 15, 69] of algorithms in an extra complexity.

We have studied the problem—[23, 30, 31] of incorporating shared variables into a distributed system. In this paper, we introduce several methods (provided in [36]). The output of the algorithm is included in the state of the algorithm and is closely related to the output in a distributed system.

Approximately 100 processes are followed, by using a standard approach; all other processes follow their output. Figure 1 shows that the complexity of the algorithm depends linearly on the input values. The complexity of the algorithm is given by:

\[ O(n) \]

Our first algorithm, whose time
operate at approximately the same speed. We have selected this particular subset of the executions for making the comparison, because it is only reasonable to compare the behavior of the algorithms in cases where both are required to terminate. Since wait-free algorithms terminate even when some processes fail, while non-wait-free algorithms may fail to terminate in this case, the comparison should only be made in executions in which no process fails, i.e., in failure-free executions. The time measure we have used is the one introduced in [54, 55], and used to evaluate the time complexity of asynchronous algorithms, in, e.g., [2, 15, 69, 70, 85]. To summarize, we are interested in measuring the time cost imposed by the wait-free property, as measured in terms of extra computation time in the most normal (failure-free) case.

We have addressed the general question by considering a specific problem—the approximate agreement problem studied, for example, in [23, 30, 31, 73]; we have studied this problem in the context of a particular shared-memory primitive—single-writer multi-reader atomic registers. In this problem, each process starts with a real-valued input, and (provided it does not fail) must eventually produce a real-valued output. The outputs must all be within a given distance $\varepsilon$ of each other, and must be included within the range of the inputs. This problem, a weaker variant of the well-studied problem of distributed consensus (e.g., [33, 60]), is closely related to the important problem of synchronizing local clocks in a distributed system.

Approximate agreement can be achieved very easily if waiting is allowed, by having a designated process write its input to the shared memory; all other processes wait for this value to be written and adopt it as their outputs. In terms of the time measure described above, it is easy to see that the time complexity of this algorithm is constant—indeed independent of $n$, the range of inputs and $\varepsilon$. On the other hand, there is a relatively simple wait-free algorithm for this problem which is based on successive averaging of intermediate values. The time complexity of this algorithm depends linearly on $n$, and logarithmically on the size of the range of input values and on $1/\varepsilon$. A natural question to ask is whether the time complexity of this algorithm is optimal for wait-free approximate agreement algorithms.

Our first major result is an algorithm for the special case where $n = 2$, whose time complexity is constant, i.e., it does not depend on the range
of inputs or on \( \varepsilon \). The algorithm uses a novel method of overcoming the uncertainty that is inherent in an asynchronous environment, without resorting to synchronization points (cf. [39]) or other waiting mechanisms (cf. [15]): this method involves ensuring that the two processes base their decisions on information that is approximately, but not exactly, the same.

Next, using a powerful technique of integrating wait-free (but slow) and non-wait-free (but fast) algorithms, together with an \( O(\log n) \) wait-free input collection function, we generalize the key ideas of the 2-process algorithm to obtain our second major result: a wait-free algorithm for approximate agreement whose time complexity is \( O(\log n) \). Thus, the time complexity of this algorithm does not depend on either the size of the range of input values or on \( \varepsilon \), but it still depends on \( n \), the number of processes.

At this point, it was natural to ask whether the logarithmic dependence on \( n \) is inherent for wait-free approximate agreement algorithms, or whether, on the other hand, there is a constant-time wait-free algorithm (independent of \( n \)). Our third major result shows that the \( \log n \) dependency is inherent: any wait-free algorithm for approximate agreement has time complexity at least \( \log n \).\(^5\) This implies an \( \Omega(\log n) \) time separation between the non-wait-free and wait-free computation models.

We note that the constant time 2-process algorithm behaves rather badly if one of the processes fails. The work performed in an execution of an algorithm is the total number of atomic operations performed in that execution by all processes before they decide. We have proved a tradeoff between the time complexity of and the work performed by any wait-free approximate agreement algorithm. We have shown that for any wait-free approximate agreement algorithm for 2 processes, there exists an execution in which the work exhibits a nontrivial dependency on \( \varepsilon \) and the range of inputs.

In practice, the design of distributed systems is often geared towards optimizing the time complexity in “normal executions,” i.e., executions where no failures occur and processes run at approximately the same pace, while building in safety provisions to protect against failures (cf. [61]). Our results indicate that, in the asynchronous shared-memory

\(^5\)The lower bound is attained in an execution where processes run synchronously and no process fails.

7 Further Research

Our future work will focus on the following problems, which are the ones we are currently investigating:

- More extensive analysis of existing algorithms.
- Experiments with existing systems.
- A problem on the optimistic side of the spectrum.
- Problem solving and algorithm development.
- Adding features to the system.
- The design and implementation of a large-scale distributed system.
setting, there are problems for which building in such safety provisions must result in performance degradation in the normal executions. This situation contrasts with that occurring, for example, in synchronous systems that solve the distributed consensus problem. In that setting, there are early-stopping algorithms (e.g., [25, 28, 79]) that tolerate failures, yet still terminate in constant time when no failures occur. The exact cost imposed by fault-tolerance on normal executions, was studied, for example, in [14, 28, 79]. It has been shown, for synchronous message passing systems, that non-blocking commit protocols take twice as much time, in failure-free executions, as blocking protocols ([29]).

Recent work has addressed the issue of adapting the usual synchronous shared-memory PRAM model to better reflect implementation issues, by reducing synchrony ([15, 16, 39, 81, 74]) or by requiring fault-tolerance ([50, 51]). To the best of our knowledge, the impact of the combination of asynchrony and fault-tolerance (as exemplified by the wait-free model) on the time complexity of shared-memory algorithms has not previously been studied.

(The work described in this section appears in [6].)

7 Further research

Our future research plans include more work on upper and lower bounds for problems of interest in real-time computing. Some specific problems we are currently working on are as follows.

- More complex resource allocation problems than simple mutual exclusion.
- A problem of probabilistic processor fault diagnosis.
- Probabilistic versions of agreement problems.
- Extensions of the consensus problem described in Section 4 to the case where more severe faults than simple stopping faults can occur.
- The problem of determining the power of a bounded capacity link.

overcoming the fact, without mechanisms base their e (but slow) (log n) wait- the 2-process algorithm for . Thus, the or the size of the number

hmic depen- algorithms, ait-free algo-
mat the log n imate agree-
) time tion models.
have rather an execution performed in have proved a
formed by any that for any , there exists tendency on ε
ared towards ., executions rely the same ; failures (cf. shared-memory synchronously
We also plan to continue our work on modeling and verification. Specifically, this includes the following.

- An attempt to relate recent work on process algebraic approaches to reasoning about real-time systems to state-machine models such as [78].
- An attempt to develop timing analysis techniques for randomized algorithms.
- An effort to apply the mapping method to verify substantial algorithms of interest in the fields of communication and/or real-time processing. A particular example we are working on is a new link-state packet distribution algorithm proposed as a standard by DEC.
- An effort to simplify our mapping method described in Section 5.

### 7.0.1 Acknowledgements:

We owe great thanks to the other researchers who worked with us on these papers: Cynthia Dwork, Marios Mavronicolas, Nir Shavit and Larry Stockmeyer. We also thank Nancy Leveson, Michael Merritt, Stephen Ponzo, Mark Tuttle and Jennifer Welch for many helpful discussions and for their reading of drafts of our papers.

### References


than agree-
ributed Soft-

; asynchrony

parallel Algo-


synchrony,”

architectures,

irements for
EE Transac-
ember 1983),

ems: Con-
hem,” IEEE
o. 1 (January

aphic proto-
Computing,

mal synchro-
e ACM, Vol.

. R. Strong,
,” Informa-

omic era: l-
he Theory of

l, “Reaching
ournal of the


al method
Proc. 10th
er 1989.
ification of
re Systems
pecification
ort 89-14,
or extended
ager-Verlag
eties in real-
Real-Time
iming mod-
d Architec-
sheduling,”
9, pp. 285–
uputations,
cial Report
University.
precision in
col. 1 (1985),
E Transac-
ember 1981),
t objects,”
Computing,


HMS Model of Time

An overview of the HMS model and its formal representation is presented. The HMS model is designed to be used in the formal verification of protocols and systems. The model is based on a formal framework for describing and verifying time-dependent systems. The framework provides a way to formally define and reason about the behavior of real-time systems. The HMS model is particularly useful for verifying the correctness of real-time protocols and systems.

**INTRODUCTION**

Various techniques have been developed for the formal verification of real-time systems. These techniques include time Petri nets, timed automata, and real-time temporal logic. These techniques allow for the verification of the correctness of real-time protocols and systems. The HMS model provides a way to formalize the behavior of real-time systems. The model is based on a framework for describing and verifying time-dependent systems. The framework provides a way to formally define and reason about the behavior of real-time systems. The HMS model is particularly useful for verifying the correctness of real-time protocols and systems.