Time-Constrained Automata*

(Extended Abstract)

Michael Merritt
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
merritt@research.att.com

Francesmary Modugno
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
fmm@cs.cmu.edu

Mark R. Tuttle
DEC Cambridge Research Lab
One Kendall Sq., Bldg. 700
Cambridge, MA 02139
tuttle@crl.dec.com

Abstract

In this paper, we augment the input-output automaton model in order to reason about time in concurrent systems, and we prove simple properties of this augmentation. The input-output automata model is a useful model for reasoning about computation in concurrent and distributed systems because it allows fundamental properties such as fairness and compositionality to be expressed easily and naturally. A unique property of the model is that systems are modeled as the composition of *autonomous* components. This paper describes a way to add a notion of time to the model in a way that preserves these properties. The result is a simple, compositional model for real-time computation that provides a convenient notation for expressing timing properties such as bounded fairness.

1 Introduction

This paper augments the *input-output automaton* model [LT87] with a notion of time that allows us to reason about timed behaviors, especially behaviors in real-time systems where real-time constraints on systems' reaction times must be satisfied. The (untimed) input-output automaton model is a natural model of computation that has been used extensively to study concurrent systems. The model has many appealing properties. For example, it is especially helpful when describing the interfaces between system components, and it provides a clean compositional model for fair computation. The motivation for our work is to find an equally intuitive generalization of the model to timed computation that preserves these properties. Our generalization results in a compositional model for timed

*This article appeared in J. C. M. Baeten and J. F. Groote, editors, *Proceedings of the Second International Conference on Concurrency Theory (Concur'91)*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423, Springer-Verlag, Berlin, August 1991.

computation with a time-bounded notion of fair computation in which many interesting real-time constraints can be described simply. This model has been used to study problems in real-time systems, and simple proof rules have been developed for it [LA90, AL89].

The input-output automaton model is unique in that it is especially well-suited for modeling concurrent systems as the composition of autonomous components. A system component is *autonomous* if it has complete control over its generation of output. More precisely, its generation of output is a function of its own local state, not the global state, and cannot be blocked simply because no other system component is ready to receive the output. Our intuition is that these autonomous components represent the physically realizable components of the system: a network node can transmit a message over the network independent of the state of other nodes. Any natural model of concurrent computation should make it easy to describe systems as the composition of autonomous components.

While most models contain a submodel that supports such descriptions, these models are so expressive that they include notions of composition that have no correspondence to physical reality, and this unwanted expressive power complicates the models' semantics considerably. One example is CSP [Hoa85, KSdR$^+$88]. In CSP, each process P has an alphabet of actions it can perform, and the parallel composition $P||Q$ of two processes P and Q requires that any action a in the intersection of their alphabets must be performed simultaneously by both if it is performed at all. In part because there is no semantic distinction between input and output actions in CSP, there is no notion of any individual process determining the performance of an action, and hence there is (in general) no notion of autonomy in the model. The parallel composition $P||Q$ of P and Q enables Q to keep P from performing any action a in the intersection of their alphabets simply by refusing to perform a itself. Thinking of Q as P's environment, this means that any action the environment can observe is an action the environment can synchronize with and block. This makes some problems almost too easy to solve. For example, consider the solution to the Dining Philosopher's problem in [Hoa85]. Here the philosophers are described in terms of actions like picking up and setting down forks, and the philosophers are placed in an environment (the definitions of the forks) that can simply block a philosopher when it tries to pick up a fork. The philosophers have no autonomy over their actions, and deadlock can be avoided by composing with any process whose definition is simply a description of the desired behaviors (cf. [CM84]). Because of the powerful operators in CSP, specifying an acceptable solution to a problem can also require more than specifying the desired external behavior (or traces). In contrast, in the input-output automaton model, it is natural to accept as a solution to a problem any system with the desired external behavior that can be expressed in the model.

Input-output automata can be viewed as a restriction of CSP and related models [Mil80, Yi90, MT90, GL90] to a simple submodel, with a simple semantics, that captures the notion of autonomy. A primary difference between the input-output automaton model and these models is that the former makes a clear distinction between input and output actions. In this model, each system component is modeled as an automaton with actions labeling the state transitions. These actions are partitioned into input and output actions. This partition is used to state two restrictions that guarantee that system components are autonomous. First, automata are *input-enabled*, which means that any input action a can be performed in any state s (there is a transition from s labeled a). Second, two

automata P and Q may be composed (essentially by identifying actions, as described above) only when the output actions of P and Q are disjoint. Consequently, P has complete control over its generation of output in the composition of P and Q: if a is an output action of P, then it must be an input action of Q (if it is an action of Q at all); and since Q is input-enabled, it is willing to accept a as input in any state. Early work involving continually enabled inputs appears in [LF81], and more recently in [Dil88].

A second difference is that fairness plays an important role in the input-output automaton model. A system computation is *fair* if every system component is given the chance to take a step infinitely often. The definition of fairness together with the weak (relative to CSP) composition operator result in a simple compositional model of fair concurrent computation. In this model automata generate fair behaviors, and when automata are composed, the fair behaviors of the composition are a composition of the fair behaviors of the components. Definitions of fairness in the same spirit appear in [LF81, Fra86, Jon87].

There are two natural approaches to extending the input-output automaton model to include timing information. The first is to record time and timing constraints directly in the automaton states and transition relation. This approach is exemplified by the work of Shankar and Lam [SL87] (and also [HLP90]), in which time is modeled as a component of the system state, and predicates on the time control system executions. The second approach is to model time and timing constraints as external conditions imposed on the executions of standard input-output automata. This approach is adopted here. A timed execution is essentially an ordered pair (e, t), where e is the execution of an input-output automaton, and t is a function assigning times to the events occurring in e (cf. [AH90, Lam91]). A timed automaton is a pair (A, P) consisting of an input-output automaton A and a predicate P on the timed executions of A.

Separating time from the local state makes it easy to define a clean notion of automaton composition. If instead time is recorded in the local state, then—in the straightforward composition of such automata—the times in the local states will bear little relation to one another. Some additional axioms or rules for automaton composition must be imposed to keep the times more or less synchronized. Furthermore, there is no longer one single variable in the state of the composition that records the current time, but rather a tuple of variables, and the complexity increases with additional composition. On the other hand, if time is externally assigned to events in a computation via a timing function t as is done here, then there is a simple syntactic mechanism for distinguishing the time component that allows a simple definition of composition in which components are synchronized.

After augmenting the input-output model to include a notion of time, a timing condition called a *boundmap* is defined, essentially a bounded fairness condition that restricts the amount of time that may elapse between consecutive steps of a system component.[1] Both the fair and unfair computations of the untimed input-output model are natural special cases of such boundmaps. One of the important results in this paper is that our augmentation of the input-output automaton model to incorporate time is a compositional model for timed computations. The fact that it is a compositional model for fair computation now follows as a special case. This modularity is one of the primary advantages of our work.

The rest of this paper is organized as follows. In Section 2 we review the input-output

[1] Lewis [Lew90] also assigns bounds to state transitions. His motivation is quite different from ours, but we can generalize boundmaps slightly and capture his assignments.

automaton model. In Section 3 we augment the model to include time, and in Section 4 we define the composition of timed automata. In Section 5 we define a simple notation for real-time constraints, and in Section 6 we define boundmaps as a special case. Finally, in Section 7 we define what it means for one timed automaton to solve a problem described by another timed automaton. Due to space limitations, we have omitted the proofs of our results. We have also omitted any significant examples of how to use our framework, but examples do appear in [LA90, AL89]. A full version of this paper will contain both proofs and examples.

2 Input-Output Automata

An *input-output automaton* A is defined by the following four components:

- A set of states, $states(A)$, (possibly an infinite set) with a subset of start states, $start(A)$.
- A set of actions, $acts(A)$, partitioned into sets of *input*, *output* and *internal* actions, $in(A)$, $out(A)$, and $int(A)$, respectively. The output and internal actions are called the *locally-controlled* actions, and the input and output actions are called *external* actions, denoted $ext(A)$.
- A transition relation $steps(A)$ is a set of (state,action,state) triples, such that for any state s' and input action π, there is a transition (s', π, s) for some state s.
- An equivalence relation $part(A)$ partitioning the locally-controlled actions of A. We interpret each class of the partition as the set of locally-controlled actions of separate, autonomous components of the system being modeled by the automaton.

An *execution* of A is a finite or infinite sequence $s_0\pi_1 s_1 \ldots$ of alternating states and actions such that s_0 is a start state, (s_{i-1}, π_i, s_i) is a transition of A for all i, and if e is finite then e ends with a state. The *schedule* of an execution is the subsequence of actions appearing in e. The *behavior* of a schedule or execution σ is the subsequence of external actions appearing in σ. An action π is *enabled* in state s' if there is a transition (s', π, s) for some state s; otherwise π is *disabled*. Since every input action is enabled in every state, automata are said to be *input-enabled*.

An execution of a system is fair if each component is given a chance to take a step infinitely often. Of course, a component can't take a step when given the chance if none of its actions are enabled. Formally, an execution e of automaton A is *fair* if for each class C of $part(A)$—that is, for each system component—the following two conditions hold:

- If e is finite, then no action of C is enabled in the final state of e.
- If e is infinite, then either actions from C appear infinitely often in e, or states in which no action of C is enabled appear infinitely often in e.

Automata can only be composed if their output actions are disjoint, and they do not share any internal actions. This restriction, together with the input-enabling condition, preserves the autonomy of independent components within a composition. To capture

this restriction we define the *action signature* of an automaton A, denoted $sig(A)$, to be the triple $(in(A), out(A), int(A))$. In general, an action signature S is a triple consisting of three disjoint sets $in(S)$, $out(S)$, and $int(S)$. The union of these sets is denoted by $acts(S)$.

The action signatures $\{S_i : i \in I\}$ are *compatible* if for all $i, j \in I$ $out(S_i) \cap out(S_j) = \emptyset$ and $int(S_i) \cap acts(S_j) = \emptyset$. The composition $S = \prod_{i \in I} S_i$ of compatible action signatures $\{S_i : i \in I\}$ is defined to be the action signature with $in(S) = \bigcup_{i \in I} in(S_i) - \bigcup_{i \in I} out(S_i)$, $out(S) = \bigcup_{i \in I} out(S_i)$, and $int(S) = \bigcup_{i \in I} int(S_i)$.

The composition $A = \prod_{i \in I} A_i$ of a set $\{A_i : i \in I\}$ of compatible automata (automata with compatible action signatures) is defined to be the automaton with

- $states(A) = \prod_{i \in I} states(A_i)$,
- $sig(A) = \prod_{i \in I} sig(A_i)$,
- $start(A) = \prod_{i \in I} start(A_i)$,
- $part(A) = \bigcup_{i \in I} part(A_i)$, and
- $steps(A)$ equal to the set of triples $(\{a_i\}, \pi, \{a_i'\})$ such that for all $i \in I$
  - if $\pi \in acts(A_i)$ then $(a_i, \pi, a_i') \in steps(A_i)$, and
  - if $\pi \notin acts(A_i)$ then $a_i = a_i'$.

(The products $states(A)$ and $start(A)$ are standard Cartesian products.) Since the automata A_i are input-enabled, so is their composition, and hence their composition is indeed an automaton. Notice that all output actions of an automaton A_i (some representing communication with other automata A_j) become output actions of the composition, and not internal actions. The definition of an operation internalizing output actions is straightforward. See [LT87, Tut87] for a more complete exposition of the model that includes such extensions.

3 Timed Automata

We introduce time into the model by introducing function t assigning times t_i to the states s_i appearing in executions $e = s_0 \pi_1 s_1 \ldots$; actually, t maps the indices i to times t_i. A *timing* t is a mapping from a nonempty (possibly infinite) prefix of $0, 1, 2, \ldots$ to the nonnegative reals satisfying

- t is nondecreasing: $i \leq j$ implies $t(i) \leq t(j)$
- t is unbounded: for every interval $[t_1, t_2]$ of the real line, $t(i) \in [t_1, t_2]$ for at most finitely many i.

The *length* of an execution e is the number of actions (and hence state transitions) appearing in e. The *length* of a timing t is k if t's domain is the finite set $\{0, \ldots, k\}$, and infinite if t's domain is the entire set of nonnegative integers.

A *timed execution* of an automaton A is an (untimed) execution e of A together with a timing t of the same length; we denote this timed execution by e^t. In other words, a timed execution is an execution together with a timing assigning times to states appearing in the execution. Notice that a timing also induces an assignment of times to actions. Intuitively,

since the action π_i is the cause of the (instantaneous) transition from state s_{i-1} to s_i, and since the system entered the state s_i at time $t(i)$, we can view the action π_i—or, perhaps more accurately, the completion of π_i—as having occurred at time $t(i)$. In fact, when $t(0) = 0$, it is convenient to represent the timed execution e^t by $s_0(\pi_1, t_1)s_1(\pi_2, t_2)s_2 \ldots$, where $t_i = t(i)$ (see [AL89]).

Timed schedules and behaviors of A are defined in a similar way. A *timed sequence* α^t consists of a sequence α of actions of A and a timing t of the same length, giving an initial time $t(0)$ and a time for each action in α. When $t(0) = 0$, it is convenient to denote the timed sequence α^t by $(\pi_1, t_1)(\pi_2, t_2)\ldots$, where $\alpha = \pi_1\pi_2\ldots$ and $t_i = t(i)$. A *timed schedule* of A is a timed sequence σ^t where σ is a schedule of A, and a *timed behavior* of A is a timed sequence β^t where β is a behavior of A.

A *timing property* P for an automaton A is any predicate on timed executions of A: given any timed execution e^t of A, the predicate P is either true or false of e^t. For example, a timing property could describe a desirable property that the timed executions of an automaton should exhibit.

A *timed automaton* is an ordered pair (A, P) consisting of an automaton A and a timing property P for A. Our intuition is that the automaton A describes the possible computations of the system, and the property P describes how these computations progress with time.

A *timed execution* of (A, P) is a timed execution e^t of A that satisfies P. We denote the set of timed executions of (A, P) by *timed-execs*(A, P). Given a timed execution e^t of (A, P), the timed schedule obtained by deleting the states appearing in e is denoted by $sched(e^t)$. For example, when $t(0) = 0$, if $e^t = s_0(\pi_1, t_1)s_1(\pi_2, t_2)\ldots$, then $sched(\sigma^t) = (\pi_1, t_1)(\pi_2, t_2)\ldots$. Similarly, given a timed schedule σ^t of (A, P), the timed behavior obtained by deleting the internal actions of A appearing in σ is denoted by $beh(\sigma^t)$. As a shorthand, we write $beh(e^t) = beh(sched(e^t))$. The set *timed-scheds*$(A, P)$ of timed schedules of (A, P) is the set of all timed schedules $sched(e^t)$ of all timed executions e^t of (A, P). Similarly, the set *timed-behs*(A, P) of timed behaviors of (A, P) is the set of all timed behaviors $beh(e^t)$ of all timed executions e^t of (A, P).

4 Composition of Timed Automata

Timed automata can be composed to yield other timed automata. Composition has the property that the behavior of a composition is a composition of the behaviors of the components. This compositionality is an important aspect of our model.

Like untimed automata, the composition of timed automata is defined only for compatible automata. Unlike untimed automata, however, composition is defined only for finite collections of automata. This guarantees that timings in the resulting composition are unbounded: if we try to compose an infinite collection of automata (A_i, P_i) where each P_i requires that an action is performed at time 1, then an infinite number of actions are performed at time 1 in an execution of the composition, violating the requirement that timings are unbounded. In this paper, compositions are assumed to be compositions of finite collections of compatible automata.

To motivate the definition of timed composition, we note that every execution e of an untimed composition $A = \prod A_i$ induces an execution $e|A_i$ of A_i: if $e = s_0\pi_1 s_1 \ldots$,

then $e|A_i$ is the result of deleting $\pi_j s_j$ whenever π_j is not an action of A_i and replacing the remaining global states s_j with A_i's local state $s_j|A_i$ in s_j. Intuitively, $e|A_i$ is the sequence of state transitions through which A_i moves during the execution e of A. Similarly, every *timed* execution e^t of A induces a *timed* execution $e^t|A_i$ of A_i: when $t(0) = 0$, if $e^t = s_0(\pi_1, t_1)s_1(\pi_2, t_2)\ldots$, then $e^t|A_i$ is the result of deleting $(\pi_j, t_j)s_j$ whenever π_j is not an action of A_i and replacing the remaining s_j with $s_j|A_i$. Given a timed sequence α^t of actions of A, the timed sequence $\alpha^t|A_i$ of actions of A_i is derived similarly.

The *composition* $\prod(A_i, P_i)$ of a finite collection of timed automata (A_i, P_i) is the timed automaton (A, P) where

- $A = \prod A_i$ is the composition of the A_i, and
- $P = \prod P_i$ is the timing property for A that is true of a timed execution e^t iff P_i is true of $e^t|A_i$ for every i.

Another way to formulate the definition of $\prod P_i$ is to extend each local property P_i to a global property, and then to define $\prod P_i$ to be the conjunction of the resulting global properties. More precisely, given a collection of timed automata (A_i, P_i), let $A = \prod A_i$ and define P_i^A to be the timing property for A defined as follows: a timed execution e^t of A satisfies P_i^A iff $e^t|A_i$ satisfies P_i. We now have the following:

Proposition 1: If $(A, P) = \prod(A_i, P_i)$, then $P \equiv \bigwedge P_i^A$.

It is interesting to explore the relationship between the global executions of $\prod(A_i, P_i)$ and the local executions of the (A_i, P_i). First of all, it is easy to see that every execution of $\prod(A_i, P_i)$ induces an execution of (A_i, P_i):

Proposition 2: Let $(A, P) = \prod(A_i, P_i)$. If e^t is a timed execution of (A, P), then $e^t|A_i$ is a timed execution of (A_i, P_i) for every i.

On the other hand, we can prove a kind of converse:

Proposition 3: Let $(A, P) = \prod(A_i, P_i)$, let e be any sequence of alternating states and actions of A, and let t be any timing of the same length. If $e^t|A_i$ is a timed execution of (A_i, P_i) for every i, then e^t is a timed execution of (A, P).

More generally, one might wonder when it is possible to take a collection of arbitrary timed executions $e_i^{t_i}$ of the (A_i, P_i) and "paste" them together to construct a timed execution e^t of the composition $\prod(A_i, P_i)$ such that $e^t|A_i = e_i^{t_i}$. In the case of untimed automata, if there is a total ordering α of the actions appearing in the e_i such that $\alpha|A_i = sched(e_i)$ for every i, then there is an execution e of $\prod A_i$ such that $\alpha = sched(e)$ and $e|A_i = e_i$ for every i. In the case of timed automata, the existence of a global timing t consistent with the local timings t_i is also required:

Proposition 4: Let $(A, P) = \prod(A_i, P_i)$, and suppose $e_i^{t_i}$ is a timed execution of (A_i, P_i) for every i. If there exists a timed sequence α^t of actions of A such that $\alpha^t|A_i = sched(e_i^{t_i})$ for every i, then there exists a timed execution e^t of (A, P) such that $\alpha^t = sched(e^t)$ and $e^t|A_i = e_i^{t_i}$ for every i.

Analogous results hold for schedules and behaviors:

Proposition 5: If e^t is a timed execution of $\prod(A_i, P_i)$, then $sched(e^t)|A_i = sched(e^t|A_i)$ and $beh(e^t)|A_i = beh(e^t|A_i)$.

Finally, we can use these results to prove the main result of this section: that our model is a compositional model of timed behavior. In other words, the observable behavior of a composition of timed automata is a composition of the observable behaviors of the component timed automata. First, we must define this composition of behaviors. Let $(A, P) = \prod(A_i, P_i)$, and define

$$\prod \textit{timed-execs}(A_i, P_i)$$

to be the set of e^t where e is a sequence of alternating states and actions of A and t is a timing of the same length such that $e^t|A_i \in \textit{timed-execs}(A_i, P_i)$ for each i. The definitions of $\prod$ *timed-scheds*(A_i, P_i) and $\prod$ *timed-behs*(A_i, P_i) are the obvious analogs. We can prove the following:

Proposition 6: If $(A, P) = \prod_{i \in I}(A_i, P_i)$, then

1. $\textit{timed-execs}(A, P) = \prod_{i \in I} \textit{timed-execs}(A_i, P_i)$,
2. $\textit{timed-scheds}(A, P) = \prod_{i \in I} \textit{timed-scheds}(A_i, P_i)$, and
3. $\textit{timed-behs}(A, P) = \prod_{i \in I} \textit{timed-behs}(A_i, P_i)$.

5 Timing Properties: Response Times

The idea of being "fair" to each component in a composition of automata comes up repeatedly in the theory of input-output automata. Informally, we view each class C in the partition of an automaton's locally-controlled actions as the locally-controlled actions of a single component in the system being modeled by the automaton. Being fair to each system component means being fair to each class of actions. This means each class is given an infinite number of chances to perform an action. On each chance, either some action of C is enabled and is performed, or no action of C is enabled and this class must pass on its chance to perform an action. More than just giving each class C an infinite number of chances to perform an action from C, we might require that the time between chances actually falls in some interval $\{l, u\}$. What we actually define is a bound on the elapsed time from the moment an action is enabled to the time it is performed. Since this is really a special case of bounding response times, the time that elapses between two events, we first define a simple notion for bounding response times, and return to bounded fairness in Section 6. This more general definition is useful in its own right when we are specifying desired response times at a level of abstraction where the ultimate partitioning of the system into components is not yet apparent (or desired).

To begin with an example, suppose one requires that the time elapsing between a request for a resource and the satisfaction of that request not exceed time ϵ. In order to be able to respond to requests in a timely manner, the system must be given the chance

(or time) to respond. For example, if a user is allowed to withdraw a request before it is fulfilled, then we might weaken our requirement to say that if a request remains unfulfilled for time ϵ, then it will be fulfilled within that time (that is, a request cannot remain unfulfilled for longer than time ϵ). We want to be able to capture statements of the form "if condition X holds for enough time, then condition Y becomes true." On the other hand, other considerations may require or depend upon certain response times taking more than a certain amount of time: "condition X must hold for enough time before condition Y becomes true." These considerations motivate us to formulate general notation for specifying upper and lower bounds on response time.

In our case, the conditions X and Y of interest are that the system is in a certain state or has performed a certain action. Let A be an automaton, let S be a subset of A's states and Π be a subset of A's actions. We denote by (S, Π) the event (or condition) corresponding to entering a state in S or performing an action in Π. Given an execution $e = s_0\pi_1 \ldots$, we denote the finite prefix $s_0\pi_1 \ldots s_k$ of e by $e[k]$. A finite prefix $e[k]$ *satisfies* (S, Π) iff

- $k = 0$ and $s_0 \in S$, or
- $k \geq 0$ and either $s_k \in S$ or $\pi_k \in \Pi$.

Intuitively, (S, Π) is true at time k if either the state entered at time k is in S or the action performed at time k (implying that $k > 0$) is in Π.

5.1 Upper Bounds on Response Times

Let A be an automaton, let S and S' be subsets of A's states, let Π and Π' be subsets of A's actions, and let $u > 0$ be a nonnegative real number. We say that a timed execution e^t of A satisfies the *upper bound*

$$(S, \Pi) \stackrel{\leq u}{\rightsquigarrow} (S', \Pi'),$$

which we read as "(S, Π) leads to (S', Π') in time at most u," iff for every $i \geq 0$,

> if $e[i]$ satisfies (S, Π)
> then, for some $j > i$ with $t(j) \leq t(i) + u$,
>
> > either $e[j]$ satisfies (S', Π') or $e[j]$ does not satisfy (S, Π).

If t is a strictly increasing function, meaning that successive states are assigned distinct times, then this condition is equivalent to saying that if (S, Π) is continuously true for the next u time units, then (S', Π') becomes true within the next u time units.

For notational convenience, we often omit reference to a set S or Π when it is empty, and we denote a singleton set $\{x\}$ by x. For example, we write $S \stackrel{\leq u}{\rightsquigarrow} \Pi'$ in place of $(S, \Pi) \stackrel{\leq u}{\rightsquigarrow} (S', \Pi')$ when Π and S' are empty. As another example, notice that if $\mathit{enabled}(\pi)$ is the set of states where the action is π is enabled, then $\mathit{enabled}(\pi) \stackrel{\leq \epsilon}{\rightsquigarrow} \pi$ says that if action π is continuously enabled for the next ϵ time units, then π is performed within time ϵ.

In a similar manner, we say that e^t satisfies the *strict upper bound*

$$(S, \Pi) \stackrel{< u}{\rightsquigarrow} (S', \Pi')$$

just as above, except that we replace the condition $t(j) \leq t(i) + u$ with $t(j) < t(i) + u$. Notice that when $u = \infty$, this strict upper bound requires that (S, Π) cannot be true forever without (S', Π') becoming true, although there is no finite bound on the delay until this event occurs. In contrast, we find it convenient to define $(S, \Pi) \lesssim^{\infty}_{\rightarrow} (S', \Pi')$ to be true for any S, S', Π and Π'. With this convention, these two conditions allow us to express as extreme cases the classes of fair and unfair executions of an automaton, respectively. Finally, as one would expect, increasing the upper bound u weakens the conditions $(S, \Pi) \lesssim^{u}_{\rightarrow} (S', \Pi')$ and $(S, \Pi) \lesssim^{u}_{\rightarrow} (S', \Pi')$.

5.2 Lower Bounds on Response Times

Consider an execution e in which π is continuously enabled. In this case, the upper bound $\mathit{enabled}(\pi) \lesssim^{\epsilon}_{\rightarrow} \pi$ says that π will be performed at least once every ϵ time units, and it seems that a lower bound $\mathit{enabled}(\pi) \gtrsim^{\epsilon}_{\rightarrow} \pi$ ought to say that π will be performed at *most* once every ϵ time units. Consider, however, an execution e in which π is intermittently enabled. In this case, the upper bound $\mathit{enabled}(\pi) \lesssim^{\epsilon}_{\rightarrow} \pi$ says that π cannot remain enabled for more than ϵ time units without being performed, and it seems that a lower bound $\mathit{enabled}(\pi) \gtrsim^{\epsilon}_{\rightarrow} \pi$ ought to say that π must be enabled at *least* ϵ time units before being performed. Combining these remarks, $\mathit{enabled}(\pi) \gtrsim^{\epsilon}_{\rightarrow} \pi$ should mean that π must be enabled at least ϵ time units between performances.

A natural way of capturing this intuition is to say that a timed execution e^t of A satisfies the lower bound $(S, \Pi) \gtrsim^{l}_{\rightarrow} (S', \Pi')$ iff for every $j > 0$,

> if $e[j]$ satisfies (S', Π')
> then for some $i < j$ with $t(i) \leq t(j) - l$
>
> > $e[k]$ satisfies (S, Π) for all k with $i \leq k < j$, and
> > $e[k]$ does *not* satisfy (S', Π') for any k with $i < k < j$.

Consider once again the condition $\mathit{enabled}(\pi) \gtrsim^{\epsilon}_{\rightarrow} \pi$. Given an execution e^t with a strictly increasing timing t (meaning that each state is assigned a distinct time), this condition says that in order for π to be performed at time τ, it must be enabled throughout the time interval $[\tau - \epsilon, \tau)$, and must not be performed in the interval $(\tau - \epsilon, \tau)$. Notice, for example, that it is perfectly acceptable for π to be performed at both times $\tau - \epsilon$ and τ, as long as π is enabled throughout the intervening interval (and, in particular, in the state at time $\tau - \epsilon$ immediately following the first performance of π).

While this definition is sufficient for our definition of bounded fairness in Section 6, it does have one weakness that it easy to repair: it says that (S, Π) must hold for l time units before (S', Π') can hold, but suppose there are two, independently timed paths by which (S', Π') might become true. For example, consider an automaton with a single output action *response* and two independent input actions, *fast-request* and *slow-request*, that enable the *response* action. The automaton has three states, including an initial state *start*. The input actions *fast-request* and *slow-request* take each state to the states *fast* and *slow*, respectively, and the output action *response* takes both *fast* and *slow* to *start* again. Intuitively, *fast-request* and *slow-request* are high- and low-priority requests, respectively, that *response* be performed: the delay between *fast-request* and *response* is to be at least 5 time units, while the delay between *slow-request* and *response* is to be at

least 10 time units. We want to say that *response* may be performed only if the automaton has been in the state *fast* for 5 time units or in the state *slow* for 10 time units, but the definition of a lower bound given above does not let us express this in a natural way. This is because it does not allow us to distinguish the performance of *response* via the state *fast* from the performance of *response* via the state *slow*.

Such examples have led us to the following definition of a lower bound. Given a nonnegative real number l, we say that a timed execution e^t of A satisfies the *lower bound*

$$(S, \Pi) \stackrel{\geq l}{\rightsquigarrow} (S', \Pi'),$$

which we read as "(S, Π) leads to (S', Π') in time at least l," iff for every $j > 0$,

> if $e[j-1]$ satisfies (S, Π) and $e[j]$ satisfies (S', Π')
> then for some $i < j$ with $t(i) \leq t(j) - l$
>
> > $e[k]$ satisfies (S, Π) for all k with $i \leq k < j$, and
> > $e[k]$ does *not* satisfy (S', Π') for any k with $i < k < j$.

This definition recognizes the fact that (S', Π') may become true via several computational paths, and says that if it becomes true via the path satisfying (S, Π), then (S, Π) must have been satisfied for the preceding l time units. Returning to the example above, notice that this definition of a lower bound allows us to express the different timing requirements with the conditions *fast* $\stackrel{\geq 5}{\rightsquigarrow}$ *response* and *slow* $\stackrel{\geq 10}{\rightsquigarrow}$ *response*.

Similarly, we say that e^t satisfies the *strict lower bound*

$$(S, \Pi) \stackrel{> l}{\rightsquigarrow} (S', \Pi')$$

just as above, except that we replace the condition $t(i) \leq t(j) - l$ with $t(i) < t(j) - l$. Again, decreasing the lower bound l weakens the conditions $(S, \Pi) \stackrel{\geq l}{\rightsquigarrow} (S', \Pi')$ and $(S, \Pi) \stackrel{> l}{\rightsquigarrow} (S', \Pi')$.

5.3 Combining Upper and Lower Bounds

We can combine the upper and lower bound conditions given above into a single condition as follows. We define the timing property

$$(S, \Pi) \stackrel{[l,u]}{\rightsquigarrow} (S', \Pi')$$

to be the conjunction of the timing properties $(S, \Pi) \stackrel{\geq l}{\rightsquigarrow} (S', \Pi)$ and $(S, \Pi) \stackrel{\leq u}{\rightsquigarrow} (S', \Pi)$; that is, a timed execution must satisfy both the upper and lower bounds. For example, the condition $\mathit{enabled}(\pi) \stackrel{[l,u]}{\rightsquigarrow} \pi$ says that π must be enabled at least l time units between performances of π, and that π cannot remain enabled from longer that u time units without being performed. Notice, by the way, that since the conditions $(S, \Pi) \stackrel{\leq \infty}{\rightsquigarrow} (S', \Pi')$ and $(S, \Pi) \stackrel{\geq 0}{\rightsquigarrow} (S', \Pi')$ are equivalent to *true* (that is, they are valid), the condition $(S, \Pi) \stackrel{\leq u}{\rightsquigarrow} (S', \Pi')$ is equivalent to $(S, \Pi) \stackrel{[0,u]}{\rightsquigarrow} (S', \Pi')$, and the condition $(S, \Pi) \stackrel{\geq l}{\rightsquigarrow} (S', \Pi')$ is equivalent to $(S, \Pi) \stackrel{[l,\infty]}{\rightsquigarrow} (S', \Pi')$. We note that, in an analogous way, we can define the conditions $(S, \Pi) \stackrel{(l,u]}{\rightsquigarrow} (S', \Pi)$, $(S, \Pi) \stackrel{(l,u)}{\rightsquigarrow} (S', \Pi)$, and $(S, \Pi) \stackrel{[l,u)}{\rightsquigarrow} (S', \Pi)$. We use $\{l, u\}$ to denote any one of these intervals when its open or closed nature is unimportant. As expected, enlarging the interval $\{l, u\}$ weakens the the timing property.

Given a collection of timing properties P_i for automata A_i, Proposition 1 says that the timing property $P = \prod P_i$ for $A = \prod A_i$ can be viewed as the conjunction $\wedge P_i^A$ of timing properties for A, where each P_i^A is the extension of the local property P_i for A_i to a global property for A. The following proposition shows how to perform this extension for the upper and lower bounds defined in this section.

Proposition 7: Let $(A, P) = \prod(A_i, P_i)$. Let S and S' be subsets of A_i's states, and let Π and Π' be subsets of A_i's actions. Define

$$S^A = \{s \in \mathit{states}(A) : s|A_i \in S\} \text{ and}$$

$$S'^A = \{s \in \mathit{states}(A) : s|A_i \in S'\}.$$

If $P_i \equiv (S, \Pi) \overset{\{l,u\}}{\rightsquigarrow} (S', \Pi')$, then $P_i^A \equiv (S^A, \Pi) \overset{\{l,u\}}{\rightsquigarrow} (S'^A, \Pi')$.

6 Timing Properties: Boundmaps

With the notation just defined, it is now easy to capture our notion of bounded fairness, the notion that a class C is given an infinite number of chances to perform an action, and that the time between chances actually falls in some interval $\{l, u\}$. Given an automaton A, a *boundmap* b for A is a mapping that maps each class C of $\mathit{part}(A)$ to an interval $b(C) = \{l(C), u(C)\}$ of the real line. Given an automaton A and a class C of $\mathit{part}(A)$, we denote by $\mathit{enabled}(A, C)$ (or just $\mathit{enabled}(C)$ when A is clear from context) the set of A's states in which some action of C is enabled. We often abuse notation and denote by b both the boundmap b and the timing property

$$P_b \overset{def}{=} \bigwedge_{C \in \mathit{part}(A)} \mathit{enabled}(C) \overset{b(C)}{\rightsquigarrow} C.$$

We refer to (A, b) as a *time-bounded automaton*, a special case of a timed-automaton.

Given that the definition of a boundmap is motivated by the definition of a fair execution, it is not surprising that the fair executions of an automaton A can be characterized as the timed executions of a timed automaton (A, b) with a fair boundmap b. The *fair boundmap* of A is the boundmap b defined by $b(C) = [0, \infty)$ for all $C \in \mathit{part}(A)$. Notice that if C is continuously enabled from some point of an execution, then this boundmap requires the *eventual* performance of an action in C, since some action of C must be performed before time ∞.

Proposition 8: Let A be an automaton, and let b be the fair boundmap for A. Given any timed execution e^t of A, e is a fair execution of A iff e^t is a timed execution of (A, b).

Similarly, we define the *unfair boundmap* of A as the boundmap b defined by $b(C) = [0, \infty]$ for all $C \in \mathit{part}(A)$; the following is immediate.

Proposition 9: Let A be an automaton, and let b be the unfair boundmap for A. Given any timed execution e^t of A, e is an execution of A iff e^t is a timed execution of (A, b).

These results show that the classes of fair and unfair computations can be understood in terms of extreme cases of boundmaps.

There is a very simple relationship between the boundmap of a composition of timed automata and the boundmaps of the individual component automata.

Proposition 10: Let $A = \prod A_i$. Suppose b_i is a boundmap for A_i for each i, and suppose b is a boundmap for A defined by $b(C) = b_i(C)$ if $C \in part(A_i)$. Then $(A, b) = \prod(A_i, b_i)$.

Given this result, we define $\prod(A_i, b_i)$ to be (A, b) where b is the boundmap defined as stated in this proposition. This result together with Proposition 6 shows that our model is a compositional model of time-bounded fair computation. Again, we can view the composition $\prod(A_i, b_i)$ in terms of extending local timing properties to global properties:

Proposition 11: Suppose $(A, b) = \prod(A_i, b_i)$. If $P_C \equiv enabled(A_i, C) \overset{b_i(C)}{\rightsquigarrow} C$, then $P_C^A \equiv enabled(A, C) \overset{b(C)}{\rightsquigarrow} C$.

7 Solvability

In addition to describing implementations of concurrent systems, input-output automata are useful for expressing specifications of such systems [LT87]. Accordingly, given two (untimed) automata A and A', we say that A *solves* A' if they have the same external actions—that is, $in(A) = in(A')$ and $out(A) = out(A')$—and every fair behavior of A is a fair behavior of A'. Intuitively, the fair behaviors of A are the behaviors that can be witnessed by an external observer of A—someone who cannot see the inner workings of A, its internal actions. Since every behavior of A is a behavior of A', any correctness condition satisfied by the behaviors of A' is satisfied by the behaviors of A as well. In particular, any problem "solved" by A' is also "solved" by A.

The definition of solvability has a natural extension to timed automata: given two timed automata (A, P) and (A', P'), we say that (A, P) *solves* (A', P') if they have the same external actions and $timed\text{-}behs(A, P) \subseteq timed\text{-}behs(A', P')$. As with the untimed case, solvability for timed automata has properties that support hierarchical and modular verification techniques. For example, an immediate result of the definition is that "solves" is a transitive relation:

Proposition 12: If (A, P) solves (A', P') and (A', P') solves (A'', P''), then (A, P) solves (A'', P'').

One consequence of this result is that we can prove that an implementation (A, P) satisfies its specification (A', P') by constructing a sequence of intervening models $(A, P) = (A_0, P_0), \ldots, (A_k, P_k) = (A', P')$ and proving that (A_i, P_i) solves (A_{i+1}, P_{i+1}) for every i. This means that hierarchical proof strategies are possible in this model, where each (A_i, P_i) is a model of the system at increasingly higher levels of conceptual abstraction. Elsewhere [LA90], refinement mappings have been used to construct this sort of hierarchical proof in this model.

Notice that if P is in some sense a stronger timing property than P', then is should immediately follow that (A, P) solves (A, P'). Unfortunately, it is difficult to give general

syntactic conditions on timing properties P and P' that imply that P is stronger than P'. In the case of boundmaps, however, such a characterization is quite simple. Given two boundmaps b and b' with the same domain (that is, b and b' are defined on the same sets of classes C), we define $b \subseteq b'$ if $b(C) \subseteq b'(C)$ for all C. Intuitively b makes stronger requirements than b'. It is easy to see that A with the stronger boundmap b solves A with a weaker boundmap b':

Proposition 13: For any automaton A, if $b \subseteq b'$ then (A, b) solves (A, b').

Finally, since—like the untimed input-output model—this model of timed computation is a compositional model, one technique for proving that one composition of timed automata solves another composition is to prove that each component of the first composition solves the corresponding component of the second:

Proposition 14: Suppose $(A, P) = \Pi_{i \in I}(A_i, P_i)$ and $(A', P') = \Pi_{i \in I}(A'_i, P'_i)$. If (A_i, P_i) solves (A'_i, P'_i) for every $i \in I$, then (A, P) solves (A', P').

8 Conclusion

We have presented a model for reasoning about time in concurrent systems. Our decision to base the model on the input-output automaton model was motivated by (in our judgment) the naturalness and utility of the model in the context of asynchronous concurrent systems. The model has been used extensively to model concurrency control and recovery in transaction systems, resource allocation, concurrent data structures, network communication, and other problems (e.g., [LT89, LM88, LMWF88, Blo87, WLL88, LMF88, Her88]). It has been used to specify these problems, to describe and analyze algorithmic solutions, and to prove lower bounds and impossibility results. The model has many natural properties (such as compositionality), and this work was motivated by our desire to find an equally intuitive generalization to real-time concurrent systems. The simple definition of a timed execution results in a modular, compositional model of timed concurrent computation; and in the definition of a boundmap we feel we have found the natural generalization of both fair and unfair executions of the input-output automaton model. We note that our goal has been only to devise a natural semantic model of timed computation. We have not considered logics for expressing general timing properties—although we feel the "leads to" notation does cover a lot of the interesting timing constraints—nor have we considered proof systems for such logics. It appears, however, that our model is a suitable semantic model for most logics and proofs systems appearing in the literature (such as [HLP90, AH90, ACD90]). However, simple proof techniques for timed automata have already been investigated [LA90].

References

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE, June 1990.

[AH90] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 390–401. IEEE, June 1990.

[AL89] Hagit Attiya and Nancy Lynch. Time bounds for real-time process control in the presence of timing uncertainty. Technical Memo MIT/LCS/TM-403, MIT Laboratory for Computer Science, July 1989.

[Blo87] Bard Bloom. Constructing two-writer atomic registers. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 249–259. ACM, August 1987.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[Dil88] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, Department of Computer Science, Carnegie Mellon University, February 1988. Available as Technical Report CMU-CS-88-119.

[Fra86] Nissim Francez. *Fairness.* Springer-Verlag, Berlin, 1986.

[GL90] Richard Gerber and Insup Lee. CCSR: A calculus for communicating shared resources. In J. C. M. Baeten and J. W. Klop, editors, *Lecture Notes in Computer Science, volume 458, Proceedings of Concur '90*, pages 263–277. Springer-Verlag, August 1990.

[Her88] Maurice Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290. ACM, August 1988.

[HLP90] Eyal Harel, Orna Lichtenstein, and Amir Pnueli. Explicit clock temporal logic. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 401–413. IEEE, June 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[Jon87] Bengt Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Uppsala University, Uppsala, Sweden, 1987. Published by Direkt Offset, Nyström & Co AB, Uppsala.

[KSdR+88] R. Koymans, R. K. Shyamasundar, W. P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79:210–256, 1988.

[LA90] Nancy A. Lynch and Hagit Attiya. Using mappings to prove timing properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 265–280. ACM, August 1990.

[Lam91] Leslie Lamport. A temporal logic of actions. Research Report 57, DEC Systems Research Center, January 1991.

[Lew90] Harry R. Lewis. A logic of concrete time intervals. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 380–389. IEEE, June 1990. Also available at Harvard Technical Report TR-07-90.

[LF81] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13(1):17–43, January 1981.

[LM88] Nancy A. Lynch and Michael Merritt. Introduction to the theory of nested transactions. *Theoretical Computer Science*, 62:123–185, 1988. Earlier versions appeared in *Proceedings of the International Conference on Database Theory*, 1986, and as MIT Technical Report MIT/LCS/TR-367.

[LMF88] Nancy A. Lynch, Yishay Mansour, and Alan Fekete. Data link layer: Two impossibility results. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 149–170. ACM, August 1988. Also available as MIT Technical Report MIT/LCS/TM-355.

[LMWF88] Nancy A. Lynch, Michael Merritt, William E. Weihl, and Alan Fekete. A theory of atomic transactions. In *Proceedings of the International Conference on Database Theory*, 1988. Also available as MIT Technical Memo MIT/LCS/TM-362.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR–387.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989. Also available as MIT Technical Memo MIT/LCS/TM-373.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag, Berlin, 1980.

[MT90] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In J. C. M. Baeten and J. W. Klop, editors, *Lecture Notes in Computer Science, volume 458, Proceedings of Concur '90*, pages 401–415. Springer-Verlag, August 1990.

[SL87] A. Udaya Shankar and Simon S. Lam. Time-dependent distributed systems: Proving safety, liveness and real-time properties. *Distributed Computing*, pages 61–79, 1987.

[Tut87] Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1987. Available as MIT Technical Report MIT/LCS/TR-387.

[WLL88] Jennifer L. Welch, Leslie Lamport, and Nancy A. Lynch. A lattice-structured proof of a minimum spanning tree algorithm. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 28–43. ACM, August 1988.

[Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In J. C. M. Baeten and J. W. Klop, editors, *Lecture Notes in Computer Science, volume 458, Proceedings of Concur '90*, pages 502–520. Springer-Verlag, August 1990.