Simulation Techniques for Proving Properties of Real-Time Systems

Nancy Lynch

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract. The method of *simulations* is an important technique for reasoning about real-time and other timing-based systems. It is adapted from an analogous method for untimed systems. This paper presents the simulation method in the context of a very general automaton (i.e., labelled transition system) model for timing-based systems. Sketches are presented of several typical examples for which the method has been used successfully. Other complementary tools are also described, in particular, invariants for safety proofs, progress functions for timing proofs, and execution correspondences for liveness proofs.

Keywords: Simulation, timing-based system, real-time system, untimed system, invariant, invariant assertion, progress function, execution correspondence, time bound, upper bound, lower bound, clock synchronization, mutual exclusion, leader election.

Table of Contents

1 Introduction

In the years that have elapsed since the REX workshop series began, a good deal has been learned about how to reason about real-time and other timing-based systems. In many cases, the methods that have been developed have been adaptations of methods that had previously been used for untimed systems.

In this paper, I present a method that I have found useful for verifying properties of timing-based systems: the method of *simulations*. This has been adapted from the simulation method that has been widely used for untimed systems. The simulation method falls into the general category of *assertional techniques*, and includes refinement mappings, forward and backward simulations, and history and prophecy mapping techniques as special cases.

I illustrate how simulations can be used for timing-based systems by introducing them in the context of a very general automaton (i.e., labelled transition system) model for such systems. I present sketches of a sizable collection of typical examples for which the method has been used successfully. These examples include proofs of ordinary safety properties, as well as time bound properties. Along the way, I describe other complementary tools, most notably, invariants for safety proofs, progress functions for timing proofs, and execution correspondences for liveness proofs.

In more detail, the paper proceeds as follows. In Section 2, I describe the very general and basic timed automaton model of Lynch and Vaandrager [26], and use it to model a simple bounded clock system. In Section 3, I express the various notions of simulations from the literature, together with their basic soundness properties, all in terms of the basic model. As an example, I describe a simple clock synchronization algorithm and show, using a refinement mapping, that it implements the bounded clock system.

Next, I impose some useful structure on the model and simulations, and present several examples of simulation proofs that take advantage of this structure. Specifically, in Section 4, I define an important special case of the general timed automaton model – the timed automaton model of Merritt, Modugno and

Tuttle [29]. I use this model to describe Fischer's timing-based mutual exclusion algorithm [10], and to verify that the algorithm in fact satisfies the mutual exclusion property. Then in Section 5, I illustrate how simulations, in particular, forward simulations, can be used to prove time bounds as well as ordinary safety properties. I do this using five examples, including Fischer's and Dijkstra's mutual exclusion algorithms. All these examples are described using the special case model of Merritt et al.

Section 6 indicates how liveness proofs can be integrated with the safety and time bound proofs, and Section 7 concludes with a discussion.

2 The Basic Timed Automaton Model

The basic model that I use for describing timing-based systems is the simple and very general model of Lynch and Vaandrager [26, 27, 41].[1] This section contains the relevant definitions.

2.1 Timed Automata

A *timed automaton* A consists of:

- a set $states(A)$ of states;
- a nonempty subset $start(A)$ of start states;
- a set $acts(A)$ of actions, including a special *time-passage* action ν; the actions are partitioned into *external* and *internal* actions, where ν is considered external; the *visible* actions are the non-ν external actions; the visible actions are partitioned into *input* and *output* actions;
- a set $steps(A)$ of steps (transitions); this is a subset of $states(A) \times acts(A) \times states(A)$;
- a mapping $now_A : states \to \mathsf{R}^+$. ($\mathsf{R}^+$ denotes the nonnegative reals.)

I write $s' \xrightarrow{a}_A s$ as shorthand for $(s', \pi, s) \in steps(A)$. I usually write the $s.now_A$ in place of $now_A(s)$. I sometimes suppress the subscript or argument A when no confusion seems likely.

There are several simple axioms that a timed automaton is required to satisfy:

[A1] If $s \in start$ then $s.now = 0$.

[A2] If $s' \xrightarrow{\pi} s$ and $\pi \neq \nu$ then $s'.now = s.now$.

[A3] If $s' \xrightarrow{\nu} s$ then $s'.now < s.now$.

[A4] If $s' \xrightarrow{\nu} s''$ and $s'' \xrightarrow{\nu} s$, then $s' \xrightarrow{\nu} s$.

In order to state the last axiom, I need a preliminary definition of a *trajectory*, which describes restrictions on the state changes that can occur during time-passage. Namely, if I is any interval of R^+, then an *I-trajectory* is a function $w : I \to states$, such that

[1] There are a few tiny technical distinctions among the definitions in the listed papers. The definitions I use here are restatements of those of [41], except that I classify external actions as input or output, allow named internal actions, and also correct an obvious omission in the *trajectory axiom*.

1. $w(t).now = t$ for all $t \in I$, and
2. $w(t_1) \xrightarrow{\nu} w(t_2)$ for all $t_1, t_2 \in I$ with $t_1 < t_2$.

That is, w assigns, to each time t in interval I, a state having the given time t as its *now* component. This assignment is done in such a way that time-passage steps can span between any pair of states in the range of w. If w is an I-trajectory and I is left-closed, then define $w.ftime = min(I)$ and $w.fstate = w(w.ftime)$, while if I is right-closed, then define $w.ltime = max(I)$ and $w.lstate = w(w.ltime)$. If I is a closed interval, then an I-trajectory w is said to span from state s' to state s if $w.fstate = s'$ and $w.lstate = s$. The final axiom is:

[A5] If $s' \xrightarrow{\nu} s$ then there exists a trajectory from s' to s.

Axiom [A1] says that the current time is always 0 in a start state. Axiom [A2] says that non-time-passage steps do not change the time; that is, they occur "instantaneously", at a single point in time. Axiom [A3] says that time-passage steps must cause the time to increase; this is a convenient technical restriction. Axiom [A4] allows repeated time-passage steps to be combined into one step. Axiom [A5] is a kind of converse to [A4]; it says that any time-passage step can be "filled in" with states for each intervening time, in a "consistent" way. This axiom is a strengthening of a similar axiom used elsewhere which, rephrased in the terminology of this paper, reads: If $s' \xrightarrow{\nu} s$ and $s'.now < t < s.now$, then there is an s'' with $s''.now = t$ such that $s' \xrightarrow{\nu} s''$ and $s'' \xrightarrow{\nu} s$.

Note that this model is sufficiently general to allow description of *hybrid systems* [28], because it allows rather general changes to the state during time-passage steps.

2.2 Timed Executions

In this subsection, I define a notion of "timed execution" for a timed automaton. The most obvious formulation of a timed execution might be as a sequence of visible, internal and time-passage actions, interspersed with their intervening states. I augment this information slightly by including the trajectories for each time-passage action.

Formally, a *timed execution fragment* is a finite or infinite alternating sequence $\alpha = w_0\pi_1w_1\pi_2w_2\cdots$, where:

1. Each w_j is a trajectory and each π_j is a non-time-passage action.
2. If α is a finite sequence, then it ends with a trajectory.
3. If w_j is not the last trajectory in α then its domain is a closed interval. If w_j is the last trajectory then its domain is left-closed (and either right-open or right-closed).
4. If w_j is not the last trajectory then $w_j.lstate \xrightarrow{\pi_{j+1}} w_{j+1}.fstate$.

The trajectories describe the changes of state during the time-passage steps. The last item says that the actions in α span between successive trajectories.

A *timed execution* is a timed execution fragment for which the first state of the first trajectory, w_0, is a start state. In this paper, I am mainly interested in a particular subclass of the set of timed executions: the *admissible* timed

executions. These are defined to be the timed executions in which the supremum of the set of *now* values occurring in the states is ∞.

A state of a timed automaton is defined to be *reachable* if it is the final state of the final trajectory in some finite timed execution of the automaton.

Note that, as I have described them so far, timed automata have no features for expressing liveness or fairness properties (with the exception of admissibility). In general, such features are less important in the timed setting than they are in the untimed setting, since they are often replaced by time bound requirements. However, in Section 6, I will say more about how liveness can be added in.

Note that there exist timed automata that have *no* admissible timed executions. To rule out this case, one generally restricts attention to timed automata that are *feasible*, i.e., in which each "finite" timed execution can be extended to an admissible timed execution. I will not address issues of feasibility in this paper; for a discussion of feasibility, I refer the reader to [12].

2.3 Timed Traces

In order to describe the problems to be solved by timed automata, I require a definition for their visible behavior. I use the notion of *timed traces*. The *timed trace* of any timed execution is just the sequence of visible events that occur in the timed execution, paired with their times of occurrence. The *admissible timed traces* of the timed automaton are just the timed traces that arise from all the admissible timed executions. If a problem P is formulated as a set of (finite and infinite) sequences of actions paired with times, then a timed automaton A is said to *solve* P if all its admissible timed traces are in P. Often, it is natural to express a problem P as the set of admissible timed traces of another timed automaton B. Thus, the notion of admissible timed traces induces a preorder on timed automata: $A \leq B$ is defined to mean that the set of admissible timed traces of A is a subset of the set of admissible timed traces of B.

2.4 Discrete Executions

Sometimes it is useful in proofs about timed automata to use another notion of execution, one that omits the trajectory information in favor of just recording time-passage steps. I define a *discrete execution fragment* of a timed automaton to be a finite or infinite alternating sequence $\alpha = s_0 \pi_1 s_1 \pi_2 s_2 \cdots$, where:

1. Each s_j is a state and each π_j is an action (possibly a time-passage action).
2. If α is a finite sequence, then it ends with a state.
3. If s_j is not the last state then $s_j \xrightarrow{\pi_{j+1}} s_{j+1}$.

A *discrete execution* is a discrete execution fragment whose first state is a start state. Again, I am mainly interested in the *admissible* discrete executions – those in which the supremum of the *now* values occurring in the states is ∞. Note that any admissible discrete execution must be an infinite sequence.

The *timed trace* of an admissible discrete execution is the sequence of visible events that occur in the execution, paired with their times of occurrence, i.e., the *now* values in the preceding states.

An admissible discrete execution α is said to *sample* an admissible timed execution α' if its sequence of actions consists of exactly the actions of α', occurring at the same times, interspersed with time-passage actions; several consecutive time-passage actions can be used to span a trajectory. The states appearing in α must be extracted in the natural way from the trajectories in α'.

Lemma 1. *If α' is an admissible timed execution then there exists an admissible discrete execution α that samples it. Conversely, if α is an admissible discrete execution, then there exists an admissible timed execution α' such that α samples α'.*

Lemma 2. *If α samples α', then the timed trace of α is the same as that of α'.*

Lemma 3. *A state of a timed automaton is reachable exactly if it is the final state of some finite discrete execution.*

These definitions and relationships are presented in detail in [26].

2.5 Composition

I define a simple binary parallel composition operator for timed automata. Let A and B be timed automata satisfying the following *compatibility* conditions:

1. A and B have no output actions in common.
2. No internal action of A is an action of B, and vice versa.

Then the *composition* of A and B, written as $A \times B$, is the timed automaton defined as follows.

- $states(A \times B) = \{(s_A, s_B) \in states(A) \times states(B) : s_A.now_A = s_B.now_B\}$;
- $start(A \times B) = start(A) \times start(B)$;
- $acts(A \times B) = acts(A) \cup acts(B)$; an action is *external* in $A \times B$ exactly if it is external in either A or B, and likewise for *internal* actions; a visible action of $A \times B$ is an *output* in $A \times B$ exactly if it is an output in either A or B, and is an *input* otherwise;
- $(s'_A, s'_B) \longrightarrow^{\pi}_{A \times B} (s_A, s_B)$ exactly if
  1. $s'_A \xrightarrow{\pi}_A s_A$ if $\pi \in acts(A)$, else $s'_A = s_A$, and
  2. $s'_B \xrightarrow{\pi}_B s_B$ if $\pi \in acts(B)$, else $s'_B = s_B$;
- $(s_A, s_B).now_{A \times B} = s_A.now_A$.

It is not hard to show that $A \times B$ is indeed a timed automaton, and that the parallel composition operator is substitutive for the admissible timed trace inclusion ordering, $\leq$, on timed automata.

2.6 Example: Bounded Clock System

I close this section with a simple example of a timed automaton. This example is adapted from [38]. It is a fairly standard-looking description of a clock system, consisting of a collection of "local clocks", each of whose values is always within a bound ϵ of real time. The automaton simply maintains this property, while permitting real time to pass.

In the given code, the state is described in a structured fashion, as a collection of values for a collection of state components. Likewise, the start state is described as a collection of initial values for the components. The actions are listed explicitly. The steps are described in a guarded command style, organized by actions (including the time-passage action), each with a "precondition" (guard) describing conditions on the state that enable the action to occur, and an "effect" describing the state changes that accompany the action. The time-passage action ν is parameterized with an incremental time Δt, describing the amount of time that passes. The *now* component appears as an explicit state component.

Let I be a nonempty, finite set of node indices.

Automaton B: Bounded Clock System

Actions:
- Output:
  - $report_i(c)$, $i \in I$
- Internal:
  - $tick_i(c)$, $i \in I$

State components:
- $now \in \mathsf{R}^+$, initially 0
- $clock_i \in \mathsf{R}^+$, $i \in I$, initially 0

$tick_i(c)$
- Precondition:
  - $c \geq clock_i$
  - $|c - now| \leq \epsilon$
- Effect:
  - $clock_i := c$

$report_i(c)$
- Precondition:
  - $c = clock_i$
- Effect:
  - none

$\nu(\Delta t)$
- Precondition:
  - $t = now + \Delta t$
  - for all i, $|t - clock_i| \leq \epsilon$
- Effect:
  - $now := t$

Thus, any local clock is allowed to "tick" (i.e., advance to a new specified value c) if the new value is at least as big as the old value, and is within ϵ of real

time. Moreover, real time is allowed to pass, as long as it remains within ϵ of all the local clock values. Finally, any current local clock value can be reported at any time. The following lemma captures the key synchronization property.

Lemma 4. *The following is true of every reachable state of B:*
For all i, $|clock_i - now| \leq \epsilon$.

Proof. In view of Lemma 3, it suffices to prove the property for all states that occur as final states of finite discrete executions of B. The proof proceeds by induction on the number of actions in a finite discrete execution. Correctness follows from the explicit checks performed by the *tick* and ν actions. □

Consider the admissible timed executions of B – those in which the time components of the states approach infinity. In order for time to pass to infinity, it is necessary that the clocks all tick infinitely often (and by an appropriate amount) so they can stay close to real time. The *report* actions are optional.

It is not hard to see that timed automaton B is feasible. For, starting from any finite timed execution of B, it is not hard to construct a sequence of synchronized clock ticks and time-passage actions that allows time to pass to infinity.

Clock B is *discrete*, in the sense that the increases in the values of the various clocks all happen in discrete *tick*'s. It is also possible to define a corresponding *continuous* clock within the same model. Such a clock would eliminate the *tick* actions, and would instead allows continuous increases in the values of the local clocks, as part of time-passage steps. Such clocks are described in [5].

2.7 Discussion

In [41], parallel composition and several other useful operations on timed automata are defined. These include standard "untimed operations" such as hiding, renaming, internal and external choice, sequential composition, and the CSP interrupt operator [13] (i.e., A and B are both started; if B performs a visible action then A is interrupted and B continues to run). They also include some "timed operations" such as the timed CSP timeout [6, 35] (i.e., A is started; if A does not perform a visible action by real time d, then A is interrupted and B is started), and the ATP execution delay operator [31]. The admissible timed trace inclusion relation (more precisely, a variant of it that includes certain kinds of "finite timed traces" as well) is shown to be substitutive with respect to all of these operations.

3 Simulations for Timed Automata

In this section, I introduce the basic types of simulations that can be used for proving properties of systems described as timed automata. Simulation methods are just a few among many possible formal tools for reasoning about systems expressed as timed automata, but they are among the most powerful for proving safety properties.

The value of the simulation method for verifying safety properties of untimed systems is now well established. Many papers and books, e.g., [3, 15, 19, 21, 23, 32, 38, 42], contain substantial examples of its use. Also see [14] for a persuasive discussion of the value of the technique. The use of this method for timed systems is much newer, but appears very promising. Preliminary results appear in [20, 38].

3.1 Simulations

In this subsection, I define the basic types of simulations: refinements, forward simulations, and backward simulations, for timed automata. These definitions are paraphrased from [26, 27].[2] As described in [25–27], they are all straightforward extensions of similar definitions for untimed automata.

Suppose A and B are timed automata. A *refinement* from A to B is a function $r : \mathit{states}(A) \to \mathit{states}(B)$ that satisfies:

1. $r(s).\mathit{now} = s.\mathit{now}$.
2. If $s \in \mathit{start}(A)$ then $r(s) \in \mathit{start}(B)$.
3. If $s' \xrightarrow{\pi}_A s$ then there is a timed execution fragment from $r(s')$ to $r(s)$ having the same sequence of timed visible actions (that is, the same sequence of visible actions, with the same associated times) as the given step.

Note that π is allowed to be the time-passage action in the third item of this definition; the same is true in the succeeding definitions.

In the following definitions, I use the notation $r[s]$, where r is a binary relation, to denote $\{u : (s, u) \in r\}$.

A *forward simulation* from A to B is a relation f over $\mathit{states}(A)$ and $\mathit{states}(B)$ that satisfies:

1. If $u \in f[s]$ then $u.\mathit{now} = s.\mathit{now}$.
2. If $s \in \mathit{start}(A)$ then $f[s] \cap \mathit{start}(B) \neq \emptyset$.
3. If $s' \xrightarrow{\pi}_A s$ and $u' \in f[s']$, then there exists $u \in f[s]$ such that there is a timed execution fragment from u' to u having the same sequence of timed visible actions as the given step.

A *backward simulation* from A to B is a total relation b over $\mathit{states}(A)$ and $\mathit{states}(B)$ that satisfies:

1. If $u \in b[s]$ then $u.\mathit{now} = s.\mathit{now}$.
2. If $s \in \mathit{start}(A)$ then $b[s] \subseteq \mathit{start}(B)$.
3. If $s' \xrightarrow{\pi}_A s$ and $u \in b[s]$, then there exists $u' \in b[s']$ such that there is a timed execution fragment from u' to u having the same sequence of timed visible actions as the given step.

A backward simulation is said to be *image-finite* provided that $b[s]$ is a finite set for every state s of A. Note that every refinement is a forward simulation, and is also an image-finite backward simulation.

[2] In the earlier papers, they are called "timed refinements", etc. Here I omit the adjective "timed" for brevity.

I write $A \leq_R B$, $A \leq_F B$ and $A \leq_B B$ to denote the existence of a refinement, forward simulation, or backward simulation from A to B, respectively. Also, I write $A \leq_{iB} B$ to denote the existence of an image-finite backward simulation from A to B.

The most important fact about these simulations is captured by a set of results saying that they are sound for admissible timed trace inclusion. More specifically:

Theorem 5 (Soundness). *$A \leq_R B$, $A \leq_F B$ and $A \leq_{iB} B$ all imply that $A \leq B$.*

Note that $A \leq_B B$ does not by itself imply admissible timed trace inclusion; a weaker soundness result, involving inclusion of sets of "finite timed traces", does hold in this case.

The soundness results are all proved in [26, 27], based on corresponding results for untimed automata. For untimed automata, the first two are proved by induction on the number of steps in an execution, while the last is proved by a backwards induction together with König's Lemma. Alternatively, the timed results can be proved directly by such inductions and König's Lemma, but the proof is best done using the "discretized" version of a timed execution mentioned earlier, which includes discrete time-passage steps rather than trajectories.

Another important fact about these simulations is a completeness result, also proved in [26, 27], for the methods used in combination. Namely, define a timed automaton to have *finite invisible nondeterminism* if, for every sequence of timed visible actions and every real time t, there are only finitely many states that can result from finite timed executions that generate the given sequence of timed visible actions and have t as the final time.

Theorem 6 (Completeness). *If $A \leq B$ and B has finite invisible nondeterminism then there exists a timed automaton C such that $A \leq_F C \leq_{iB} B$.*

3.2 Invariants and Weak Simulations

In using the simulation methods for actual proofs, the first thing that one usually wants to do is to divide the work of the proof, by first proving some invariants about either or both of the two automata involved. The use of such invariants must be justified; doing this requires augmenting the simulation definitions and soundness results to incorporate the invariants explicitly. In this subsection, I describe this extension.

For the purposes of this paper, I define an *invariant* of a timed automaton to be any property that is true of all reachable states; I do not make the assumption sometimes made elsewhere, that it is actually preserved by all steps of the automaton.

I call the newly-defined simulations *weak refinements*, *weak forward simulations*, etc., since (in most cases) they have fewer proof obligations.

A *weak refinement* from A to B with respect to invariants I_A and I_B (of A and B, respectively) is a function $r : states(A) \rightarrow states(B)$ that satisfies:

1. $r(s).now = s.now$.
2. If $s \in start(A)$ then $r(s) \in start(B)$.
3. If $s' \xrightarrow{\pi}_A s$, $\{s', s\} \subseteq I_A$, and $r(s') \in I_B$, then there is a timed execution fragment from $r(s')$ to $r(s)$ having the same timed visible actions as the given step.

A *weak forward simulation* from A to B with respect to I_A and I_B is a relation f over $states(A)$ and $states(B)$ that satisfies:

1. If $u \in f[s]$ then $u.now = s.now$.
2. If $s \in start(A)$ then $f[s] \cap start(B) \neq \emptyset$.
3. If $s' \xrightarrow{\pi}_A s$, $\{s', s\} \subseteq I_A$, and $u' \in f[s'] \cap I_B$, then there exists $u \in f[s]$ such that there is a timed execution fragment from u' to u having the same timed visible actions as the given step.

A *weak backward simulation* from A to B with respect to I_A and I_B is a relation b over $states(A)$ and $states(B)$ that satisfies:

1. If $u \in b[s]$ then $u.now = s.now$.
2. If $s \in start(A)$ then $b[s] \cap I_B \subseteq start(B)$.
3. If $s' \xrightarrow{\pi}_A s$, $\{s', s\} \subseteq I_A$, and $u \in b[s] \cap I_B$, then there exists $u' \in b[s'] \cap I_B$ such that there is a timed execution fragment from u' to u having the same timed visible actions as the given step.
4. If $s \in I_A$ then $b[s] \cap I_B \neq \emptyset$.

A weak backward simulation is said to be *image-finite* provided that $b[s]$ is a finite set for every state s of A.

Each of these three new definitions says that it is permissible to use the invariants on all the hypothesized states, in proving the existence of the required timed execution fragment. Note that in the case of a backward simulation, there is an extra proof obligation – to show that the invariant for B gets preserved "in reverse". In this sense, it is not strictly correct to say the notion of a weak backward simulation is "weaker" than the original notion of a backward simulation. Every weak refinement is a weak forward simulation, but not necessarily a weak backward simulation.

I extend the notation defined earlier, writing $A \leq_{\mathrm{WR}} B$, $A \leq_{\mathrm{WF}} B$, $A \leq_{\mathrm{WB}} B$, and $A \leq_{\mathrm{WiB}} B$ to denote that there exists a weak refinement, weak forward simulation, weak backward simulation, or weak image-finite backward simulation, from A to B, respectively, with respect to some invariants. The extended soundness results are:

Theorem 7 (Soundness). *$A \leq_{\mathrm{WR}} B$, $A \leq_{\mathrm{WF}} B$ and $A \leq_{\mathrm{WiB}} B$ all imply that $A \leq B$.*

3.3 Example: Clock Synchronization Algorithm

In this subsection, I describe a very simple implementation (in the sense of the $\leq$ preorder) of the bounded clock system B (where $|I| = 2$, i.e., the system

has two nodes). The algorithm consists of two nodes connected by a one-way channel, with message delay in the known range $[0, d]$. Node 1 maintains its own local clock, assumed always to be within δ of real time. Node 1 informs node 2 whenever its own clock changes, and node 2 simply adopts the maximum clock value it has seen as its own. Although it would probably be most natural to model this algorithm as the composition of three timed automata (the two nodes and the channel), for brevity, I just model it as a single timed automaton A.

Automaton A: Clock Synchronization Algorithm

Actions:

Output:

$report_i(c)$, $i \in \{1, 2\}$

Internal:

$tick_1(c)$

$deliver(c)$

State components:

$now \in \mathsf{R}^+$, initially 0

$clock_i \in \mathsf{R}^+$, $i \in \{1, 2\}$, initially 0

$channel$, a multiset of $\mathsf{R}^+ \times \mathsf{R}^+$, initially empty

$tick_1(c)$

Precondition:

$c \geq clock_1$

$|c - now| \leq \delta$

Effect:

$clock_1 := c$

add $(c, now + d)$ to $channel$

$\nu(\Delta t)$

Precondition:

$t = now + \Delta t$

$|t - clock_1| \leq \delta$

for all $(c, v) \in channel$, $t \leq v$

Effect:

$now := t$

$deliver(c)$

Precondition:

$(c, v) \in channel$

Effect:

remove (c, v) from $channel$

$clock_2 := max(clock_2, c)$

$report_i(c)$

Precondition:

$c = clock_i$

Effect:

none

The *tick* action for node 1 is just like the *tick* actions of B, except that, in addition to just updating the local clock, it also causes a copy of the new clock value to be put into the channel. The second component v of the message that is put into the channel represents a real time by which that message is supposed to get delivered to node 2. Note that this second component is not a "normal" component of the algorithm; it is only introduced in order to encode a real-time restriction on the algorithm's behavior. This strategy – representing a real-time

deadline by an explicit deadline component in the state – is a frequently-used technical device in defining timed automata.

The *deliver* action causes node 2 to reset its clock to the newly received value (provided that the new value is not less than the old value). Now the time-passage action is required explicitly to maintain the appropriate relationship with clock 1, but there is no direct requirement that it remain close to clock 2. However, there is a new constraint on real time: time is constrained not to pass beyond the scheduled last delivery time for any message in the channel. The reports are as in B.

I claim that, provided that $\epsilon \geq \delta + d$, this algorithm A "implements" system B, in the sense that $A \leq B$ (inclusion of sets of admissible timed traces). To show this, I use a trivial weak refinement, r, defined as follows. Here, record notation is used to indicate state components.

- $r(s).now = s.now$.
- $r(s).clock_i = s.clock_i$, $i \in \{1, 2\}$.

In order to show that r is a weak refinement, some invariants are helpful. Specifically, I_A is defined to be the set of states of A in which $now - \epsilon \leq clock_2 \leq now + \delta$ and $now - \delta \leq clock_1 \leq now + \delta$. I_B is defined to be the set of all states of B – no particular properties of B will be needed in the proof. I now prove that I_A holds of all reachable states of A; this requires a series of simple lemmas.

Lemma 8. *The following is true of every reachable state of A:*
$|clock_1 - now| \leq \delta$.

Proof. By induction on the number of actions in a discrete execution, using the explicit checks in the $tick_1$ and ν actions. □

Lemma 9. *The following are true of every reachable state of A:*

1. *If $(c, v) \in channel$ then $c \leq clock_1$.*
2. *$clock_2 \leq clock_1$.*
3. *$clock_2 \leq now + \delta$.*

Proof. The first two parts are by an easy induction. The last part follows from the second part and Lemma 8. □

It remains to prove that $now - \epsilon \leq clock_2$, i.e., that $clock_2$ does not lag too far behind real time. In order to prove this, I prove several intermediate lemmas. The next lemma asserts that the value of $clock_1$ is in fact communicated to node 2.

Lemma 10. *The following is true of every reachable state of A:*
Either $clock_2 = clock_1$ or there is some $(c, v) \in channel$ such that $c = clock_1$.

The next lemma says that every message in the channel is scheduled to be delivered at most time d in the future.

Lemma 11. *The following is true of every reachable state of A:*
If $(c, v) \in$ *channel then* $now \leq v \leq now + d$.

The following is the key lemma. It implies that, for $0 \leq l \leq d$, the value that $clock_2$ will have l time units from now is at least $now - \delta - (d - l) \geq now - \epsilon + l$. The lemma explicitly describes the smallest value that $clock_2$ can take on, in terms of the current value of $clock_2$ and the messages that are guaranteed to be delivered strictly before time l from now.[3]

Lemma 12. *The following is true of every reachable state of A:*
For any l, $0 \leq l \leq d$, *either* $clock_2 \geq now - \delta - (d - l)$ *or there is some* $(c, v) \in$ *channel such that* $c \geq now - \delta - (d - l)$ *and* $v < now + l$.

Proof. By induction on the number of actions in a discrete execution. The interesting case is the time-passage action ν. Suppose that ν increases the time from t' to $t = t' + \Delta t$, while allowing the state to change from s' to s. Fix l, $0 \leq l \leq d$. There are two cases:

1. $\Delta t + l \leq d$.
   Then let $l' = l + \Delta t$; then $0 \leq l' \leq d$. By inductive hypothesis, either $s'.clock_2 \geq t' - \delta - (d - l')$ or there is some $(c, v) \in s'.channel$ such that $c \geq t' - \delta - (d - l')$ and $v < t' + l'$. But $t + l = t' + l'$, and so $t - \delta - (d - l) = t' - \delta - (d - l')$. Moreover, $s.clock_2 = s'.clock_2$. So either $s.clock_2 \geq t - \delta - (d - l)$ or there is some $(c, v) \in s.channel$ such that $c \geq t - \delta - (d - l)$ and $v < t + l$. This is as needed.
2. $\Delta t + l > d$.
   The definition for ν says that $s.clock_1 = s'.clock_1 \geq t - \delta$. Then Lemma 10 implies that either $s.clock_2 = s.clock_1 \geq t - \delta$ or there is some $(c, v) \in s.channel$ such that $c = s.clock_1 \geq t - \delta$. The first of these alternatives suffices for the lemma. In the latter case, it must be that $v \leq t' + d$, by Lemma 11. By the defining condition of this case, this implies that $v < t + l$. This suffices.

□

Lemma 13. *The following is true of every reachable state of A:*
$now - \epsilon \leq clock_2$.

Proof. By Lemma 12, for $l = 0$, it must be that in any reachable state, either $clock_2 \geq now - \delta - d \geq now - \epsilon$ or there is some $(c, v) \in channel$ such that $c \geq now - \delta - d$ and $v < now$. But the latter is impossible, by Lemma 11. So the former holds, which is as needed. □

[3] Note that the model allows for several actions to occur at the same real time. This makes it necessary to be especially careful about strict vs. non-strict inequalities.

This proves I_A. It still remains to show that r is a weak forward simulation with respect to invariants I_A and I_B. Fortunately, for this case, proving the invariants has already accomplished most of the work.

Lemma 14. *r is a weak refinement from A to B, with respect to invariants I_A and I_B.*

Proof. The time condition and start condition are easy to see; it remains to show the step condition, Condition 3. Suppose $s' \xrightarrow{\pi}_A s$, $s', s \in I_A$, $r(s') \in I_B$. The proof is by cases:

1. $\pi = tick_1(c)$
   Then I claim that $r(s') \overset{tick_1(c)}{\longrightarrow}_B r(s)$. This is straightforward because $\delta \leq \epsilon$ – the precondition for $tick_1(c)$ in A is at least as strong as in B.
2. $\pi = deliver(c)$, $c > s'.clock_2$
   Then I claim that $r(s') \overset{tick_2(c)}{\longrightarrow}_B r(s)$. Since $s \in I_A$, it must be that $|s.clock_2 - s.now| \leq \epsilon$. But $s.clock_2 = c$, by the effect of the $deliver(c)$ action. Also, $s'.now = s.now$. So $|c - s'.now| \leq \epsilon$. Thus, the precondition of the $tick_2(c)$ action in B holds. The effect clearly corresponds.
3. $\pi = deliver(c)$, $c \leq s'.clock_2$
   Then $r(s') = r(s)$, which suffices.
4. $\pi = report_i(c)$
   Then it is straightforward to show that $r(s') \overset{report_i(c)}{\longrightarrow}_B r(s)$.
5. $\pi = \nu$, increasing by Δt.
   Then I claim that $r(s') \xrightarrow{\nu}_B r(s)$. Define $t = s'.now + \Delta t$. The precondition of π in A implies that $|t - s'.clock_1| \leq \delta \leq \epsilon$. Since $s \in I_A$, it follows that $|s.clock_2 - s.now| \leq \epsilon$. But $t = s.now$ and $s.clock_2 = s'.clock_2$, so that $|t - s'.clock_2| \leq \epsilon$. This yields the precondition for $\nu(\Delta t)$ in B. The effect corresponds.

□

This proves the following theorem.

Theorem 15. *Let A be the clock synchronization algorithm and B the bounded clock system. Then $A \leq_{\mathrm{WR}} B$, and therefore $A \leq B$.*

This example showed a simple weak refinement. For an example of a weak forward simulation, consider A', which is defined to be the same system as A, but instead of sending the full clock values, node 1 just send the "low-order bits". More precisely, in place of sending c, node 1 sends $c' = c \bmod \gamma$, for some fixed γ such that $\gamma > 2\epsilon + 2\delta$. (I use the notation $c \bmod \gamma$ to denote the remainder when c is divided by γ, i.e., $c/\gamma - \lfloor c/\gamma \rfloor$.) The second component, v, of each message, is still allowed to be an unbounded time, because it does not represent an actual component to be included in the message, but rather just a conceptual real-time deadline.

The key idea is that from any state s of algorithm A, the range of c values that might arrive at node 2 in a *deliver* step in algorithm A is $[s.clock_2 - \epsilon - \delta, s.clock_2 + \epsilon + \delta]$. Thus, node 2 can correctly decode an arriving condensed clock value c' into the unique clock value c in the given range such that $c' = c \bmod \gamma$, and sets $clock_2$ to $max(clock_2, c)$.

In more detail, the modified actions are as follows. Here, the *decode* is defined to be the partial function such that $decode(c, d)$ is the (unique) value $c' \in [d - \epsilon - \delta, d + \epsilon + \delta]$ such that $c = c' \bmod \gamma$, if one exists, else undefined.

$tick_1(c)$
  Precondition:
    $c \geq clock_1$
    $|c - now| \leq \delta$
  Effect:
    $clock_1 := c$
    add $(c \bmod \gamma, now + d)$ to *channel*

$deliver(c)$
  Precondition:
    $(c, v) \in$ *channel*
  Effect:
    remove (c, v) from *channel*
    $clock_2 := max(clock_2, decode(c, clock_2))$

There is a (multivalued) forward simulation from A' to A, defined by $(s, u) \in f$ if and only if all state components are the same in s and u, with the following exception. For each message (c, v) in $u.channel$, the corresponding message $(c \bmod \gamma, v)$ appears in $s.channel$. Correctness of this simulation rests on first proving the invariant for A that all clock values appearing in messages in the channel are in the indicated interval; this follows in turn from the claim that $clock_2$ and all the clocks in the channel are in the interval $[now - \epsilon, now + \delta]$.

This example shows a typical use for forward simulations – describing an optimized version of an algorithm in terms of a simple, less efficient original version. In such a case, the correspondence generally needs to be multi-valued, since the original algorithm typically contains more information than the optimized version.

I do not have a related example to show here of a backward simulation. In fact, it seems hard to find practical examples where backward simulations are needed. They arise in situations where a choice is made earlier in the specification automaton than in the implementation automaton. I will not mention backward simulations any further in this paper; all the remaining examples will involve forward simulations only.

4 A Specialized Model

So far, I have presented the basic concepts for simulation proofs in the setting of a very general timed automaton model. But when one carries out interesting

verifications, it is often the case that the implementation and/or specification has some specialized structure that can help to "stylize" the proofs. Next I will describe a special case of the general timed automaton model that I have found to be suitable for describing most implementations, and many specifications as well.

4.1 MMT Automata

The specialized model is based on one defined by Merritt, Modugno and Tuttle [29], hence I call it the *MMT automaton* model. An MMT automaton is basically an I/O automaton [23, 24] together with some upper and lower bounds on time.

An I/O automaton A consists of

- a set $states(A)$ of states;
- a nonempty subset $start(A)$ of start states;
- a set $acts(A)$ of actions, partitioned into *external* and *internal* actions; the external actions are further partitioned into *input* and *output* actions;
- a set $steps(A)$ of steps; this is a subset of $states(A) \times acts(A) \times states(A)$;
- a partition $part(A)$ of the locally controlled (i.e., output and internal) actions into at most countably many equivalence classes.

An action π is said to be *enabled* in a state s' provided that there exists a state s such that $(s', \pi, s) \in steps(A)$, i.e., such that $s' \xrightarrow{\pi}_A s$. A set of actions is said to be *enabled* in s' provided that at least one action in that set is enabled in s'. It is required that the automaton be *input-enabled*, by which is meant that π is enabled in s' for every state s' and input action π. Note that there is no explicit time-passage action. The final component, *part*, is sometimes called the *fairness partition*. Each class in this partition groups together actions that are supposed to be part of the same "task". *Fair executions* are defined in such a way as to allow "fair turns" to each class of the partition. That is, for each partition class C, either (a) the execution is finite and ends in a state in which C is not enabled, or (b) the execution is infinite and either contains infinitely many C actions or infinitely many states in which C is not enabled. The I/O automaton model is a simple, yet rather expressive model for asynchronous concurrent systems. Typical examples of its use in describing and reasoning about such systems appear in [22].

The I/O automaton model, however, does not have any facilities for describing timing-based systems. An MMT automaton is obtained by augmenting an I/O automaton with certain upper and lower time bound information. In this paper, I use a special case of the MMT model that is described formally in [20]. Namely, let A be an I/O automaton with only finitely many partition classes. For each class C, define lower and upper time bounds, $lower(C)$ and $upper(C)$, where $0 \leq lower < \infty$ and $0 < upper(C) \leq \infty$; that is, the lower bounds cannot be infinite and the upper bounds cannot be 0.

A timed execution of an MMT automaton A is defined to be an alternating sequence of the form $s_0, (\pi_1, t_1), s_1, \cdots$ where now the π's are input, output or

internal actions. For each j, it must be that $s_j \xrightarrow{\pi_{j+1}} s_{j+1}$. The successive times are nondecreasing, and are required to satisfy the given *lower* and *upper* bound requirements. More specifically, define j to be an *initial index* for a class C provided that C is enabled in s_j, and either $j = 0$, or else C is not enabled in s_{j-1}, or else $\pi_j \in C$; initial indices are the points at which the bounds for C begin to be measured. Then for every initial index j for a class C, the following conditions must hold:

1. (Upper bound)
   If $upper \neq \infty$, then there exists $k > j$ with $t_k \leq t_j + upper(C)$ such that either $\pi_k \in C$ or C is not enabled in s_k.
2. (Lower bound)
   There does not exist $k > j$ with $t_k < t_j + lower(C)$ and $\pi_k \in C$.

Note that an *upper* bound of ∞ does not impose any requirement that actions in the corresponding class ever occur. Finally, *admissibility* is required: if the sequence is infinite, then the times of actions approach ∞. More formal statements of these conditions appear in [20].

Each timed execution of an MMT automaton A gives rise to a *timed trace*, which is just the subsequence of external actions and their associated times. The *admissible timed traces* of the MMT automaton A are just the timed traces that arise from all the timed executions of A.

MMT automata can be composed in much the same way as ordinary I/O automata, using synchronization on common actions. More specifically, define two MMT automata A and B to be *compatible* according to the same definition of compatibility for general timed automata. Then the *composition* of the two automata is the MMT automaton consisting of the I/O automaton that is the composition of the two component I/O automata (according to the definition of composition in [23, 24]), together with the bounds arising from the components. This composition operator is substitutive for the admissible timed trace inclusion ordering on MMT automata.

The MMT model just described is useful for describing many real-time systems. It is especially good as a low-level model for computer systems, since the class structure and associated time bounds are natural ways of modelling physical components and their speeds. However, it cannot be used for describing hybrid systems, in which state changes can accompany time-passage actions. Also, the MMT model does not appear to be general enough to provide a good model for arbitrary specifications or high-level system descriptions. For example, the model does not seem to be appropriate for describing the bounded clock system in Section 2.6.

Note that MMT automata, as presented so far, are not exactly a special case of the general (Lynch-Vaandrager) timed automata I described earlier. This is because the MMT model uses an "external" way of specifying the time bound restrictions, via the added lower and upper bounds. The Lynch-Vaandrager model, in contrast, builds the time-bound restrictions explicitly into the time-passage steps. However, it is not hard to transform any MMT automaton A into a

naturally-corresponding Lynch-Vaandrager timed automaton A'. This can be done using a construction similar to the one in Section 3 of [20], as follows.

First, the state of the MMT automaton A is augmented with a *now* component, plus *first*(C) and *last*(C) components for each class. The *first*(C) and *last*(C) components represent, respectively, the earliest and latest time in the future that an action in class C is allowed to occur. The rest of the state is called *basic*. The *now*, *first* and *last* components all take on values that represent *absolute* times, not incremental times. The time-passage action ν is also added.

The *first* and *last* components get updated in the natural way by the various steps, according to the *lower* and *upper* bounds specified in the MMT automaton A. The time-passage action has explicit preconditions saying that time cannot pass beyond any of the *last*(C) values, since these represent deadlines for the various tasks. Note that this usage of the *last*(C) components as deadlines is similar to the usage of deadline components in messages in the clock synchronization algorithm above. Restrictions are also added on actions in any class C, saying that the current time *now* must be at least equal to *first*(C).

In more detail, each state of A' is a record consisting of a component *basic*, which is a state of A, a component $now \in \mathsf{R}^+$, and, for each class C of A, components *first*(C) and *last*(C), each in $\mathsf{R}^+ \cup \{\infty\}$. Each start state s of A' has $s.basic \in start(A)$, and $s.now = 0$. Also, if C is enabled in $s.basic$, then $s.first(C) = lower(C)$ and $s.last(C) = upper(C)$; otherwise $s.first(C) = 0$ and $s.last(C) = \infty$. The actions of A' are the same as those of A, with the addition of the time-passage action ν. Each non-time-passage action is classified as an input, output or internal action according to its classification in A.

The steps are defined as follows. If $\pi \in acts(A)$, then $s' \xrightarrow{\pi}_{A'} s$ exactly if all the following conditions hold:

1. $s'.now = s.now$.
2. $s'.basic \xrightarrow{\pi}_{A} s.basic$.
3. For each $C \in part(A)$:
   (a) If $\pi \in C$ then $s'.first(C) \leq s'.now$.
   (b) If C is enabled in both s and s', and $\pi \notin C$, then $s.first(C) = s'.first(C)$ and $s.last(C) = s'.last(C)$.
   (c) If C is enabled in s and either C is not enabled in s' or $\pi \in C$ then $s.first(C) = s'.now + lower(C)$ and $s.last(C) = s'.now + upper(C)$.
   (d) If C is not enabled in s then $s.first(C) = 0$ and $s.last(C) = \infty$.

On the other hand, if $\pi = \nu$, then $s' \xrightarrow{\pi}_{A'} s$ exactly if all the following conditions hold:

1. $s'.now < s.now$.
2. $s.basic = s'.basic$.
3. For each $C \in part(A)$:
   (a) $s.now \leq s'.last(C)$.
   (b) $s.first(C) = s'.first(C)$ and $s.last(C) = s'.last(C)$.

The resulting timed automaton A' has exactly the same admissible timed traces as the MMT automaton A. Moreover, this transformation commutes with the operation of composition, up to isomorphism. From now on in this paper, I will often refer to an MMT timed automaton and to its transformed version interchangeably, relying on the context to distinguish them.

Suppose that two MMT automata are given, one (A) describing an implementation and the other (B) describing a specification. Then by regarding both A and B as timed automata, it is possible to use the simulation techniques defined in Section 3 to show that A implements B (in the sense of admissible timed trace inclusion).

4.2 Example: Fischer's Mutual Exclusion Algorithm

In this subsection, I use MMT automata to model a simple algorithm – the well-known Fischer mutual exclusion algorithm using read-write shared memory [10]. This algorithm has become a standard example for demonstrating the power of formal methods for reasoning about real-time systems. It can be verified in several ways, but to fit it into this paper, I express the proof as a simulation.

The most important correctness property of this algorithm is mutual exclusion. Other properties may also be of interest, for example, a time bound property, limiting the time that it takes from when anyone requests the resource until someone gets it, and a liveness property, stating that if anyone is trying to obtain the resource, then someone succeeds. In this subsection, I will just argue mutual exclusion, but will return to this example twice later in the paper to prove time bounds and liveness properties. Both of these proofs will also be based on simulations, but they will require a little more machinery (which I will introduce shortly).

I begin with the problem specification. It consists of a set of *users*, U_i, $1 \leq i \leq n$, each an MMT automaton, plus a *mutex object* M, also an MMT automaton. Let U denote the composition of all the U_i.

Each U_i has a state containing its current *region*, either trying, critical, exit or remainder. The outputs are try_i and $exit_i$, while the inputs are $crit_i$ and rem_i. Each action moves the user to the indicated region, The input-output behavior is intended to be cyclical, in the order try_i, $crit_i$, $exit_i$, $rem_i \cdots$; the definition of the user guarantees that it will not be the first to violate the cyclic condition.

The try_i and $exit_i$ actions are placed in separate singleton classes of the fairness partition. There are no special bounds on when try actions must occur, or when the critical region must be exited; therefore, the bounds are just the trivial $[0, \infty]$ for each class.

Automaton U_i: User

Actions:
  Input:
    $crit_i$
    rem_i

Output:
try_i
$exit_i$

State components:
$region_i \in \{rem, try, crit, exit\}$, initially rem

try_i
Precondition:
$region_i = rem$
Effect:
$region_i := try$

$exit_i$
Precondition:
$region_i = crit$
Effect:
$region_i := exit$

$crit_i$
Effect:
$region_i := crit$

rem_i
Effect:
$region_i := rem$

Classes and Bounds:
$\{try_i\}$, bounds $[0, \infty]$
$\{exit_i\}$, bounds $[0, \infty]$

The mutex object models the high-level behavior of a mutual exclusion system. It interacts with users by receiving the try_i and $exit_i$ inputs and producing the $crit_i$ and rem_i outputs. It keeps track of the regions for all the users, and ensures that it does not issue two *crit* actions before the first has exited. All *crit* actions are placed in one class, while each rem_i is in a class by itself. Again, all the classes only have the trivial bounds $[0, \infty]$. (Recall that for now, I am only going to prove mutual exclusion, not time bounds or liveness, so no interesting bounds are included in the specification.)

Automaton M: Mutex Object

Actions:
Input:
try_i, $1 \leq i \leq n$
$exit_i$, $1 \leq i \leq n$
Output:
$crit_i$, $1 \leq i \leq n$
rem_i, $1 \leq i \leq n$

State components:
$region_i$, $1 \leq i \leq n$, each in $\{rem, try, crit, exit\}$, initially rem

try_i
Effect:
$region_i := try$

$crit_i$
Precondition:
$region_i = try$
for all j, $region_j \neq crit$
Effect:
$region_i := crit$

$exit_i$
Effect:
$region_i := exit$

rem_i
Precondition:
$region_i = exit$
Effect:
$region_i := rem$

Classes and Bounds:
$crit = \{crit_i, 1 \leq i \leq n\}$, bounds $[0, \infty]$
$\{rem_i\}$, $1 \leq i \leq n$, bounds $[0, \infty]$

Finally, I present the algorithm. It is modelled formally as a single MMT automaton containing several processes sharing read-write memory. The state consists of the state of the shared memory (in this case, just the single variable x), plus the local states of all the processes. Each class of the fairness partition consists of actions of just one of the processes.

In this algorithm, each process i that is trying to obtain the resource tests the shared variable x until it finds the value equal to 0. After it finds this value, process i sets x to its own index i. Then it checks that x is still equal to i. If so, process i obtains the resource, and otherwise, it goes back to the beginning, testing for $x = 0$. When a process i exits, it resets x to 0.

A possible problem with the algorithm as described so far is that two processes, i and j, might *both* test x and find its value to be 0. Then i might set $x := i$ and immediately check and find its own index, and then j might do the same. This execution is illustrated in Figure 1.

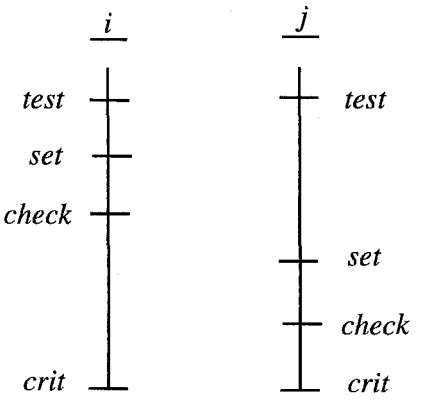

Fig. 1. Bad interleaving prevented by Fischer mutual exclusion algorithm

In order to avoid this bad interleaving, a simple time restriction is used. Each of the actions $test_i$, set_i, $check_i$, $crit_i$, $reset_i$, rem_i, for each i, comprises a singleton class. The bounds assigned to all the classes are $[0, \infty]$, with the following exceptions: set_i gets assigned $[0, a]$ and $check_i$ gets assigned $[b, \infty]$, for some constants a, b, where $a < b$.

The two bounds a and b prevent the bad interleaving in Figure 1 as follows. Any process i that sets $x := i$ is made to wait long enough before checking to ensure that any other process j that tested x before i set x (and therefore might subsequently set x to its own index) has already set x to its index. That is, there should be no processes left at the point of setting, when i finally checks.

Automaton F: Fischer Mutual Exclusion Algorithm

Actions:

Input:

try_i, $1 \leq i \leq n$

$exit_i$, $1 \leq i \leq n$

Output:

$crit_i$, $1 \leq i \leq n$

rem_i, $1 \leq i \leq n$

Internal:

$test_i$, $1 \leq i \leq n$

set_i, $1 \leq i \leq n$

$check_i$, $1 \leq i \leq n$

$reset_i$, $1 \leq i \leq n$

State components:

pc_i, $1 \leq i \leq n$, each in $\{rem, test, set, check, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem

x, an integer in $[0, n]$, initially 0

try_i
Effect:
$pc_i := test$

$test_i$
Precondition:
$pc_i = test$
Effect:
if $x = 0$ then $pc_i := set$

set_i
Precondition:
$pc_i = set$
Effect:
$x := i$
$pc_i := check$

$check_i$
Precondition:
$pc_i = check$
Effect:
if $x = i$
then $pc_i := leave\text{-}try$
else $pc_i := test$

$crit_i$
Precondition:
$pc_i = leave\text{-}try$
Effect:
$pc_i := crit$

$exit_i$
Effect:
$pc_i := reset$

$reset_i$
Precondition:
$pc_i = reset$
Effect:
$x := 0$
$pc_i := leave\text{-}exit$

rem_i
Precondition:
$pc_i = leave\text{-}exit$
Effect:
$pc_i := rem$

Classes and Bounds:
Assume $a < b$.
$\{test_i\}$, $1 \leq i \leq n$, bounds $[0, \infty]$
$\{set_i\}$, $[0, a]$
$\{check_i\}$, $[b, \infty]$
$\{crit_i\}$, $[0, \infty]$
$\{reset_i\}$, $[0, \infty]$
$\{rem_i\}$, $[0, \infty]$

Consider the composition $F \times U$. When this is transformed into a timed automaton, the only nontrivial state components that are added by the transformation are now, and $last(set_i)$ and $first(check_i)$, $1 \leq i \leq n$. (Here and elsewhere, I am using the convention of naming a singleton class by the single action contained in the class. Also, for simplicity, I ignore trivial *first* and *last* components.) Likewise, when $M \times U$ is transformed into a timed automaton, the only nontrivial added state component is now. Note that the external actions in each of the compositions $F \times U$ and $M \times U$ are try_i, $crit_i$, $exit_i$ and rem_i, $1 \leq i \leq n$.

I claim that the composition $F \times U$ is an implementation of $M \times U$, in the sense that $F \times U \leq M \times U$. To show the implementation, I define a mapping r from $F \times U$ to $M \times U$. Here, dot notation is used to indicate component automata, as well as state components.

- $r(s).now = s.now$.
- $r(s).U.region_i = s.U.region_i$.
- $r(s).M.region_i = \begin{cases} try & \text{if } s.F.pc_i \in \{test, set, check, leave\text{-}try\}, \\ crit & \text{if } s.F.pc_i = crit, \\ exit & \text{if } s.F.pc_i \in \{reset, leave\text{-}exit\}, \\ rem & \text{if } s.F.pc_i = rem. \end{cases}$

In order to show that r is a weak refinement, I first prove some invariants. The main invariant is *mutual exclusion*, i.e., that there do not exist two different users whose regions are both *crit*. Mutual exclusion is proved by means of a series of auxiliary invariants; these invariants and their proofs are due to Luchangco [18], and are based on those used by Abadi and Lamport [1]. The first property is obvious from the general definitions of the *last* functions – it says that the *last* value is no later than the current time plus the upper bound for the class.

Lemma 16. *The following is true of every reachable state of $F \times U$:*
If $pc_i = set$ then $last(set_i) \leq now + a$.

This lemma can be used to prove the following key claim. It says that the earliest time a successful $check_i$ can happen is after the set_j of any j that has already passed the test. This lemma serves to rule out the bad interleaving in Figure 1, in which the sequence set_i, $check_i$, set_j, $check_j$ occurs, and both checks are successful.

Lemma 17. *The following is true of every reachable state of $F \times U$:*
If $pc_i = check$ and $x = i$ and $pc_j = set$ then $first(check_i) > last(set_j)$.

Proof. By induction. Again, I consider steps of the form $s' \xrightarrow{\pi} s$. Here, the only interesting cases are:

1. $\pi = set_i$
   Then $s.first(check_i) = s.now + b$ and $last(set_j) \leq s.now + a$, by Lemma 16. Since $a < b$, the inequality follows.
2. $\pi = test_j$ and $s'.x = 0$ (i.e., the test is successful)
   Then $s.x = 0$, making the statement true vacuously.

□

The next lemma says that if a process i is in the critical region (or just before or just after it), then $x = i$ and no other process can be about to set.

Lemma 18. *The following is true of every reachable state of $F \times U$:*
If $pc_i \in \{leave\text{-}try, crit, reset\}$ then $x = i$ and for all j, $pc_j \neq set$.

Proof. By induction. The interesting cases are:

1. $\pi = check_i$ and $s'.x = i$ (i.e., the check is successful)
   Then $s.x = s'.x = i$, and Lemma 17, together with the time requirements, imply that no j has $s'.pc_j = set$, nor (therefore) $s.pc_j = set$.

2. $\pi = set_j$, $j \neq i$
   This is impossible because the inductive hypothesis implies that there can be no j with $s'.pc_j = set$.
3. $\pi = reset_j$, $j \neq i$
   This is impossible because if it were true, the inductive hypothesis applied to both i and j would imply the contradictory requirements $s'.x = i$ and $s'.x = j$.
4. $\pi = test_j$, $j \neq i$, and $s'.x = 0$ (i.e., the test is successful)
   Then the inductive hypothesis implies that $s'.pc_i \notin \{leave\text{-}try, crit, reset\}$, so $s.pc_i \notin \{leave\text{-}try, crit, reset\}$, which implies that the condition is true vacuously.

□

Now it is not hard to see that Lemma 18 implies the mutual exclusion property.

Lemma 19 (Mutual Exclusion). *The following is true of every reachable state of $F \times U$:*
There do not exist i, j, $i \neq j$, such that $pc_i = pc_j = crit$.

Now, using the mutual exclusion invariant, it is possible to argue that r is a weak refinement. This claim says a bit more than just that the algorithm satisfies the mutual exclusion *property*: it says that the algorithm actually implements a *mutual exclusion system*, with a particular interface, input/output conventions, etc.

Lemma 20. *The function r is a weak refinement from $F \times U$ to $M \times U$.*

Proof. Straightforward. The dangerous steps to check are those of the form $s' \xrightarrow{crit_i} s$, where some process j is in the critical region in state s'. But then both i and j would be in the critical region, in state s, which violates the mutual exclusion property for s, a contradiction. □

Theorem 21. *Let F be the Fischer mutual exclusion algorithm, U the composed user automaton and M the mutex object. Then $F \times U \leq_{\mathrm{WR}} M \times U$, and therefore $F \times U \leq M \times U$.*

Note that this proof, while technically a simulation proof, does not really demonstrate the power of the method, since the key property being proved, mutual exclusion, is shown just using invariants. The remaining examples in this paper better illustrate the power of the simulation method. I will return to the Fischer mutual exclusion example twice more, once to argue time bounds and once to argue liveness, both using simulations.

For later use, I also state the following invariant:

Lemma 22. *The following is true of every reachable state of $F \times U$: If $x = i$ then $pc_i \in \{check, leave\text{-}try, crit, reset\}$.*

5 Using Simulations to Prove Time Bounds

In the previous section, I introduced the MMT model and used it to describe Fischer's mutual exclusion algorithm. I then used invariants and simulations to prove that the algorithm satisfies the mutual exclusion property. In this section, I show how simulations can be used to prove something more than just basic safety properties – they can also be used to prove time bounds.

In the Fischer example, the implementation automaton had upper and lower bound assumptions, which were used in proving the basic safety property. The specification, however, did not include any time bounds. The main idea for proving time bounds via simulations is to include lower and upper time bounds on the classes of the specification MMT automaton. I demonstrate the power of this method with three examples: a simple counting process, a two-process race system, and Fischer's mutual exclusion algorithm. I also indicate how the same method can be used to prove time bounds for asynchronous algorithms.

I believe that these examples demonstrate that the power of simulation methods is much greater in the real-time setting than it is in the asynchronous setting. For in the asynchronous setting there are usually *liveness conditions* rather than *time bounds* to be proved. Proofs of liveness conditions require some extra machinery, e.g., temporal logic, in addition to simulations, but time bounds can be proved just using simulations. I will say more about liveness in Section 6.

This method of proving timing properties is derived from [20]. Also, the first two examples are simplifications of examples in that paper.

5.1 Example: Counting Process

The first example involves a simple automaton that just counts down from some fixed positive integer k and then reports its completion. If the time between the automaton's steps is always in a limited range, say $[c_1, c_2]$, then it should be possible to prove a corresponding range of times until the report occurs.

Automaton *Count*: Counting Automaton

Actions:
- Output:
  - *report*
- Internal:
  - *decrement*

State components:
- *count*, initially $k > 0$
- *reported*, Boolean, initially *false*

decrement
Precondition:
$count > 0$
Effect:
$count := count - 1$

report
Precondition:
$count = 0$
$reported = false$
Effect:
$reported := true$

Classes and Bounds:
$\{report\}$, bounds $[c_1, c_2]$
$\{decrement\}$, each with bounds $[c_1, c_2]$

Informally, it is easy to see that the time until a *report* occurs can be any time in the interval $[(k+1)c_1, (k+1)c_2]$. In order to prove this formally, I express these time bound assumptions by a trivial high-level reporting automaton called *Report*. In the following formal description, I parameterize the name of this automaton by the time bounds that it is to guarantee. This is for the purpose of disambiguation, because in the next example, I will use another *Report* automaton, with different time bounds.

Automaton *Report*$[d_1, d_2]$: Reporting Automaton

Actions:
Output:
report

State components:
reported, Boolean, initially *false*.

report
Precondition:
$reported = false$
Effect:
$reported := true$

Classes and Bounds:
$\{report\}$, bounds $[d_1, d_2]$.

I show that *Count* implements *Report*$[(k+1)c_1, (k+1)c_2]$, using a weak forward simulation. The multiple values permitted by a forward simulation are needed because the simulation is expressed in terms of inequalities. Specifically, I define $(s, u) \in f$ provided that the following hold:

- $u.now = s.now$,
- $u.reported = s.reported$,

- $u.last(report) \geq$
  $\begin{cases} s.last(decrement) + s.count \cdot c_2 & \text{if } s.count > 0, \\ s.last(report) & \text{otherwise.} \end{cases}$
- $u.first(report) \leq$
  $\begin{cases} s.first(decrement) + s.count \cdot c_1 & \text{if } s.count > 0, \\ s.first(report) & \text{otherwise.} \end{cases}$

The idea of the simulation is as follows. The *now* and *reported* component definitions are straightforward. The $last(report)$ component is constrained to be at least as large as a quantity that is calculated in terms of the state (including time components) of *Count*. This quantity is a calculated upper bound on the latest time until a *report* action is performed by *Count*. There are two cases: If $count > 0$, then this time is bounded by the last time at which the first *decrement* can occur, plus the additional time required to do $count - 1$ *decrement* steps, followed by a *report*; since each of these *count* steps could take at most time c_2, this additional time is at most $count \cdot c_2$. On the other hand, if $count = 0$, then this time is bounded by the last time at which the *report* can occur. The inequality expresses the fact that this calculated bound on the actual time until *report* is at most equal to the upper bound to be proved. The interpretation of the $first(report)$ component is symmetric – it should be no larger than a calculated lower bound on the earliest time until a *report* action is performed by *Count*.

In order to prove that f is a weak forward simulation, I use the simple invariant "if $count > 0$ then $reported = false$", plus basic properties of the *Count* automaton, of the style of Lemma 16.

Lemma 23. *The relation f is a weak forward simulation from Count to $Report[(k+1)c_1, (k+1)c_2]$.*

Proof. The proof proceeds in the usual way for forward simulations, verifying the three properties in the definition one by one. The inequalities are treated in the same manner as any other type of relation between the states. The correspondence between *now* values is immediate.

For the correspondence between start states, let s and u be the unique start states of *Count* and R, respectively. I show that $(s, u) \in f$. The first two parts of the definition of f are immediate; consider the third part. The definition of R implies that $u.last(report) = (k+1)c_2$, while the definition of *Count* implies that $s.count > 0$ and $s.last(decrement) + s.count \cdot c_2 = c_2 + kc_2 = (k+1)c_2$. Therefore, $u.last(report) = s.last(decrement) + s.count \cdot c_2$, which shows the third part of the definition of f. The fourth part is analogous to the third.

Finally, for the correspondence between steps, consider cases based on types of transitions. For example, consider a transition $s' \xrightarrow[Count]{decrement} s$, where $u' \in f[s']$. Since *decrement* is enabled in s', it must be that $s'.count > 0$. Suppose that also $s.count > 0$. The fact that $u' \in f[s']$ means that $u'.now = s'.now$, $u'.reported = s'.reported$, $u'.last(report) \geq s'.last(decrement) + s'.count \cdot c_2$, and $u'.first(report) \leq s'.first(decrement) + s'.count \cdot c_1$. It suffices to show that $u' \in f[s]$.

The first two conditions in the definition of f carry over immediately. For the third condition, the left-hand side of the inequality, $last(report)$, does not change, while on the right-hand side, $last(decrement)$ is increased by at most c_2, while the second term decreases by exactly c_2. (The reason why $last(decrement)$ is increased by at most c_2 is as follows: the construction of the timed automaton from the MMT automaton for *Count* — captured in the invariants — implies that $s'.now \leq s'.last(decrement)$, but note that $s.last(decrement) = s.now + c_2$ and $s.now = s'.now$.) So the inequality still holds after the step.

Similar arguments can be made for the lower bound, and for the case of decrementing to 0. □

Theorem 24. *Count* $\leq_{\text{WF}}$ *Report*$[(k+1)c_1, (k+1)c_2]$, *and therefore Count* $\leq$ *Report*$[(k+1)c_1, (k+1)c_2]$.

The main content of this theorem is that *Count* satisfies the timing requirements. In the rest of the paper, I will focus on upper bound results and proofs; lower bounds can be stated and proved similarly.

5.2 Example: Two-Process Race

This is an example suggested by Pnueli [34] as a test case for proof methods for timing-based systems. Consider an MMT automaton *Race* with state variables *count*, *flag*, and *reported*. The automaton can be thought of as consisting of two tasks. The *main* task increments the variable *count* as long as the *flag* is false, then decrements *count* back to 0. When the value of the *count* has been restored to 0, the *main* task reports its completion. There is a separate *interrupt* task whose sole job is to set the *flag* to *true*.

Automaton *Race*: Two-Process Race System

Actions:
- Output:
  - *report*
- Internal:
  - *increment*
  - *decrement*
  - *set*

State components:
- *count*, a nonnegative integer, initially 0
- *flag*, a Boolean, initially *false*
- *reported*, a Boolean, initially *false*

increment
Precondition:
$flag = false$
Effect:
$count := count + 1$

decrement
Precondition:
$flag = true$
$count > 0$
Effect:
$count := count - 1$

set
Precondition:
$flag = false$
Effect:
$flag := true$

report
Precondition:
$flag = true$
$count = 0$
$reported = false$
Effect:
$reported := true$

Classes and Bounds:
$main = \{increment, decrement, report\}$, bounds $[c_1, c_2]$
$int = \{set\}$, bounds $[0, a]$

Let $C = c_2/c_1$.

The correctness specification is the automaton $Report[0, a + c_2 + Ca]$, where *Report* is defined in the previous example. (I am only proving the upper bound here, so I use a lower bound of 0.) The reason why $a + c_2 + Ca$ is a correct upper bound is, intuitively, as follows. Within time a, the *int* task sets the *flag* to *true*. During this time, the *count* could reach at most a/c_1. Then it takes at most time $(a/c_1)c_2 = Ca$ for the *main* task to decrement *count* to 0, and another c_2 to report.

I show that this bound is correct by a simple weak forward simulation from *Race* to $Report[0, a + c_2 + Ca]$: Specifically, I define $(s, u) \in f$ provided that the following hold:

- $u.now = s.now$.
- $u.reported = s.reported$.
- $u.last(report) \geq$
$$\begin{cases} s.last(int) + (s.count + 2)c_2 + C(s.last(int) - s.first(main)) \\ \quad \text{if } s.flag = false \text{ and } s.first(main) \leq s.last(int), \\ s.last(main) + (s.count)c_2 \\ \quad \text{otherwise.} \end{cases}$$

The idea of the last inequality is as follows. If $flag = true$, then the time remaining until *report* is just the time for the *main* task to do the remaining *decrement* steps, followed by the final *report*. The same reasoning holds if *flag* is still *false*, but must become *true* before there is time for another *increment* to occur, i.e., if $s.first(main) > s.last(int)$. Otherwise, there is time for at least one more *increment* to occur, i.e., $s.flag = false$ and $s.first(main) \leq s.last(int)$; then the first case of the inequality for $last(report)$ applies.

In this case, after the *set*, it might take as long as time $(count + 1)c_2$ for the *main* task to count down from the current *count*, and then to *report*. But the current *count* could be increased by some additional *increment* events that

happen before the *set*. The largest number of these that might occur is $1 + (last(int) - first(main))/c_1$. Multiplying this by c_2 gives the extra time required to decrement this additional count.

Again, the only invariants needed are general properties of the sort in Lemma 16. Now the standard proof methods yield:

Lemma 25. *The relation f is a weak forward simulation from Race to Report*$[0, a + c_2 + Ca]$.

Theorem 26. *Race* $\leq_{\mathrm{WF}}$ *Report*$[0, a+c_2+Ca]$, *and therefore Race* $\leq$ *Report*$[0, a + c_2 + Ca]$.

5.3 Example: Fischer Mutual Exclusion Algorithm

As the third example of proving time bounds via simulations, I return to the Fischer mutual exclusion algorithm, F. This time, I prove an upper bound for the time from when some process is trying to obtain the resource and no one is critical, until some process is critical. There is also a corresponding bound (trivial to prove) for the remainder region.

In order to prove these time bounds, I must assume additional time bounds for process steps, besides the upper bound of a on *set* steps and lower bound of b on *check* steps already used for proving mutual exclusion. For simplicity, I assign the same upper bound of a used for *set* steps to all of the other locally controlled steps except for the *check* steps, i.e., to the *test*, *crit*, *reset* and *rem* steps. (Upper bounds are not needed for *try* or *exit* steps.) I also assign an upper bound of c to the *check* steps, for some $c \geq b$. The result is a new MMT automaton, which I call F'.

I also express the time bound requirements using an MMT automaton, M'. M' is the same as M except that the class *crit* of $crit_i$ actions has the bounds $[0, 2c+5a]$ and each class rem_i has the bounds $[0, 2a]$. (Note: It appears that the upper bound can be improved to $2c + 5a - b$, at the cost of some complication. Details appear in [18].)

For the proof, I could give a direct weak forward simulation from $F' \times U$ to $M' \times U$, but it seems useful instead to introduce an intermediate level of abstraction. The intermediate level expresses certain *milestones* toward the goal of some process reaching the critical region. (This general approach is derived from [18].) Specifically, from the point when actions in the class *crit* becomes enabled (that is, when some process enters the trying region when no process is critical, or when some process leaves the critical region), there is a later step at which some process first converts x from 0 to a process index; I call this a *seize* step. Then there is a later step at which some process last sets x to an index, leaving no other processes with program counters equal to *set* (this means that no one will do another *set*, before some process reaches the critical region); I call this a *stabilize* step. The milestones I will consider are just the *seize* and *stabilize* steps.

I will argue that, from the time of enabling of *crit*, a *seize* step occurs at most time $c+3a$ later. Then from the time of a *seize* step, a *stabilize* step occurs at most time a later. And from the time of a *stabilize* step, a *crit* step occurs at most time $c + a$ later. The total is $2c + 5a$, as claimed. To express these milestones, I describe an intermediate MMT automaton I'.

Automaton I': Intermediate Automaton for Fischer Algorithm

Actions:

Input:

try_i, $1 \leq i \leq n$

$exit_i$, $1 \leq i \leq n$

Output:

$crit_i$, $1 \leq i \leq n$

rem_i, $1 \leq i \leq n$

Internal:

seize, $1 \leq i \leq n$

stabilize, $1 \leq i \leq n$

State components:

$region_i$, $i \in I$, an element of $\{rem, try, crit, exit\}$, initially *rem*

status, an element of $\{start, seized, stab\}$, initially *start*

try_i

Effect:

$region_i := try$

seize

Precondition:

$\exists i$, $region_i = try$

$\forall i$, $region_i \neq crit$

$status = start$

Effect:

$status := seized$

stabilize

Precondition:

$status = seized$

Effect:

$status := stab$

$crit_i$

Precondition:

$region_i = try$

$status = stab$

Effect:

$region_i := crit$

$status := start$

$exit_i$

Effect:

$region_i := exit$

rem_i

Precondition:

$region = exit$

Effect:

$region := rem$

Classes and Bounds:

$\{seize\}$, bounds $[0, c + 3a]$

$\{stabilize\}$, bounds $[0, a]$

$crit = \{crit_i : 1 \leq i \leq n\}$, bounds $[0, c + a]$

$\{rem_i\}$, $1 \leq i \leq n$, bounds $[0, 2a]$

There is a simple weak forward simulation from $I' \times U$ to $M' \times U$. Namely, define $(s, u) \in f$ provided that the following hold.

- $u.now = s.now$.
- $u.U.region_i = s.U.region_i$ for all i.
- $u.M'.region_i = s.I'.region_i$ for all i.
- $u.last(crit) \geq \begin{cases} s.last(seize) + c + 2a & \text{if } s.status = start, \\ s.last(stabilize) + c + a & \text{if } s.status = seized, \\ s.last(crit) & \text{if } s.status = stab. \end{cases}$
- $u.last(rem_i) \geq s.last(rem_i)$.

Here, the inequality for $last(crit)$ uses a calculated upper bound on the time until $I' \times U$ performs a *crit* action. This calculation is based on a series of cases. Working backwards, in the last case, where $status = crit$, *crit* is enabled in $I' \times U$, and a calculated upper bound is just $last(crit)$. In the next-to-last case, *stabilize* is enabled, and after it occurs, only the worst-case time $c + a$ for *crit* remains. In the first case, *seize* is enabled, and after it occurs, the additional remaining time is at most the worst-case time for *stabilize* and *crit* to occur, in succcession.

To show that f is a weak forward simulation, I use the following invariant of $I' \times U$: If $region_i = crit$ for some i, then $status = start$. (Also, once again, I need some properties of the sort in Lemma 16. From now on, I will omit explicit mention of such properties.)

Lemma 27. *The relation f is a weak forward simulation from $I' \times U$ to $M' \times U$.*

Theorem 28. *$I' \times U \leq_{\mathrm{WF}} M' \times U$, and therefore $I' \times U \leq M' \times U$.*

Now I consider the simulation from $F' \times U$ to $I' \times U$. Define $(s, u) \in g$ if the following hold. (All unbound uses of process indices are implicitly universally quantified.)

- $u.now = s.now$.
- $u.U.region_i = s.U.region_i$.
- $u.I'.region_i = \begin{cases} rem & \text{if } s.F'.pc_i = rem, \\ crit & \text{if } s.F'.pc_i = crit, \\ exit & \text{if } s.F'.pc_i \in \{reset, leave\text{-}exit\}, \\ try & \text{otherwise.} \end{cases}$
- $u.status = \begin{cases} start & \text{if } s.x = 0, \text{ or } \exists i : s.pc_i \in \{crit, reset\}, \text{ else} \\ seized & \text{if } \exists i : s.pc_i = set, \text{ else} \\ stab. \end{cases}$

- $u.last(seize) \geq s.last(reset_i) + c + 2a$ if $s.pc_i = reset$.
- $u.last(seize) \geq \min_i\{h(i)\}$ if $s.x = 0$,

  where $h(i) = \begin{cases} s.last(check_i) + 2a & \text{if } s.pc_i = check, \\ s.last(test_i) + a & \text{if } s.pc_i = test, \\ s.last(set_i) & \text{if } s.pc_i = set, \\ \infty, & \text{otherwise.} \end{cases}$

- $u.last(stabilize) \geq s.last(set_i)$ if $s.pc_i = set$.

- $u.last(crit) \geq \begin{cases} s.last(check_i) + a & \text{if } s.pc_i = check \wedge s.x = i, \\ s.last(crit_i) & \text{if } s.pc_i = leave\text{-}try. \end{cases}$

- $u.last(rem_i) \geq \begin{cases} s.last(reset_i) + a & \text{if } s.pc_i = reset, \\ s.last(rem_i) & \text{if } s.pc_i = leave\text{-}exit. \end{cases}$

The *now* and region correspondences and *status* definition are straightforward. The first inequality for *seize* says that if some process is about to *reset*, then the time until the variable is seized is at most an additional $c+2a$ after the *reset* occurs. The second inequality for *seize* says that if $x = 0$ (which means that no process is about to *reset*), then the time until the variable is seized is determined by the minimum of a set of possible times, each corresponding to some candidate process that might set x. For instance, if some process i is about to set x, then the corresponding time is only the maximum time until it does so, while if i is about to test x, then the corresponding time is an additional a after the *test* occurs. The interpretations for the remaining inequalities are similar.

To show that g is a weak forward simulation, I use the invariant of $F' \times U$ given in Lemma 18. Then:

Lemma 29. *The relation g is a weak forward simulation from $F' \times U$ to $I' \times U$.*

Proof. (Rough sketch) Each external step simulates a corresponding external step of $I' \times U$. A *set* step that changes x from 0 to a process index simulates *seize*, while a *set* step that occurs when no process is in *critical* or *reset*, and after which there are no other processes with $pc = set$, simulates *stabilize*. A *set* step that satisfies both of these conditions simulates both *seize* and *stabilize*, in that order. All other steps simulate a trivial timed execution fragment with no actions. □

Theorem 30. *Let F' be the Fischer mutual exclusion algorithm with time bounds, U the composed user automata, and I' the Fischer intermediate automaton with time bounds. Then $F' \times U \leq_{\mathrm{WF}} I' \times U$, and therefore $F' \times U \leq I' \times U$.*

Corollary 31. *Let F' be the Fischer mutual exclusion algorithm with time bounds, U the composed user automaton, and M' the mutex object with time bounds. Then $F' \times U \leq M' \times U$.*

5.4 Example: Dijkstra's Mutual Exclusion Algorithm

Using these simulation methods, it is also possible to carry out rigorous time complexity analysis for asynchronous algorithms. It might not be clear to the reader what it means to analyze the time complexity for an asynchronous algorithm, since asynchronous algorithms, by definition, have no timing assumptions on their steps; thus, it is impossible to prove any unconditional time bounds for

their user-visible behavior. However, it is reasonable to assume upper bounds on time for various steps, since just assuming upper bounds (and no lower bounds) does not restrict the possible executions of the algorithm. Given such upper bounds on steps, it is often possible to prove upper bounds on the time for interesting behavior.

Analyzing the time complexity of asynchronous algorithms is often a very difficult task. Generally, such analysis has been done in an informal and operational style. But it is not hard to see that the simulation methods described in this paper can also be used in the asynchronous setting, yielding proofs that express the key insights, yet can be done in complete formal detail.

I illustrate with sketches of two examples. In this subsection, I consider Dijkstra's asynchronous algorithm [8] for mutual exclusion using read-write shared memory. In the following subsection, I consider a simple leader election algorithm.

The Dijkstra algorithm, rewritten to fit the precondition-effect notation, is as follows.

Automaton D: Dijkstra's Mutual Exclusion Algorithm

Actions:

Input:

try_i, $1 \leq i \leq n$

$exit_i$, $1 \leq i \leq n$

Output:

$crit_i$, $1 \leq i \leq n$

rem_i, $1 \leq i \leq n$

Internal:

$announce_i$, $1 \leq i \leq n$

$test1_i$ $1 \leq i \leq n$

$test2(j)_i$, $1 \leq i, j \leq n$

set_i, $1 \leq i \leq n$

$advance_i$, $1 \leq i \leq n$

$check_i(j)$, $1 \leq i, j \leq n$

$reset_i$, $1 \leq i \leq n$

State components:

pc_i, $1 \leq i \leq n$, an element of
$\{rem, ann, test1, test2(j), set, adv, check, leave\text{-}try, crit, reset, leave\text{-}exit\}$,
each initially rem

x, an element of I, initially arbitrary

$flag_i$, $i \in I$, an element of $\{0, 1, 2\}$, initially 0

S_i, $i \in I$, a set of process indices, initially $\{i\}$

try_i
Effect:
$pc_i := ann$

$announce_i$
Precondition:
$pc_i = ann$
Effect:
$flag_i := 1$
$pc_i := test1$

$test1_i$
Precondition:
$pc_i = test1$
Effect:
if $x = i$ then $pc_i := adv$
else $pc_i := test2(x)$

$test2(j)_i$
Precondition:
$pc_i = test2(j)$
Effect:
if $flag_j = 0$ then $pc_i := set$
else $pc_i := test1$

set_i
Precondition:
$pc_i = set$
Effect:
$x := i$
$pc_i := test1$

$advance_i$
Precondition:
$pc_i = adv$
Effect:
$flag_i := 2$
$pc_i := check$
$S_i := \{i\}$

$check_i(j)$
Precondition:
$pc_i = check$
$j \notin S_i$
Effect:
if $flag_j = 2$ then $pc_i := ann$
else
$S_i := S_i \cup \{j\}$
if $|S_i| = n$ then $pc_i := leave\text{-}try$

$crit_i$
Precondition:
$pc_i = leave\text{-}try$
Effect:
$pc_i := crit$

$exit_i$
Effect:
$pc_i := reset$

$reset_i$
Precondition:
$pc_i = reset$
Effect:
$flag_i := 0$
$pc_i := leave\text{-}exit$

rem_i
Precondition:
$pc_i = leave\text{-}exit$
Effect:
$pc_i := rem$

Classes and Bounds:
For each i, there is a separate class for each type of locally controlled action; each class has bounds $[0, a]$.

It is not difficult to prove that algorithm $D \times U$ satisfies mutual exclusion, where U is the composed user automaton used in the Fischer example. The proof is by induction as usual, with the following as a key auxiliary invariant: $\neg[\exists i, j : (i \neq j) \wedge (i \in S_j) \wedge (j \in S_i)]$. Then a weak forward simulation can be

given from $D \times U$ to $M \times U$, where M is the mutex object used in the Fischer algorithm.

In order to prove an upper bound on time, I once again modify the specification M by adding time bounds. This time, however, the time bound for the *crit* class is $(3n+11)a$, while the bound for each rem_i is still $2a$. I call the resulting specification MD.

The proof strategy is similar to the one for the Fischer algorithm. I define an intermediate automaton ID, also with *seize* and *stabilize* milestones, plus an additional *dropback* milestone. The new code is as follows:

ID

dropback
Precondition:
$status = stab$
Effect:
$status := drop$

$crit_i$
Precondition:
$region_i = try$
$status = drop$
Effect:
$region_i := crit$
$status := start$

Classes and Bounds:
$\{seize\}$, bounds $[0, (n+5)a]$
$\{stabilize\}$, bounds $[0, 2a]$
$\{dropback\}$, bounds $[0, (n+1)a]$
$crit = \{crit_i : 1 \leq i \leq n\}$, bounds $[0, (n+3)a]$
$\{rem_i\}$, $1 \leq i \leq n$, bounds $[0, 2a]$

The action *seize* is deemed to occur when no process is in the critical region or at the point of *reset*, some process is trying, and x acquires a value that is the index of a trying process with $flag \neq 0$ (i.e., an announced trying process). Once *seize* is enabled, (when no process is in the critical region but some process is trying, and x does not have such a value), it is at most one step (time a) until any newly-exiting process j has reset its flag to 0, then at most $n+4$ steps until some process reaches *test1*, then *test2*, then *set* – this accomplishes a *seize*. (There is a technicality: a process could be interrupted at *test2* by the arrival (more particular, the announcement) of the process j whose index is already in x. But such an announcement itself serves to accomplish the *seize*.)

The *stabilize* action is deemed to occur when x settles down to a value that cannot be changed before someone reaches the critical region. By a similar argument to the one used for the Fischer algorithm, it takes at most 2 steps until residual effects are removed, thus producing stabilization.

The *dropback* action is deemed to occur when all but the process whose index is in x drop back to the first stage of the algorithm, where $flag = 1$ (or when someone goes critical before this happens). This takes at most $n+1$ steps. Then the remaining process goes to the critical region in at most $n+3$ steps.

The simulation from $ID \times U$ to $MD \times U$ is similar to that from $I' \times U$ to $M' \times U$. The simulation from $D \times U$ to $ID \times U$ is similar in style to that from

$F' \times U$ to $I' \times U$, but is more complicated because of the additional technical complications in this algorithm.

- $u.now = s.now$.
- $u.U.region_i = s.U.region_i$.
- $u.ID.region_i = \begin{cases} rem & \text{if } s.D.pc_i = rem, \\ crit & \text{if } s.D.pc_i = crit, \\ exit & \text{if } s.D.pc_i \in \{reset, leave\text{-}exit\}, \\ try & \text{otherwise.} \end{cases}$
- $u.status = \begin{cases} start & \text{if } \exists i, s.pc_i \in \{crit, reset\}, \text{ or } s.flag_{s.x} = 0, \text{ else} \\ seized & \text{if } \exists i \neq s.x : s.pc_i \in \{set\} \cup \{test2(j) : j \neq s.x\}, \text{ else} \\ stab & \text{if } \exists i \neq s.x : s.pc_i = adv \vee s.flag_i = 2, \text{ else} \\ drop. \end{cases}$

- $u.last(seize) \geq s.last(reset_i) + (n+4)a$ if $s.pc_i = reset$.
- $u.last(seize) \geq \min_i\{h(i)\}$ if $\forall i : s.pc_i \neq reset$,

$$\text{where } h(i) = \begin{cases} s.last(advance_i) + (n+3)a & \text{if } s.pc_i = adv, \\ s.last(check_i) + (n - |s.S_i| + 3)a & \text{if } s.pc_i = check, \\ s.last(announce_i) + 3a & \text{if } s.pc_i = ann \wedge s.x \neq i, \\ s.last(test2_i) + 3a & \text{if } s.pc_i = test2(j) \wedge j \neq s.x, \\ s.last(test1_i) + 2a & \text{if } s.pc_i = test1, \\ s.last(test2_i) + a & \text{if } s.pc_i = test2(s.x), \\ s.last(set_i) & s.pc_i = set, \\ s.last(announce_i) & s.pc_i = ann \wedge s.x = i, \\ \infty & \text{otherwise.} \end{cases}$$

- $u.last(stabilize) \geq \begin{cases} s.last(test2_i) + a & \text{if } s.pc_i = test2(j) \wedge j \neq s.x, \\ s.last(set_i) & \text{if } s.pc_i = set. \end{cases}$

- $u.last(dropback) \geq \begin{cases} s.last(advance_i) + na & \text{if } s.pc_i = adv \wedge i \neq s.x, \\ s.last(check_i) + (n - |s.S_i|)a & \text{if } s.pc_i = check \wedge i \neq s.x, \\ s.last(announce_i) & \text{if } s.pc_i = ann \wedge s.flag_i = 2, \\ s.last(crit_i) & \text{if } s.pc_i = leave\text{-}try. \end{cases}$

- $u.last(crit) \geq \min_i\{g(i)\}$ where $g(i) = \begin{cases} s.last(announce_i) + (n+2)a & \text{if } s.pc_i = ann \wedge s.x = i, \\ s.last(test1_i) + (n+1)a & \text{if } s.pc_i = test1 \wedge s.x = i, \\ s.last(advance_i) + na & \text{if } s.pc_i = adv, \\ s.last(check_i) + (n - |s.S_i|)a & \text{if } s.pc_i = check, \\ s.last(crit_i) & \text{if } s.pc_i = leave\text{-}try, \\ \infty & \text{otherwise.} \end{cases}$

- $u.last(rem_i) \geq \begin{cases} s.last(reset_i) + a & \text{if } s.pc_i = reset, \\ s.last(rem_i) & \text{if } s.pc_i = leave\text{-}exit. \end{cases}$

This definition may look formidable because of its size. However, as for thet Fischer algorithm, the *now* and region correspondences and *status* definition are straightforward. The remaining pieces of the definition are inequalities describing the progress toward the various goals. For each goal, the cases in the corresponding inequalities just trace this progress step-by-step.

In order to show that this is a weak forward simulation, we use the following invariant:

Lemma 32. *The following is true of every reachable state of $D \times U$:*

1. *If $pc_i \in \{leave\text{-}exit, rem\}$ then $flag_i = 0$.*
2. *If $pc_i \in \{test1, test2(j), set, adv\}$, then $flag_i = 1$.*
3. *If $pc_i \in \{check, leave\text{-}try, crit, reset\}$ then $flag_i = 2$.*
4. *If $pc_i = ann$, then $flag_i \in \{0, 2\}$.*

Lemma 33. *This relation is a weak forward simulation from $D \times U$ to $ID \times U$.*

Theorem 34. *Let D be the Dijkstra mutual exclusion algorithm, U the composed user automaton, and ID the Dijkstra intermediate automaton. Then $D \times U \leq_{\mathrm{WF}} ID \times U$, and therefore $D \times U \leq ID \times U$.*

Corollary 35. *Let D be the Dijkstra mutual exclusion algorithm, U the composed user automaton, and MD the mutex object with time bounds $[0, (3n+11)a]$ for the crit class and $[0, 2a]$ for each rem_i class. Then $D \times U \leq MD \times U$.*

5.5 Example: LeLann-Chang-Roberts Leader Election Algorithm

Luchangco [18] gives a formal proof of an upper bound on time for the LeLann-Chang-Roberts leader election algorithm for ring networks [4, 16]. The algorithm is simple: every processor in the network sends its processor identifier clockwise, and smaller identifiers that encounter larger identifiers are discarded. If a node receives its own identifier in a message, it elects itself as leader. An upper bound of c is assumed on the step time of each processor, and an upper bound of d is assumed for the time to deliver the oldest message in each channel. Under these assumptions, the time until a processor is elected is at most $(n+1)c + nd$, if there are n processors in the ring.

The difficulties in the proof involve the possible pile-up of identifiers in channels, if some processors and channels operate faster than others. Luchangco's proof is again based on the "milestone" idea. Here, there are n milestones; milestone i, $1 \leq i \leq n$, is said to be reached when the *slowest* token has progressed distance i around the ring. Formally, the specification automaton is just $Report[0, (n+1)c + nd]$, and the intermediate automaton describes milestones that are time at most $c + d$ apart, followed by a leader election *report* occurring at most time c after the final milestone. The mapping from the algorithm to the intermediate automaton computes how many milestones have been reached based on the least progress made by any identifier. Details can be found in [18].

5.6 Progress Functions

It should be apparent that the proofs in this section all have a similar style. In each case, the correspondence is a weak forward simulation. In each case, the simulation includes a set of inequalities involving calculated upper and lower

bounds. It is possible to formalize this common structure, and to establish general sufficient conditions for a relation with this structure to be a weak forward simulation. Doing this can systematize and slightly shorten the proofs.

In some more detail, the heart of each proof is a collection of definitions of "progress functions", one corresponding to each upper or lower bound to be proved, i.e., an upper bound expression $ub(C)$ and a lower bound expression $lb(C)$ for each class of the specification automaton. Then the portion of the simulation involving the *last* and *first* components is of the form: for all C, $u.last(C) \geq s.ub(C)$ and $u.first(C) \leq s.lb(C)$. The rest of the forward simulation is defined by the equations $u.now = s.now$ and $u.basic = f(s)$, where f is some function of the implementation state.

There are certain conditions that the progress functions have to satisfy in order for this correspondence to be a forward simulation. They are somewhat technical, so I paraphrase them roughly here. (A formal presentation, which includes some additional technicalities not mentioned here, appears in [20]; however, I remind the reader that there are some technical differences between the model of that paper and the one used here.)

For all classes C of the specification automaton:

1. If s is a start state then $f(s)$ is a start state; moreover, if C is enabled in $f(s)$ then $s.ub(C) \leq upper(C)$ and $s.lb(C) \geq lower(C)$.
2. For each non-time-passage step from s' to s, there is a "corresponding" fragment of the specification automaton, beginning with $f(s')$ and ending with $f(s)$, such that:
   (a) If a C step occurs in this fragment, then $s'.now \geq s'.lb(C)$.
   (b) If C remains enabled and no action in C occurs, then $s.ub(C) \leq s'.ub(C)$ and $s.lb(C) \geq s'.lb(C)$.
   (c) If C becomes newly enabled then $s.ub(C) \leq s'.now + upper(C)$ and $s.lb(C) \geq s'.now + lower(C)$.
3. For each time-passage step from s' to s:
   (a) $f(s) = f(s')$.
   (b) $s.now \leq s'.ub(C)$.
   (c) $s.ub(C) \leq s'.ub(C)$ and $s.lb(C) \geq s'.lb(C)$.

A general theorem in [20] says that if a collection of progress functions satisfies these conditions, then combining the functions as indicated above yields a forward simulation. The examples in that paper are developed in terms of the formal notion of progress functions. It appears to be straightforward to carry out the technical modifications of the theorem to fit the model of this paper, as well as to incorporate invariants for the implementation automaton into the conditions, thereby obtaining similar sufficient conditions for a weak forward simulation. However, this work remains to be done.

6 Liveness

It is sometimes desirable to prove liveness properties, e.g., properties that say that something eventually happens, even for systems with time bounds. In do-

ing this, it is sometimes useful to make liveness assumptions as well as timing assumptions. In this section, I give a way of describing systems with liveness assumptions, and a way, based on simulations and an "Execution Correspondence lemma", to verify that timed systems satisfy liveness properties. These notions are taken from [12]. I illustrate these methods with a proof of liveness for Fischer's mutual exclusion algorithm.

6.1 Augmented Timed Automata and Execution Correspondence

In order to describe liveness properties, I augment the timed automaton model. An *augmented timed automaton* consists of a timed automaton A, together with a subset L of the admissible timed executions called the *live timed executions*. (Normally, L is required to be of a restricted form – to contain an extension of every "finite" timed execution of A; however, I will not address this issue further in this paper, but refer the reader to [12].) A timed automaton A can be regarded as a special case of an augmented timed automaton, where the live timed executions are just the entire set of admissible executions. Define an admissible timed trace of A to be a *live timed trace* of (A, L) provided that it is the timed trace of some live timed execution of (A, L).

If (A, L) and (B, M) are augmented timed automata, and A and B are compatible, then I define the *composition* of (A, L) and (B, M) to be the augmented timed automaton $(A \times B, N)$, where N is the set of admissible executions of $A \times B$ that project onto A and B to give timed executions in L and M, respectively.

If (A, L) and (B, M) are augmented timed automata, I define $(A, L) \leq (B, M)$ provided that all the live timed traces of (A, L) are also live timed traces of (B, M). Then composition is substitutive with respect to $\leq$.

For any augmented timed automaton (A, L), I define L_d, the *live discrete executions*, to be the set of admissible discrete executions of A that sample timed executions in L.

Given a timing-based algorithm with some additional liveness assumptions, it is natural to express the algorithm as an augmented timed automaton (A, L). If one wants to show that the algorithm satisfies certain high-level liveness properties, then an effective strategy is to express the entire specification, safety plus liveness conditions, as another augmented timed automaton (B, M). Then showing that the algorithm satisfies the required liveness properties amounts to showing that $(A, L) \leq (B, M)$.

Now I describe one strategy for showing that $(A, L) \leq (B, M)$. This strategy involves first showing a simulation from A to B, yielding safety and timing properties as usual. But more strongly, it turns out that a simulation yields a close correspondence between any admissible discrete execution of A and some admissible discrete execution of B. For instance, consider the following definition of a correspondence between discrete executions of two timed automata. The definition is adapted slightly from [12]. Let A and B be timed automata with the same visible actions and let R be a relation over $states(A)$ and $states(B)$ that only relates states with the same *now* component. Let $\alpha = s_0 \pi_1 s_1 \pi_2 s_2 \cdots$ and $\alpha' = s'_0 \pi'_1 s'_1 \pi'_2 s_2 \cdots$ be admissible discrete executions of A and B, respectively. I

say that α and α' are *related by* R, or $(\alpha, \alpha') \in R$, provided that there is a total nondecreasing mapping m from natural numbers (i.e., indices of states in α) to natural numbers (i.e., indices of states in α'), such that:

1. $m(0) = 0$.
2. $(s_i, s'_{m(i)}) \in R$ for all i.
3. The execution fragment $s'_{m(i)} \cdots s'_{m(i+1)}$ contains the same sequence of timed visible actions as the step s_i, π_{i+1}, s_{i+1}.

That is, the initial states of α and α' correspond, corresponding states are R-related, and the fragment corresponding to any step has the same sequence of timed visible actions. I also say that A and B are *related by* R (or $(A, B) \in R$) if for every admissible discrete execution of A, there is an R-related admissible discrete execution of B.

The next result says that all of the simulations that have been presented in this paper yield a stronger correspondence than just inclusion of sets of admissible timed traces – they yield that the two automata are related by the simulation relation.

Lemma 36 (Execution Correspondence). *Suppose that R is a refinement, forward simulation, or image-finite backward simulation from A to B (or a weak version thereof). Then $(A, B) \in R$.*

To show that $(A, L) \leq (B, M)$, one first produces a simulation from A to B, in order to obtain a close correspondence between admissible discrete executions as just described. Given a live timed trace β of (A, L), one can obtain a timed execution α_1 in L that gives rise to β. Let α be any admissible discrete execution of A that samples α_1; Lemma 1 implies that α exists. Then $\alpha \in L_d$, by definition of L_d, and Lemma 2 implies that α also has β as its timed trace. Next, one uses the Execution Correspondence Lemma to obtain a corresponding admissible discrete execution α' of B, again with β as its timed trace. Then one shows that $\alpha' \in M_d$, by performing a case analysis based on the correspondence with α and the definition of L_d; this is the part of the proof that is specially tailored for each pair of augmented timed automata. Let α_2 be a live timed execution of (B, M) that is sampled by α'; α_2 exists by definition of M_d. Lemma 2 implies that α_2 also has β as its timed trace. This implies that β is a live timed trace of (B, M), as needed.

In this way, the liveness proof can be built incrementally on top of the simulation proof.

6.2 Example: Fischer Mutual Exclusion Algorithm

I consider a version F'' of Fischer's algorithm that is similar to the time-bounded version F', but instead of the explicit upper bounds of a on the steps, it just has *eventual* upper bounds for each type of step of each individual process. However, the upper bound of a on each set_i and the lower bound of b on each $check_i$ are retained, because they are needed to guarantee the safety property (mutual

exclusion). Formally, F'' can be described as an augmented timed automaton (F, L_F), where F is the original Fischer algorithm presented in Section 3 and L_F is a liveness condition for F giving the eventual bounds for all the non-*set* steps.

Similarly, I use a version I'' of the intermediate algorithm I', giving eventual bounds for the classes *seize*, *stabilize*, *crit* and rem_i for each i. Formally, by removing the time bounds from I', I can obtain a version I having neither time bounds nor liveness conditions; then I'' can be described as an augmented timed automaton (I, L_I), where L_I is a liveness condition for I giving eventual bounds for all the classes.

Finally, I use an eventual version M'' of M, giving eventual bounds for the classes *crit* and rem_i for each i. Formally, M'' can be described as an augmented timed automaton (M, L_M), where M is the untimed mutex object presented in Section 3 and L_M is a liveness condition for M giving the eventual bounds. Thus, the liveness at each level is described using fairness conditions in the usual style for I/O automata.

I carry out the liveness proof in two stages, first showing that every live timed trace of $I'' \times U$ is a live timed trace of $M'' \times U$, and then showing that every live timed trace of $F'' \times U$ is also a live timed trace of $I'' \times U$. I use the Execution Correspondence Lemma for each of these steps.

First, consider the mapping from $I'' \times U$ to $M'' \times U$. This proof rests on a simulation from $I \times U$ to $M \times U$. Note that I never actually gave such a simulation, but only a timed version, from $I' \times U$ to $M' \times U$. However, the untimed version of this simulation can be derived straightforwardly from the timed version, just by dropping the *first* and *last* components in the simulation.

A simple invariant for $I \times U$ is useful:

Lemma 37. *The following is true in all reachable states of $I \times U$: If status $\neq$ start then $region_i$ = try for some i.*

Let α be a live discrete execution of $I'' \times U$. Then α is an admissible discrete execution of $I \times U$. Then by the Execution Correspondence Lemma, get α', a corresponding admissible discrete execution of $M \times U$. It suffices to show that α' is also a live discrete execution of $M'' \times U$. I work by contradiction, supposing that α' is not live. There are two liveness conditions that α' can fail to satisfy: fairness for the class *crit*, or fairness for one of the classes rem_i. I illustrate the method by sketching the argument for *crit*.

Suppose that the liveness condition for *crit* fails. Then *crit* must be enabled from some point on in α', but no *crit* action ever occurs after that point. Since *crit* is enabled from the given point on in α', it follows that $region_i$ = *try* for some i and $region_i \neq$ *crit* for all i, from that point on. By the execution correspondence, from some corresponding point on in α, no *crit* action ever occurs, and the same region conditions hold. I consider cases.

1. *status* = *stab* occurs sometime after the designated point in α.
   Then Lemma 37 implies that $region_i$ = *try* at that point, for some i. Since no *crit* action ever occurs, this persists. Also, the condition *status* = *stab* must

persist, since the *crit* actions are the only ones that can change *status* from *stab* to anything else. Then fairness of α for *crit* implies that eventually some *crit* action occurs, a contradiction. Therefore, $status = stab$ never occurs.

2. $status = seized$ occurs sometime after the designated point in α.
   Then this condition persists, since the only action that can cause it to change is *stabilize*, which would lead to $status = stab$, contradicting the first case. But then fairness of α for *stabilize* implies that eventually *stabilize* must occur, a contradiction.
3. $status = start$ everywhere after the designated point in α.
   Then the correspondence implies that *seize* is enabled throughout that portion of α, so the fairness of α for *seize* implies that eventually *seize* occurs. But this leads to $status = seized$, contradicting the second case.

Next, consider the mapping from $F'' \times U$ to $I'' \times U$. This proof rests on a simulation from $F \times U$ to $I \times U$. Again, I have only given a timed version of this simulation, from $F' \times U$ to $I' \times U$, but the untimed simulation can be derived from the timed version by dropping the *first* and *last* components.

It is possible to use the Execution Correspondence Lemma based on the same kind of execution correspondence defined above, but a more efficient approach is to define a stronger (but messier) kind of correspondence, also implied by the simulations of this paper. This correspondence preserves not only the visible actions, but also certain information about internal actions. A little more precisely, to each internal step of the implementation automaton, I assign a particular sequence of internal actions of the specification automaton. Provided that, in the simulation proof, each internal step of the implementation always corresponds to an execution fragment with the assigned sequence, such a correspondence also holds for the complete admissible timed executions. That is, an additional condition 4 is added to the correspondence definition, saying that, if π_{i+1} is an internal action, then the execution fragment $s'_{m(i)} \cdots s'_{m(i+1)}$ contains exactly the sequence of internal actions assigned to the step $s_i \xrightarrow{\pi_{i+1}} s_{i+1}$.

In the current example, I assign the *seize* action to each *set* step that changes x from 0 to a process index. I assign the *stabilize* action to each *set* step occurring when no process is in *crit* or *reset*, and after which there are no other processes with $pc = set$. Note that both actions can be assigned to the same *set* step; in this case, they are assigned in the order *seize*, *stabilize*. The empty sequence is assigned to all other internal steps. These assigned sequences of internal actions are exactly what is simulated by the given internal steps, in the simulation proof I sketched earlier.

Now for the proof. Let α be a live discrete execution of $F'' \times U$. Then α is an admissible discrete execution of $F \times U$. Then by the Execution Correspondence Lemma (using the stronger correspondence just defined), get α', a corresponding admissible discrete execution of $I \times U$. It suffices to show that α' is also a live discrete execution of $I'' \times U$. I work by contradiction, supposing that α' is not live. There are four liveness conditions that α' can fail to satisfy: fairness for the class *seize*, *stab*, or *crit*, or fairness for one of the classes rem_i.

1. *seize*

   Suppose that the liveness condition for *seize* fails. Then *seize* must be enabled from some point on in α', but no *seize* action ever occurs after that point. Since *seize* is enabled from the given point on in α', it follows that $region_i = try$ for some i, $region_i \neq crit$ for all i, and $status = start$, from that point on. By the execution correspondence, from some corresponding point on in α, the same region conditions hold, either $x = 0$ or else some process has $pc = reset$, and no *set* step ever occurs that changes x from 0 to a process index. I consider cases.

   (a) A later point p is reached in α where no process is at *reset*.

   Then it must be that $x = 0$ at point p, and consequently from that point onward. This implies by Lemma 18 that, from point p onward, no process is ever at *leave-try*.

   If a point is reached after p at which some process i is at *set*, then fairness for set_i in α implies that a set from 0 to an index eventually occurs, a contradiction; therefore, no process is ever at *set*.

   If a point is reached after p at which some process i is at *test*, then fairness for $test_i$ says that $test_i$ eventually occurs, and since x stays equal to 0, the test succeeds, causing pc_i to becomes *set*, a contradiction. Therefore, no process is ever at *test*.

   The only remaining possibility is that some process i remains at *check* from point p onward. But then fairness for $check_i$ in α implies that $check_i$ eventually occurs, fails because $x = 0$, and results in process i going to *test*. This is again a contradiction.

   This covers all the possibilities (because some process must be in the trying region from point p onward), so this entire case is contradicted.

   (b) At every later point in α, some process is as *reset*.

   Note that in this fragment, no new process ever reaches *reset*, because no process is ever in the critical region. Then repeated use of fairness for *reset* implies that eventually there is no process at *reset*, a contradiction.

2. *stabilize*:

   Suppose that the liveness condition for *stabilize* fails. Then *stabilize* must be enabled from some point on in α', but no *stabilize* action ever occurs after that point. Since *stabilize* is enabled from the given point on in α', it follows that $status = seized$ from that point on. By the execution correspondence, from some corresponding point on in α, $x \neq 0$, no process is at *crit* or *reset*, and some process is at *set*. Note that in this fragment, no new process ever reaches *set*, because $x \neq 0$. Then repeated use of fairness for *set* implies that eventually there is no process at *set*. This is a contradiction.

3. *crit*

   Suppose that the liveness condition for *crit* fails. Then *crit* must be enabled from some point on in α', but no *crit* action ever occurs after that point. Since *crit* is enabled from the given point on in α', it follows that $status = stab$ from that point on. By the execution correspondence, from some corresponding point on in α, $x \neq 0$, no process is at *crit* or *reset*, and no process is at

set. Moreover, no *crit* action occurs. This implies that x remains equal to i for some fixed i. By Lemma 22, it must be that from this point onward, this process i must either be at *check* or *leave-try*. If it is ever at *leave-try*, then fairness to *crit* in α easily implies that a *crit* action eventually occurs, a contradiction. So it must be that process i is at *check* throughout the fragment. Then fairness implies that $check_i$ eventually occurs, and succeeds since $x = i$. But this leads to $pc_i = leave\text{-}try$, which is again a contradiction.

4. rem_i
   Left to the reader.

Such a proof can be written more formally using a temporal language that allows mention of both states and actions, such as the ones used in [36, 40].

7 Discussion

In this paper, I have given a comprehensive survey of simulation methods and other related methods for reasoning about timing-based systems. The main concepts and techniques that I have presented are the following:

1. The general timed automaton model.
2. Refinements, forward and backward simulations, and their weak versions.
3. The special-case MMT model.
4. Building time, in particular, the current time and timing predictions, into the state.
5. Invariants, especially those involving time predictions.
6. Milestones.
7. Progress functions.
8. Execution correspondence.
9. Weak fairness for automaton classes.

I have illustrated these methods with a substantial number of examples.

I do not mean to claim that simulations provide an all-purpose proof method for timing-based systems. Even though there are completeness results for the combination of forward and backward simulations, there are important examples for which simulations do not provide the most *natural* proof of correctness. For example, some algorithms are best understood as the result of transformations of automata that *reorder* the steps of executions, shifting the part of the execution that occurs at one node relative to the part that occurs at another. For examples of such algorithms in the timed setting, consider the transformations described by Neiger and Toueg [30] and by Chaudhuri et al. [5]. In the untimed setting, this style of reasoning occurs in arguments about database concurrency control [21], and about synchronizers [2, 7]. It also occurs in arguments about algorithms that have the structure of *communication-closed layers* [9].

Several other examples have been verified, or partially verified, using the methods of this paper. These include a timed protocol for *at-most-once* message delivery due to Liskov, Shrira and Wroclawski [17]; the proof appears in [36, 38].

They also include a *bounded message queue* example designed to show that, with certain limitations on the rate of message arrival and message processing, the length of a message queue stays bounded. The proof appears in [18]. A third example is a timing-based link-state packet distribution protocol designed by Perlman [33]; Although efficient in practice, this protocol has bad worst-case behavior, which was identified in the course of sketching an analysis using simulation methods.

Future work includes continuing to apply these methods to additional problems. Telecommunications and real-time process control are general areas that should serve as rich sources for appropriate timing-based algorithms.

It also remains to systematize and formalize liveness proofs of the sort outlined here. This will probably involve fixing a suitable temporal language and logic.

Finally, proofs of the sort given in this paper appear to be excellent candidates for mechanical verification using automatic theorem-provers. Several researchers have already done work using automatic theorem-provers to assist in carrying out simulation proofs [32,37]. Work in progress [39] involves using the Larch Prover [11] to carry out some simple simulation proofs for timing properties.

8 Acknowledgements

I would like to thank my co-workers, Hagit Attiya, Victor Luchangco, Jorgen Søgaard-Andersen and Frits Vaandrager, for their great contributions to the ideas in this paper. Victor deserves special thanks for helping with the many details of the examples in this paper. Roberto Segala and Rainer Gawlick also made many useful suggestions on the presentation.

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27, Mook, The Netherlands, June 1991. Springer-Verlag.
2. Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.
3. K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, 1988.
4. E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, May 1979.
5. Soma Chaudhuri, Rainer Gawlick, and Nancy Lynch. Designing algorithms for distributed systems with partially synchronized clocks. In *Proceedings of the Twelve Annual ACM Symposium on the Principles of Distributed Computing*, August 1993.
6. J. Davies and S. Schneider. An introduction to CSP, August 1989. Technical Monograph PRG-75.

7. Harish Devarajan. Correctness proof for a network synchronizer. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, May 1993.
8. E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
9. T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Progamming*, 2:155–173, 1982.
10. Michael Fischer. Re: Where are you? E-mail message to Leslie Lamport. Arpanet message number 8506252257.AA07636@YALE-BULLDOG.YALE.ARPA (47 lines), June 25, 1985 18:56:29EDT.
11. Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical report, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, December 1991. Research Report 82.
12. R. Gawlick, N. Lynch, R. Segala, and J. Søgaard-Andersen. Liveness in timed and untimed systems. In preparation, 1993.
13. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
14. Butler Lampson. Principles for computer system design, 1993. Turing Award Talk.
15. Butler Lampson, William Weihl, and Eric Brewer. 6.826 Principles of computer systems, Fall 1991. MIT/LCS/RSS 19, Massachusetts Institute of Technology, 1992. Lecture notes and Handouts.
16. G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, Toronto, 1977.
17. B. Liskov, Luiba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. Technical Report MIT/LCS/TR-476, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1990.
18. Victor Luchangco. Using simulation techiniques to prove timing properties. Master's thesis, MIT Electrical Engineering and Computer Science, 1993. In progress.
19. N. Lynch. Multivalued possibilities mappings. In *Rex Workshop*, volume 430 of *Lecture Notes in Computer Science*, Mook, The Netherlands, May 1989. Springer-Verlag. Also, MIT/LCS/TM-422.
20. N. Lynch and H. Attiya. Using mappings to prove timing properties. *Distrib. Comput.*, 6(2):121–139, 1992.
21. N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.
22. N. Lynch and B. Patt-Shamir. Distributed algorithms. MIT/LCS/RSS 20, Massachusetts Institute of Technology, 1992. Lecture notes for 6.852.
23. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, April, 1987. Also, MIT/LCS/TR-387.
24. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Also, MIT/LCS/TM-373.
25. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – Part I: Untimed systems. Submitted for publication. Also, MIT/LCS/TM-486.
26. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. Submitted for publication. Also, MIT/LCS/TM-487.
27. Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446, Mook, The Netherlands, June 1991. Springer-Verlag. Also, MIT/LCS/TM-458.
28. O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes*

in Computer Science, pages 447–484, Mook, The Netherlands, June 1991. Springer-Verlag.

29. M. Merritt, F. Modugno and M. Tuttle. Time constrained automata. In *CONCUR'91 Proceedings Workshop on Theories of Concurrency: Unification and Extension*, Amsterdam, August 1991.
30. Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *J. ACM*, 40(2):334–367, April 1993.
31. X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application, November 1991. Technical Report RT-C26, LGI-IMAG (revised version).
32. Tobias Nipkow. Formal verification of data type refinement. In *Proceedings of REX Workshop "Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness"*, volume 430 of *Lecture Notes in Computer Science*, pages 561–591, Mook, The Netherlands, June 1989. Springer-Verlag.
33. Radia Perlman. *Interconnections: bridge and routers.* Addison Wesley, 1992.
34. A. Pnueli, 1988. Personal Communication.
35. G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theor. Comp. Sci.*, pages 249–261, 58 1988.
36. Jørgen Søgaard-Andersen. Reliable at-most-once message delivery protocols, a protocol verification example. PhD thesis in progress.
37. Jørgen Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In *Proceedings of the Conference on Computer-Aided Verification*, Heraklion, Crete, Greece, June 1993.
38. Jørgen Søgaard-Andersen, Nancy A. Lynch, and Butler Lampson. Correctness of at-most-once message delivery protocols. In *FORTE '93 - Sixth International Conference on Formal Description Techniques*, pages 387–402, Boston, MA, October 1993.
39. Ekrem Soylemez. Automatic verification of the timing properties of MMT automata. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, 1993. In progress.
40. E.W. Stark. *Foundations of a Theory of Specification for Distributed Systems.* PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA 02139, August 1984. Also, MIT/LCS/TM-342.
41. F.W. Vaandrager and N.A. Lynch. Action transducers and timed automata. In *Proceedings of CONCUR '92, 3rd International Conference on Concurrency Theory*, Lecture Notes in Computer Science, Stony Brook, NY, August 1992. Springer Verlag.
42. J.L. Welch, L. Lamport, and N. Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proceedings of the 7^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 28–43, Toronto, Canada, August 1988. Also, MIT/LCS/TM-361.