Designing Algorithms for Distributed Systems with Partially Synchronized Clocks*

Soma Chaudhuri Rainer Gawlick Nancy Lynch
Massachusetts Institute of Technology

Abstract

Much of modern systems programming involves designing algorithms for distributed systems in which the nodes have access to information about time. Time information can be used to estimate the time at which system or environment events occur, to detect process failures, to schedule the use of resources, and to synchronize activities of different system components. In this paper we propose a simple programming model that is based on the timed automaton model of Lynch and Vaandrager [9], which gives algorithms direct access to perfectly accurate time information.

Unfortunately, this programming model is not realistic. In a realistic distributed system, clocks have skew and a finite granularity. Furthermore, other details neglected by the timed automaton model such as processor step times must also be considered. We provide two simulations that show how to transform an algorithm designed in the simple programming model to run in a more realistic distributed system. One of our simulations is an extension of previous results on the use of inaccurate clocks by Lamport [5], Neiger and Toueg [13], and Welch [17]. Our extensions suggest several powerful design techniques for algorithms that are to be run in distributed systems with clocks whose divergence from real time is bounded. We demonstrate these techniques by providing a new algorithm for distributed linearizable read-write objects. This algorithm significantly improves over previous results [10] in terms of time complexity and algorithmic simplicity.

1 Introduction

In most real distributed systems, the timing information that is available is imprecise. Individual nodes in a distributed system are usually provided with clocks, which provide estimates of the real time. These clocks often have a small *skew* ϵ, which bounds the amount by which their values may differ from real time. Clocks with a small skew are now achievable by means of time services such as NTP [12] and the Digital Time Service [3]. For example, [12] states that its protocol is "capable of accuracies in the order of a millisecond, even after extended periods when synchronization to primary reference sources has been lost."

Programming with imprecise clocks can be quite difficult. Additional complexity is introduced by a host of mundane but significant issues such as the granularity of the clocks and the processor speeds. For example, a processor might only be able to obtain new clock estimates every so often, which means that it might "miss seeing" a particular clock value, and might not be capable of performing a particular action when its local clock reaches exactly a particular value.

This paper provides a very simple programming model for the design and verification of realistic timing based distributed algorithms. By "realistic", we mean that the algorithms can run in systems that suffer from all of the complications we have discussed above.

The programming model we propose is based on the *timed automaton* model of [4, 8, 9, 16]. We have chosen this model because it is very general and simple. Furthermore, it has a powerful set of proof techniques for reasoning about timing-based distributed systems. In fact, a substantial practical verification project [6] which uses some of these proof methods is being carried out. The timed automaton model presumes direct access to *real time*. Furthermore, a timed automaton is able to schedule actions at exactly a predetermined real time.

Our model of a more "realistic system" uses a simple special case of the model of [7, 11] which we call the MMT model. The MMT model is an extension of the I/O automaton model [15] that includes time bound restrictions on the tasks performed by the automaton. The knowledge that each node has of the time is derived from a separate clock subsystem, which occasionally informs the node of its time. We suppose that the clock subsystem always provides a node with a clock value that is within ϵ of real time. However, the clock may change in discrete jumps, so that any particular time value might be missed. While the MMT model accounts for many of the complications present in real systems, it still contains some unrealistic assumptions that we hope to eliminate in future work.

The main result of this paper is a transformation from algorithms written in the simple abstract model into algorithms in the realistic model, more precisely, from algorithms written in the timed automata model to algorithms written in the MMT model. The transformation is accomplished with two simulations. Both simulations preserve the real-time behavior of the system, to within a small amount.

*Supported by NSF grant CCR-89-15206, by DARPA contracts N00014-89-J-1988 and N00014-92-J-4033, and by ONR contract N00014-91-J-1046.

12th ACM Symposium on Principles on Distributed Computing, Ithaca NY
© 1993 ACM 0-89791-613-1/93/0008/0121....$1.50

The purpose of the transformation is to permit algorithms to be designed and verified in the simple abstract model. The algorithms can then be transformed using our transformation to run correctly in the realistic model. This frees the algorithm designer from worrying about the complications of the realistic model.

The first simulation transforms an arbitrary timed automaton conforming to our network interface to a special case of a timed automaton called a *clock automaton*. A clock automaton is a timed automaton with a special clock component, where the automaton's behavior depends only on the clock, not on real time. The transformation is essentially the same as the one used by Neiger and Toueg [13] and by Welch [17], reformulated in terms of timed automata. This transformation preserves not only logical correctness properties ("internal specifications", in the terminology of [13]), but also real-time behavior. In this way, it extends the result of Neiger and Toueg. This extension leads to several new design techniques that apply to specifications with real-time behavior as well as internal specifications.

The second simulation transforms any clock automaton (with some simple restrictions) to a corresponding MMT automaton. Recall that the MMT model deals with the messy details of clock granularity and processor step time. The key observation of the second simulation is that the MMT automaton might miss seeing particular clock values. Thus, it will carry out a delayed simulation, continually "catching up" whenever it has an opportunity to take a step.

We also provide an application of the powerful design techniques provided by our results. In [10], Mavronicolas presents two algorithms for implementing linearizable read-write objects in a network, one for our simple abstract programming model and a second complicated algorithm for a realistic distributed system model that corresponds closely to our clock automaton model. We use the algorithm suggested by [10] for the simple abstract programming model and apply our first simulation result to obtain an algorithm for linearizable read-write objects in the clock automaton distributed system model. The algorithm that results from the transformation is simpler and has better complexity than the second algorithm of [10]. In addition to applications, we discuss some practical implementation issues surrounding our results.

Our model at present only considers safety conditions and not liveness conditions. In addition, we do not consider failures. However, it appears that the results will extend to cases involving faulty nodes and also faulty message channels. See [17] for an indication of how the first simulation applies to faulty processes.

The timed automaton and clock automaton models are introduced in Section 2. Section 3 introduces the simple programming model. The transformation to the clock automaton model is the focus of Section 4. Section 5 presents the transformation to the MMT automaton model. In Section 6 we demonstrate the new design techniques made available by our results with a new algorithm for distributed linearizable read-write objects. Finally, Section 7 offers some discussion of the results.

2 Model and Definitions

2.1 Timed Automaton Model

Timed automata are based on the timed automaton model of Lynch and Vaandrager [9], an extension of that model in [4], and the I/O automaton model of Lynch and Tuttle [15]. A timed automaton is essentially a (finite or infinite) state machine. The transitions of this state machine are labeled using action names. The action names are the means by which a timed automaton communicates with its environment. The actions of a timed automaton are divided into four disjoint sets, *input actions*, *output actions*, *internal actions* and a *time-passage action*. Input actions are controlled by the environment. To model this fact, each state must have a state transition for each input action[1]. Output actions are controlled by the timed automaton. The time-passage action, ν, signals the passage of time. Internal actions are also controlled by the timed automaton. But, Internal actions are not observable by the environment.

Definition 2.1 (timed automata) A *timed automaton*[2] A consists of four components:

- a set $states(A)$ of states. Each state s of A has a state component $s.now$ which represents the real time in state s. The domain for the *now* component is the non-negative reals, $\Re^+$. We denote by $s.tbasic$ all state components of s except the *now* component.
- a nonempty set $start(A) \subseteq states(A)$ of start states.
- an action signature $sig(A) = (in(A), out(A), int(A))$ where $in(A)$, $out(A)$, $int(A)$ are disjoint sets of input, output and internal actions, respectively. We denote by $vis(A)$ the set $in(A) \cup out(A)$ of visible actions. We denote by $ext(A)$ the set $vis(A) \cup \{\nu\}$ of external actions where ν is the special *time-passage* action. We denote by $acts(A)$ the set $ext(A) \cup int(A)$ of all actions. Finally we denote by $uacts(A)$ the set $vis(A) \cup int(A)$ of non time-passage actions.
- a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$.

A must satisfy the following five axioms:

S1 If $s \in start(A)$ then $s.now = 0$.

S2 If $(s, a, s') \in trans(A)$ and $a \neq \nu$, then $s'.now = s.now$.

S3 If $(s, \nu, s') \in trans(A)$ then $s'.now > s.now$.

S4 If $(s, \nu, s') \in trans(A)$ and $(s', \nu, s'') \in trans(A)$, then $(s, \nu, s'') \in trans(A)$.

S5 if $(s, \nu, s'') \in trans(A)$ where $s''.now = s.now + \Delta_t$ then for any Δ_t' such that $0 < \Delta_t' < \Delta_t$ there exists state s' such that $s'.now = s.now + \Delta_t'$, $(s, \nu, s') \in trans(A)$, and $(s', \nu, s'') \in trans(A)$. ∎

In any state s of a timed automaton A an action a is said to be *enabled* if there exists a state s' such that $(s, a, s') \in trans(A)$. An *execution* of A is a (finite or infinite) sequence of alternating states and actions starting with a start state and, if the execution is finite, ending in a state. For example

[1] Since we distinguish between input and output actions and require that each state have a state transition for each input action, it would be notationally more correct to call our model the timed I/O automaton model rather than the timed automaton model. We simplify the presentation by dropping the I/O prefix from the model name.

[2] There are two small inconsistencies between our definition of timed automata and the definition of timed I/O automata given in [4]. We define the *now* as a state component while [4] defines it as a function from $states(A)$ to $\Re^+$. Also, [4] has a slightly stronger version of **S5**. Our version of **S5** can be strengthened to the version in **S5** by choosing *now* from the non-negative rationals [9].

$\alpha = s_0 a_0 s_1 a_1 s_2 \cdots$ where each $(s_{i-1}, a_i, s_i) \in trans(A)$ and $s_0 \in start(A)$. We denote by $execs(A)$ the set of executions of A.

We now introduce the notion of a *timed sequence* over a set B of non time-passage actions. A timed sequence is a sequence of *action-time pairs*, (a, t), where $a \in B$ and $t \in \Re^+$. If the pair (a, t) precedes the pair (a', t') in a timed sequence then $t \leq t'$. The *timed schedule* of an execution α of timed automaton A, denoted by $t\text{-}sched(\alpha)$, is the timed sequence constructed as follows. Suppose that $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$. Define $t_i = s_{i-1}.now$. Then $t\text{-}sched(\alpha) = ((a_1, t_1)(a_2, t_2)\cdots)|(uacts(A) \times \Re^+)$. Denote the timed schedules of A by $t\text{-}scheds(A)$. We extend the definition of a timed schedule to timed traces. Specifically, $t\text{-}trace(\alpha) = t\text{-}sched(\alpha)|(vis(A) \times \Re^+)$. Denote the timed schedules of A by $t\text{-}scheds(A)$ and the timed traces of A by $t\text{-}traces(A)$.

For any execution α, let $\alpha.ltime$ be the smallest number larger than or equal to $s.now$ for all states s in α. An execution for which $ltime = \infty$ is said to be *admissible*. If β is a timed schedule or a timed trace derived from an admissible execution α, then β is admissible. Denote the admissible executions of timed automaton A by $execs^\infty(A)$. Similarly, denote the admissible timed schedules and admissible timed traces of timed automaton A by $t\text{-}scheds^\infty(A)$ and $t\text{-}traces^\infty(A)$ respectively. A timed automaton is said to be *feasible* if every finite execution can be extended to an admissible execution.

We introduce the following notation. Let α be an execution of timed automaton A. Then $a_i(\alpha)$ is the i^{th} action in α, $s_i(\alpha)$ is the i^{th} state of α not counting the initial state, and $t_i(\alpha)$ is the value of the *now* component in state $s_{i-1}(\alpha)$. The initial state is denoted by $s_0(\alpha)$. Let α be a timed sequence. Then $a_i(\alpha)$ is the action of the i^{th} action-time pair in α and $t_i(\alpha)$ is the time of the i^{th} action-time pair in α. Furthermore, if α is an execution or timed sequence then i is the *index* of action $a_i(\alpha)$ in α. If α is an execution then $\Omega(\alpha)$ is the set of indices of all non time-passage actions in α. Similarly, if α is a timed trace then $\Omega(\alpha)$ is the set of indices of all actions in α.

Next we consider the composition, hiding, and renaming operators. The composition of timed automata provides a means by which complex timed automata can be constructed from simpler ones. The constituent timed automata of a composition communicate on their shared actions. Let A_i for all $i \in I$ be a finite set of timed automata. We say that these timed automata are *compatible* iff for all $i, j \in I$, $out(A_i) \cap out(A_j) = \emptyset$ and $int(A_i) \cap acts(A_j) = \emptyset$.

Definition 2.2 (timed automata composition) Define $tbasic(A_i) = \{s.tbasic \mid s \in states(A_i)\}$. Let $\{A_i \mid i \in I\}$ be a finite set of compatible timed automata where $I = \{1, \ldots, n\}$. The *composition* $A = \prod_{i \in I} A_i$ is the timed automaton defined as follows[3]:

- $states(A) = \prod_{i \in I} tbasic(A_i) \times \Re^+$. If $s \in states(A)$, i.e. $s = (s_1, \ldots, s_n, now)$, then for all $i \in I$, $s|A_i = (s_i, now)$.
- $start(A) \subseteq states(A)$ such that for all $i \in I$, $s|A_i \in start(A_i)$.
- $sig(A) = (in(A), out(A), int(A))$ where $in(A) = \bigcup_{i \in I} in(A_i) - \bigcup_{i \in I} out(A_i)$, $out(A) = \bigcup_{i \in I} out(A_i)$, and $int(A) = \bigcup_{i \in I} int(A_i)$.
- $trans(A)$ is the set of triples (s, a, s') such that $s|A_i = s'|A_i$ when $a \notin acts(A_i)$ and $(s|A_i, a, s'|A_i) \in trans(A_i)$ when $a \in acts(A_i)$. ■

The projection operator $|$ extends to executions of compositions. Let $A = \prod_{i \in I} A_i$, where $\{A_i \mid i \in I\}$ is a finite set of compatible timed automata. If α is an execution of A then $\alpha|A_i$ is the sequence that results from deleting all $a_j(\alpha), s_j(\alpha)$ when $a_j(\alpha)$ is not in $acts(A_i)$, and replacing all remaining states $s_j(\alpha)$ by $s_j(\alpha)|A_i$. The projection operator extends naturally to schedules, traces, and timed sequences. The following lemmas are analogs of lemmas in [15].

Lemma 2.1 *Let $A = \prod_{i \in I} A_i$, where $\{A_i \mid i \in I\}$ is a finite set of compatible timed automata. If $\alpha \in execs^\infty(A)$ then $\alpha|A_i \in execs^\infty(A_i)$ for all A_i.*

Lemma 2.2 *Let $A = \prod_{i \in I} A_i$, where $\{A_i \mid i \in I\}$ is a finite set of compatible timed automata. Define a sequence of action-time pairs, $\alpha = (a_1, t_1)(a_2, t_2)\cdots$, such that $a_i \in acts(A)$ and $t_i \in \Re^+$ for all i. Then $\alpha|A_i \in t\text{-}scheds^\infty(A_i)$ for all A_i iff $\alpha \in t\text{-}scheds^\infty(A)$.*

Two additional operators that are used for timed automata are *hiding* and *renaming*. The hiding operator is used to reclassify output actions to be internal actions so that they are no longer visible to the environment. The renaming operator renames some subset of the actions of a timed automaton. See [4, 16] for a formal discussion of the hiding and renaming operators for timed automata.

2.2 Clock Automaton Model

As a special case of a timed automaton we define a *clock automaton*. A clock automaton is a timed automaton with a state component, *clock*, which represents a clock time.

Definition 2.3 (clock automata) A *clock automaton* A is a timed automaton where each state contains a state component, $s.clock$, which indicates the current clock time in state s. Denote by $s.cbasic$ all the state components of s except the *now* and *clock* components. A must satisfy the following four additional axioms:

C1 If $s \in start(A)$ then $s.clock = 0$.

C2 If $(s, a, s') \in trans(A)$ and $a \neq \nu$ then $s'.clock = s.clock$.

C3 If $(s, \nu, s') \in trans(A)$ then $s'.clock > s.clock$.

C4 If $(s, \nu, s'') \in trans(A)$ where $s''.now = s.now + \Delta_t$ and $s''.clock = s.clock + \Delta_c$ then for any Δ_t' such that $0 < \Delta_t' < \Delta_t$ and any Δ_c' such that $0 < \Delta_c' < \Delta_c$ there exists state s' such that $s'.now = s.now + \Delta_t'$, $s'.clock = s.clock + \Delta_c'$, $(s, \nu, s') \in trans(A)$, and $(s', \nu, s'') \in trans(A)$. ■

For some clock automata we wish to specify the behavior of the clock with respect to real time. This is done be restricting the set of *reachable* states of the clock automaton. A state is reachable if it is the final state of some finite execution. To formalize this notion, we introduce the concept of a *clock predicate*.

Definition 2.4 (clock predicate) A *clock predicate*, C, is a binary relation from $\Re^+$ to $\Re^+$. A state s of any clock automaton *satisfies* the predicate C iff $(s.now, s.clock) \in C$. We say that a particular clock automaton A *satisfies* C iff for every reachable state $s \in states(A)$, it is the case that s satisfies C. ■

[3] The $\prod$ symbol used to define $states(A)$ represents the normal Cartesian product.

In the following definition we consider a specific type of clock predicate that bounds the difference between the real time and the clock time.

Definition 2.5 (C_ϵ) C_ϵ is the clock predicate such that $(x, y) \in C_\epsilon$ iff $|x - y| \leq \epsilon$. ∎

To formalize the concept that clock automata are not supposed to use the *now* component in the transition decision, we introduce the ϵ-time independence property.

Definition 2.6 (ϵ-**time independent**) Let A be a clock automaton. We say that A is *ϵ-time independent* iff, for every $(s, a, s') \in trans(A)$ and all states $u, u' \in states(A)$ such that $u.clock = s.clock$, $u.cbasic = s.cbasic$, u satisfies clock predicate C_ϵ, $u'.clock = s'.clock$, $u'.cbasic = s'.cbasic$, and u' satisfies clock predicate C_ϵ, the following holds: $(u, a, u') \in trans(A)$ when $a \neq \nu$ and $u.now = u'.now$ or when $a = \nu$ and $u.now < u'.now$ ∎

Finally we define a composition operator that applies only to clock automata. The difference between this operator and the composition operator for timed automata is that the *clock* component as well as the *now* component is a global entity in the composed clock automaton.

Definition 2.7 (clock automaton composition) Define $cbasic(A_i) = \{s.cbasic \mid s \in states(A_i)\}$. Let $\{A_i \mid i \in I\}$ be a finite set of compatible clock automata where $I = \{1, \ldots, n\}$. The *composition* $A = \prod_{i \in I} A_i$ is the following clock automaton. The $states(A) = \prod_{i \in I} cbasic(A_i) \times \Re^+ \times \Re^+$. If $s \in states(A)$, i.e. $s = (s_1, \ldots, s_n, clock, now)$, then for all $i \in I$, $s|A_i = (s_i, clock, now)$. The other components, $start(A)$, $sig(A)$, and $trans(A)$, are as in Definition 2.2. ∎

Lemma 2.3 *Let $A = \prod_{i \in I} A_i$, where $\{A_i \mid i \in I\}$ is a finite set of compatible clock automata. If $\alpha \in execs^\infty(A)$ then $\alpha|A_i \in execs^\infty(A_i)$ for all A_i.*

2.3 Relations on Traces

Consider the following equivalence relations on timed sequences. The first equivalence relation preserves the order between specified sets of actions. It also preserves the real time of each action to within some constant.

Definition 2.8 ($=_{\epsilon,K}$) Let α_1 and α_2 be timed sequences, $\epsilon \in \Re^+$, and K a set of disjoint sets of actions. Then $\alpha_1 =_{\epsilon,K} \alpha_2$ iff there exists a bijective function f from $\Omega(\alpha_1)$ to $\Omega(\alpha_2)$ such that for all i, j:

- $a_{f(i)}(\alpha_2) = a_i(\alpha_1)$, and
- if $k \in$ K and $a_i(\alpha_1), a_j(\alpha_1) \in k$ then $i < j$ iff $f(i) < f(j)$, and
- $|t_{f(i)}(\alpha_2) - t_i(\alpha_1)| \leq \epsilon$. ∎

The next relation, $\leq_{\delta,K}$, just shifts the occurrence of some actions by δ in real time. Consider a set of actions $k \in$ K. The occurrence of these actions may be shifted up to δ time units into the future. The ordering of the actions in k may change with respect to actions not in k but not with respect to each other.

Definition 2.9 ($\leq_{\delta,K}$) Let α_1 and α_2 be timed sequences, $\delta \in \Re^+$, and K a set of disjoint sets of actions. Then $\alpha_1 \leq_\delta \alpha_2$ iff there exists a bijective function f from $\Omega(\alpha_1)$ to $\Omega(\alpha_2)$ such that for all i, j:

- $a_{f(i)}(\alpha_2) = a_i(\alpha_1)$,
- if there exists no $k \in$ K such that $a_i(\alpha_1) \in k$, and $a_j(\alpha_1) \notin k$ then $i < j$ iff $f(i) < f(j)$,
- if there exists no $k \in$ K such that $a_i(\alpha_1) \in k$ then $t_i(\alpha_1) = t_{f(i)}(\alpha_2)$,
- if $k \in$ K and $a_i(\alpha_1) \in k$ then $t_i(\alpha_1) \leq t_{f(i)}(\alpha_2) \leq t_i(\alpha_1) + \delta$. ∎

2.4 Algorithms and Problems in Distributed Systems

A distributed system is characterized by a set of processes or nodes that are interconnected with a communication network consisting of unidirectional links. The topology of the distributed system is represented by a graph (V, E) where the set of nodes in the distributed system is given by $V = \{v_1 \ldots v_n\}$ and a set of edges connecting the nodes in V is given by E where $e_{i,j} \in E$ if there is a link from node v_i to node v_j. Let $N = \{1 \ldots n\}$. The nodes share no memory and communicate only using messages passed over the links represented by the edges in E. The message system can have various delay characteristics. A message system where the real time delay of any message is between d_1 and d_2 will have its delay characterized using the notation $[d_1, d_2]$. We assume the distributed system to be reliable, in other words, messages are neither duplicated nor lost. However, messages may be reordered[4].

In each of the models we represent each of the nodes and each of the edges of the distributed system by an automaton. The algorithm run by the distributed system is encoded in the automata that model the nodes. The composition of the automata modeling the nodes and the edges is the automaton that models the entire distributed system.

A problem P defined on the graph (V, E) consists of a set of external actions $sig(P) = (in(P), out(P))$, a partition $part(P)$ of the actions in $acts(P) = in(P) \cup out(P)$, and a set of timed sequences over the actions in $sig(P)$ called $tseq(P)$. For partition $part(P) = \{p_1 \ldots p_n\}$, where $p_i \subseteq acts(P)$, we denote the input actions of p_i by $in(p_i)$ and the output actions of p_i by $out(p_i)$. The purpose of the partition is to associate particular actions with particular nodes in the graph of the distributed system. Let P_1 and P_2 be two problems defined on the same graph. We say that a problem P_1 is a subset of problem P_2, denoted $P_1 \subseteq P_2$, iff $sig(P_1) = sig(P_2)$, $part(P_1) = part(P_2)$, and $tseq(P_1) \subseteq tseq(P_2)$.

Definition 2.10 (solve) Consider a graph (V, E). Let D be an automaton that models a distributed system with topology (V, E) such that node v_i is modeled by automaton A_i for all $v_i \in V$. Furthermore let P be a problem defined on (V, E). Let $part(P) = \{p_1 \ldots p_n\}$. D is said to *solve* problem P iff: $(in(A_i), out(A_i)) = (in(p_i), out(p_i))$ for all i such that $v_i \in V$ and $t\text{-}traces^\infty(D) \subseteq tseq(P)$. ∎

We now define two types of generalizations for problems.

Definition 2.11 (P_ϵ) Suppose P is a problem with partition $\{p_1 \ldots p_n\}$. Let K $= \{p_1, \ldots, p_n\}$. Define P_ϵ as follows: (1) $sig(P_\epsilon) = sig(P)$, (2) $part(P_\epsilon) = part(P)$, (3) $tseq(P_\epsilon) = \{\alpha \mid \alpha' =_{\epsilon,K} \alpha$ for some $\alpha' \in tseq(P)\}$. ∎

Definition 2.12 (P^δ) Suppose P is a problem with partition $\{p_1 \ldots p_n\}$. Let K $= \{out(p_1), \ldots, out(p_n)\}$. Define P^δ as follows: (1) $sig(P^\delta) = sig(P)$, (2) $part(P^\delta) = part(P)$, (3) $tseq(P^\delta) = \{\alpha \mid \alpha' \leq_{\delta,K} \alpha$ for some $\alpha' \in tseq(P)\}$. ∎

[4] Our results also hold for the case where messages cannot be reordered

3 Programming Model (Timed Automaton Model)

We specify formally the simple abstract programming model using timed automata. This is the model in which algorithm designers should specify their algorithms and prove them to be correct. In this model the programmer has direct access to real time using the *now* state component[5]. The messages sent over edges are taken from an arbitrary message set M. In order to simplify the proofs we assume that each message sent is *unique*, i.e. the same message cannot be sent twice in a given execution. It is easy, though tedious, to remove this restriction [1]. In the following sections we describe how each component of a distributed system is modeled by timed automata.

3.1 Algorithm

The timed automaton modeling node v_i is referred to as A_i. Since the timed automata A_i encode the algorithm, they must be supplied and proved to be correct by the algorithm designer. A_i is arbitrary except for a few restrictions. The action signature of A_i must include output actions $\text{SENDMSG}_i(j,m)$ for each j such that $e_{i,j} \in E$ and input actions $\text{RECVMSG}_i(j,m)$ for each j such that $e_{j,i} \in E$, where $m \in M$. All other actions in $acts(A_i)$ are restricted only by the requirement that $acts(A_i) \cap acts(A_j) = \{\nu\}$ when $j \neq i$. This restriction ensures that all communication between nodes uses the edges in E. The $\text{SENDMSG}_i(j,m)$ action sends the message m from v_i to v_j. The $\text{RECVMSG}_i(j,m)$ action receives message m sent by v_j to v_i. Finally, A_i must be feasible.

3.2 Communication

Denote the timed automaton that models edge $e_{i,j}$ with communication delay $[d_1, d_2]$ by $E_{i,j,[d_1,d_2]}$. The state of $E_{i,j}$ consists of an initially empty buffer $b_{i,j}$ whose elements are pairs (m,t) where $m \in M$ and $t \in \Re^+$. The action signature of $E_{i,j}$ consists of input actions $\text{SENDMSG}_i(j,m)$ and output actions $\text{RECVMSG}_j(i,m)$. The action $\text{SENDMSG}_i(j,m)$ adds (m,t), where t is the *now* value in the state preceding the action, to buffer $b_{i,j}$. The action $\text{RECVMSG}_i(j,m)$ deletes (m,t), where t is the *now* value in the state preceding the corresponding $\text{SENDMSG}_i(j,m)$ action, from buffer $b_{i,j}$. The transitions of $E_{i,j,[d_1,d_2]}$ are shown in Figure 1. The notation $\nu(\Delta_t)$ refers to a time-passage action that increases the *now* component by Δ_t.

3.3 System

Consider a distributed system with topology $G = (V,E)$ where each node v_i is modeled by timed automaton A_i and each edge $e_{i,j}$ is modeled by timed automaton $E_{i,j,[d_1,d_2]}$. Let A be a mapping assigning a timed automaton to each node in V such that timed automaton A_i is assigned to node v_i. Similarly, $E_{[d_1,d_2]}$ is a mapping from edges in E to timed automata such that edge $e_{i,j}$ is mapped to $E_{i,j,[d_1,d_2]}$. The timed automaton created by the composition of all the A_i and $E_{i,j,[d_1,d_2]}$, and the subsequent hiding of the $\text{RECVMSG}_j(i,m)$ and $\text{SENDMSG}_i(j,m)$ actions, denoted by $D_T(G,A,E_{[d_1,d_2]})$, represents the distributed system running the algorithm described by the mapping A. The subscript T on D is meant to indicate that $D_T(G,A,E_{[d_1,d_2]})$ is based on the timed automaton model.

[5] In addition to real time, the *now* state component can also be viewed as a perfectly accurate clock.

4 Inaccurate Clock Model (Clock Automaton Model)

The clock automaton model is more realistic than the timed automaton model since it only gives programmers access to a clock rather than real time. The clock keeps time with some minimum predetermined accuracy ϵ. In other words, the clock differs by at most ϵ from real time. The clock is modeled by the *clock* component of the clock automaton.

We show how the collection A_i of timed automata that are used to solve a problem in the timed automaton distributed system model can be transformed to solve a similar problem in the clock automaton distributed system model when the clock at each node has a minimum predetermined accuracy ϵ. The intuition behind the transformation is that an execution in the clock automaton model should look to the algorithm at each node like a possible execution of the timed automaton model. Since the algorithms have no access to real time, this is achieved automatically except in situations where a message arrives at a clock time that is less than the clock time at which the message was sent. The problem with this situation is that it cannot occur in the timed automaton model as long as message delivery times cannot be negative. We avoid these situations in the transformed clock automata by introducing message buffers at each node[6]. In the next two sections we formally specify a distributed system in the clock automaton model and give the transformation.

4.1 Algorithm, Communication, System

The node v_i is modeled by a clock automaton denoted by $A^c_{i,\epsilon}$. $A^c_{i,\epsilon}$ is arbitrary except for a few restrictions. $A^c_{i,\epsilon}$ must be feasible. Furthermore, $A^c_{i,\epsilon}$ must satisfy clock predicate C_ϵ, be ϵ-time independent, and conform to the edge interface.

Denote the timed automaton that models edge $e_{i,j}$ with communication delay $[d_1, d_2]$ by $E^c_{i,j,[d_1,d_2]}$. $E^c_{i,j,[d_1,d_2]}$ is the same as the timed automaton $E_{i,j,[d_1,d_2]}$ in the timed automaton model except that the messages are taken from the domain $M \times \Re^+$, and the actions $\text{RECVMSG}_j(i,m)$ and $\text{SENDMSG}_i(j,m)$ are renamed to $\text{ERECVMSG}_j(i,(m,c))$ and $\text{ESENDMSG}_i(j,(m,c))$ respectively.

The automaton modeling the distributed system in the clock automaton model, denoted $D_C(G, A^c_\epsilon, E^c_{[d_1,d_2]})$, is constructed as the composition of all $A^c_{i,\epsilon}$ and $E^c_{i,j,[d_1,d_2]}$ in the same manner as in the timed automaton distributed system model. In the clock automaton distributed system model the actions $\text{ERECVMSG}_j(i,(m,c))$ and $\text{ESENDMSG}_i(j,(m,c))$ are hidden since they constitute the edge interface.

4.2 Transformation

Definition 4.1 ($C(A_i,\epsilon)$) Let A_i be a timed automaton. Then $C(A_i,\epsilon)$ is the following clock automaton:

- $states(C(A_i,\epsilon)) = states(A_i) \times \Re^+$. Consider the state $s \in states(C(A_i,\epsilon))$, i.e. $s = (s_i, c)$. Then $s.cbasic = s_i.tbasic$, $s.now = s_i.now$, and $s.clock = c$. Furthermore define $s.A_i$ to be the state of A_i such that $(s.A_i).tbasic = s.cbasic$ and $(s.A_i).now = s.clock$.
- $s \in start(C(A_i,\epsilon))$ iff $s.A_i \in start(A_i)$ and $s.now = 0$.

[6] The requirement that the clock time at which a message is delivered is never less than the clock time at which it was sent was first identified by Lamport in [5]. The use of buffering to achieve this property was suggested by Welch in [17] and Neiger and Toueg in [13].

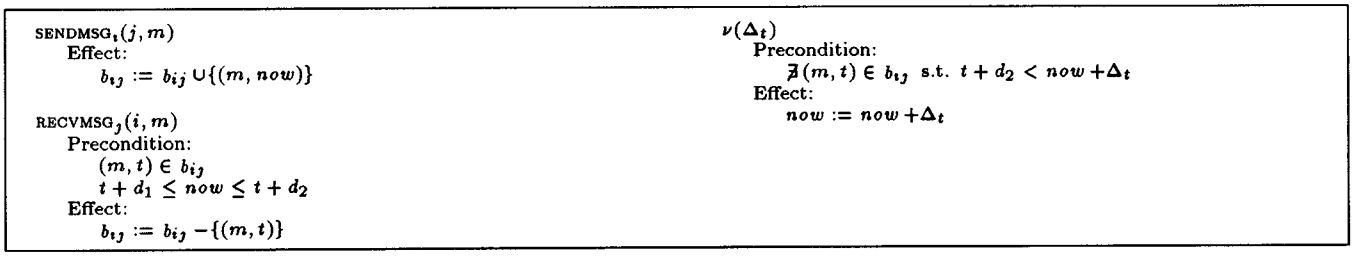

Figure 1: Transitions for $E_{ij,[d_1,d_2]}$.

- $sig(C(A_i,\epsilon)) = sig(A_i)$.
- $(s,a,s') \in trans(C(A_i,\epsilon))$ iff $(s.A_i, a, s'.A_i) \in trans(A_i)$, $a \neq \nu$ implies $s'.now = s.now$, and $a = \nu$ implies $s'.now > s.now$ and s' satisfies clock predicate C_ϵ. ■

The following lemmas are a consequence of Definition 4.1.

Lemma 4.1 *Let A_i be a timed automaton. Then $C(A_i,\epsilon)$ satisfies clock predicate C_ϵ. Furthermore, $C(A_i,\epsilon)$ is ϵ-time independent.*

Lemma 4.2 *Consider timed automaton A_i. For any $\alpha \in execs^\infty(C(A_i,\epsilon))$ define $\beta = (a_1(\alpha), c_1(\alpha))(a_2(\alpha), c_2(\alpha))\cdots$, where $c_i(\alpha) = s_{i-1}(\alpha).clock$. Then $\beta|(uacts(A_i) \times \Re^+) \in t\text{-}scheds^\infty(A_i)$.*

Section 4.2.2 and Section 4.2.1 introduce the clock automata $S_{ij,\epsilon}$ and $R_{ji,\epsilon}$ that implement the buffering needed to ensure that a message never arrives at a clock time that is strictly less than the clock time at which it was sent. Using $S_{ij,\epsilon}$ and $R_{ji,\epsilon}$ along with transformation C we specify the transformation used to make A_i function properly in the clock automaton model. Let $A^c_{i,\epsilon}$ be the parallel composition for clock automata of $C(A_i,\epsilon)$, $S_{ij,\epsilon}$ and $R_{ji,\epsilon}$ for all $e_{i,j} \in E$ and the subsequent hiding of the SENDMSG$_i(j,m)$ and RECVMSG$_i(j,m)$ actions.

The following lemma, which follows immediately from Lemma 4.2 and the definitions of $S_{ij,\epsilon}$ and $R_{ji,\epsilon}$, is used to compose the simulation of this section with the simulation to the MMT model.

Lemma 4.3 *Let ℓ, k be arbitrary constants. Assume that for any $\alpha \in execs(A_i)$ and $t \in \Re^+$ there are at most k output actions $a_j(\alpha)$ such that $t_j(\alpha) \in (t, t + k\ell]$ and at most k output actions $a_j(\alpha)$ such that $t_j(\alpha) \in [t, t + k\ell)$. Then for any $\alpha \in execs(A^c_{i,\epsilon})$ and $c \in \Re^+$ there are at most k output actions $a_j(\alpha)$ such that $c_j(\alpha) \in (c, c + k\ell]$ and at most k output actions $a_j(\alpha)$ such that $c_j(\alpha) \in [c, c + k\ell)$, where $c_j(\alpha) = s_{i-}(\alpha).clock$.*

4.2.1 Send Buffer

The sole purpose of clock automaton $S_{ij,\epsilon}$ is to tag the outgoing messages of $C(A_i,\epsilon)$ with the clock time at which they are sent. $S_{ij,\epsilon}$ satisfies clock predicate C_ϵ. The state of $S_{ij,\epsilon}$ consists of an initially empty queue q_{ij} with elements from $M \times \Re^+$. The operations defined on the queue are $enqu(q_{ji},(m,c))$ which adds (m,c) to the end of the queue, $dequ(q_{ji})$ which removes the front element of the queue, and $front(q_{ji})$ which returns the front element of the queue. The actions of $S_{ij,\epsilon}$ are input actions SENDMSG$_i(j,m)$ and output actions[7] ESENDMSG$_i(j,(m,c))$. The SENDMSG$_i(j,m)$ action causes $S_{ij,\epsilon}$ to receive message m from $A_{i,\epsilon}$. $S_{ij,\epsilon}$ delivers the message (m,c) to E_{ij} with the ESENDMSG$_i(j,(m,c))$ action. The transitions for $S_{ij,\epsilon}$ are shown in Figure 2.

[7] The E in ERECVMSG stands for "external".

4.2.2 Receive Buffer

The clock automaton $R_{ji,\epsilon}$ is used to ensure that the clock time at which a message is received by $C(A_i,\epsilon)$ is not strictly less than the clock time at which the message was sent. $R_{ji,\epsilon}$ receives messages from the communication channel and holds them in a buffer until the clock at the node is greater than or equal to the clock time at which the message was sent. $R_{ji,\epsilon}$ satisfies clock predicate C_ϵ. The state consists of an initially empty queue q_{ji} with elements from $M \times \Re^+$. The actions of $R_{ji,\epsilon}$ are input actions ERECVMSG$_i(j,(m,c))$ and output actions RECVMSG$_i(j,m)$. ERECVMSG$_i(j,(m,c))$ causes $R_{ji,\epsilon}$ to receive message (m,c) from $E^c_{ji,[d_1,d_2]}$. The message (m,c) was sent by node v_j when the clock at that node had the value c. The RECVMSG$_i(j,m)$ action causes $R_{ji,\epsilon}$ to deliver message m to $C(A_i,\epsilon)$. The transitions for $R_{ji,\epsilon}$ are shown in Figure 2.

4.3 Simulation Proof

Consider a distributed system modeled in the timed automaton model by $D_T(G,A,E_{[d'_1,d'_2]})$. Let $D_C(G,A^c_\epsilon,E^c_{[d_1,d_2]})$ be a distributed system modeled in the clock automaton model where each $A^c_{i,\epsilon}$ is generated from A_i based on the transformation defined in Section 4.2. Assume that $d'_1 = \text{MAX}(d_1 - 2\epsilon, 0)$ and $d'_2 = d_2 + 2\epsilon$.

We introduce some notational conventions. As shorthand we refer to $D_T(G,A,E_{[d'_1,d'_2]})$ and $D_C(G,A^c_\epsilon,E^c_{[d_1,d_2]})$ by D_T and D_C respectively. This section makes extensive use of the $=_{\epsilon,K}$ relation. $K = \{uacts(A_1),\ldots,uacts(A_n)\}$ in all cases. Consequently, we drop the subscript K for the remainder of Section 4.3. Suppose α is an execution of D_C. Since $uacts(D_C) = \bigcup_{i\in N} uacts(A^c_{i,\epsilon})$ and $uacts(A^c_{i,\epsilon}) \cap uacts(A^c_{j,\epsilon}) = \emptyset$ when $i \neq j$, we can associate a clock time with each non time-passage action in α using the *clock* components of the $A^c_{i,\epsilon}$. Formally, for action $a_i(\alpha)$ let $c_i(\alpha) = (s_{i-1}(\alpha)|A^c_{j,\epsilon}).clock$ when $a_i(\alpha) \in uacts(A_j)$.

Definition 4.2 Suppose α is an execution of D_C. We define two timed sequences γ'_α and γ_α. Let $\gamma'_\alpha = ((a_1(\alpha),c_1(\alpha)),(a_2(\alpha),c_2(\alpha))\cdots)|(uacts(D_T) \times \Re^+)$. Thus γ'_α is essentially the projection of the timed schedule of α onto the actions of the D_T except that the time component of each action-time pair is a *clock* value rather than the *now* value. Reorder γ'_α in non-decreasing order of the time components, retaining the original order of action-time pairs with the same time values. Call the resulting timed sequence γ_α. ■

The following lemmas give some properties of γ'_α and γ_α.

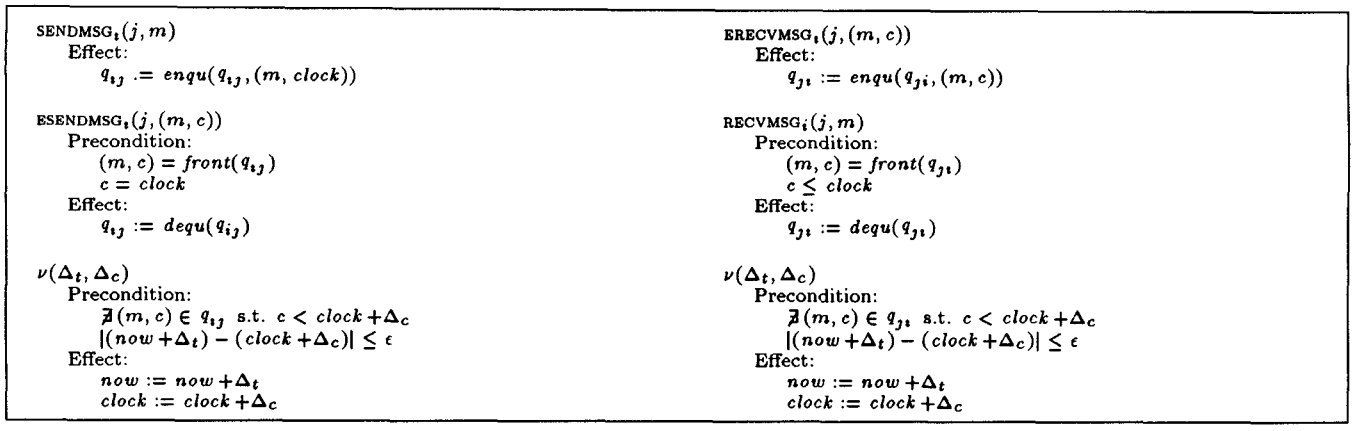

Figure 2: Transitions for $S_{ij,\epsilon}$ on the left and for $R_{ji,\epsilon}$ on the right

Lemma 4.4 *Let α be an admissible execution of D_C. Then $\gamma_\alpha|A_i$ is an admissible timed schedule of A_i.*

Proof: Let $\gamma_i = \gamma_\alpha|A_i$. Using Definition 4.2 is is easy to see that $\gamma_\alpha|A_i = \gamma'_\alpha|A_i$. Thus $\gamma_i = \gamma'_\alpha|A_i$. Let admissible execution α_i of $C(A_i,\epsilon)$ be defined as follows: $\alpha_i = (\alpha|A^c_{i,\epsilon})|C(A_i,\epsilon)$. In other words α_i is the projection of execution α onto the clock automaton modeling all parts of the node v_i except the send and receive buffers. Let $\beta_i = ((a_1(\alpha_i),c_1(\alpha_i))(a_2(\alpha_i),c_2(\alpha_i))\cdots$ where we define $c_j(\alpha_i) = s_{j-1}(\alpha_i).clock$. Now the construction of γ'_α shows $\gamma'_\alpha|A_i = \beta_i|(uacts(A_i)\times\Re^+)$. Since $\gamma_i = \gamma'_\alpha|A_i$ we know that $\gamma_i = \beta_i|(uacts(A_i)\times\Re^+)$. Using Lemma 4.2 we can now conclude that $\gamma_i \in t\text{-}scheds^\infty(A_i)$. ∎

Lemma 4.5 *Let α be an admissible execution of D_C. Then $\gamma_\alpha|E_{ij,[d'_1,d'_2]}$ is an admissible timed schedule of $E_{ij,[d'_1,d'_2]}$.*

Proof: We provide an informal sketch of the proof. The times associated with the actions in $\gamma_\alpha|E_{ij,[d'_1,d'_2]}$ are the clock times of the actions in α. The proof notes that the clock time of any send action is at most ϵ greater than the real time of the send action. Similarly, the clock time of any receive action is at most ϵ less than the real time of the receive action. Thus the clock time used by a message is at least $d_1 - 2\epsilon$. We note that the buffering now gives the desired result which is that the clock time used by a message is at least $\max(0, d_1 - 2\epsilon)$. The argument showing that the clock time used by a message is at most $d_2 + 2\epsilon$ is similar. ∎

Theorem 4.6 *Suppose α is an admissible execution of D_C. Then there exists an admissible execution β of D_T, such that $t\text{-}trace(\alpha) =_\epsilon t\text{-}trace(\beta)$.*

Proof: Lemma 4.4 shows that $\gamma_\alpha|A_i \in t\text{-}scheds^\infty(A_i)$, and Lemma 4.5 shows that $\gamma_\alpha|E_{ij,[d'_1,d'_2]} \in t\text{-}scheds^\infty(E_{ij,[d'_1,d'_2]})$. As a consequence we can conclude from Lemma 2.2 that $\gamma_\alpha \in t\text{-}scheds^\infty(D_T)$. Thus there exist an admissible execution β of D_T such that $\gamma_\alpha = t\text{-}sched(\beta)$. It is straightforward to show that $t\text{-}trace(\alpha) =_\epsilon \gamma_\alpha|(vis(D_T)\times\Re^+)$. Since $\gamma_\alpha = t\text{-}sched(\beta)$, we know that $\gamma_\alpha|(vis(D_T)\times\Re^+) = t\text{-}trace(\beta)$. Thus we conclude that $t\text{-}trace(\alpha) =_\epsilon t\text{-}trace(\beta)$. ∎

We are now ready to state the main result about the first simulation. The result states that any algorithm that solves a problem P in the timed automaton distributed system model can be transformed to solve the problem P_ϵ in the clock automaton distributed system model where the clock error is bounded by ϵ.

Theorem 4.7 *Suppose $D_T(G,A_i,E_{[d'_1,d'_2]})$ solves P. Consider $D_C(G,A^c_\epsilon,E^c_{[d_1,d_2]})$ where each $A^c_{i,\epsilon}$ is generated from A_i based on the transformation defined in Section 4.2. Assume that $d'_1 = \text{MAX}(d_1 - 2\epsilon, 0)$ and $d'_2 = d_2 + 2\epsilon$. Then $D_C(G,A^c_\epsilon,E^c_{[d_1,d_2]})$ solves P_ϵ.*

Proof: Consider any $\alpha \in execs^\infty(D_C(G,A^c_\epsilon,E^c_{[d_1,d_2]}))$. By Theorem 4.6, there exists $\beta \in execs^\infty(D_T(G,A,E_{[d'_1,d'_2]}))$, such that $t\text{-}trace(\alpha) =_\epsilon t\text{-}trace(\beta)$. Since β is an admissible execution of $D_T(G,A,E_{[d'_1,d'_2]})$, we know that $t\text{-}trace(\beta) \in tseq(P)$. Consequently, $t\text{-}trace(\alpha) \in tseq(P_\epsilon)$. ∎

Previous work by Lamport [5] and Neiger and Toueg [13] has focused on *internal specifications*, which in our terminology are specifications where $P = P_\infty$. In the context of asynchronous distributed systems, Lamport in [5] was the first to show that an algorithm solving an internal specification in a system with access to real time can be transformed to solve the same specification in a system with clocks that are not related to real time. In [13] Neiger and Toueg give the first formal definition for the concept of an internal specification. They also extend Lamport's original result to synchronous systems and systems where the maximum amount by which clocks differ from each other is bounded.

Since Theorem 4.7 covers non-internal specifications, or real time specifications, our results extend those of [5] and [13]. The ability to use real time specifications provides very powerful design techniques that are demonstrated in Section 6 and further discussed in Section 7. It should be noted though that our model and those of [5] and [13] differ. In contrast to the asynchronous systems, we assume that the delay on the links is bounded. Furthermore, in the synchronous setting [13] only bounds the amount by which clocks differ from each other. However, we note that the clock model where the clocks differ by at most ϵ from each other is similar to our clock model if some of the nodes in the distributed system are attached to real time sources such as atomic clocks[8].

5 Realistic Model (MMT Automaton Model)

In this section we consider a distributed system model based on the MMT automaton model. The MMT automaton

[8] We note that we do not claim any formal equivalence between the two clock models so that the comparison stated here is informal

model is more realistic than the clock automaton model in several important ways. MMT automata as defined in this section model finite step times by only guaranteeing that locally controlled actions occur within some time ℓ of each other. This limits the speed at which locally controlled actions are guaranteed to execute. Another consequence is that the clock may increase in discrete "jumps", skipping some values and making it impossible for the automaton to perform some output at a specified clock time.

We show how any collection of clock automata that are used to solve a problem in the clock automaton distributed system model can be transformed to solve a similar problem in the MMT automaton distributed system model. The minimum clock accuracy is assumed to be ϵ in both models. Furthermore, if the locally controlled actions in the MMT model are only guaranteed to occur within ℓ time units of each other, then for some constant k, at most k outputs can occur in the clock model at each node in any clock time interval of length $k\ell$. The transformation involves maintaining a buffer of pending outputs. Every time an output step can occur in the MMT model, the first output action in the buffer is allowed to take place. This causes each output to be delayed by real time at most $k\ell$, if the output is in the kth position in the buffer when it is first added to the buffer. Because of the restriction on the rate of output, the buffer remains bounded in size, which implies that the outputs are only delayed by a constant amount of time. In the next three sections we introduce the MMT model, specify a distributed system in the MMT automaton model, and give the transformation.

5.1 MMT Model

We give a brief introduction to MMT automata and refer the reader to [7, 11] for a more complete discussion. An MMT automaton A, like a timed automaton, consists of a set of states, $states(A)$, a subset of the states, $start(A)$, an action signature, $sig(A)$, and a set of transitions, $trans(A)$. It does not include the special state component *now* or the special time-passage action ν. Timing constraints are encoded by a *partition*, $part(A)$, of the locally controlled actions, $out(A) \cup int(A)$, into a set of equivalence classes and a *boundmap* which maps each class in the partition to a closed subinterval of $[0, \infty]$. The boundmap provides timing constraints for the actions in each class of the partition. Informally, the boundmap gives the minimum and maximum amount of time some action from a class must be enabled before an action from the class is executed.

An *execution* α of an MMT automaton A with boundmap b is a (finite or infinite) alternating sequence of states and action-time pairs starting in a start state, ending in a state if it is finite, with the following additional conditions. If $\alpha = s_0(a_1, t_1)s_1(a_2, t_1)\cdots$, then $(s_{i-1}, a_i, s_i) \in trans(A)$ for all i and $t_{i-1} \leq t_i$ for all i. Furthermore, if $b(C) = [l, u]$ for $C \in part(A)$ and $a_i \in C$, then there exists a consecutive set of states $s_j \ldots s_i$ in which a_i is enabled and $t_i - t_j \geq l$. Also, there exists no sequence of consecutive states $s_j \ldots s_i$ in which some action of C is enabled and $t_i - t_j > u$, but there is no $j < k \leq i$ such that $a_k \in C$. Finally, if α is infinite, the time values are unbounded. A timed trace of an execution α, denoted $t\text{-}trace(A)$ is the projection of the execution onto $ext(A) \times \Re^+$. The executions and the timed traces of MMT automaton A are denoted by $execs^\infty(A)$ and $t\text{-}traces^\infty(A)$ respectively. Finally, we note that a MMT automaton solves a problem as defined in Definition 2.10.

5.2 Algorithm, Communication, System

The node v_i is modeled by an MMT automaton denoted by $A^m_{i,\epsilon,\ell}$. $A^m_{i,\epsilon,\ell}$ is arbitrary except for a few restrictions. Its boundmap should map classes of the partition to the interval $[0, \ell]$. In the construction of the automaton representing the entire system, $A^m_{i,\epsilon,\ell}$ is composed with an MMT automaton $C^m_{i,\epsilon,\ell}$ whose sole output action is $\text{TICK}(c)$, where c represents the current clock time. The value c is always within ϵ of real time. Finally, $A^m_{i,\epsilon,\ell}$ must conform to the network interface described in Section 4.1.

Denote the automaton that models edge $e_{i,j}$ with communication delay $[d_1, d_2]$ by $E^m_{ij,[d_1,d_2]}$. $E^m_{ij,[d_1,d_2]}$ is the same as timed automaton $E^c_{ij,[d_1,d_2]}$.

In [7], there is a transformation T from MMT automata to timed automata. We use this transformation so that the MMT automata $A^m_{i,\epsilon,\ell}$ and $C^m_{i,\epsilon,\ell}$ can be composed with the timed automata $E^m_{ij,[d_1,d_2]}$. Since T is trace preserving, in other words $t\text{-}trace^\infty(A) = t\text{-}trace^\infty(T(A))$ for any MMT automaton A, we do not lose any of the realistic characteristics of the MMT automata when using T. We compose $T(A^m_{i,\epsilon,\ell})$ with $T(C^m_{i,\epsilon,\ell})$ to construct the automaton representing node v_i. The automaton modeling the distributed system in the MMT automaton model, denoted $D_M(G, A^m_{\epsilon,\ell}, E^m_{[d_1,d_2]})$, is constructed as the composition of all automata representing the nodes, $T(A^m_{i,\epsilon,\ell})$ composed with $T(C^m_{i,\epsilon,\ell})$, and all automata representing the edges, $E^m_{ij,[d_1,d_2]}$, using timed automata composition.

5.3 Transformation

Consider a clock automaton $A^c_{i,\epsilon}$ modeling node v_i in the clock automata model. We transform $A^c_{i,\epsilon}$ into an MMT automaton for node v_i with the transformation M. The simulation in this section requires that the clock automaton $A^c_{i,\epsilon}$ to which the transformation is applied satisfy the following restrictions. $A^c_{i,\epsilon}$ must be feasible, satisfy clock predicate C_ϵ, and be ϵ-time independent. Furthermore, for some constant k, any $\alpha \in execs(A^c_{i,\epsilon})$, and $c \in \Re^+$ there are at most k output actions $a_j(\alpha)$ such that $c_j(\alpha) \in (c, c + k\ell]$ and at most k output actions $a_j(\alpha)$ such that $c_j(\alpha) \in [c, c + k\ell)$, where $c_j(\alpha) = s_{i-}(\alpha).clock$. Since the clock automata generated by the transformation in Section 4.2, satisfy these restrictions, as long as the original timed automaton A_i satisfies the appropriate restrictions, the simulation to the clock model can be composed with the simulation to the MMT model. See Theorem 5.2.

Definition 5.1 ($M(A^c_{i,\epsilon}, \ell)$) Let $A^c_{i,\epsilon}$ be a clock automaton. Then $M(A^c_{i,\epsilon}, \ell)$ is the following MMT automaton:

- $states(M(A^c_{i,\epsilon}, \ell)) = states(A^c_{i,\epsilon}) \times \Re^+ \times Q(out(A^c_{i,\epsilon}))$, where $Q(out(A^c_{i,\epsilon}))$ is the set of all queues whose elements are in $out(A^c_{i,\epsilon})$. If $s \in states(M(A^c_{i,\epsilon}, \ell))$, i.e. $s = (s_i, c, q)$, then $s.simstate = s_i$, $s.mmtclock = c$, and $s.pending = q$. Each state s also has the following derived state components: $s.frag$, an arbitrary execution fragment[9] of $A^c_{i,\epsilon}$ containing no input actions, with initial state $s.simstate$ and some final state u such that $u.clock = s.mmtclock$, $s.fragstate$, the final state of $s.frag$, and $s.fragoutputs$, the projection of $s.frag$ onto $out(A^c_{i,\epsilon})$.
- $s \in start(M(A^c_{i,\epsilon}, \ell))$ iff $s.simstate \in start(A^c_{i,\epsilon})$, $s.mmtclock = 0$, and $s.pending$ is empty.
- $sig(M(A^c_{i,\epsilon}, \ell)) = (in(A^c_{i,\epsilon}) \cup \{\text{TICK}(c)\}, out(A^c_{i,\epsilon}), \{\tau\})$.

- $part(M(A^c_{i,\epsilon}, \ell)) = \{out(A^c_{i,\epsilon}) \cup \{\tau\}\}$, with boundmap value $[0, \ell]$ for the single class.
- $(s, a, s') \in trans(M(A^c_{i,\epsilon}, \ell))$ iff one of the following holds:
  - $a = \text{TICK}(c)$, $s.simstate = s'.simstate$, $s'.mmtclock = c$, and $s.pending = s'.pending$.
  - $a \in in(A^c_{i,\epsilon})$, $(s.fragstate, a, s'.simstate) \in trans(A^c_{i,\epsilon})$, $s'.mmtclock = s.mmtclock$, and $s'.pending =$ $s.pending$ plus $s.fragoutputs$.
  - $a \in out(A^c_{i,\epsilon})$, a is first on $s.pending$, $s'.simstate = s.fragstate$, $s'.mmtclock = s.mmtclock$, and $s'.pending =$ $s.pending$ minus first action, plus $s.fragoutputs$.
  - $a = \tau$, $s.pending$ is empty, $s'.simstate = s.fragstate$, $s'.mmtclock = s.mmtclock$, and $s'.pending =$ $s.pending$ plus $s.fragoutputs$.

In the transformed automaton the state component *simstate* constitutes the simulated state of the clock automaton. The *mmtclock* component is the clock value of the MMT automaton, and the *pending* component is the buffer of output actions that must still be executed. The derived state components *frag*, *fragstate*, and *fragoutputs* are used to calculate *pending* and *simstate* at the transitions of the transformed automaton.

5.4 Simulation Proof

The main result about this simulation states that any algorithm that solves a problem P in the clock automaton distributed system model can be transformed to solve the problem $P^{k\ell+2\epsilon+3\ell}$ in the MMT automaton distributed system model where the clock error is bounded by ϵ, the step times are bounded by ℓ, and k characterizes the speed at which the algorithm in the clock automaton model generates output actions. Recall from Definition 2.12 that $P^{k\ell+2\epsilon+3\ell}$ is the same as P except that the output actions at each node can be shifted by at most $k\ell + 2\epsilon + 3\ell$ into the future. Theorem 5.1 is stated without the proof since its proof is very similar to the one given for Theorem 4.7.

Theorem 5.1 *Suppose $D_C(G, A^c_\epsilon, E_{[d'_1,d'_2]})$ solves P. Consider $D_M(G, A^m_{\epsilon,\ell}, E^m_{[d_1,d_2]})$ where each $A^M_{i,\epsilon,\ell}$ is generated from $A^c_{i,\epsilon}$ based on the transformation defined in Section 5.3. Assume $d'_1 = d_1$ and $d'_2 = d_2 + k\ell$. Then $D_M(G, A^m_{\epsilon,\ell}, E^m_{[d_1,d_2]})$ solves $P^{k\ell+2\epsilon+3\ell}$.*

Using Lemma 4.1 and Lemma 4.3, the result of Theorem 5.1 can be composed with the result of Theorem 4.7.

Theorem 5.2 *Suppose $D_T(G, A_i, E_{[d'_1,d'_2]})$ solves P, where A_i conforms to the timing restrictions of Lemma 4.3. Consider $D_M(G, A^m_{\epsilon,\ell}, E^m_{[d_1,d_2]})$ where each $A^M_{i,\epsilon,\ell}$ is generated from $A^c_{i,\epsilon}$ based on the transformation defined in Section 5.3, and each $A^c_{i,\epsilon}$ is in turn generated from A_i based on the transformation defined in Section 4.2. Assume that $d'_1 = \text{MAX}(d_1 - 2\epsilon, 0)$ and $d'_2 = d_2 + 2\epsilon + k\ell$. Then $D_M(G, A^m_{\epsilon,\ell}, E^m_{[d_1,d_2]})$ solves $(P_\epsilon)^{k\ell+2\epsilon+3\ell}$.*

[9] The feasibility requirement and the clock predicate C_ϵ of $A^c_{i,\epsilon}$, together with the invariant that $s.mmtclock \geq (s.simstate).clock$ for all states s, imply that such a fragment always exists when ϵ is finite.

6 Linearizable Read-Write Objects

We consider a network implementation of concurrent, distributed shared memory objects satisfying *linearizability*. Each READ or WRITE operation does not take place instantaneously, but has invocation and response events, which may be separated by other intervening events and might even occur at different times. In a concurrent execution, operations may therefore overlap. Linearizability requires that a concurrent execution have the same behavior as a sequential execution where each operation takes place instantaneously at some point in time between its invocation and response.

We present an algorithm for linearizable read-write objects in the clock automaton distributed system model. We consider the simple algorithm for this problem in the timed automaton distributed system model given by Mavronicolas in [10] (the algorithm is a generalization of an algorithm in [2].) We then transform the algorithm using our simulation techniques to arrive at a simple algorithm in the clock automaton system. We proceed by defining ϵ-*superlinearizability* (a property stronger than linearizability), and modifying the algorithm for linearizability in the timed automaton model, to one that solves superlinearizability in the same model. We then use our first simulation to show that the algorithm solves linearizability in the clock automaton model.

Our algorithm is much simpler and improves on the complexity bounds of the algorithm for the clock automaton model presented in [10]. We present our algorithm and complexity results in the clock automaton model rather than the more realistic MMT model, so as to be consistent with previous results. It is easy to transform this algorithm to run in the realistic model using Theorem 5.2. We generalize our results to other shared memory objects in the full paper.

6.1 An Algorithm for Linearizability in the Timed Automaton Model

We define linearizability formally below. Given a timed schedule α, each operation op is defined by a pair of actions, the invocation $Inv(op)$ and the response $Res(op)$. A timed schedule α is *linearizable* if it is possible to create a timed sequence β by inserting an operation-time pair (op, t) into α for each operation op such that, (op, t) is ordered between $Inv(op)$ and $Res(op)$, and a READ operation returns the value written by the last WRITE operation that precedes it in the projection of β onto the operation-time pairs. We call this projection $\tau_l(\alpha)$. We say that operation op occurs at time t in $\tau_l(\alpha)$ provided that the pair (op, t) appears in the sequence $\tau_l(\alpha)$.

We now give an informal description of the algorithm presented in [10]. The algorithm is presented in a model similar to the timed automaton model with message delay $[d'_1, d'_2]$. The model in [10] assumes that at any particular time, all inputs at a node arrive before any outputs are executed. We modify the algorithm in [10] slightly to work in the timed automaton model, which does not have this property, by waiting an arbitrarily small amount of extra time δ before executing an output action that depends on all the inputs at a particular time. We define c to be any value between 0 and $d'_2 - 2\epsilon$, where ϵ is the clock inaccuracy in the clock model. By varying the value of c, the user can achieve a tradeoff between the time complexity of a read and write.

Algorithm L The inputs to node i are the invocation actions READ_i and $\text{WRITE}_i(v)$ from the external environment and messages of the form $\text{UPDATE}_j(v,t)$ from node j. The outputs are $\text{RETURN}_i(v)$ (the response to READ_i) and ACK_i (the response to $\text{WRITE}_i(v)$) to the external environment, and messages of the form $\text{UPDATE}_i(v,t)$ to other nodes. When a READ_i occurs, the processor (node) waits $c+\delta$ time, reads the value v in its local memory, and does a $\text{RETURN}_i(v)$. When a $\text{WRITE}_i(v)$ occurs, i sends an $\text{UPDATE}_i(v,t)$ message to all processors (including itself), where t is the sending time. It then waits $d_2' - c$ time and does an ACK_i. On receiving an update message $\text{UPDATE}_j(v,t)$, the processor waits until the time component *now* equals $t + d_2' + \delta$ and then changes the value in its local memory to v. If it receives more than one update message at the same time, it picks the one with largest the index j, and ignores the others.

In any execution of the algorithm L, every read can be linearized just after its invocation and every write can be linearized just before its response. To prove this, we notice that updates to each processor's local memory happen at exactly the same time. Thus, all local memories are always consistent after each real time. Note that, intuitively, a read actually occurs at the time of response (that is when the local copy is read), while a write occurs at the time of the local update. However, the update is not necessarily within the range of the write, but exactly $d_2' + \delta$ after its invocation. We therefore shift the linearization points of both reads and writes $c + \delta$ earlier, causing the reads to be linearized at the time of invocation (duration of a read is $c+\delta$), and the writes at the time of response (duration of a write is $d_2' - c$). This satisfies the requirement that reads and writes be linearized between their invocation and response.

We note that the algorithm only guarantees linearizability when an execution has the property that the actions at each node alternate between an invoke action and the corresponding response action. An execution with this property is said to satisfy the *alternation condition*. We say that the environment is the first to violate the alternation condition for an execution if the smallest prefix of the execution not satisfying the alternation condition contains two consecutive invoke operations at some node. We define the problem, P, of a linearizable read-write object as follows. The timed traces of P are the union of the set of traces in which the environment is the first to violate the alternation condition, and the set of traces that satisfy the alternation condition and are linearizable.

$D_T(G, L, E_{[d_1', d_2']})$ is the timed automaton system with message delay bounds $[d_1', d_2']$ running the algorithm L. We define the time complexity of an operation to be the maximum time between the invocation and response of any instance of the operation in any execution of the system. A variation of the following lemma is proven in [10]. We have the added time complexity of δ on the read.

Lemma 6.1 *$D_T(G, L, E_{[d_1', d_2']})$ solves P with the time complexity of $c + \delta$ for a read operation and the time complexity of $d_2' - c$ for a write operation.*

6.2 An Algorithm for Superlinearizability in the Timed Automaton Model

We describe our ϵ-superlinearizability property. A timed schedule α is *ϵ-superlinearizable* if it is possible to create a timed sequence β by inserting a pair (op, t) into α for each operation op such that, (op, t) is ordered between $Inv(op)$ and $Res(op)$, the time of $Inv(op)$ is less than or equal to $t - 2\epsilon$, and a READ operation returns the value written by the last WRITE operation that precedes it in the projection of β onto the operation-time pairs. We call this projection $\tau_s(\alpha)$.

We define the problem, Q, of an ϵ-superlinearizable read-write object as follows. The timed traces of Q are the union of the set of traces in which the environment is the first to violate the alternation condition, and the set of traces that satisfy the alternation condition and are ϵ-superlinearizable.

We remark informally that many linearizable algorithms can be transformed into ϵ-superlinearizable algorithms, by allowing each operation to delay an extra 2ϵ before starting the normal processing of the operation. So, if an operation takes time t in the original algorithm, it would take time $t + 2\epsilon$ in the new algorithm.

We modify the algorithm L described above to solve ϵ-superlinearizability. While adding a delay of 2ϵ before every operation as suggested by the above transformation is sufficient for correctness, we can improve on the performance of the algorithm by adding delays more judiciously. Specifically, it is sufficient to add a delay at the beginning of a READ operation, but it is not necessary to modify a WRITE operation. It is easy to see, intuitively, that each operation is now linearized at least 2ϵ after its invocation, and no later than its response.

Our modified algorithm S (for ϵ-superlinearizability) is described formally below, where each processor (or node) i is represented by a timed automaton, S_i. The state components of the automaton are: the *now* component; *value*, a value in V, initially v_0; and the records *read*, *write* and *updates*, as described below.

- The components of the *read* are
  status, in $\{inactive, active\}$, initially *inactive*; and
  time, a nonnegative real or *nil*, initially *nil*.
- The components of the *write* are
  status, in $\{inactive, send, ack\}$, initially *inactive*;
  send-value, in $V \cup \{nil\}$, initially *nil*;
  send-procs, $\subseteq P$, initially $\emptyset$;
  send-time, a nonnegative real or *nil*, initially *nil*; and
  ack-time, a nonnegative real or *nil*, initially *nil*.
- *updates* is a set of records r, each with components
  proc, in $P \cup \{nil\}$, initially *nil*;
  value, in $V \cup \{nil\}$, initially *nil*; and
  update-time, a nonnegative real or *nil*, initially *nil*.

In addition, there is a derived variable *mintime*, defined as the minimum of the components *read.time*, *write.send-time*, *write.ack-time*, and *r.update-time* (r in updates). The input actions are READ_i, $\text{WRITE}_i(v)$, and $\text{RECVMSG}_i(j,(v,t))$; the output actions are $\text{RETURN}_i(v)$, ACK_i, and $\text{SENDMSG}_i(j,(v,t))$; and the internal action is UPDATE. The transition relation is given in Figure 3.

The following lemma shows that our new algorithm S linearizes each operation at least 2ϵ after the invocation. $D_T(G, S, E_{[d_1', d_2']})$ represents the timed automaton system with message delay bounds $[d_1', d_2']$ running algorithm S.

Lemma 6.2 *$D_T(G, S, E_{[d_1', d_2']})$ solves Q, with time complexity $2\epsilon + c + \delta$ for a read and time complexity $d_2' - c$ for a write.*

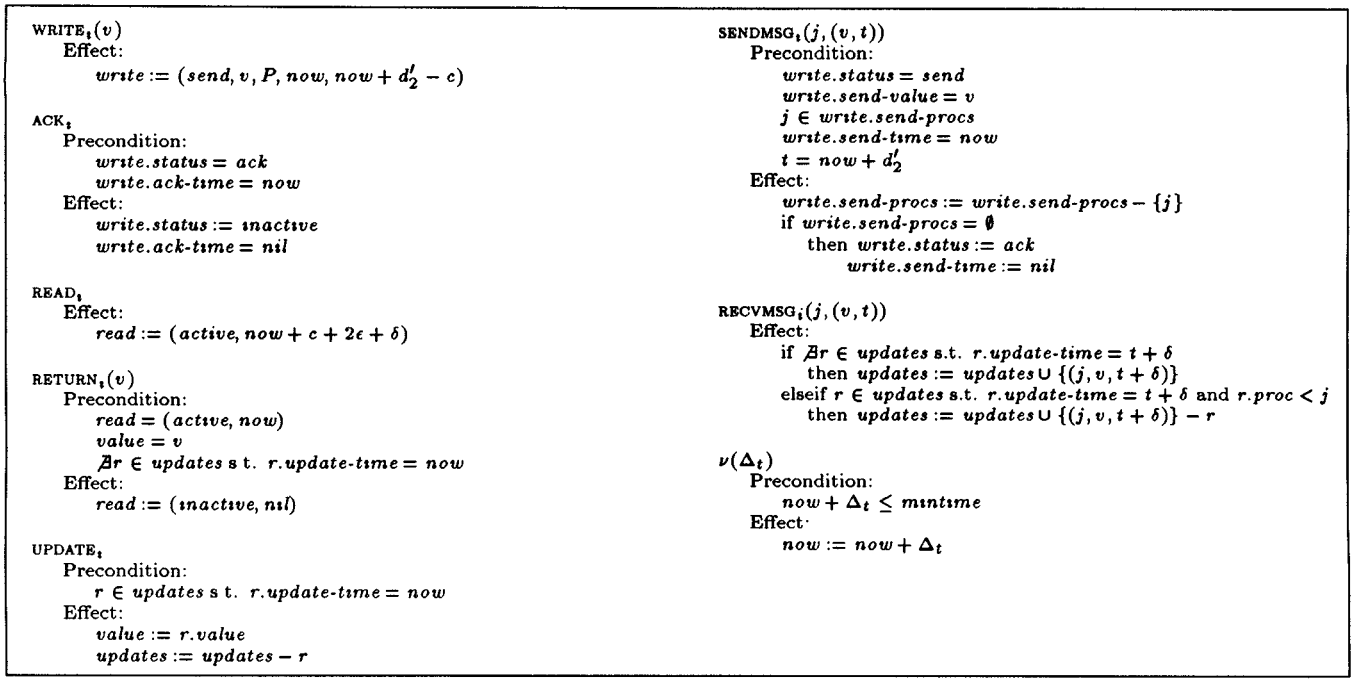

Figure 3: The Transition Relation for *Read-Write*

6.3 An Algorithm for Linearizability in the Clock Automaton Model

As claimed above, algorithm S solves ϵ-superlinearizability in the timed automaton model. We use our first simulation to transform the algorithm to run in the clock automaton model with delay $[d_1, d_2]$. Assume that $d_1' = max(d_1 - 2\epsilon, 0)$ and $d_2' = d_2 + 2\epsilon$.

Lemma 6.3 *$D_C(G, S_\epsilon^c, E_{[d_1,d_2]}^c)$ solves Q_ϵ, with read time complexity $2\epsilon + \delta + c$ and write time complexity $d_2 + 2\epsilon - c$.*

To show that our algorithm actually solves linearizability in the clock model, it remains to prove that $Q_\epsilon \subseteq P$.

Lemma 6.4 $Q_\epsilon \subseteq P$.

Proof: Let β be a timed trace in Q_ϵ. Then β is obtained by modifying some $\beta' \in Q$ by moving actions by at most ϵ without changing local node order.

Suppose that β' is a trace in which the environment is first to violate the alternation condition. Then there is a prefix of β' in which there are two more invocations than responses at some node i; let γ' be the shortest such prefix. Consider the subsequence of invocations and responses for node i in γ'. The same subsequence must appear in a prefix of β since β and β' satisfy the same local node order. Therefore, the environment is first to violate the alternation condition in β as well.

Now suppose that β' satisfies the alternation condition and is ϵ-superlinearizable. Since local node orders are the same in β and β', there is correct alternation in β as well. From a timed sequence $\tau_s(\beta')$ that is used to satisfy superlinearizability in β', we construct the timed sequence τ, where the time for each operation is shifted earlier by ϵ. Since the order of the operations in τ and $\tau_s(\beta')$ are the same, it is now straightforward to show that there exists some $\tau_l(\beta)$ such that $\tau = \tau_l(\beta)$, i.e. τ satisfies linearizability in β. Therefore, β satisfies the alternation condition and is linearizable. It follows that β is a timed trace in P.

■

Theorem 6.5 follows from Lemma 6.3 and Lemma 6.4.

Theorem 6.5 *$D_C(G, S_\epsilon^c, E_{[d_1,d_2]}^c)$ solves P, with read time complexity $2\epsilon + \delta + c$ and write time complexity $d_2 + 2\epsilon - c$.*

The complexity bounds of our algorithm represent a significant improvement over the algorithm for linearizable read-write objects in the clock automaton model presented in [10]. It is important to note here that the clock automaton model in [10] is a slight variant of ours. While our model allows clocks to be at most ϵ away from real time, the clock automaton model in [10] allows clocks to be at most a constant u from each other, while proceeding at a constant real time rate. We claim informally that the clock model in [10] is similar to ours if $u = 2\epsilon$, and at least one of the nodes in the system are attached to a real time source such as an atomic clock. In particular, our results apply to this model as well. Translated into the model in [10], our algorithm would have a read time complexity of $c + u$, and a write time complexity of $d_2 - c + u$, giving a combined read and write complexity bound of $d_2 + 2u$. The algorithm presented in [10], which involves some complicated time-slicing, achieves the read time complexity $4u$ and the write time complexity $d_2 + 3u$.

We can extend our result to the MMT model. All we need to show is that $(Q_\epsilon)^{k\ell+2\epsilon+3\ell} \subseteq P$. This is a straightforward argument based on the observation that the response actions in an execution in $(Q_\epsilon)^{k\ell+2\epsilon+3\ell}$ are just shifted into the future with respect to a corresponding execution in Q_ϵ.

7 Discussion

7.1 Design Techniques

We discuss two approaches for using our results in practice. To simplify the discussion, we focus our attention on the

first simulation between the timed automaton model and the clock automaton model.

The basic observation of the first approach is that it is often sufficient to solve P_ϵ instead of P. This is especially true in practice when ϵ is small and the system interacts with humans. In situations where it is sufficient to solve P_ϵ instead of P, an algorithm designer proceeds as follows. Assume that you have a physical distributed system with topology $G = (V, E)$ where the clocks have an accuracy of ϵ and the communication has delay $[d_1, d_2]$. Construct an algorithm, in other words design an A_i for each node v_i, such that $D_T(G, A, E_{[d_1', d_2']})$, where $d_1' = max(d_1 - 2\epsilon, 0)$ and $d_2' = d_2 + 2\epsilon$ solves problem P. Then Theorem 4.7 shows that $D_T(G, A_\epsilon^c, E_{[d_1, d_2]}^c)$ solves problem P_ϵ when A_ϵ^c is constructed based on the transformation in Section 4.2. Using this approach simplifies both the algorithm's design and correctness proof since both are carried out in a model with access to real time.

The second approach deals with situations where it is not sufficient to solve P_ϵ. In this situation one should design a problem Q such that $Q_\epsilon \subseteq P$. Once the problem Q is defined the algorithm designer proceeds as in the first approach to find an algorithm for Q and hence for P. The application in Section 6 uses this technique. The degree of difficulty associated with finding a specification Q such that $Q_\epsilon \subseteq P$ varies with the type of property that the specification requires the environment and the system to maintain. For ordering properties such as the well-formed property of Section 6 it is generally easy to find the appropriate Q. For real time properties, finding the appropriate Q is often complicated or impossible.

7.2 Practical Issues

Many algorithms designed assuming access to real time are run, without implementing the buffering added by our transformation, in systems where there is only access to inaccurate clocks. Recall that the buffering is designed to prevent the arrival of messages at clock times that are less than the clock times at which the messages are sent. There are situations where the buffering is not required. For example, when the minimum message delay is greater than 2ϵ no message can ever arrive at a clock time that is greater than the clock time at which it was sent. Thus, when the minimum message delay in greater than 2ϵ, the buffering is not needed.

However, even when it is required, the buffering is not too expensive. In [12], Mills indicates that clock accuracies of a few milliseconds are possible, even under adverse conditions. Thus, even if the clocks of two communicating nodes differ by the maximum possible amount, the largest amount of time that messages would have to be buffered is a few milliseconds. For all but the most exotic applications, such as real time video[10], the task of delaying the messages can be handled with little overhead in the device drivers.

7.3 Future Work

While MMT automaton model captures many of the complications of real systems, we believe that additional work is needed in this area. For example, permitting inputs to occur without timing constraints while at the same time permitting the MMT automaton to perform local calculations on each input seems unrealistic. We also hope to apply the techniques developed in this paper to additional applications.

[10] One would expect that real time video would fall under one of the situations where buffering is not necessary.

References

[1] Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck, Reliable Communication over an Unreliable Channel, LCS TM-447, October, 1992. Also submitted to JACM.

[2] H. Attiya and J. Welch, Sequential Consistency Versus Linearizability, *Proceedings of 3rd ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

[3] Digital Time Service Functional Specification Version T.1.0.5, Digital Equipment Corporation, 1989.

[4] R. Gawlick, N. Lynch, R. Segala, and J. Søgaard-Andersen, Liveness Properties for Timed and Untimed Systems, work in progress.

[5] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol. 21, Number 7, July 1978.

[6] B. Lampson, N. Lynch, and J. Søgaard-Andersen, Reliable At-Most-Once Message Delivery Protocols, work in progress.

[7] N. Lynch and H. Attiya, Using mappings to prove timing properties, *Distributed Computing*, 6:121-139, 1992.

[8] N. Lynch and F. Vaandrager, Forward and Backward Simulations for Timing-Based Systems, *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, 1992.

[9] N. Lynch and F. Vaandrager, Forward and Backward Simulations Part II: Timing-Based systems, submitted for publication.

[10] M. Mavronicolas, Timing-Based, Distributed Computation: Algorithms and Impossibility Results, Ph.D. Thesis, Harvard University, 1992.

[11] M. Merritt, F. Modugno, and M. Tuttle, Time Constrained Automata, *CONCUR'91 Proceedings Workshop on Theories of Concurrency: Unification and Extension*, August, 1991.

[12] D. Mills, Network Time Protocol(Version 3) Specification, Implementation, analysis, DARPA Networking Group Report, July 1990.

[13] G. Neiger and S. Toueg, Simulating Synchronized Clocks and Common Knowledge in Distributed Systems. *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, to appear in the *Journal of the ACM*.

[14] R. Perlman, An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN, *Journal of the ACM*, 1985.

[15] M. Tuttle and N. Lynch, Hierarchical Correctness Proofs for Distributed Algorithms, Technical Report MIT/LCS/TR-387, MIT, 1987.

[16] F. Vaandrager and N. Lynch, Action Transducers and Timed Automata, *Proceedings of CONCUR '92, 3rd International Conference on Concurrency Theory*, August 1992.

[17] J. Welch, Simulating Synchronous Processors, *Information and Computation*, 74(2):159-171, August 1987.