LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-412.e

USING MAPPINGS TO PROVE TIMING PROPERTIES

(Replacing TM 412.d)

Nancy Lynch
Hagit Attiya

April 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Using Mappings to Prove Timing Properties*

Nancy A. Lynch† and Hagit Attiya‡

April 1, 1992

†Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge, MA 02139
‡Department of Computer Science, The Technion, Haifa 32000, ISRAEL
*This work was supported by ONR contracts N00014-85-K-0168 and N00014-91-J-1046, by NSF grants CCR-8611442 and CCR-8915206, and by DARPA contracts N00014-87-K-0825 and N00014-89-J-1988.

Abstract

A new technique for proving timing properties for timing-based algorithms is described; it is an extension of the mapping techniques previously used in proofs of safety properties for asynchronous concurrent systems. The key to the method is a way of representing a system with timing constraints as an automaton whose state includes predictive timing information. Timing assumptions and timing requirements for the system are both represented in this way. A multi-valued mapping from the "assumptions automaton" to the "requirements automaton" is then used to show that the given system satisfies the requirements. One type of mapping is based on a collection of "progress functions" providing measures of progress toward timing goals. The technique is illustrated with two examples, a simple resource manager and a two-process race system.

Keywords: Timing properties, timing-based algorithms, formal specification, formal verification, assertional reasoning, possibilities mappings, timed automata, I/O automata, progress functions.

1 Introduction

Assertional reasoning is a useful technique for proving safety properties of sequential and concurrent algorithms. This proof method involves describing the algorithm of interest as a state machine, and defining a predicate known as an *assertion* on the states of the machine. One proves inductively that the assertion is true of all the states that are reachable in a computation of the machine, *i.e.*, that it is an *invariant* of the machine. The assertion is defined so that it implies the safety property to be proved. Assertional reasoning is a rigorous, simple and general proof technique. Furthermore, the assertions usually provide an intuitively appealing explanation of *why* the algorithm satisfies the property.

One kind of assertional reasoning uses a mapping to describe a correspondence between the given algorithm and a higher-level algorithm used as a specification of correctness. (See, for example, [17, 21, 25].) Such mappings may be single-valued or multi-valued.

So far, assertional reasoning has been used primarily to prove properties of sequential algorithms and synchronous and asynchronous concurrent algorithms. We would also like to use this technique to prove properties of concurrent algorithms whose operation depends on time, *e.g.*, ones that arise in real-time systems or ones that rely on clocks that tick at approximately known rates. Also, the kinds of properties generally proved using assertional reasoning have been "ordinary" safety properties; we would like to use similar methods to prove timing properties (upper and lower bounds on time) for algorithms that have timing assumptions. Predictable performance is often a desirable characteristic of real-time systems [40]; assertional techniques could be very helpful in proving such performance properties.

In this paper, we describe one way in which assertional reasoning can be used to prove timing properties for algorithms that have timing assumptions. Our method involves constructing a multi-valued mapping from an automaton representing the given algorithm to another automaton representing the timing requirements. The key to our method is a way of representing a system with timing constraints as an automaton whose state includes predictive timing information. Timing assumptions and timing requirements for the system are both represented in this way, and the mappings we construct map from the "assumptions automaton" to the "requirements automaton". One type of mapping is based on a collection of "progress functions" providing measures of progress toward timing goals.

We describe our method in terms of the *timed automaton* model, a slight variant of the *time constrained automaton* model of [29]. We use this model to state the requirements to be satisfied, to define the basic architectural and timing assumptions, to describe the algorithms, and to prove their correctness and timing properties. A timed automaton is a pair (A, b), consisting of an *I/O automaton* [25, 26], A, together with a *boundmap*, b, which is a formal description of the timing assumptions for the components of the system. A timed automaton generates a set of *timed executions* which describe the operation of the algorithm, and a corresponding set of *timed behaviors* which describe the algorithm's externally-visible activity. In this paper, a timed automaton (A, b) is used to describe the given system (including its timing

assumptions), and another timed automaton (A', b') is used to describe the correctness and timing requirements.

While convenient for specifying timing assumptions and requirements, timed automata are not directly suited for carrying out assertional proofs about timing properties, because timing properties are described externally (by boundmaps) rather than being built into the automaton itself. We therefore introduce a way of incorporating timing conditions into an automaton definition. For a given timed automaton (A, b), we define the automaton $time(A, b)$ to be an ordinary I/O automaton (not a timed automaton) whose state includes predictive information describing the first and last times at which various events can next occur; this information is designed to enforce the timing conditions expressed by the boundmap b. The I/O automaton $time(A, b)$ is related to the timed automaton (A, b) in that a certain subset of the behaviors of $time(A, b)$, which we call the "admissible" behaviors, is exactly equal to the set of timed behaviors of (A, b).

We apply this construction to both the system description (A, b) and the requirements description (A', b'); our "assumptions automaton" is defined to be $time(A, b)$ and our "requirements automaton" is $time(A', b')$. Then the problem of showing that a given algorithm (A, b) satisfies the timing requirements amounts to that of showing that any admissible behavior of the automaton $time(A, b)$ is also an admissible behavior of $time(A', b')$. We do this by using invariant assertion techniques; in particular, we demonstrate a multi-valued mapping from states of $time(A, b)$ to states of $time(A', b')$.

We define a special class of multi-valued mappings that appears to be especially useful. Each such mapping is defined by a collection of inequalities relating the time bounds to be proved (those expressed by b') to the values of a collection of "progress functions" defined on the states of $time(A, b)$. These progress functions provide upper and lower bound measures of progress toward the timing goals expressed by b'. These functions generalize the notion of progress function commonly used to prove termination of sequential programs and asynchronous concurrent programs (see, *e.g.*, the description of the method of well-founded sets in [28]), to allow real-valued rather than just discrete measures, and to allow proofs of lower bounds as well as upper bounds.

In order to demonstrate the use of our technique, we apply it to two examples. The first example is a simple timing-dependent resource granting system, consisting of two concurrently-operating components, a *clock* and a *manager*. The manager monitors the clock ticks, which occur at an approximately known rate, and whenever a certain number have occurred, it grants the resource. We prove upper and lower bounds on the amount of time prior to the first grant and between each successive pair of grants.

The second example involves one process incrementing a counter until another process modifies a flag, and then decrementing the counter. When the counter reaches 0, the first process announces that it is done. We show upper and lower bounds on the time until the "done" announcement occurs.

Technically, mapping techniques of the sort used in this paper are only capable of proving safety properties, but not liveness properties. Timing properties have aspects of both safety

and liveness. A timing lower bound asserts that an event cannot occur before a certain amount of time has elapsed; a violation of this property is detectable after a finite prefix of a timed execution, and so a timing lower bound can be regarded as a safety property. A timing upper bound asserts that an event must occur before a certain amount of time has elapsed. This can be regarded as making two separate claims: that the designated amount of time does in fact elapse (a liveness property), and that this amount of time cannot elapse without the event having occurred (a safety property). In this paper, we assume the liveness property that time increases without bound, so that all the remaining properties that need to be proved in order to prove either upper or lower time bounds are safety properties. Thus, our mapping technique provides complete proofs for timing properties without requiring any additional techniques for arguing liveness.

There has been some other work on using assertional reasoning to prove timing properties. In particular, Haase [10], Hooman [13], Shankar and Lam [37], Tel [41], Schneider [36], Lewis [19], Abadi and Lamport [2, 18], Lamport and Neumann [31] and Shaw [38] have all developed models for timing-based systems that incorporate time information into the state. In [41] and [19], in fact, the information that is included is similar to ours in that it is also predictive timing information (although not exactly the same information as ours). Some of the work mentioned above has used invariant assertions to prove timing properties; however, none of this work is based on mappings.

Lynch and Vaandrager [27] describe a wide range of mapping proof techniques for timing-based systems, in the setting of a very general timed automaton model. One of the techniques considered there, *forward simulation*, is very similar to our general multi-valued mapping method. However, the model in [27] has less structure than the one considered here; in particular, it lacks the component structure that is needed to describe our progress function technique.

Several other, quite different formal approaches to proving timing properties have also been developed, based on state machines (*e.g.*, [9]), model-checking (*e.g.*, [19], [4]), ω-automata (*e.g.*, [5]) first-order logic (*e.g.*, [14, 15]), temporal logic (*e.g.*, [3, 7, 11, 32, 34, 12]), Petri nets (*e.g.*, [8, 39]) and process algebras (*e.g.*, [16, 42]). (See the survey by Ostroff [33].)

An earlier version of this paper appears in [23].

The rest of the paper is organized as follows. Section 2 contains a description of the underlying formal models: I/O automata and timed automata. Section 3 contains the construction used to incorporate timing conditions into I/O automata, and some basic properties of these automata. Section 4 contains our definitions for mappings and for collections of progress functions, and shows that the existence of such mappings and collections imply that a given algorithm satisfies a given set of timing requirements. Section 5 contains our examples, the simple resource-granting system and the two-process race system. For each of these examples, this section contains a description of the system, a description of the corresponding requirements automaton, and a correctness proof using mappings. We conclude with a discussion in Section 6.

2 Formal Model

In this section, we present the definitions for the underlying formal model. In particular, we define I/O automata, timed automata and timing conditions. We also present some of their relevant properties.

2.1 I/O Automata

We begin by summarizing some of the key definitions for the I/O automaton model. We refer the reader to [25, 26] for a complete presentation of the model and its properties.

An *I/O automaton*, A, consists of the following pieces: $acts(A)$, a set of *actions*, classified as *output*, *input* and *internal* (input and output actions are called *external*); $states(A)$, a set of *states*, including a distinguished subset, $start(A)$, of *start states*; $steps(A)$, a set of steps, where a *step* is defined to be a (*state, action, state*) triple; and $part(A)$, a *partition* of the locally controlled (output and internal) actions into equivalence classes; the partition groups together actions that are to be thought of as under the control of the same underlying process.

An action π is said to be *enabled* in a state s' provided that there is a step of the form (s', π, s). An automaton is required to be *input enabled*, which means that every input action must be enabled in every state. For any set $\Pi \subseteq acts(A)$, we denote by $enabled(A, \Pi)$ the set of states of A in which some action in Π is enabled, and by $disabled(A, \Pi)$ be the set of all states of A not in $enabled(A, \Pi)$, that is, $disabled(A, \Pi) = states(A) \setminus enabled(A, \Pi)$. We use the term *event* to refer to an occurrence of an action in a sequence.

An *execution fragment* of an I/O automaton A is a sequence (finite or infinite) of alternating states and actions

$$s_0, \pi_1, s_1, \ldots, s_{i+1}, \pi_i, s_i, \ldots$$

where for every i, $(s_{i-1}, \pi_i, s_i) \in steps(A)$. (If the sequence is finite, then it is required to end with a state.) An *execution* is an execution fragment with $s_0 \in start(A)$. The *schedule* of an execution α is the subsequence of α consisting of all the events appearing in α, and the *behavior* of α is the subsequence consisting of all the external events. The *schedules* and *behaviors* of A are just those of the executions of A. An *extended step* is a triple (s', β, s) for which there is an execution fragment that starts and ends with s' and s, respectively, and whose schedule is β.

Concurrent systems are modeled by compositions of I/O automata, as defined in [25, 26]. In order to be composed, automata must be *strongly compatible*; this means that no action can be an output of more than one component, that internal actions of one component are not shared by any other component, and that no action is shared by infinitely many components. The result of such a composition is another I/O automaton. The *hiding* operator can be applied to reclassify output actions as internal actions.

2.2 Timed Automata

In this subsection, we augment the I/O automaton model to allow discussion of timing properties. The treatment here is similar to the one described in [6] and is a special case of the definitions proposed in [29]. A *boundmap* for an I/O automaton A is a a mapping that associates a closed subinterval of $[0, \infty]$ with each class in $part(A)$, where the lower bound of each interval is not ∞ and the upper bound is nonzero.[1] Intuitively, the interval associated with a class C by the boundmap represents the range of possible lengths of time between successive times when C "gets a chance" to perform an action. We sometimes use the notation $b_\ell(C)$ to denote the lower bound assigned by boundmap b to class C, and $b_u(C)$ for the corresponding upper bound. A *timed automaton* is a pair (A, b), where A is an I/O automaton and b is a boundmap for A.

We require notions of "timed execution", "timed schedule" and "timed behavior" for timed automata, corresponding to executions, schedules and behaviors for ordinary I/O automata. These will all include time information. We begin by defining the basic type of sequence that underlies the definition of a timed execution.

Definition 2.1 *A* timed sequence *(for an I/O automaton A) is a (finite or infinite) sequence of alternating states and (action,time) pairs,*

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

satisfying the following conditions.

1. *The states s_0, s_1, ... are in $states(A)$.*
2. *The actions π_1, π_2,... are in $acts(A)$.*
3. *The times t_1, t_2,... are successively nondecreasing nonnegative real numbers.*
4. *If the sequence is finite, then it ends in a state s_i.*
5. *If the sequence is infinite then the times are unbounded.*

For a given timed sequence, we use the convention that $t_0 = 0$. For any finite timed sequence α, we define $endtime(\alpha)$ to be the time of the last event in α, if α contains any (action,time) pairs, or 0, if α contains no such pairs. Also, we define $endstate(\alpha)$ to be the last state in α. We denote by $ord(\alpha)$ (the "ordinary" part of α) the sequence

$$s_0, \pi_1, s_1, \pi_2, \ldots ,$$

[1] In [29], the model is defined in a more general manner, to allow boundmaps to yield open or semi-open intervals as well as closed intervals. This restriction is not crucial in this paper, but allows us to avoid considering extra cases in some of the technical arguments.

i.e., α with time information removed.

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

Definition 2.2 *Suppose (A, b) is a timed automaton. Then a timed sequence α is a* timed execution *of (A, b) provided that $ord(\alpha)$ is an execution of A and α satisfies the following conditions, for each class $C \in part(A)$ and every initial index i for C in α.*

1. *If $b_u(C) < \infty$ then there exists $j > i$ with $t_j \leq t_i + b_u(C)$ such that either $\pi_j \in C$ or $s_j \in disabled(A, C)$.*
2. *There does not exist $j > i$ with $t_j < t_i + b_\ell(C)$ and π_j in C.*

The first condition says that, starting from an initial index for C, within time $b_u(C)$ either some action in C occurs or there is a point at which no such action is enabled. Note that if $b_u(C) = \infty$, no upper bound requirement is imposed. The second condition says that, again starting from an initial index for C, no action in C can occur before time $b_\ell(C)$ has elapsed. Note in particular that if a class C becomes disabled and then enabled once again, the lower bound calculation gets "restarted" at the point where the class becomes re-enabled.

The *timed schedule* of a timed execution of a timed automaton (A, b) is the subsequence consisting of the (action,time) pairs, and the *timed behavior* is the subsequence consisting of the (action,time) pairs for which the action is external. The *timed schedules* and *timed behaviors* of (A, b) are just those of the timed executions of (A, b).

We model each timing-dependent concurrent system as a single timed automaton (A, b), where A is a composition of ordinary I/O automata (possibly with some output actions hidden).[2] We also model problem specifications, including timing properties, in terms of timed automata.

We note that the definition we use for timed automata may not be sufficiently general to capture all interesting systems and timing requirements. It does capture many, however; we discuss this further in Section 6.

3 Incorporating Timing Conditions into I/O Automata

In order to use invariant assertion techniques to reason about timed automata, we define an ordinary I/O automaton $time(A, b)$ corresponding to a given timed automaton (A, b). This

[2] An equivalent way of looking at each system is as a composition of timed automata. An appropriate definition for a composition of timed automata is developed in [29], together with theorems showing the equivalence of the two viewpoints.

new automaton has the timing restrictions imposed by b on A built into its transition rules, based on predictions about when the next event from each set of actions will occur. In this section, we give the construction of $time(A, b)$ and also give results that relate the executions and behaviors of $time(A, b)$ to the timed executions and timed behaviors of (A, b).

The close relationships between (A, b) and $time(A, b)$ suggest the possibility of avoiding the timed automaton definition entirely, instead using the $time(A, b)$ notion as the starting point for our work. We prefer to begin with the timed automaton definition because we regard that definition as the more fundamental of the two, expressed as it is in terms of a traditional asynchronous system with some additional timing restrictions. As will be seen below, the $time(A, b)$ definition introduces special constructs (*e.g.*, special $NULL$ actions and special variables such as *now*), which are quite useful in proofs, but which do not seem to be fundamental parts of system descriptions. Another reason we prefer to begin with the timed automaton definition is that it has already been used elsewhere ([29, 6]). Moreover, we believe that the elegant relationship between the two expressed by Theorem 3.1 is interesting in its own right.

3.1 Definition of $time(A, b)$

Given any timed automaton (A, b), we define the ordinary I/O automaton $time(A, b)$. The automaton $time(A, b)$ has as its actions all pairs of the form (π, t), where π is an element of $acts(A) \cup \{NULL\}$ and t is a nonnegative real number; here $NULL$ is a "null action" that represents the passage of time. The classification of actions into input, output and internal actions is derived from that for A, with the additional stipulation that each $(NULL, t)$ is an internal action. (The $NULL$ action is similar to the unit action, 1, of SCCS [30] and to the time-passage actions of [27].) Each of the states of $time(A, b)$ consists of a state, *basic*, of A, augmented with a variable *now*, and, for each class C of the partition of A, two variables $first(C)$ and $last(C)$. The value of the *now* variable represents the time of the last preceding event. The values of the $first(C)$ and $last(C)$ variables represent, respectively, the first and last times at which an event in class C is permitted to occur.

We use record notation to denote the various components of the state of $time(A, b)$: for instance, $s.basic$ denotes the state of A included in state s of $time(A, b)$. Each start state of $time(A, b)$ consists of a start state $s.basic$ of A, plus $now = 0$, plus values of $first(C)$ and $last(C)$ with the following property: if there is an action in C enabled in s, then $s.first(C) = b_l(C)$ and $s.last(C) = b_u(C)$; otherwise, $s.first(C) = 0$ and $s.last(C) = \infty$. That is, if the start state of A has an action in C enabled, then the predicted times are the ones specified in the boundmap for C; otherwise, they are set to default values.

If (π, t) is an action of $time(A, b)$, then $(s', (\pi, t), s)$ is defined to be a step of $time(A, b)$ exactly if all of the following conditions hold.

1. If $\pi \in acts(A)$ then:

(a) $s'.now = t = s.now$.

(b) $(s'.basic, \pi, s.basic) \in steps(A)$.

(c) For each $C \in part(A)$:

i. If $\pi \in C$ then $s'.first(C) \leq t$.

ii. If $s.basic \in enabled(A, C)$ and $\pi \notin C$ and $s'.basic \in enabled(A, C)$ then $s.first(C) = s'.first(C)$ and $s.last(C) = s'.last(C)$.

iii. If $s.basic \in enabled(A, C)$ and either $\pi \in C$ or $s'.basic \in disabled(A, C)$ then $s.first(C) = t + b_l(C)$ and and $s.last(C) = t + b_u(C)$.

iv. If $s.basic \in disabled(A, C)$, then $s.first(C) = 0$ and $s.last(C) = \infty$.

2. If $\pi = NULL$ then

(a) $s'.now \leq t = s.now$.

(b) $s.basic = s'.basic$.

(c) $t \leq s'.last(C)$, for each $C \in part(A)$.

(d) $s.first(C) = s'.first(C)$ and $s.last(C) = s'.last(C)$, for each $C \in part(A)$.

The meaning of these conditions is as follows. Condition 1 describes restrictions for the case where π is an action of A. Condition 1(a) says that time does not pass during the performance of non-null actions, and Condition 1(b) says that the steps associated with non-null actions correctly simulate steps of A. Condition 1(c) describes the use and manipulation of the *first* and *last* variables during non-null steps. Condition 1(c)i says that a locally controlled step is only permitted to occur at a time that is at least as great as the first time specified for that action's partition class. Condition 1(c)ii says that an action not in a particular class that keeps the class enabled does not alter the timing predictions for that class. Condition 1(c)iii says that an action that enables a particular class sets the timing predictions for that class to the values specified by the boundmap. Finally, Condition 1(c)iv says that an action that leaves a particular class disabled sets the timing predictions to the default values.

Similarly, Condition 2 describes restrictions for the case where π is the special null action. Condition 2(a) says that time cannot move backwards when a null action is performed, and Condition 2(b) says that the steps associated with null actions do not cause any changes to the underlying state of A. Condition 2(c) says that time cannot pass beyond the latest time specified for any class, and Condition 2(d) says that timing predictions are unaltered by the passage of time.

It is easy to check that for any reachable state of $time(A, b)$ and any class C of the partition, the following facts are true. First, it must be the case that $s.last(C) \geq s.now$ (although it is possible to have $s.first(C) < s.now$). Second, if $s.basic \in enabled(A, C)$ then $s.last(C) \leq s.now + b_u(C)$ and $s.first \leq s.now + b_l(C)$. Third, if $s.basic \in disabled(A, C)$ then both the $last(C)$ and $first(C)$ variables have their default values (∞ and 0, respectively).

The partition classes for $time(A, b)$ are derived one-for-one from those of A, with the addition of a single new class for all the $(NULL, t)$ actions.[3] Note that a similar automaton was defined in [6, 23]; it differs in not containing special "null" actions.

We will be particularly interested in a subset of the executions of $time(A, b)$ that we call the "admissible executions". Informally, the admissible executions are those in which time continues to pass without bound.

Definition 3.1 *An execution of $time(A, b)$ is said to be* admissible *provided the times associated with the NULL events in the execution are unbounded. The* admissible schedules *and* admissible behaviors *of $time(A, b)$ are defined to be the schedules and behaviors, respectively, of admissible executions of $time(A, b)$.*

Note that any admissible execution must have infinitely many *NULL* events, in order that the associated times might be unbounded. In each of our examples in this paper, we will apply the $time(A, b)$ construction to a timed automaton A modeling the entire system under consideration.

3.2 Relationship between (A, b) and $time(A, b)$

In this subsection, we relate a timed automaton (A, b) to the corresponding I/O automaton $time(A, b)$; specifically, we prove the following main theorem, Theorem 3.1, which relates the timed behaviors of (A, b) and the admissible behaviors of $time(A, b)$. (Note that both behaviors are sequences of pairs of the form (π, t), where π is an action and t is a time.)

Theorem 3.1 *The set of timed behaviors of (A, b) is the same as the set of admissible behaviors of $time(A, b)$.*

This theorem implies that properties of timed behaviors of a timed automaton (A, b) can be proved by proving them about the set of admissible behaviors of the corresponding I/O automaton $time(A, b)$. The latter task is more amenable to treatment using assertional techniques.

The rest of this subsection is devoted to proving Theorem 3.1. The concepts and lemmas used in this proof are not needed outside of the proof, so the reader may wish to skip the rest of this subsection on a first reading.

First, the definition of a timed execution contains aspects of both safety and liveness. In the proof, it is useful to focus first on the safety aspects alone. We thus define the notion of a "timed semi-execution" to capture the safety part of the definition of a timed execution.

[3] We will not need these classes in this paper, however, since the purpose of I/O automaton partition classes is to enforce fairness to the components of the system, and we will not require such fairness conditions.

Definition 3.2 *Suppose (A, b) is a timed automaton. Then a finite timed sequence α is a* timed semi-execution *of (A, b) provided that $ord(\alpha)$ is an execution of A and α satisfies the following conditions, for each class C of $part(A)$ and every initial index i for C in α.*

1. *If $b_u(C) < \infty$ then either $endtime(\alpha) \leq t_i + b_u(C)$ or there exists $j > i$ with $t_j \leq t_i + b_u(C)$ such that either $\pi_j \in C$ or $s_j \in disabled(A, C)$.*
2. *There does not exist $j > i$ with $t_j < t_i + b_\ell(C)$ and π_j in C.*

This definition is identical to that of a finite timed execution (Definition 2.2), except for the "either" clause in the first item. This clause allows an action to fail to occur if insufficient time has passed by the end of the execution. (Recall that *endtime*(α) refers to the time of the last event in α.) We prove two technical lemmas about the properties of timed semi-executions. The first lemma gives a condition on a timed semi-execution that ensures that it is a timed execution.

Lemma 3.2 *Suppose that α is a timed semi-execution of a timed automaton (A, b). Then α is a timed execution if and only if each locally controlled action of A that is enabled in state $endstate(\alpha)$ is in a partition class C in $part(A)$ such that $b_u(C) = \infty$.*

Proof: Straightforward. ∎

The second lemma says that the limit of a sequence of timed semi-executions in which the times are unbounded must be a timed execution.

Lemma 3.3 *Let $\{\alpha_i\}_{i=1}^{\infty}$ be a sequence of timed semi-executions of (A, b) such that the following conditions hold.*

1. *For any $i \geq 1$, α_i is a prefix of α_{i+1}.*
2. *$\lim_{i \to \infty} endtime(\alpha_i) = \infty$.*

Then the limit of the α_i under the prefix ordering is a timed execution of (A, b)

Proof: Straightforward. ∎

We now show a simple correspondence between the timed semi-executions of (A, b) and the finite executions of $time(A, b)$. We require an auxiliary definition. Namely, if α is an execution of $time(A, b)$, we define $project(\alpha)$ to be the timed sequence obtained from α by mapping each occurrence of a state s in α to $s.basic$ while keeping the (action,time) pairs intact, and then removing any *NULL* events, together with their immediately following states.

Lemma 3.4 *Let (A, b) be a timed automaton.*

1. *If α' is a timed semi-execution of (A, b), then there exists a finite execution α of $time(A, b)$ such that $\alpha' = project(\alpha)$.*
2. *If α is a finite execution of $time(A, b)$, then $project(\alpha)$ is a timed semi-execution of (A, b).*

Proof: 1. Suppose that α' is a timed semi-execution of (A, b). First we construct α'', an alternating sequence of states of A and actions of $time(A, b)$, by inserting exactly one $NULL$ event before the first event in α' and between every pair of events in α'; more precisely, if s and (π, t) occur consecutively in α', then α'' replaces this pair with the sequence $s, (NULL, t), s, (\pi, t)$. (The reason we need to insert the $NULL$ events is that they are the only kinds of events of $time(A, b)$ that allow time to pass.)

Now we modify α'' to obtain α, a finite sequence of alternating states and actions of $time(A, b)$, by adding *now*, *last* and *first* variables to all the states in α'. We do this in the unique way that guarantees that the first state is a start state of $time(A, b)$ and that Conditions 1(a), 1(c)ii-iv, 2(a) and 2(d) of the definition of $time(A, b)$ are satisfied. Then $\alpha' = project(\alpha)$. We show that α is an execution of $time(A, b)$ by showing that each step of α satisfies the remaining conditions of the definition of $time(A, b)$.

The fact that α' is a timed semi-execution of (A, b) implies Condition 1(b), and Condition 2(b) holds by construction. Condition 1 of Definition 3.2 ensures Condition 2(c) of the definition of $time(A, b)$, while Condition 2 of Definition 3.2 ensures Condition 1(c)i of the definition of $time(A, b)$.

2. Let $\alpha' = project(\alpha)$. By Conditions 1(b) and 2(b) of the definition of $time(A, b)$, $ord(\alpha')$ is an execution of the ordinary I/O automaton A. It remains to show that for every class C, α' satisfies Conditions 1 and 2 of Definition 3.2 for C (and every $i \geq 0$).

The initialization and Condition 1(c)iii of the definition of $time(A, b)$ imply that the correct upper bounds are assigned to the $last(C)$ variable whenever C becomes enabled, and Conditions 1(c)ii and 2(d) imply that those bounds do not change until an action in C occurs or C becomes disabled. Condition 2(c) then implies that the upper bounds are respected, which implies Condition 1 of Definition 3.2 for C. Similarly, the initialization and Condition 1(c)iii imply that the correct lower bounds are assigned to the $first(C)$ variable whenever C becomes enabled, and Conditions 1(c)ii and 2(d) imply that those bounds do not change until an action in C occurs or C becomes disabled. Condition 1(c)i then implies that the lower bound is respected, which implies Condition 2 of Definition 3.2 for C.

■

Next, we show a correspondence between the timed executions of (A, b) and the admissible executions of $time(A, b)$.

Lemma 3.5 *1. If α' is a timed execution of (A, b), then there exists an admissible execution α of $time(A, b)$ such that $\alpha' = project(\alpha)$.*

2. If α is an admissible execution of $time(A, b)$, then $project(\alpha)$ is a timed execution of (A, b).

Proof: 1. Suppose α' is a timed execution of (A, b). We carry out a similar construction to that in Part 1 of Lemma 3.4, except that if α' is finite, we augment α with an infinite suffix of *NULL* actions, associated with times that increase without bound. The argument is similar to before; the main difference is that we must argue that that Condition 2(c) of the definitions of $time(A, b)$ is not violated by the trailing *NULL* events. More specifically, if α' is finite, then since it is a timed execution, Lemma 3.2 implies that each locally controlled action that is enabled in state $endstate(\alpha')$ is in a partition class C with $b_u(C) = \infty$. Then the definition of $time(A, b)$ implies that $last(C) = \infty$ for each $C \in part(A)$, in the state of α just prior to each of the trailing *NULL* events. This implies that the trailing *NULL* events cannot cause violations of 2(c).

2. Suppose that $\alpha = s_0, (\pi_1, t_1), s_1, \ldots$ is an admissible execution of $time(A, b)$, and let $\alpha' = project(\alpha)$. Let α_i be the prefix of α ending with s_i, and let $\alpha'_i = project(\alpha_i)$, for each $i \geq 0$. Then each α'_i is a prefix of α'_{i+1}, and α' is the limit of the α'_i under the prefix ordering. Since α_i is a finite execution of $time(A, b)$, Part 2 of Lemma 3.4 implies that α'_i is a timed semi-execution of (A, b), for each $i \geq 0$. We consider two cases.

   First, suppose α' is infinite. Then α does not have a suffix consisting entirely of *NULL* events. Since the times of the actions in α are unbounded, and α does not have a suffix consisting entirely of *NULL* events, it follows that $\lim_{i \to \infty} endtime(\alpha'_i) = \infty$. Then Lemma 3.3 implies that α' is a timed execution of (A, b).

   Second, suppose that α' is finite. Then α has a suffix consisting entirely of *NULL* events, say starting after s_j, for some fixed j, and $\alpha' = \alpha'_j$. As argued above, α_j is a timed semi-execution of (A, b), so α' is a timed semi-execution of (A, b). Condition 2(c) of the $time(A, b)$ definition and the fact that times increase without bound in α imply that each locally controlled action of A that is enabled in state $s_j.basic$ is in a partition class C in $part(A)$ such that $b_u(C) = \infty$. Since $endstate(\alpha') = s_j.basic$, Lemma 3.2 implies that α' is a timed execution of (A, b).

■

Proof: (of Theorem 3.1) Immediate by Lemma 3.5. ■

4 Sufficient Conditions for Inclusion of Timed Behavior Sets

In this section, we describe a method for showing that the timed behaviors of one timed automaton, (A, b), are also timed behaviors of another timed automaton, (A', b'). This method

uses the construction in Section 3; *i.e.*, it involves showing that the admissible behaviors of $time(A, b)$ are also admissible behaviors of $time(A', b')$. As we describe in Subsection 4.1, our basic method involves mapping states of $time(A, b)$ to sets of states of $time(A', b')$ and is a special case of the *possibilities mapping* method described in [25, 26].

In the examples later in this paper (as well as others to which we have applied this mapping method), the mappings that are constructed are expressible in a particular form: in terms of inequalities involving the values of the state variables of the $time(A, b)$ and $time(A', b')$ automata. In particular, these inequalities assert that the value of each $last(C)$ variable of $time(A', b')$ is at least as great as a certain real-valued "progress function" of the values of the state variables of $time(A, b)$, and also that the value of each $first(C)$ variable of $time(A', b')$ is no greater than another such function. These functions can be thought of as measures of progress of the system $time(A, b)$ toward the goals of producing events from the various partition classes C of $time(A', b')$. In Subsection 4.2, we define our notion of progress function and show how they can be used to generate correct mappings.

Our notion of progress function is similar to the notion of progress function commonly used to prove liveness properties of sequential and asynchronous concurrent programs (*e.g.*, in [28]); however, our notion generalizes the usual notion in that ours allows real-valued rather than just discrete measures, and that ours applies to lower bounds as well as upper bounds.

4.1 Strong Possibilities Mappings

In this subsection, we define the notion of a *strong possibilities mapping* from an automaton of the form $time(A, b)$ to another automaton $time(A', b')$.[4] We then prove our basic theorem about strong possibilities mappings, namely, that the existence of such a mapping implies that the timed behaviors of (A, b) are all timed behaviors of (A', b').

Recall from Section 2.1 the definition of an *extended step* of an arbitrary I/O automaton.

Definition 4.1 *Let (A, b) and (A', b') be timed automata with the same set Π of external actions. Let f be a mapping from states of $time(A, b)$ to sets of states of $time(A', b')$. The mapping f is a* strong possibilities mapping *from $time(A, b)$ to $time(A', b')$ provided that the following conditions hold:*

1. *For every start state s of $time(A, b)$, there is a start state u of $time(A', b')$ such that $u \in f(s)$.*
2. *If s' is a reachable state of $time(A, b)$, $u' \in f(s')$ is a reachable state of $time(A', b')$ and $(s', (\pi, t), s)$ is a step of $time(A, b)$, then there is an extended step (u', β, u) of $time(A', b')$, such that $u \in f(s)$ and $\beta|(\Pi \times \Re) = (\pi, t)|(\Pi \times \Re)$.*[5]

[4] This is a strengthened version of the definition of "possibilities mapping" in [26], where the strengthening involves the addition of the third condition. The term "possibilities" is used to suggest the different possible states in an image set. An alternative formulation is in terms of relations rather than mappings, as is described in [27].

[5] We use the notation $\Re$ in this paper to represent the nonnegative real numbers.

3. *If s and u are reachable states of $time(A, b)$ and $time(A', b')$, respectively, and $u \in f(s)$, then $u.now = s.now$.*

The first condition in the mapping definition establishes a correspondence between start states of the two automata, while the second condition establishes a correspondence between steps of $time(A, b)$ and extended steps (as defined in Section 2.1) of $time(A', b')$; this correspondence must preserve the sequences of timed external events. The third condition simply asserts that the current times of corresponding states must be identical.

The following key lemma says that the existence of a strong possibilities mapping is a sufficient condition for the inclusion of admissible behaviors.

Lemma 4.1 *Suppose that there is a strong possibilities mapping from $time(A, b)$ to $time(A', b')$. Then any admissible behavior of $time(A, b)$ is an admissible behavior of $time(A', b')$.*

Proof: Let β be an admissible behavior of $time(A, b)$, and let α be an admissible execution of $time(A, b)$ whose behavior is β. For each finite prefix α_i of α that ends with a state, it is possible to construct a finite execution, α'_i, of $time(A', b')$ having the same behavior as α_i and such that the values of the *now* variables of the final states of both executions are identical. Moreover, it is possible to do this in such a way that each α'_i is a prefix of α'_{i+1}. (The construction is by induction on i, using Conditions 1 and 2 of Definition 4.1.) Let α' be the limit of the α'_i; then α' is an execution of $time(A', b')$, and the behavior of α' is the same as the behavior of α, which is β.

Since α is admissible, the values of the *now* variables of the final states of the α_i increase without bound as i approaches infinity. Since the values of the *now* variables are the same in the final states of α_i and α'_i, the values of the *now* variables of the final states of the α'_i also increase without bound as i approaches infinity. It follows that α' is an admissible execution of $time(A', b')$ with behavior β. Thus, β is an admissible behavior of $time(A', b')$. ■

Now we give the main theorem of this subsection, which expresses the basic mapping technique for timed automata.

Theorem 4.2 *Suppose that there is a strong possibilities mapping from $time(A, b)$ to $time(A', b')$. Then any timed behavior of (A, b) is a timed behavior of (A', b').*

Proof: Immediate from Lemma 4.1 and Theorem 3.1. ■

This theorem says that the existence of a strong possibilities mapping is sufficient by itself to yield the desired inclusion result for timed behaviors. Since the timed behaviors of a timed automaton embody both safety and liveness restrictions, it follows that this mapping technique suffices to show both types of properties. This is in contrast to the situation for non-timed

systems, where analogous mapping techniques only yield safety properties. (In [1], for example, extra machinery in the form of a "supplementary property" is added to the mapping machinery in order to allow proofs of liveness properties.)

Lynch and Vaandrager [27] generalize our Lemma 4.1 to the setting of a more general and abstract timed automaton model. However, there is no corollary analogous to our Theorem 4.2 in that paper; also, the model in [27] lacks the partition class structure of the model of this paper, which is needed to describe the progress function technique we describe in the following subsection.

4.2 Progress Function Collections

In this subsection, we define our notion of progress functions and show how they can be used to generate strong possibilities mappings.

The progress function definition is presented in terms of a pair of timed automata, (A, b) and (A', b'), where (A, b) describes the system under study and (A', b') describes the requirements to be satisfied. The underlying automaton, A', of (A', b') is used to describe correctness requirements that do not involve time, whereas the boundmap b' is used to describe timing requirements; more specifically, b' specifies upper and lower bounds for various kinds of events to occur, where each "kind of event" corresponds to a partition class C of A'. Thus, for each class C, the definition mentions one progress function ub_C to describe progress toward guaranteeing the upper bound requirement given by $b'_u(C)$, and another progress function lb_C to describe progress toward guaranteeing the lower bound requirement given by $b'_l(C)$. Each of these progress functions is a function from the state of automaton $time(A, b)$ to $\Re \cup \infty$. Along with the functions ub_C and lb_C, the definition also uses another function $\hat{f}$ that describes a correspondence between states of the underlying automata A and A'.[6] The various conditions in the definition assert that the function $\hat{f}$ is a correct correspondence between states of A and A', and that the functions ub_C and lb_C provide correct measures of progress toward their respective goals.

We caution the reader that this definition is somewhat technical. One aspect that may seem confusing is that it is based on a mixture of the two styles of definition, $time(A, b)$ versus (A', b'). However, note that the mixture is completely consistent, always using the $time(A, b)$ definition at the lower level and the (A', b') at the higher level. The $time(A, b)$ definition is used at the lower level because the progress measures are naturally defined in terms of states of $time(A, b)$ (in particular, in terms of the values of the *first* and *last* variables). On the other hand, the (A', b') definition is used at the higher level because it permits decomposition of the properties that need to be shown to demonstrate the existence of a strong possibilities mapping into very small pieces.

In Section 5, we verify timing properties for two examples using progress functions. We note that it is possible to avoid the progress function definition entirely, and verify correctness

[6] This function could also be replaced by a multi-valued mapping, but this causes notational complications we thought it best to avoid.

and timing properties for our examples directly from Theorem 4.2. (In fact, that is how similar proofs are carried out in the preliminary version of this paper [23].) However, examination of our proofs based on Theorem 4.2 shows that they all use the notion of progress function implicitly. This subsection is our attempt to make this strategy explicit.

Definition 4.2 *Let (A, b) and (A', b') be timed automata with the same set Π of external actions. Let $\hat{f}$ be a mapping from states of time(A, b) to states of A'. For each $C \in part(A')$, let ub_C and lb_C be mappings from states of time(A, b) to $\Re \cup \infty$. Then the collection of mappings $(\hat{f}, (ub_C, lb_C)_{C \in part(A')})$ is a* progress function collection *from (A, b) to (A', b') provided that the following conditions hold:*

1. *If s is a start state of time(A, b) and $v = \hat{f}(s)$, then v is a start state of A'. Moreover, for each $C \in part(A')$ such that $v \in enabled(A', C)$, we have $ub_C(s) \leq b'_u(C)$ and $lb_C(s) \geq b'_\ell(C)$.*
2. *Suppose s' is a reachable state of time(A, b) and $(s', (\pi, t), s)$ is a step of time(A, b), where $\pi \neq NULL$. Suppose $v' = \hat{f}(s')$, $v = \hat{f}(s)$, and v' is a reachable state of A'. Then there is an execution fragment α of A' beginning and ending with v' and v respectively, such that:*
   (a) $\alpha|\Pi = \pi|\Pi$.
   (b) *For each $C \in part(A')$:*
      i. *If $b'_\ell(C) > 0$ and a C event occurs in α, then there is only one C event in α, all states occurring in α prior to the C event are in enabled(A', C) and $t \geq lb_C(s')$.*
      ii. *If all states in α are in enabled(A', C) and if no C events occur in α then $ub_C(s) \leq ub_C(s')$ and $lb_C(s) \geq lb_C(s')$.*
      iii. *If $v \in enabled(A', C)$, and if either there is a state in α in disabled(A', C) or if a C event occurs in α, then $ub_C(s) \leq t + b'_u(C)$ and $lb_C(s) \geq t + b'_\ell(C)$.*
3. *Suppose s' is a reachable state of time(A, b) and $(s', (NULL, t), s)$ is a step of time(A, b). Suppose $v' = \hat{f}(s')$, $v = \hat{f}(s)$, and v' is a reachable state of A'. Then:*
   (a) $v' = v$.
   (b) *For each $C \in part(A')$:*
      i. $t \leq ub_C(s')$.
      ii. $ub_C(s) \leq ub_C(s')$ *and* $lb_C(s) \geq lb_C(s')$.

The meaning of these conditions is as follows. Condition 1 asserts that any start state s of *time*(A, b) corresponds to a start state of A'; moreover, the value for each progress function in state s is defined in an appropriate way to enable proof of the desired bound. For example, consider the upper bound requirement for class C, as specified by the boundmap value $b'_u(C)$. If class C is enabled in state v and remains enabled, then we will wish to prove that some

action in C will occur by time at most $b'_u(C)$. In order to use the progress function ub_C as a progress measure to prove this upper bound, we require that the initial value of ub_C should be no greater than the bound $b'_u(C)$ to be proved.

Condition 2 asserts that each non-null step of $time(A, b)$ has a corresponding execution fragment of A' satisfying certain properties. Condition 2(a) says that the execution fragment exhibits the same external behavior as the given step, while Condition 2(b) says that the values of the progress function are handled appropriately to enable proof of the desired bounds. Condition 2(b)i says that each progress function lb_C does in fact describe a lower bound on the time by which an action in C may occur. If the lower bound specified by the boundmap b' for C is 0, then there is nothing to show for this condition; if it is nonzero, then a C event should only occur if the time at which it occurs is at least as great as the time $lb_C(s')$. However, there is a technicality that arises in this condition: recall that the lower bound requirement for C is restarted whenever C becomes enabled or a C event occurs. This means that a violation of the lower bound requirement given by $b'_l(C)$ could occur in the given execution fragment if class C becomes enabled in the fragment or a C event occurs, and then a subsequent event of C occurs; even though the time for this C event is at least $lb_C(s')$, that time might not be sufficiently great to satisfy the restarted lower bound requirement. In order to cope with this troublesome situation, we simply rule out this pattern from the execution fragments we consider.

Condition 2(b)ii simply says that the progress functions are maintained properly when no relevant steps occur; for example, consider the upper bound requirement for class C. If no events in C occur and C remains enabled, then the progress function used as a progress measure for C's upper bound may decrease, but it should not be allowed to increase. Finally, Condition 2(b)iii says that the progress functions are restarted properly when a class C becomes enabled or when an event in C occurs. The considerations are analogous to those for proper initialization.

Condition 3 describes what must happen whan a null step of $time(A, b)$ occurs. Condition 3(a) says that a null step does not change the state of A'. Condition 3(b)i says that each progress function ub_C does in fact describe an upper bound on the time by which an action in C must occur. That is, if the system $time(A, b)$ is in state s', then it is not permissible for time to pass beyond time $ub_C(s')$ without some action in C occurring. Condition 3(b)ii is similar to Condition 2(b)ii, in that it says that the progress functions are maintained properly when nothing of interest occurs.

We now show how progress function collections can be used to generate strong possibilities mappings. Let $(\hat{f}, (ub_C, lb_C)_{C \in part(A')})$ be a progress function collection from (A, b) to (A', b'). Then we define a mapping f from states of $time(A, b)$ to sets of states of $time(A', b')$ by: $u \in f(s)$ iff

1. $u.basic = \hat{f}(s)$,

2. $u.now = s.now$,

3. $u.last(C) \geq ub_C(s)$ for each $C \in part(A')$, and

4. $u.first(C) \leq lb_C(s)$ for each $C \in part(A')$.

The next lemma shows that f is a strong possibilities mapping.

Lemma 4.3 *Suppose that (A, b) and (A', b') are timed automata with the same set of external actions, and suppose that $(\hat{f}, (ub_C, lb_C)_{C \in part(A')})$ is a progress function collection from (A, b) to (A', b'). Let f be the corresponding mapping defined just above. Then f is a strong possibilities mapping from $time(A, b)$ to $time(A', b')$.*

Proof: We show the three conditions of Definition 4.1. Condition 3 is immediate by definition.

For Condition 1, let s be a start state of $time(A, b)$. Then Condition 1 of Definition 4.2 yields a start state v of A' such that $v = \hat{f}(s)$ and, for all $C \in part(A')$, if $v \in enabled(A', C)$ then $ub_C(s) \leq b'_u(C)$ and $lb_C(s) \geq b'_l(C)$. Define u to be the (unique) start state of $time(A', b')$ having $u.basic = v$. By definition of the start states of $time(A', b')$, it follows that $u.now = 0 = s.now$, $u.last(C) = b'_u(C)$ if $v \in enabled(A', C)$ and $u.last(C) = \infty$ otherwise, and $u.first(C) = b'_l(C)$ if $v \in enabled(A', C)$ and $u.first(C) = 0$ otherwise. Then we have $u.basic = v = \hat{f}(s)$, $u.now = s.now$, and $u.last(C) \geq ub_C(s)$ and $u.first(C) \leq lb_C(s)$ for all C, which implies that $u \in f(s)$, as needed.

Now we show Condition 2 of Definition 4.1. Let Π be the common set of external actions for (A, b) and (A', b'). Suppose that s' is a reachable state of $time(A, b)$, $u' \in f(s')$ is a reachable state of $time(A', b')$, and $(s', (\pi, t), s)$ is a step of $time(A, b)$. Since $u' \in f(s')$, it follows that $u'.basic = \hat{f}(s')$, $u'.now = s'.now$, and $u'.last(C) \geq ub_C(s')$ and $u'.first(C) \leq lb_C(s')$ for all $C \in part(A')$. Also, since u' is a reachable state of $time(A', b')$, it follows that $u'.basic$ is a reachable state of A'.

We consider two cases:

1. $\pi \neq NULL$.

   Then Condition 2 of Definition 4.2 yields an execution fragment α of A' with the properties detailed in that definition. We modify α to obtain an execution fragment α' of $time(A', b')$, by using the same sequence of events as in α, associating time t with each event, and filling in the values of the *now*, *last* and *first* variables as determined by the definition of $time(A', b')$.

   In order to show that the resulting α' is an execution fragment of $time(A', b')$, we must argue that the designated times of events are within the bounds allowed by the definition of $time(A', b')$. The only interesting condition to show is Condition 1(c)i of the definition of $time(A', b')$, for a class C that has $b'_l(C) > 0$: we must show that if any action in such a class C occurs in α', then $u''.first(C) \leq t$, where u'' is the state of $time(A', b')$ just prior

to that C event. By Condition 2(b)i of Definition 4.2, there is only one C event in α, and all states in α prior to the given C event are in $enabled(A', C)$; by the definition of $time(A', b')$, this implies that $u''.first(C) = u'.first(C)$. Condition 2(b)i of Definition 4.2 also implies that $t \geq lb_C(s')$; since $u'.first(C) \leq lb_C(s')$, this implies that $u'.first(C) \leq t$, so that $u''.first(C) \leq t$, as needed.

Now we define the extended step (u', β, u) of $time(A', b')$ that arises from α'; that is, u is the last state in α' and β is the schedule of α'. We show that this extended step satisfies the conditions required in Definition 4.1. First, we must show that $u \in f(s)$, that is, that $u.basic = \hat{f}(s)$, $u.now = s.now$, and that $u.last(C) \geq ub_C(s)$ and $u.first(C) \leq lb_C(s)$ for all C. But $u.basic = \hat{f}(s)$ by the definition of α, and $u.now = t = s.now$, showing the first two of these conditions. To see that $u.last(C) \geq ub_C(s)$, note that $u'.last(C) \geq ub_C(s')$ since $u' \in f(s')$; Conditions 2(b)ii and 2(b)iii of Definition 4.2 and the definition of $time(A, b)$ then imply the needed inequality. A similar argument holds for the lower bound condition.

Also, since $\alpha|\Pi = \pi|\Pi$, it follows that $\beta|\Pi \times \Re = (\pi, t)|\Pi \times \Re$. Thus, Condition 2 of Definition 4.1 is satisfied.

2. $\pi = NULL$.

Define state u of $time(A', b')$ to be the same as state u', except that $u.now = t$. We claim that $(u', (NULL, t), u)$ is the required extended step of $time(A', b')$.

First, we argue that $(u', (NULL, t), u)$ is a step of $time(A', b')$. By definition of $time(A', b')$, the only interesting condition to check is that $t \leq u'.last(C)$ for all $C \in part(A')$. So fix $C \in part(A')$. Condition 3(b)i of Definition 4.2 implies that $t \leq ub_C(s')$; since $u'.last(C) \geq ub_C(s')$, we have $t \leq u'.last(C)$, as needed.

Now we check the remaining requirements for Condition 2 of Definition 4.1. The correspondence between external action sequences is easy to see. We argue that $u \in f(s)$. Since $u.basic = u'.basic$, $\hat{f}(s) = \hat{f}(s')$ (by Condition 3(a) of Definition 4.2), and $u'.basic = \hat{f}(s')$, it follows that $u.basic = \hat{f}(s)$. Also, $u.now = t = s.now$. Let $C \in part(A')$. Then $u.last(C) = u'.last(C) \geq ub_C(s')$, and $ub_C(s') \geq ub_C(s)$ by Condition 3(b)ii of Definition 4.2. Therefore, $u.last(C) \geq ub_C(s)$. A similar argument shows that $u.first(C) \leq lb_C(s)$. Therefore, Condition 2 of Definition 4.1 holds, as needed.

■

Now we give the main theorem about progress function collections, saying that their existence implies timed behavior inclusion.

Theorem 4.4 *Suppose that (A, b) and (A', b') are timed automata with the same set of external actions. If there exists a progress function collection from (A, b) to (A', b'), then every timed behavior of (A, b) is a timed behavior of (A', b').*

Proof: By Lemma 4.3 and Theorem 4.2. ■

5 Examples

In this section, we present two examples for which we prove time upper and lower bounds using our mapping techniques, (in particular, using progress function collections).

5.1 Resource Manager

Our first example is a simple resource-granting system adapted from an algorithm in [6]. The system consists of two components, a *clock* and a *manager*. The clock ticks at an approximately-predictable rate, and the manager counts ticks in order to decide when to grant a resource. We wish to analyze the time until the first grant, and the time between each successive pair of grants.

We describe the algorithm and its timing assumptions as a timed automaton (A, b). The required timing behavior is presented as a timed automaton (A', b'); we prove that the algorithm satisfies the requirements by exhibiting a progress function collection from (A, b) to (A', b').

5.1.1 The Algorithm

The algorithm consists of two components, a *clock* and a *manager*. The *clock* has only one action, the output *TICK*, which is always enabled, and has no effect on the clock's state. It can be described as the particular one-state I/O automaton with the following steps.[7]

TICK
Precondition:
    true
Effect:
    none

The partition contains a single class, which contains the single output event *TICK*. For convenience, we overload the notation and designate this singleton class as *TICK* also.

The manager can be described as another I/O automaton, this one having one input action, *TICK* and one output action, *GRANT*. The manager waits a particular number $k > 0$ of clock ticks before issuing each *GRANT*, counting from the beginning or from the last preceding *GRANT*. The manager's state has one variable: *timer*, holding an integer, initially k.

The manager's algorithm is as follows:

[7] In the notation we use for automata, a separate description is given for the steps involving each action. Instead of listing the steps, we provide a "precondition" which describes the set of states in which the action is enabled, and an "effect" which describes the changes caused by the action. Input actions do not have a precondition, because they are always enabled.

TICK
Effect:
$timer := timer - 1$

GRANT
Precondition:
$timer \leq 0$
Effect:
$timer := k$

Thus, in the situation we are modeling, when the *GRANT* action's precondition becomes satisfied, the action does not occur instantly – the action waits until the automaton's next local step occurs. The partition has a single class, containing the single output action *GRANT*; we call this class *GRANT* as well. Fix A to be the I/O automaton which is the composition of the clock and manager automata, with the *TICK* output action hidden (using the I/O automaton hiding operator to convert it to an internal action); thus, the only external action of A is the output action *GRANT*.

The boundmap b associates the lower bound c_1 and upper bound c_2 with the class *TICK*, where $0 < c_1 \leq c_2 < \infty$; this means that the times between successive *TICK* events, and the time of the first *TICK* event, are in the interval $[c_1, c_2]$. The boundmap b also associates the lower bound 0 and upper bound l with the class *GRANT*, where $0 < l < \infty$; which means that the times between successive chances for the manager to take a step, and the time of the first such chance, are in the interval $[0, l]$. We assume that $c_1 > l$.[8] We wish to show that all the timed behaviors of (A, b) satisfy certain upper and lower bounds on the time up to the first *GRANT* and the time between consecutive pairs of *GRANT* events.

We begin our analysis by stating some useful invariant properties of the algorithm. In order to do this, we need timing information to be included in the state, so we consider the automaton $time(A, b)$, constructed as described in Section 3. Note that in this case, the automaton $time(A, b)$ has the following variables: *basic*, *now*, *first*(*TICK*), *last*(*TICK*), *first*(*GRANT*), and *last*(*GRANT*). The next lemma states invariant properties of the automaton $time(A, b)$. Notice that the second property involves the time prediction variables.

We again use record notation to designate state components, *e.g.*, we use *s.timer* to denote the value of the *timer* component of *s.basic*.

Lemma 5.1 *The following are true about any reachable state s of $time(A, b)$.*

1. $s.timer \geq 0$.
2. *If* $s.timer = 0$ *then* $s.first(TICK) \geq s.last(GRANT) + c_1 - l$.

[8] This assumption is needed, for example, for Lemma 5.1. Other assumptions could be used, but they would lead to slightly different bounds.

Proof: By induction on the length of an execution leading to s. If the length is 0, then $s.timer = k > 0$, so the conditions are easily seen to be true. So suppose that $(s', (\pi, t), s)$ is a step of $time(A, b)$, where s' is reachable in n steps and the conditions are true for s'. We consider cases.

1. $\pi = GRANT$.

   Then the effect of the $GRANT$ action implies that $s.timer = k > 0$, which implies both conditions.

2. $\pi = TICK$.

   Suppose that $s.timer < 0$. Then $s'.timer = 0$, by the effect of the step and the inductive hypothesis. The inductive hypothesis also implies that $s'.first(TICK) \geq s'.last(GRANT) + c_1 - l$. Since $c_1 > l$ (by assumption), this implies that $s'.first(TICK) > s'.last(GRANT)$. Since $s'.last(GRANT) \geq s'.now = t$, it follows that $s'.first(TICK) > t$. But then the definition of $time(A, b)$ implies that $TICK$ is not enabled in s', a contradiction. Thus, $s.timer \geq 0$, showing the first condition.

   Now, $s.first(TICK) = t + c_1$ and $s.last(GRANT) \leq t + l$. This implies that $s.first(TICK) \geq s.last(GRANT) + c_1 - l$, showing the second condition.

3. $\pi = NULL$.

   Then all of the terms involved in the two conditions are the same in s' and s, so the conditions are preserved.

■

5.1.2 The Requirements Automaton

We show the following, for any timed behavior β of (A, b):

1. There are infinitely many $GRANT$ events in β.
2. If t is the time of the first $GRANT$ event in β, then $k \cdot c_1 - l \leq t \leq k \cdot c_2 + l$.
3. If t_1 and t_2 are the times of any two consecutive $GRANT$ events in β, then

$$k \cdot c_1 - l \leq t_2 - t_1 \leq k \cdot c_2 + l.$$

We let $\mathbf{P}$ denote the set of sequences of $(action, time)$ pairs, where the only action is $GRANT$, satisfying the above three conditions.

We specify $\mathbf{P}$ in terms of another timed automaton, (A', b'). Define A' to have a single state and a single $GRANT$ output action enabled in that state, and define the boundmap b' to assign to the unique class of A' the lower and upper bounds $k \cdot c_1 - l$ and $k \cdot c_2 + l$, respectively.

Note that the timed behaviors of (A', b') are exactly the sequences in $\mathbf{P}$.

5.1.3 The Proof

In this subsection, we give a progress function collection from (A, b) to (A', b'), thereby showing that all timed behaviors of (A, b) are also timed behaviors of (A', b'). This fact yields Theorem 5.3, which says that all timed behaviors of (A, b) are in $\mathbf{P}$.

The mapping is defined by means of a progress function collection, $(\hat{f}, ub_{GRANT}, lb_{GRANT})$, where $\hat{f}(s.basic)$ is the unique state of A', for all s, and

$$ub_{GRANT}(s) = \begin{cases} s.last(TICK) + (s.timer - 1)c_2 + l & \text{if } s.timer > 0, \\ s.last(GRANT) & \text{otherwise,} \end{cases}$$

and

$$lb_{GRANT}(s) = \begin{cases} s.first(TICK) + (s.timer - 1)c_1 & \text{if } s.timer > 0, \\ s.now & \text{otherwise.} \end{cases}$$

The progress functions give explicit upper and lower bounds for the time of the next $GRANT$ event, in terms of the values of the variables in the state of $time(A, b)$. For instance, if $s.timer > 0$, a $TICK$ event must happen within time $s.last(TICK)$, and then after $s.timer - 1$ additional ticks, each happening after at most c_2 time, $timer$ will become 0, thus enabling the $GRANT$, which will happen within time at most l.

Since there is only one class in the partition of A', we drop the subscript $GRANT$ on the progress functions for the rest of this example, writing simply ub and lb in place of ub_{GRANT} and lb_{GRANT}.

Lemma 5.2 *The triple $(\hat{f}, ub, lb)$ is a progress function collection from (A, b) to (A', b').*

Proof: Let s be the unique start state of $time(A, b)$. Then $s.timer = k > 0$, $s.last(TICK) = c_2$ and $s.first(TICK) = c_1$, so that

$$ub(s) = s.last(TICK) + (s.timer - 1)c_2 + l = k \cdot c_2 + l$$

and

$$lb(s) = s.first(TICK) + (s.timer - 1)c_1 = k \cdot c_1 \geq k \cdot c_1 - l.$$

Let $v = \hat{f}(s.basic)$. Then v is the unique start state of A'. Also,

$$b'_u(GRANT) = k \cdot c_2 + l = ub(s)$$

and

$$b'_l(GRANT) = k \cdot c_1 - l \leq lb(s).$$

This shows Condition 1 of Definition 4.2.

Now we show Condition 2. Suppose that s' is a reachable state of $time(A, b)$ and $(s', (\pi, t), s)$ is a step of $time(A, b)$, where π is nonnull. Let v denote the unique state of A'. We consider cases.

1. $\pi = GRANT$.

   Then $s'.timer \leq 0$ and $s.timer = k > 0$, by the precondition and effect of $GRANT$ in A; thus, $s'.timer = 0$ by Lemma 5.1. Lemma 5.1 also implies that $s'.first(TICK) \geq s'.last(GRANT) + c_1 - l$.

   Let α be the execution fragment $(v, GRANT, v)$ of A'. Then Condition 2(a) of Definition 4.2 is immediate. For Condition 2(b)i, the enabling and uniqueness conditions are immediate; moreover,

$$\begin{aligned} t &= s'.now \text{ by definition of } time(A,b), \\ &= lb(s') \text{ since } s'.timer = 0, \end{aligned}$$

   as needed.

   Condition 2(b)ii is vacuously true, since a $GRANT$ event occurs in α. For Condition 2(b)iii, we must show that $ub(s) \leq t + b'_u(GRANT)$ and $lb(s) \geq t + b'_l(GRANT)$. For the upper bound, we have that $s.last(TICK) \leq t + c_2$, by definition of $time(A,b)$. Therefore,

$$\begin{aligned} ub(s) &= s.last(TICK) + (k-1)c_2 + l \text{ since } s.timer = k > 0, \\ &\leq t + k \cdot c_2 + l, \\ &= t + b'_u(GRANT), \end{aligned}$$

   as needed.

   For the lower bound, we have that $s.first(TICK) = s'.first(TICK)$ and $s'.last(GRANT) \geq t$, by definition of $time(A,b)$. Therefore,

$$\begin{aligned} lb(s) &= s.first(TICK) + (k-1)c_1, \text{ since } s.timer > 0, \\ &= s'.first(TICK) + (k-1)c_1, \\ &\geq s'.last(GRANT) + k \cdot c_1 - l \text{ by Lemma 5.1}, \\ &\geq t + k \cdot c_1 - l, \\ &= t + b'_l(GRANT), \end{aligned}$$

   as needed.

2. $\pi = TICK$.

   Then $s.timer = s'.timer - 1$. Let α be the trivial execution fragment v of A'. Once again, Conditions 2(a) of Definition 4.2 is immediate. Conditions 2(b)i and 2(b)iii are vacuously true. For Condition 2(b)ii, we must show that $ub(s) \leq ub(s')$ and $lb(s) \geq lb(s')$. There are two cases.

   (a) $s.timer > 0$.

For the upper bound, we have that $s.last(TICK) = t + c_2$ and $t \leq s'.last(TICK)$, by definition of $time(A, b)$; therefore, $s.last(TICK) \leq s'.last(TICK) + c_2$. Thus,

$$\begin{aligned} ub(s) &= s.last(TICK) + (s.timer - 1)c_2 + l, \\ &= s.last(TICK) + (s'.timer - 2)c_2 + l \text{ since } s.timer = s'.timer - 1, \\ &\leq s'.last(TICK) + (s'.timer - 1)c_2 + l, \\ &= ub(s'), \end{aligned}$$

as needed.

For the lower bound, we have that $s.first(TICK) = t + c_1$ and $s'.first(TICK) \leq t$ by the definition of $time(A, b)$; therefore, $s.first(TICK) \geq s'.first(TICK) + c_1$. Thus,

$$\begin{aligned} lb(s) &= s.first(TICK) + (s.timer - 1)c_1, \\ &\geq s'.first(TICK) + c_1 + (s.timer - 1)c_1, \\ &= s'.first(TICK) + (s'.timer - 1)c_1 \text{ since } s.timer = s'.timer - 1, \\ &= lb(s'), \end{aligned}$$

as needed.

(b) $s.timer = 0$.

Then $s'.timer = 1$. For the upper bound, we have that $s.last(GRANT) \leq t + l$ and $t \leq s'.last(TICK)$, so that $s.last(GRANT) \leq s'.last(TICK) + l$, by definition of $time(A, b)$. Therefore,

$$\begin{aligned} ub(s) &= s.last(GRANT), \\ &\leq s'.last(TICK) + l, \\ &= ub(s'), \end{aligned}$$

as needed.

For the lower bound, we have that $s.now = t$ and $s'.first(TICK) \leq t$, so that $s.now \geq s'.first(TICK)$. Therefore,

$$\begin{aligned} lb(s) &= s.now, \\ &\geq s'.first(TICK), \\ &= lb(s'), \end{aligned}$$

as needed.

Now consider a step $(s', (NULL, t), s)$ of $time(A, b)$, where s' is a reachable state of $time(A, b)$. Condition 3(a) of Definition 4.2 is immediate. Now,

$$ub(s') = \begin{cases} s'.last(TICK) + (s'.timer - 1)c_2 + l & \text{if } s'.timer > 0, \\ s'.last(GRANT) & \text{otherwise.} \end{cases}$$

Therefore, $ub(s') \geq \min(s'.last(TICK), s'.last(GRANT))$. By the definition of $time(A, b)$, it must be that $t \leq \min(s'.last(TICK), s'.last(GRANT))$; thus, $t \leq ub(s')$, which shows Condition 3(b)i of Definition 4.2. For Condition 3(b)ii, we must show that $ub(s) \leq ub(s')$ and $lb(s) \geq lb(s')$. But since only the value of *now* is different in s and s', and $s.now \geq s'.now$, these inequalities follow immediately from the definitions of the progress functions ub and lb. ■

Now we can put the pieces together.

Theorem 5.3 *All timed behaviors of* (A, b) *are in* $\mathbf{P}$.

Proof: Lemma 5.2 yields a progress function collection from (A, b) to (A', b'). Thus, by Theorem 4.4, any timed behavior β of (A, b) is a timed behavior of (A', b'). This implies that $\beta \in \mathbf{P}$. ■

5.1.4 Discussion

The bounds that we have proved above are nearly tight, Specifically, it is possible to produce four timed executions of (A, b) that exhibit the following types of behavior:

1. The time until the first *GRANT* is exactly $k \cdot c_1$.
2. The time until the first *GRANT* is exactly $k \cdot c_2 + l$.
3. The time between the first and second *GRANT* events is exactly $k \cdot c_1 - l$.
4. The time between the first and second *GRANT* events is exactly $k \cdot c_2 + l$.

The only discrepancy between these bounds and those proved above is a difference of l in the lower bound for the first *GRANT*.

For example, the first bound is realized by the timed execution of (A, b) that has the following timed schedule:

$$(TICK, c_1), (TICK, 2 \cdot c_1), \ldots, (TICK, k \cdot c_1), (GRANT, k \cdot c_1).$$

The second bound is realized by the timed execution that has the following timed schedule:

$$(TICK, c_2), (TICK, 2 \cdot c_2), \ldots, (TICK, k \cdot c_2), (GRANT, k \cdot c_2 + l).$$

The third bound is realized by:

$$(TICK, c_1), (TICK, 2 \cdot c_1), \ldots, (TICK, k \cdot c_1), (GRANT, k \cdot c_1 + l)$$
$$(TICK, (k+1) \cdot c_1), (TICK, (k+2) \cdot c_1), \ldots, (TICK, 2k \cdot c_1), (GRANT, 2k \cdot c_1).$$

Finally, the fourth bound is realized by:

$$(TICK, c_2), (TICK, 2 \cdot c_2), \ldots, (TICK, k \cdot c_2), (GRANT, k \cdot c_2)$$
$$(TICK, (k+1) \cdot c_2), (TICK, (k+2) \cdot c_2), \ldots, (TICK, 2k \cdot c_2), (GRANT, 2k \cdot c_2 + l).$$

Note that it is possible to modify our proof to give the tight lower bound of $k \cdot c_1$ for the first *GRANT*; the idea is to split the requirements to be proved so they are expressed by two separate partition classes in (A', b'), one for the first *GRANT* and one for the time between pairs of *GRANT* events. The two classes will have different lower bounds. There is a slight technical difficulty in that the algorithm (A, b) would have to be modified slightly in order to distinguish the first *GRANT* event from successive *GRANT* events, but there is no problem in principle.

Note that our resource manager is much simpler than the usual examples of resource-granting systems; in particular, there is no request input that triggers the *GRANT* output. We do not think that adding such structure would increase the conceptual difficulty of the example or expose any interesting property of the methodology we suggest here; however, it would make the analysis somewhat longer.

5.2 Two-Process Race System

We consider a system composed of two processes, X and Y. Process X increments a counter until process Y modifies a flag, and then decrements the counter. When the counter reaches 0, process X announces that it is done. We are interested in upper and lower bounds on the time until a "done" announcement occurs. An interesting aspect of this example is the fact that the worst-case time is not attained in the case where the processes both continually take steps at their slowest possible rates. Rather, it is attained when process Y takes steps at its slowest possible rate, while process X takes steps at its *fastest* rate until the flag is set, and then takes steps at its *slowest* rate until the counter reaches 0. (Actually, process X does not quite take steps at its fastest rate; more precisely, it performs the maximum number of steps it can before the flag is set, but may slow down slightly to ensure that the last step occurs at the latest possible time.)

This example was originally suggested to us by Amir Pnueli, as a test case for our proof technique. Several variants of this example, for specific bounds on the step time, have also been studied in [12].

As in the previous example, we describe the algorithm and its timing assumptions as a timed automaton (A, b), and the required timing behavior as another timed automaton (A', b'), and produce a progress function collection from (A, b) to (A', b').

5.2.1 The Algorithm

The system is described as a single timed automaton (A, b) containing two classes representing the two processes X and Y. Automaton A has state variables x, y and *done*, where x and y are integers, initially 0, and *done* is a Boolean, initially *false*. There are one output action, $DONE$, three internal actions, SET, INC and DEC, and no input actions. The partition classes are $X = \{INC, DEC, DONE\}$ and $Y = \{SET\}$. Intuitively, there are two sequential processes (using shared memory), one of which performs the SET action and one of which performs the other three actions. The transitions are as follows.

SET
Precondition:
$y = 0$
Effect:
$y := 1$

INC
Precondition:
$y = 0$
Effect:
$x := x + 1$

DEC
Precondition:
$y = 1$
$x > 0$
Effect:
$x := x - 1$

DONE
Precondition:
$y = 1$
$x = 0$
$done = false$
Effect:
$done := true$

The boundmap b for A assigns the lower bound l_1 and the upper bound l_2, where $0 < l_1 \leq l_2 < \infty$, with each of the two partition classes, indicating that the time between successive steps of each of the two processes is in the interval $[l_1, l_2]$. We are interested in determining the maximum and minimum times taken by the timed automaton (A, b) from the beginning until the $DONE$ action occurs.

5.2.2 The Requirements Automaton

We will show that any timed behavior β of (A, b) contains exactly one *DONE* event, occurring at a time in the interval $[l_1, (2 + \lfloor \frac{l_2}{l_1} \rfloor) l_2]$. The intuition for the lower bound should be clear: this is the earliest time at which the flag can be set, and hence the earliest at which the *DONE* event can occur. The intuition for the upper bound is a little more complex: if process Y sets the flag at the latest possible time l_2, then there is time for process X to take approximately $\frac{l_2}{l_1}$ steps before the flag is set, if X takes steps as quickly as possible. This will cause the counter to be set to approximately $\frac{l_2}{l_1}$. If X then decrements the counter as slowly as possible, with time l_2 between successive steps, then the total time to decrement is approximately $(\frac{l_2}{l_1}) l_2$. The precise bound involves some roundoffs and additive constants, and is obtained using some trial and error.

Let $\mathbf{P}$ denote the set of sequences of (action,time) pairs, where the only action is *DONE*, satisfying the condition that the *DONE* event occurs at a time in the interval $[l_1, (2 + \lfloor \frac{l_2}{l_1} \rfloor) l_2]$.

We specify $\mathbf{P}$ in terms of a timed automaton (A', b'), defined as follows. A' has two states, *active* and *inactive*, with start state *active*, and a single action, *DONE*, which is an output action enabled in state *active* and whose effect is to change the state to *inactive*. The boundmap b' assigns to the single class *DONE* the lower and upper bounds l_1 and $(2 + \lfloor \frac{l_2}{l_1} \rfloor) l_2$, respectively. Note that the timed behaviors of (A', b') are exactly the sequences in $\mathbf{P}$.

5.2.3 The Proof

In this subsection, we define a progress function collection from (A, b) to (A', b'), which implies that every timed behavior of (A, b) satisfies $\mathbf{P}$. The progress function collection, $(\hat{f}, ub_{DONE}, lb_{DONE})$, has $\hat{f}(s.basic) = active$ if $s.done = false$ and *inactive* if $s.done = true$, and

$$ub_{DONE}(s) = \begin{cases} s.last(Y) + (s.x + 2 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor) l_2 & \text{if } s.y = 0 \text{ and } s.first(X) \leq s.last(Y) \\ s.last(X) + s.x \cdot l_2 & \text{otherwise,} \end{cases}$$

and

$$lb_{DONE}(s) = \begin{cases} s.first(X) + (s.x + 2) l_1 & \text{if } s.y = 0 \text{ and } s.first(Y) > s.last(X) \\ s.first(X) + s.x \cdot l_1 & \text{otherwise.} \end{cases}$$

We give some intuition for the first, more complicated case of each inequality. For the upper bound, this is the case where another step of X *can* occur before the next (and only) step of Y occurs. In this case, $\lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor$ measures how many *additional* steps of X (after the indicated step of X) can fit before Y must take a step, and $(s.x + 2 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor) l_2$ is the longest time it can take from the time *SET* occurs (which is at most $s.last(Y)$) until *DONE* occurs. In more detail, at the time the *SET* occurs, the value of x is at most $s.x + 1 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor$, so it takes this number of *DEC* events (each consuming at most l_2 time) until x gets set to 0, and at most another l_2 until *DONE* occurs.

For the lower bound, the first case is the case where another step of X *must* occur before the next (and first) step of Y occurs. In this case, x will be increased at time at least $s.first(X)$ and it will take at least $x+1$ DEC operations (each consuming at least l_1 time) until x gets set to 0 and another l_1 time until *DONE* occurs. The second cases of both inequalities are similar, but simpler.

Again, since there is only one class in the partition of A', we will drop the subscript *DONE* on the progress functions for the rest of this example, writing simply ub and lb in place of ub_{DONE} and lb_{DONE}.

Lemma 5.4 *The triple $(\hat{f}, ub, lb)$ is a progress function collection from (A,b) to (A',b').*

Proof: Let s be the unique start state of $time(A,b)$. Then $s.first(X) = s.first(Y) = l_1$, $s.last(X) = s.last(Y) = l_2$, $s.x = s.y = 0$, and $s.done = false$. Then

$$\begin{aligned} ub(s) &= s.last(Y) + (s.x + 2 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor) l_2 \\ &= l_2 + (2 + \lfloor \frac{l_2 - l_1}{l_1} \rfloor) l_2 \\ &= (2 + \lfloor \frac{l_2}{l_1} \rfloor) l_2, \end{aligned}$$

and

$$lb(s) = s.first(X) + s.x \cdot l_1 = l_1.$$

Let $v = \hat{f}(s.basic)$. Then $v = active$, by definition of f, which is the start state of A'. Also, $b'_u(DONE) = (2 + \lfloor \frac{l_2}{l_1} \rfloor) l_2 = ub(s)$ and $b'_l(DONE) = l_1 = lb(s)$. This shows Condition 1 of Definition 4.2.

Now we show Condition 2. Suppose that s' is a reachable state of $time(A,b)$ and $(s', (\pi, t), s)$ is a step of $time(A,b)$, where π is nonnull. Also suppose that $v' = \hat{f}(s'.basic)$ and $v = \hat{f}(s.basic)$. We consider cases.

1. $\pi = DONE$.

   Then $s'.y = 1$, $s'.x = 0$, $s'.done = false$, and $s.done = true$, by the precondition and effect of *DONE* in A, and $s'.first(X) \leq t$, by the definition of $time(A,b)$. Also, $v' = \hat{f}(s'.basic) = active$ and $v = \hat{f}(s.basic) = inactive$.

   Let α be the execution fragment $(v', DONE, v)$ of A'. Condition 2(a) is immediate. For Condition 2(b)i, the uniqueness and enabling conditions are immediate; moreover,

$$\begin{aligned} t &\geq s'.first(X), \\ &= lb(s') \text{ since } s'.y = 1 \text{ and } s'.x = 0, \end{aligned}$$

as needed.

Condition 2(b)ii is vacuously true, since a *DONE* event occurs in α. Condition 2(b)iii is also vacuously true, since $v \notin enabled(A', DONE)$.

2. $\pi = SET$.

Then $s'.y = 0$, $s.y = 1$, $s'.x = s.x$, by the precondition and effect of SET in A. Moreover, $s'.done = s.done = false$, which implies that $v' = v = active$. Also, $s.last(X) = s'.last(X)$, $s.first(X) = s'.first(X)$, $s.last(X) \leq t + l_2$, $t \leq s'.last(Y)$, $t \leq s'.last(X)$ and $s'.first(Y) \leq t$, by definition of $time(A, b)$.

Let α be the trivial execution fragment v' of A'. Condition 2(a) is immediate, and 2(b)i and 2(b)iii are vacuously true. For Condition 2(b)ii, we must show that $ub(s) \leq ub(s')$ and $lb(s) \geq lb(s')$. For the upper bound, we consider two cases.

(a) $s'.first(X) > s'.last(Y)$.
Then

$$\begin{aligned} ub(s) &= s.last(X) + (s.x)l_2 \text{ since } s.y = 1, \\ &= s'.last(X) + (s'.x)l_2, \\ &= ub(s'), \end{aligned}$$

which suffices.

(b) $s'.first(X) \leq s'.last(Y)$.
Then

$$\begin{aligned} ub(s) &= s.last(X) + (s.x)l_2, \\ &\leq t + l_2 + (s.x)l_2, \\ &\leq t + (s'.x + 2)l_2, \\ &\leq s'.last(Y) + (s'.x + 2)l_2, \\ &\leq s'.last(Y) + (s'.x + 2 + \lfloor \frac{s'.last(Y) - s'.first(X)}{l_1} \rfloor)l_2, \\ &= ub(s'), \end{aligned}$$

as needed.

For the lower bound, we see that $s'.first(Y) \leq s'.last(X)$, since $t \leq s'.last(X)$ and $s'.first(Y) \leq t$. Therefore,

$$\begin{aligned} lb(s) &= s.first(X) + (s.x)l_1, \\ &= s'.first(X) + (s'.x)l_1, \\ &= lb(s'), \end{aligned}$$

which suffices.

3. $\pi = INC$.

Then $s'.y = s.y = 0$ and $s.x = s'.x + 1$, by the definition of INC. Also, $s'.first(X) \leq t \leq s'.last(Y)$, $s.last(Y) = s'.last(Y)$, $s.last(X) = t + l_2$, $s.first(X) = t + l_1$, and $s.first(Y) \leq t + l_1$, by definition of $time(A, b)$. Thus, $ub(s') = s'.last(Y) + (s'.x + 2 + \lfloor \frac{s'.last(Y) - s'.first(X)}{l_1} \rfloor) l_2$.

Let α be the trivial execution fragment v' of A'. As before, the only nontrivial condition to show is Condition 2(b)ii, that $ub(s) \leq ub(s')$ and $lb(s) \geq lb(s')$. For the upper bound, we consider two cases.

(a) $s.first(X) \leq s.last(Y)$.
Then $ub(s) = s.last(Y) + (s.x + 2 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor) l_2$. Now,

$$\begin{aligned} \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor + 1 &= \lfloor \frac{s.last(Y) - (t + l_1)}{l_1} \rfloor + 1, \\ &\quad \text{since } s.first(X) = t + l_1, \\ &= \lfloor \frac{s.last(Y) - t}{l_1} \rfloor, \\ &\leq \lfloor \frac{s'.last(Y) - s'.first(X)}{l_1} \rfloor \\ &\quad \text{since } t \geq s'.first(X) \text{ and } s.last(Y) = s'.last(Y). \end{aligned}$$

So

$$\begin{aligned} ub(s) &= s.last(Y) + (s.x + 2 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor) l_2, \\ &= s'.last(Y) + (s'.x + 3 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor) l_2, \\ &\leq s'.last(Y) + (s'.x + 2 + \lfloor \frac{s'.last(Y) - s'.first(X)}{l_1} \rfloor) l_2, \\ &= ub(s'), \end{aligned}$$

as needed.

(b) $s.first(X) > s.last(Y)$.
Then $ub(s) = s.last(X) + (s.x) l_2$. Then

$$\begin{aligned} ub(s) &= s.last(X) + (s.x) l_2, \\ &= s.last(X) + (s'.x + 1) l_2, \\ &= t + l_2 + (s'.x + 1) l_2, \\ &\leq s'.last(Y) + l_2 + (s'.x + 1) l_2 \\ &= s'.last(Y) + (s'.x + 2) l_2 \end{aligned}$$

$$\begin{aligned} &\leq s'.last(Y) + (s'.x + 2 + \lfloor \frac{s'.last(Y) - s'.first(X)}{l_1} \rfloor) l_2 \\ &\quad \text{since } s'.first(X) \leq s'.last(Y), \\ &= ub(s'), \end{aligned}$$

as needed.

For the lower bound, notice that

$$s.first(Y) \leq t + l_1 \leq t + l_2 = s.last(X) .$$

Thus, we have $lb(s) = s.first(X) + (s.x)l_1$. There are two cases.

(a) $s'.first(Y) \leq s'.last(X)$.
Then

$$\begin{aligned} lb(s) &= s.first(X) + (s.x)l_1, \\ &\geq s.first(X) + (s'.x)l_1, \\ &\geq t + (s'.x)l_1, \\ &\geq s'.first(X) + (s'.x)l_1, \\ &= lb(s'), \end{aligned}$$

as needed.

(b) $s'.first(Y) > s'.last(X)$.
Then

$$\begin{aligned} lb(s) &= s.first(X) + (s.x)l_1, \\ &= s.first(X) + (s'.x + 1)l_1, \\ &= s.first(X) - l_1 + (s'.x + 2)l_1, \\ &= t + (s'.x + 2)l_1, \\ &\geq s'.first(X) + (s'.x + 2)l_1, \\ &= lb(s'), \end{aligned}$$

as needed.

4. $\pi = DEC$.

Once again, let α be the trivial execution fragment v' of A'. As before, the only nontrivial condition to show is Condition 2(b)ii, that $ub(s) \leq ub(s')$ and $lb(s) \geq lb(s')$. By the definition of DEC, $s'.y = s.y = 1$ and $s.x = s'.x - 1$. Also, $s.last(X) = t + l_2$, $s.first(X) = t + l_1$, $t \leq s'.last(X)$, and $t \geq s'.first(X)$, by definition of $time(A, b)$.

For the upper bound, we have that

$$\begin{aligned} ub(s) &= s.last(X) + (s.x)l_2, \\ &= t + l_2 + (s.x)l_2, \\ &\leq s'.last(X) + l_2 + (s.x)l_2, \\ &= s'.last(X) + (s'.x)l_2, \\ &= ub(s'), \end{aligned}$$

as needed.

For the lower bound, we have that

$$\begin{aligned} lb(s) &= s.first(X) + (s.x)l_1, \\ &= t + l_1 + (s.x)l_1, \\ &\geq s'.first(X) + l_1 + (s.x)l_1, \\ &= s'.first(X) + (s'.x)l_1, \\ &= lb(s'), \end{aligned}$$

as needed.

Now consider a step $(s', (NULL, t), s)$ of $time(A, b)$, where s' is a reachable state of $time(A, b)$. Condition 3(a) of Definition 4.2 is immediate. Now,

$$ub(s') = \begin{cases} s'.last(Y) + (s'.x + 2 + \lfloor \frac{s.last(Y) - s.first(X)}{l_1} \rfloor)l_2 & \text{if } s'.y = 0 \text{ and } s'.first(X) \leq s'.last(Y), \\ s'.last(X) + s'.x \cdot l_2 & \text{otherwise.} \end{cases}$$

Thus, $ub(s') \geq \min(s'.last(Y), s'.last(X))$. By the definition of $time(A, b)$, it must be that $t \leq \min(s'.last(Y), s'.last(X))$; thus, $t \leq ub(s')$, which shows Condition 3(b)i of Definition 4.2. For Condition 3(b)ii, note that there are no changes in any of the terms involved in the definitions of ub and lb, so $ub(s) = ub(s')$ and $lb(s) = lb(s')$. ■

Theorem 5.5 *All timed behaviors of* (A, b) *are in* $\mathbf{P}$.

Proof: As for Theorem 5.3, using Lemma 5.4. ■

5.2.4 Discussion

For this example, the bounds we have proved are attainable. That is, there is a timed execution of (A, b) for which the time until a *DONE* event occurs is exactly l_1, and another timed execution for which the time until a *DONE* event occurs is exactly $(2 + \lfloor \frac{l_2}{l_1} \rfloor)l_2$.

For example, the bound l_1 is realized by the timed execution that has the timed schedule $(SET, l_1), (DONE, l_1)$. The bound $(2 + \lfloor \frac{l_2}{l_1} \rfloor) l_2$ is realized by the timed execution having the timed schedule

$$(INC, al_2), (INC, 2al_2), \ldots, (INC, \lfloor \frac{l_2}{l_1} \rfloor al_2), (SET, l_2),$$
$$(DEC, 2l_2), (DEC, 3l_2), \ldots, (DEC, (1 + \lfloor \frac{l_2}{l_1} \rfloor) l_2), (DONE, (2 + \lfloor \frac{l_2}{l_1} \rfloor) l_2),$$

where $a = 1/\lfloor \frac{l_2}{l_1} \rfloor$. This timed execution involves the SET happening at the latest possible time, l_2. The maximum possible number of INC events occur prior to the SET, and the last of these occurs at the same time as the SET. The DEC events occur as late as possible.

6 Conclusions and Further Work

In this paper, we have described a way to carry out assertional proofs for timing properties of algorithms that have timing assumptions. The method involves expressing an algorithm and its timing assumptions as a timed automaton (A, b), and expressing the timing requirements in terms of a second timed automaton (A', b'). Then we convert the timed automata (A, b) and (A', b') into ordinary (not timed) I/O automata, $time(A, b)$ and $time(A', b')$ respectively, using a general construction that builds predictive timing information into the automaton state. Then the goal of proving timing requirements can be met by demonstrating the existence of a certain type of mapping called a "strong possibilities mapping" from the "assumptions automaton" $time(A, b)$ to the "requirements automaton" $time(A', b')$. One way of demonstrating the existence of such a mapping is based on a collection of progress functions, each designed to measure progress toward the fulfillment of one of the upper or lower bound requirements expressed by (A', b'). These progress functions generalize those used elsewhere for program verification in that they are real-valued rather than discrete, and that they are used for lower as well as upper bounds.

We have applied this method in this paper to analyze the timing properties of two systems – a simple resource-granting system and a race system involving two processes. The analyses of these two examples are straightforward; they consist of case analyses based directly on the conditions specified in the definition of a progress function collection. The style and level of difficulty of these proofs is exactly the same as that of typical inductive proofs of invariant assertions. As do other proofs of that type, these remove the need for complex dynamic arguments about the behavior of the algorithm, replacing them with simple checks involving individual algorithm steps. Because of the need to check many cases, the proofs are not extremely short (the proof for each of our examples is about three pages long); however, this style should scale very well because of the local nature of the checks performed. Also, as for other assertional proofs, it seems likely that proofs using this method can someday be checked using machine-verification technology.

We do not have an easy method for finding an appropriate progress function. Just as for finding invariant assertions, finding the right progress function is a creative task, which depends on an understanding of how the system operates. There are alternative methods which do not require human intervention, *e.g.*, those based on model-checking [4, 19]. However, these methods apply only to finite-state algorithms, and are known to be expensive or even undecidable [4]. Moreover, these methods do not give the benefit of the insights provided by a good invariant or progress function.

The two examples in this paper are not the only examples to which this method has been applied. In a project carried out for Digital Equipment Corporation, several timing properties (including self-stabilization properties) were proved for a new link state packet distribution protocol [20]. Some of the timing properties proved were unexpected, and were discovered in the course of applying the methods of this paper. Although it is possible to provide some informal intuitions for these properties using ad hoc arguments, we do not know a better way than the method of this paper to provide complete and convincing proofs that these properties hold. We have found that progress functions provide a natural and intuitive way of thinking about the reasons the timing properties hold, as well as a basis for formal correctness arguments. Based on the examples that have been tried so far, we believe that the method may be practical for use in verifying timing properties for real timing-based algorithms. It remains to test this hypothesis by applying the technique to more examples; good sources for examples are the areas of real-time computing and communication.

In some of the proofs we give for the DEC protocol, we do not give bounds that are as tight as those we have given for the simple examples in this paper. This is not surprising: in general, for complex algorithms, it is often much easier to prove bounds that are somewhat loose than to prove bounds that are actually attainable by some execution. The method of this paper supports the proof of loose bounds just as easily as that of tight bounds.

A good technique for proving timing properties of systems with timing assumptions should be rigorous, simple and general. Our technique is certainly rigorous, and we think it is also reasonably simple. We consider its generality. Although it seems to us that timed automata are probably sufficiently general to describe typical implementations, they may not be sufficiently general to describe all interesting requirements specifications. For example, as currently defined, they cannot specify bounds for reaching certain states, but only for the occurrence of certain actions. In [29], the authors express a similar doubt, and address it by generalizing the notion of a boundmap to include certain more general timing conditions. While we could make a similar extension here (indeed, we do make such an extension in an earlier version of this paper [23]), the extra notation required for doing so seems to obscure the essentially simple ideas of our method. Moreover, there is no guarantee that the resulting extension will yet be sufficiently expressive. (Although we state a completeness result in [23] for the generalized specifications, this completeness result is relative to the restriction, not used in this paper, that the underlying automata A and A' are identical.) We have chosen to present our method here using a model that is possibly somewhat too restrictive, and to leave the appropriate generalization for future work.

It remains to relate our method to other methods for proving timing properties. One method we have considered is the one used for several algorithms in [24], based on bounding the time for the occurrence of intermediate milestones. Such a proof can be expressed by a series of proofs in our method, one for each intermediate milestone. A good example to consider is the tournament algorithm for mutual exclusion in [35]. The proof sketched in [24] for this algorithm uses recurrence inequalities to bound the time until a given process wins at various levels of the tournament tree. It should be possible to recast this proof as a sequence of proofs, one for each level of the tree, where the proof for each level of the tree is a generic argument based on a single use of the main recurrence inequality. Although we have not worked out this example in detail, we have done a complete proof [22, 23] of a simpler example motivated by this one (based on a line rather than a tree). In principle, it seems that the ideas should extend to the more complex example, but this remains to be done. Some other techniques to relate to this one include those based on bounded-time temporal logic (*e.g.*, [3, 7, 11, 12]). Also, it remains to see how proofs using our techniques can be applied in a modular way for the verification of timing properties of large and complex timing-based systems.

Acknowledgements.

We would like to thank Amir Pnueli for suggesting the race-system example of Section 5.2 as a test case for our proof technique. We would also like to thank Stephen Ponzio for his helpful comments on much earlier versions of this paper, and George Varghese for many useful suggestions on the final version.

References

[1] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," DEC SRC Research Report 29, August 1988.

[2] M. Abadi and L. Lamport, "An Old-Fashioned Recipe for Real Time," *Proceedings REX Workshop "Real-Time: Theory in Practice"*, June, 1991, Mook, The Netherlands.

[3] R. Alur and T. Henzinger, "Real-Time Logics: Complexity and Expressiveness," in *proc. 5th IEEE Symp. on Logic in Computer Science*, pp. 390–401, June 1990.

[4] R. Alur, C. Courcoubetis and D. Dill, "Model-Checking for Real-Time Systems," in *proc. 5th IEEE Symp. on Logic in Computer Science*, June 1990.

[5] R. Alud and D. Dill, "Automata for Modelling Real-Time Systems," in *Proc. ICALP '90*, Lecture Notes in Computer Science 443, Springer-Verlag, pp. 322–335.

[6] H. Attiya and N. Lynch, "Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty," in *proc. 10th Real-Time Systems Symposium*, pp. 268–284, December 1989. Expanded version available as Technical Report MIT/LCS/TR-403, Laboratory for Computer Science, MIT, July 1989.

[7] A. Bernstein and P. Harter, Jr. "Proving Real-Time Properties of Programs with Temporal Logic," in *Proc. 8th Symp. on Operating System Principles*, Operating Systems Review, Vol. 15, No. 5 (December 1981), pp. 1–11.

[8] J. E. Coolahan and N. Roussopoulus, "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5 (September 1983), pp. 603–616.

[9] A. Gabrielian and M. W. Franklin, "State-Based Specification of Complex Real-Time Systems," in *Proc. 9th IEEE Real-Time Systems Symp.*, 1988, pp. 2–11.

[10] V. H. Hasse, "Real-time behavior of programs," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 5 (September 1981), pp. 494–501.

[11] E. Harel, O. Lichtenstein and A. Pnueli, "Explicit Clock Temporal Logic," in *proc. 5th IEEE Symp. on Logic in Computer Science*, pp. 402–413, June 1990.

[12] T. A. Henzinger, Z. Manna and A. Pnueli, "Temporal Proof Methodologies for Real-Time Systems," in *proc. ACM Symp. on Principles of Programming Languages*, pp. 353–366, January 1991.

[13] J. Hooman, *A Compositional Proof Theory for Real-Time Distributed Message Passing*, TR. 4-1-1(1), Department of Mathematics and Computer Science, Eindhoven University of technology, March 1987.

[14] F. Jahanian and A. Mok, "A Graph-Theoretic Approach for Timing Analysis and Its Implementation," *IEEE Transactions on Computers*, Vol. C-36, No. 8 (August 1987), pp. 961–975.

[15] F. Jahanian and D. A. Stuart, "A Method for Verifying Properties of Modechart Specifications," in *Proc. 9th IEEE Real-Time Systems Symp.*, 1988, pp. 12–21.

[16] R. Koymans, J. Vytopil and W. P. deRoever, "Real-Time Programming and Asynchronous Message Passing," in *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, 1983, pp. 187–197.

[17] L. Lamport, "Specifying Concurrent Program Modules," *ACM Trans. on Programming Languages and Systems*, Vol. 5, No. 2 (April 1983), pp. 190–222.

[18] L. Lamport and M. Abadi, "Refining and Composing Real-Time Specifications," in progress.

[19] H. R. Lewis, "Finite-State Analysis of Asynchronous Circuits with Bounded Temporal Uncertainty," Technical Report TR-15-89, Aiken Computation Laboratory, Harvard University.

[20] N. Lynch, A. Harvey, R. Perlman and G. Varghese, "An Analysis of the OSI Network Layer Link State Packet Distribution Protocol," in progress.

[21] N. Lynch, "Concurrency Control for Resilient Nested transactions," *Advances in Computing Research*, Vol. 3, 1986, pp. 335–373.

[22] N. Lynch and H. Attiya, "Using Mappings to Prove Timing Properties," Technical Memo MIT/LCS/TM-412.b, Laboratory for Computer Science, MIT, March 1990.

[23] N. Lynch and H. Attiya. "Using mappings to prove timing properties," In *Proceedings of the 9^{th} Annual ACM Symposium on Principles of Distributed Computing*, Quebec, Canada, August 1990. pp. 265-280.

[24] N. Lynch and K. Goldman, *Lecture notes for 6.852.* MIT/LCS/RSS-5, Laboratory for Computer Science, MIT, 1989.

[25] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," in *Proc. 7th ACM symp. on Principles of Distributed Computing*, 1987, pp. 137–151. Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, April 1987.

[26] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI-Quarterly*, Vol. 2, No. 3, 1989. Also: Technical Memo, MIT/LCS/TM-373, Laboratory for Computer Science Massachusetts Institute of Technology, November 1988.

[27] N. Lynch and F. Vaandrager, "Forward and Backward Simulations for Timing-Based Systems," *Proceedings REX Workshop "Real-Time: Theory in Practice"*, June, 1991, Mook, The Netherlands.

[28] Z. Manna, "Mathematical Theory of Computation," McGraw-Hill Computer Science Series, MacGraw-Hill Book Company, 1974.

[29] M. Merritt, F. Modugno and M. Tuttle, "Time Constrained Automata," In J.C.M. Baeten and J.F. Groote, editors, *Proceedings CONCUR 91*, Amsterdam, Vol. 527 of *Lecture Notes in Computer Science*, pp. 408-423. Springer-Verlag, 1991.

[30] R. Milner, "Calculi for Synchrony and Asynchrony," TCS 25, 1983, pp. 267-310.

[31] Peter G. Neumann and Leslie Lamport, "Highly Dependable Distributed Systems," Technical Report, SRI International, Contract Number DAEA18-81-G-0062, SRI Project 4180, June 1983.

[32] J. S. Ostroff, "Deciding Properties of Timed Transion Models," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 170-183, April 1990.

[33] J. S. Ostroff, "Survey of Formal Methods for the Specification and Design of Real-Time Systems," IEEE Press, to appear.

[34] J. S. Ostroff and W. M. Wonham, "A Framework for Real-Time Discrete Event Control," *IEEE Trans. on Automatic Control*, April 1990.

[35] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," in *Proc. 9th ACM symp. on Theory of Computing*, May 1977, pp. 91–97.

[36] F. B. Schneider, "Real-Time Reliable Systems Project," in *Foundations of Real-Time Computing Research Initiative*, ONR Kickoff Workshop, November 1988, pp. 28–32.

[37] A. U. Shankar and S. Lam, "Time-Dependent Distributed Systems: Proving Safety, Liveness and Timing Properties," *Distributed Computing*, 2 (1987), pp. 61–79.

[38] A. C. Shaw, "Reasoning About Time in Higher-Level Language Software," *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 7 (July 1989), pp. 875–889.

[39] J. Sifakis, "Petri Nets for Performance Evaluation, in Measuring, Modeling And Evaluating Computer Systems," in *Proc. 3rd Symp. IFIP Working Group 7.3*, H. Beilner and E. Gelenbe (eds.), Amsterdam, The Netherlands, North-Holland, 1977, pp. 75–93.

[40] J. Stankovic and K. Ramamritham, "The SPRING Kernel: A New Paradigm for Real-Time Operating Systems," *ACM Operating Systems Reviews*, Vol 23, No. 3 (July 1989), pp. 54–71.

[41] G. Tel, "Assertional Verification of a Timer Based Protocol," in *Proc. ICALP '88*, Lecture Notes in Computer Science 317, Springer-Verlag, pp. 600–614.

[42] A. Zwarico, *Timed Acceptance: an Algebra of Time Dependent Computing*, Ph.D. thesis, Dept. of Computer and Information Science, University of Pennsylvania, 1988.