Proving Performance Properties (even Probabilistic Ones)

Nancy Lynch
MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
`lynch@theory.lcs.mit.edu`

September 2, 1994

Abstract: This paper surveys some new tools and methods for formally verifying time performance properties of systems that satisfy timing assumptions. The techniques are potentially of practical benefit in the validation of real-time process control and communication systems. The tools and methods include nondeterministic timed automaton models, invariant assertion and simulation techniques for proving worst-case time bounds, probabilistic timed automaton models, and Markov-style techniques for proving probabilistic time bounds. All of these techniques are well suited for (partial) mechanization.

1 Introduction

A rich collection of formal methods have become well established for proving correctness properties – usually, safety and liveness properties – for asynchronous concurrent systems. The most important of these techniques are invariant assertion methods and simulation (refinement) methods for proving safety properties, and temporal logic methods for proving liveness properties. Other important techniques include algebraic and partial ordering methods.

But basic correctness properties are not the only ones that must be proved. Performance properties – especially time performance properties – are often nearly as important as correctness. In fact, for some systems, timing properties of system components are required to ensure the correct operation of the system as a whole. Yet formal methods have not been used very much to prove performance properties; nearly all of the proofs that have been carried out for such properties are ad hoc.

The goal of this paper is to show some ways in which proofs of timing properties can be formalized and systematized. We discuss the methods in three groups: those that are used for proving worst-case time bounds for asynchronous and timing-based algorithms, those for proving timing properties for hybrid systems, and those for proving probabilistic time bounds for probabilistic systems. Another paper in this proceedings, by Luchangco, Soylemez, Garland and Lynch [12], shows how proofs using the methods in the first group can be verified mechanically.

D. Hogrefe et al. (eds.), *Formal Description Techniques VII*
© IFIP International Federation for Information Processing 1995

2 Time Bounds for Asynchronous and Timing-Based Algorithms

2.1 Theory

The basic model that we use for modelling asynchronous and timing-based systems is the nondeterministic *timed automaton* model of Lynch and Vaandrager [18, 17, 14]. We base our work on an automaton model rather than any particular specification language, programming language or proof system in order to obtain the greatest flexibility in choice of specification and proof methods.

Timed automata. A *timed automaton* A consists of a set $states(A)$ of states, a nonempty set $start(A) \subseteq states(A)$ of start states, a set $acts(A)$ of actions, including a special *time-passage* action ν, a set $steps(A)$ of steps (transitions), and a mapping $now_A : states(A) \rightarrow R^{\geq 0}$. ($R^{\geq 0}$ denotes the nonnegative reals.) 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. The set $steps(A)$ is a subset of $states(A) \times acts(A) \times states(A)$. We write $s \xrightarrow{a}_A s'$ as shorthand for $(s, \pi, s') \in steps(A)$, and usually write $s.now_A$ in place of $now_A(s)$. We sometimes suppress the subscript or argument A.

A timed automaton must satisfy five axioms: [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 $s \xrightarrow{\nu} s'$. The statement of [A5] requires the auxiliary definition of a *trajectory*, which describes restrictions on the state changes that can occur during time-passage. Namely, if I is any interval of $R^{\geq 0}$, then an *I-trajectory* is a function $w : I \rightarrow states$, such that $w(t).now = t$ for all $t \in I$, and $w(t_1) \xrightarrow{\nu} w(t_2)$ for all $t_1, t_2 \in I$ with $t_1 < t_2$. That is, w assigns a state to each time t in interval I, in such a way that time-passage steps can connect any pair of states in the range of w. If w is an I-trajectory and I is left-closed, then let $w.fstate$ to be the first state of w, while if I is right-closed, then let $w.lstate$ denote the last state of w. 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 that spans from s to s'.

Timed Executions and Timed Traces. A *timed execution fragment* is a finite or infinite alternating sequence $\alpha = w_0 \pi_1 w_1 \pi_2 w_2 \cdots$ 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 $w_j.lstate \xrightarrow{\pi_{j+1}} w_{j+1}.fstate$.

A *timed execution* is a timed execution fragment for which $w_0.fstate$ is a start state. We focus mainly on the *admissible* timed executions, i.e., those in which the *now* values occurring in the states approach ∞. 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.

In order to describe the problems to be solved by timed automata, we require a definition for their visible behavior. We use *timed traces*, where 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.

Composition. Let A and B be timed automata satisfying the following *compatibility* conditions: A and B have no output actions in common, and 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;
- $(s_A, s_B) \xrightarrow{\pi}_{A\times B} (s'_A, s'_B)$ 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 $s_B = s'_B$;
- $(s_A, s_B).now_{A\times B} = s_A.now_A$.

Then $A \times B$ is a timed automaton. If α is a timed execution of $A \times B$, we write $\alpha|A$ and $\alpha|B$ for the projections of α on A and B, respectively. For instance, $\alpha|A$ is defined by projecting all states in α on the state of A, removing actions that do not belong to A, and collapsing consecutive trajectories.

MMT automata. An important special case of the timed automaton model, describable in a particularly simple way, is the MMT timed automaton model [20], developed by Merritt, Modugno and Tuttle. We use a special case of their definition from [19, 14].

An MMT automaton is an I/O automaton [13] together with upper and lower bounds on time. An I/O automaton A consists of a set $states(A)$ of states, a nonempty set $start(A) \subseteq states(A)$ 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, and a partition $part(A)$ of the locally controlled (i.e., output and internal) actions into at most countably many "tasks". The set $steps(A)$ is a subset of $states(A) \times acts(A) \times states(A)$; 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 $s \xrightarrow{\pi}_A s'$. 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. The final component, *part*, is sometimes called the *task partition*. Each class in this partition groups together actions that are supposed to be part of the same "task".

An MMT automaton is obtained by augmenting an I/O automaton with certain upper and lower time bound information. Let A be an I/O automaton with only finitely many tasks. For each task C, define lower and upper time bounds, $lower(C)$ and $upper(C)$, where $0 \leq lower(C) < \infty$ 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 the π's are input, output or internal actions (but not time-passage 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. Finally, *admissibility* is required: if the sequence is infinite, then the times of actions approach ∞. 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.

It is not hard to transform any MMT automaton A into a naturally-corresponding timed automaton A'. First, the state of the MMT automaton A is augmented with a *now* component, plus *first*(C) and *last*(C) components for each task. The *first*(C) and *last*(C) components represent, respectively, the earliest and latest time in the future that an action of task C is allowed to occur. 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. Restrictions are also added on actions of any task C, saying that the current time *now* must be at least equal to *first*(C). The resulting timed automaton A' has exactly the same admissible timed traces as the MMT automaton A.

Invariants and simulations. We define an *invariant* of a timed automaton to be any property that is true of all reachable states.

The definition of a simulation is paraphrased from [18, 17, 14]. We use the notation $f[s]$, where f is a binary relation, to denote $\{u : (s, u) \in f\}$. Suppose A and B are timed automata and I_A and I_B are invariants of A and B, respectively. Then a *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 $s \xrightarrow{\pi}_A s'$, $s, s' \in I_A$, 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.

Note that π is allowed to be the time-passage action in the third item. The most important fact about simulations is that they imply admissible timed trace inclusion:

Theorem 2.1 *If there is a simulation from timed automaton A to timed automaton B, with respect to any invariants, then every admissible timed trace of A is an admissible timed trace of B.*

In practice, the simulations often have an interesting form: a system of inequalities [19].

2.2 Applications

We sketch several examples of time bounds for asynchronous and timing-based concurrent algorithms, proved using the tools and methods described above. The MMT model is sufficient to describe all of these. The examples are summarized from [14, 15, 11].

Counting Process. The first and simplest example involves an automaton that counts down from some fixed positive integer k and then reports its completion. The time between successive steps is assumed to be in the range $[c_1, c_2]$.

Count:

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

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

Transitions:

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

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

Tasks and bounds:
$\{report\}$, bounds $[c_1, c_2]$
$\{decrement\}$, bounds $[c_1, c_2]$

Informally, it is easy to see that the time until a *report* occurs is in the range $[(k+1)c_1, (k+1)c_2]$. In order to prove this formally, we express these time bound assumptions by a trivial high-level reporting automaton called *Report*.

Report:

Actions:
Output: *report*

State components:
reported, Boolean, initially *false*

Transitions:

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

Tasks and bounds:
$\{report\}$, bounds $[(k+1)c_1, (k+1)c_2]$.

We can show that *Count* implements *Report*, that is, that every admissible timed trace of *Count* is an admissible timed trace of *Report*. Specifically, we 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 *now* and *reported* conditions are straightforward. The *last*(*report*) component is constrained to be at least as large as a calculated upper bound on the latest time until a *report* action is performed by *Count*. If $count > 0$, then this calculated bound is 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 calculated bound is the last time at which the *report* can occur. The interpretation of the condition for *first*(*report*) is symmetric – the lower bound to be proved should be no larger than a calculated lower bound on the earliest time until a *report* action is performed by *Count*.

Lemma 2.2 *The relation f is a forward simulation from Count to Report.*

Proof: The proof involves verifying the three properties in the definition of a forward simulation (after proving some trivial invariants). The correspondence between *now* values is immediate, and it is easy to check that the unique start states of the two automata are related by f. The interesting part of the proof is the third condition – the step correspondence. For this, we consider cases based on types of transitions.

For example, consider a transition $s \xrightarrow{decrement}_{Count} s'$, where $u \in f[s]$. Then $s.count > 0$. Thus, $u.now = s.now$, $u.reported = s.reported$, $u.last(report) \geq s.last(decrement) + s.count \cdot c_2$, 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, consider first the case where $s'.count > 0$. Then the left side of the inequality, *last*(*report*), does not change, while on the right side, *last*(*decrement*) cannot increase by more than c_2, and the second term decreases by exactly c_2. (The reason why *last*(*decrement*) cannot increase by more than c_2 is as follows: The construction of the timed automaton from the MMT automaton for *Count* — captured by invariants — implies that $s.now \leq s.last(decrement)$. On the other hand, $s'.last(decrement) = s'.now + c_2$ and $s'.now = s.now$.) So the inequality still holds after the step.

Similar arguments can be made for the case where $s'.count = 0$, and for the lower bound. ■

This lemma implies the following theorem, which says that *Count* satisfies the timing requirements.

Theorem 2.3 *Every admissible timed trace of Count is an admissible timed traces of Report.*

It is possible to use an equational theorem prover to verify proofs of simulations such as this one, as well as proofs of the needed invariants. In fact, such a theorem prover can fill in some of the proof steps. Work along these lines, using the Larch theorem prover [6], is described in [12].

The *Count* algorithm is trivial; in the rest of this section, we sketch more interesting time bound results that can be proved using the same strategy. Without such a stylized method, proving such time bounds would be a difficult task.

Fischer mutual exclusion algorithm. The Fischer mutual exclusion algorithm [5] is a popular test case for formal methods for verifying real-time algorithms. In this algorithm, a collection of user processes compete for control of a resource, using a single shared variable x that they can only access using *read* and *write* operations. We model the entire assembly of processes and x as an MMT automaton, where the tasks correspond to the different types of steps performed by the processes (several tasks per process). A high-level description of the algorithm is:

```
Fischer, process $i$:
L: wait for $x = 0$
    $x := i$
    if $x \neq i$ then go to $L$
    (Critical Region)
    $x := 0$
```

As described so far, this algorithm can violate mutual exclusion. For example, 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 see its own index, and then j might do the same. In order to avoid this bad interleaving, we introduce time restrictions: for each process, the time for setting x, once x has been seen equal to 0, is in the range $[0, a]$, while the time to check x after it has been set is in the range $[b, \infty]$, for some constants a, b, where $a < b$. This rules out the bad interleaving above, since 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 $x := j$) has already set $x := j$. That is, there should be no processes left at the point of setting x, when i finally checks.

It is easy to translate this code into an MMT automaton *Fischer* (say, in precondition-effect style). The input actions are of the form try_i, by which a process i tries to get access to the resource, and $exit_i$, by which a process returns the resource, and the outputs are $crit_i$, by which the algorithm grants access to the resource (critical region) and rem_i, by which the algorithm gives permission to the user program to return to the remainder of its processing. There are also internal actions $test_i$, set_i, $check_i$ and $reset_i$. The tasks correspond to the individual actions. Then mutual exclusion can be expressed as an invariant on the algorithm's state (more precisely, on the state of a composed *Fischer* system automaton consisting of the *Fischer* automaton and and a collection of nondeterministic *user* automata). It can be proved as a consequence of the following invariant, which is in turn proved easily by induction:

Lemma 2.4 *If $pc_i = check$ and $x = i$ and $pc_j = set$ then $first(check_i) > last(set_j)$.*

That is, if i is about to check x successfully, then any other process that is about to set x is scheduled to do so before i can check x.

Mutual exclusion is the property that is usually proved for this algorithm. But since this paper is about proving time bounds, we take the example further. We assume an upper bound of c, $c \geq b$, for each *check* action, and an upper bound of a for all the other actions, and we show that the time from when *any* process is trying to obtain the resource until *some* process has it is at most $2c + 5a$.

Following the same strategy as for the *Count* example, we formulate the specification of the time bound as an MMT automaton *Mutex* with the same external actions as the *Fischer* algorithm. The *Mutex* automaton is very nondeterministic; it expresses only the mutual exclusion property plus the given time bound. Then we use the simulation method to show that the *Fischer* system implements the *Mutex* system (i.e., the composition of the *Mutex* automaton and the users).

In this example, the time bound is most easily understood in terms of reaching certain *milestones*. In particular, once the critical region is empty, the first important event is for the shared variable x to be "seized" by some contending process. The next important event is for the value of x to "stabilize" so that it can no longer be modified by any process until someone reaches the critical region. It turns out that the first milestone is reached within at most time $c + 3a$, then the second milestone is reached within additional time a, and finally some process reaches the critical region within additional time $c + a$; the total is $2c + 5a$.

Formally, we can express these milestones as the events of another intermediate MMT automaton I. A trivial simulation can be used to show that the I system implements the *Mutex* system, and then a more complicated simulation can be used to show that the *Fischer* system implements the I system. For example, a key piece of the simulation from *Fischer* to I is the set of inequalities that involve calculated bounds on time for some process to seize x:

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

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

Thus, if some process i has just exited, the calculated bound is the maximum time until process i resets $x := 0$, plus $c + 2a$. If $x = 0$, the calculated bound is the minimum of a collection of individual process bounds, where the bound for process i measures the time until i sets x (if no other process does so in the meantime).

Theorem 2.5 *Every admissible timed trace of the Fischer system is an admissible timed trace of the Mutex system.*

Again, the proof can be checked mechanically [12].

Dijkstra mutual exclusion algorithm. Dijkstra's asynchronous algorithm [4] for mutual exclusion using read/write shared memory is one of the earliest published distributed algorithms.

```
Dijkstra, process $i$:
L: $flag(i) := 1$
    while $x \neq i$ do
        if $flag(x) = 0$ then $x := i$
    $flag(i) := 2$
    for $j \neq i$ do:
        if $flag(j) = 2$ then goto L
    (Critical Region)
    $flag(i) := 0$
```

Unlike the *Fischer* algorithm, the *Dijkstra* algorithm guarantees mutual exclusion regardless of the relative speeds of processes. For a time bound, we can assume an upper bound of a for all process steps and show that the time from when any process is trying until some process is critical is at most $(3n + 11)a$. (Here and elsewhere, n is the number of processes in the system.)

The proof strategy is the same as for the Fischer algorithm, using an intermediate automaton with *seize* and *stabilize* milestones, plus an additional *dropback* milestone. The *dropback* event indicates a point by which all but the process whose index is in x have dropped back to the first stage of the algorithm, where $flag = 1$ (or by which someone goes critical, it this happens first). I uses upper bounds $(n+5)a$, $2a$, $(n+1)a$ and $(n+3)a$ for the milestones *seize*, *stabilize*, *dropback* and *critical*, respectively. Using I, we can provide two stylized simulations to show the needed time bound.

LeLann-Chang-Roberts leader election algorithm. The LeLann-Chang-Roberts leader election algorithm for ring networks [10, 3] works as follows:

> **LCR:**
> Every process sends its process identifier clockwise. Smaller identifiers that encounter larger identifiers are discarded. If a node receives its own identifier in a message, it elects itself as leader.

If we assume an upper bound of ℓ on the step time for each process, and an upper bound of d on the time to deliver the oldest message in each channel, then we can show that the time until a leader is elected is at most $(n+1)\ell+nd$. The difficulty of the proof involves the possible pile-up of messages in channels if some processes and channels operate faster than others. The proof again uses an intermediate automaton I, this time with n milestones. Milestone i, $1 \leq i \leq n$, is reached when the *slowest* token has progressed distance i around the ring. The bounds for the successive milestones are all $\ell + d$, and the bound for the final leader report is ℓ. The simulation from the *LCR* system to I determines how many milestones have been reached based on the least progress made by any identifier.

Discussion. Timed automata, MMT automata, simulations and invariants have been successful for verifying time bounds for a substantial collection of asynchronous and timing-based concurrent algorithms. These have not all been toy examples, but include algorithms for which time bounds are otherwise difficult to obtain. The method provides information (invariants, simulations) that help in documenting the operation of the algorithms. The method is systematic, and lends itself to mechanical assistance.

The proofs have more common structure than just the use of simulations and invariants. In several of the cases, an intermediate milestone automaton is used. In every case, an important part of the simulation is a set of inequalities involving calculated upper and/or lower bounds. These calculated bounds can usually be expressed as "progress functions" that measure the time until the specification-level goals are reached. If progress functions satisfy certain properties (as detailed in [19]), then a relation derived from them in a systematic way is guaranteed to be a simulation. Identifying such common structure should help in further systematizing the proofs.

3 Timing Properties for Hybrid Systems

The results of Section 2.2 show how simulation and invariant methods can be used to prove time performance properties for asynchronous and timing-based algorithms. Now we show how the same methods can be used to prove properties of "hybrid" systems containing both real world and computer components, for example, real-time process control systems. The real world components typically include physical quantities that change continuously as time passes, as well as quantities that are changed by discrete events.

3.1 Theory

It turns out that the timed automaton and MMT automaton models introduced in Section 2.1 for representing algorithms are also adequate for representing most real-time systems with real world components. The main difference is that the trajectory mechanism provided by the general timed automaton model now becomes important for modelling continuous changes in the real-world components. The simulation and invariant methods are still used in much the same way as before.

3.2 Applications

We discuss one example: the Generalized Railroad Crossing (GRC) problem [7].

Generalized railroad crossing. The problem involves real world trains and gates, interacting with a computer system via sensors and actuators. There are several parallel tracks on which trains travel through a road intersection I. Before arriving in I, each train arrives in the general region R, and triggers a sensor that notifies the computer system of its arrival. When a train leaves I, it also triggers a sensor. The computer system is capable of lowering and raising a crossing gate. Parameters ϵ_1 and ϵ_2 describe lower and upper bounds on the time from when a train triggers the arrival sensor until it reaches I, and parameters γ_{down} and γ_{up} describe upper bounds on the time to lower and raise the gate completely. There are two requirements:

1. A *safety property*, which says that if a train is in I then the gate is down.
2. A *utility property*, which says that the gate must be up unless there is a reason it should not be: either within following time ξ_1 or preceding time ξ_2 (two additional parameters), some train will be in I.

The problem is to formalize these requirements and to design and verify a computer system that satisfies them. This problem was originally proposed as a challenge problem for comparing the effectiveness of various formal methods for handling such problems.

We present the approach from [8, 9]. We begin by defining separate timed automata for the trains and gate components, plus a placeholder automaton to represent the interface between the real world components and the computer system. Initially, we give discrete automata *Trains* and *Gate*, and later infer results about the corresponding continuous components using general properties of timed automaton composition. For example, *Trains* is the following MMT automaton.

Trains:

Actions:
Output: *enterR*(r), r a train
enterI(r), r a train
exit(r), r a train

State components:
for each train r:
$r.status \in \{not\text{-}here, P, I\}$, initially *not-here*

Transitions:

enterR(r)
Precondition:
$r.status = not\text{-}here$
Effect:
$r.status := P$

enterI(r)
Precondition:
$r.status = P$
Effect:
$r.status := I$

exit(r)
Precondition:
$r.status = I$
Effect:
$r.status := not\text{-}here$

Tasks and bounds:
Each action is a separate task; all bounds are trivial except each *enterI*(r) has $[\epsilon_1, \epsilon_2]$.

Similarly, *Gate* is modelled as a discrete MMT automaton, with four states, *up*, *going-down*, *down* and *going-up*. It has inputs *lower* and *raise*, which place the gate in state *going-down* or *going-up*, respectively, and outputs *down* and *up*, which signal the arrival of the gate at the extreme positions.

For the specification, there is also a trivial *CompSpec* component describing the computer system's interface. The complete specification *AxSpec* consists of the composition of these three automata, plus two axioms corresponding to the safety and utility properties, which constrain the set of executions.

To describe an implementation, we replace *CompSpec* by *CompImpl* – a timed automaton that decides when to raise and lower the gate. It lowers the gate when there is a train that might reach I within time γ_{down}, and raises it when there is no train that could reach I within time $\gamma_{up} + \gamma_{down}$. The complete system *SysImpl* is just the composition of the three components, with no extra constraints. We must show that *SysImpl* satisfies the requirements, specifically, that each admissible timed execution of *SysImpl* projects on the real world components to look like an admissible timed execution of *AxSpec*. This is the same as saying that *SysImpl* satisfies the two axioms.

The safety property is proved as an invariant of *SysImpl*. We are more interested here in the utility property, however, since it can be classified as a performance property. As for the previous performance properties, we would like to prove this using a simulation. But to do this, we need a specification automaton, which is not the same as an axiomatic specification. Thus, we give a second specification, *OpSpec*, in the form of a timed automaton. Instead of having an axiom about time intervals, we put in explicit *last* components as before, as deadlines for certain specification-level goals. For example, *OpSpec* includes the provision that whenever the gate is lowered, some train will be in the crossing within time ξ_1. This is expressed by means of a *last* component that gets set when *CompImpl* lowers the gate to time ξ_1 in the future, and gets disabled when some train enters I.

An ad hoc argument is used to prove that *OpSpec* implements *AxSpec* (in the sense that each admissible timed execution of *OpSpec* projects on the real world to look like an admissible timed execution of *AxSpec*). (The implementation relationship does not need to go the other way.) Then the more systematic simulation methods are used as before to show that *SysImpl* implements *OpSpec*. Technically, the simulation proof only shows inclusion of sets of admissible timed traces, and not the preservation of the complete view of the real world. But general properties of timed automaton composition yield the stronger correspondence as a corollary. We obtain:

Theorem 3.1 *For every admissible timed execution α of SysImpl, there is an admissible timed execution α' of AxSpec such that $\alpha' | Trains \times Gate = \alpha | Trains \times Gate$.*

Also, as noted earlier, we can define more realistic models of the real world components, such as a *Gate'* automaton with a specific function describing the position of the gate while it is being lowered or raised. We can then infer results about the continuous models from those for the discrete models, using general properties of timed automaton composition.

Discussion. Timed automata, simulations and invariants have proved able to cope successfully with the Generalized Railroad Crossing problem, in particular, with its performance properties. The results obtained are very general – e.g., they apply to arbitrary values of the various parameters. This is in contrast, say, to model-checking methods which (usually) only work for specific constants. As before, the method provides useful information in the form of invariants and simulations, is systematic, and lends itself to mechanical assistance.

It appears that applications experts for process control systems prefer axiomatic to operational specifications. Furthermore, the axioms should be statements about the real world, and should (as far as possible) be independent of each other. On the other hand, an operational, automaton-style specification is needed for the application of invariant and simulation proof methods. We conclude that both types of specifications should be provided, and a proof that the operational specification implements the axiomatic specification should be given. In the GRC example, this proof of correspondence between the two specifications is ad hoc. It remains to develop a user-friendly language for axiomatic specification of real-time systems, and a systematic way of relating (and perhaps translating) such specifications to operational specifications.

It is useful to work initially with simplified, discrete models of the real world components, and then to use general properties of timed automaton composition to extend the results to more realistic, continuous models.

The GRC example is small. It remains to see how well these methods scale up to realistic size process control systems. We are currently looking at some problems arising in transportation systems such as the Personal Rapid Transit system currently being designed by Raytheon.

4 Probabilistic Time Bounds for Probabilistic Algorithms

We finish with a collection of new formal tools and methods for proving time bound properties for probabilistic systems. The motivating examples are the many randomized distributed algorithms that have been developed in the theory of distributed computing. The appropriate time bounds for such algorithms are generally *probabilistic time bounds*, i.e., assertions that some event occurs within a certain amount of time t, with at least a certain probability p.

It is notoriously difficult to obtain correct proofs for randomized distributed algorithms, and so it is worthwhile to develop systematic proof methods. The methods we use to prove probabilistic time bounds are somewhat different from the simulation methods discussed in the previous sections. However, the development of these methods represents work in progress; eventually, the methods used for probabilistic algorithms ought to be better unified with those used for non-probabilistic algorithms.

4.1 Theory

The theory we used is taken from [22, 16, 21].

Probabilistic timed automata. We modify the timed automaton model to incorporate probabilistic state transitions; specifically, we allow non-time-passage steps to be of the form (s, π, Π), where Π is a probability distribution whose domain is a subset of the states of the algorithm. The meaning is that when the step occurs, a probabilistic choice according to distribution Π is used to determine the new state. A probabilistic timed automaton (pta) has a combination of nondeterministic and probabilistic branching in its timed executions, where nondeterminism is involved in the choice of the next step.

Meaningful probabilistic statements about the behavior of a pta must be based on a probability distribution on its timed executions. In order to obtain such a distribution, we must resolve all nondeterministic choices, making all the branching probabilistic. We do this by hypothesizing an entity called an *adversary* that determines the next step; formally, an adversary is a function that, given the finite execution performed so far, returns the following (possibly probabilistic) step. A fixed adversary $\mathcal{A}$ gives rise to an execution tree with only probabilistic branch points, and thereby to a probability distribution on timed executions.

An adversary is said to be *admissible* provided that all the timed executions that appear in its distribution are admissible, i.e., provided that it always allows time to pass to ∞. When we claim that an algorithm (pta) guarantees some property P with at least probability p, we mean that for every admissible adversary $\mathcal{A}$, the probability of P in the probability distribution of timed executions generated by $\mathcal{A}$ is at least p.

Markov rules. One practical method for proving probabalistic time bound properties is based on formulating Markov-style statements of the form $\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$, where $\mathcal{U}$ and $\mathcal{U}'$ are sets of states. This statement means that for every admissible adversary $\mathcal{A}$, if the algorithm is started in a state in $\mathcal{U}$, then in the probability distribution generated by $\mathcal{A}$, the probability that a state in $\mathcal{U}'$ is reached within time t is at least p. Such statements can be combined:

Lemma 4.1 *If* $\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$ *and* $\mathcal{U}' \xrightarrow[p']{t'} \mathcal{U}''$ *then* $\mathcal{U} \xrightarrow[pp']{t+t'} \mathcal{U}''$.

Coin lemmas. In most situations where a property of the form $\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$ is true, the probability p arises as the probability that certain designated random choices are made in a favorable way. In such a case, the statement can usually be proved by fixing the results of those choices in the favorable way and converting all the remaining random choices to nondeterministic choices. This gives rise to an ordinary (non-probabilistic) timed automaton A. Proving that the desired time bound holds *with certainty* for A implies that it holds with probability p in the original pta. The soundness of this strategy is justified by a series of *coin lemmas* in [21].

4.2 Applications

We have so far carried out two significant proofs of probabilistic time bound properties for randomized distributed algorithms.

Lehmann-Rabin Dining Philosophers algorithm. The randomized Dining Philosophers algorithm of Lehmann and Rabin works as follows.

LehmannRabin:

Each contending process executes a loop wherein it randomly chooses a direction, *left* or

> *right*, waits to obtain the fork in that direction, and then instantaneously examines the other fork. If the second fork is free, it takes it and proceeds to the critical region; otherwise, it drops its first fork and returns to the beginning of the loop.

The safety property, i.e., that no two processes with conflicting resource requirements ever reach their critical regions simultaneously, is easy to show. A more interesting property is the following probabilistic time bound claim: from any state where some process is trying to obtain its forks, within time 14ℓ, and with probability at least $\frac{1}{16}$, some process will reach the critical region.[1] This can be expressed in the form $\mathcal{T} \xrightarrow[\frac{1}{16}]{14\ell} \mathcal{C}$. where $\mathcal{T}$ is the set of states in which some process is trying to get its forks and $\mathcal{C}$ is the set of states in which some process is in the critical region.

This property is proved in [16] using Markov rules and coin lemmas. We use a chain of five Markov rules marking progress from $\mathcal{T}$ to $\mathcal{C}$, each with its own time and probability bounds, and combine them using Lemma 4.1. Some of these rules involve probability 1, and in fact describe results that hold with certainty; these can be proved by methods such as the simulation method already developed for proving time bounds for non-probabilistic timed automata. The others all involve the results of particular random coin tosses, generally the first coin tosses by a small number of neighboring processes. These can be proved by using the coin lemmas to reduce the problem to one involving non-probabilistic timed automata, then using methods for proving time bounds for non-probabilistic timed automata. In [16], operational arguments are used to prove the claims for non-probabilistic timed automata, but it is clear in principle that the arguments could be redone more systematically using simulations.

For an example of an argument that is done for this proof, consider a situation involving a process i and its right neighbor $i+1$. Suppose that i has already obtained its left fork and is about to test its right fork (the one shared with $i+1$), while process $i+1$ is about to toss its coin. This situation is formally described as a set $\mathcal{U}$ of states of the algorithm. We claim that $\mathcal{U} \xrightarrow[\frac{1}{2}]{\ell} \mathcal{U}'$, where $\mathcal{U}'$ is the set of states in which some process has both its forks. The probability of $\frac{1}{2}$ is just the probability that $i+1$ chooses *right*. In that case, we claim that the given situation leads to $\mathcal{U}'$ with absolute certainty, within time ℓ. To see this, note that i must test its right fork within time ℓ. If it succeeds in getting it, then we are done. If not, then it must be that in the meantime, $i+1$ has obtained it. But if $i+1$ has obtained it, then it must be $i+1$'s second fork, since $i+1$'s next coin toss is *right*. Thus in this case, $i+1$ must have both its forks and again we are done.

Aggarwal-Kutten spanning tree algorithm. A larger and more complex example is the Aggarwal-Kutten randomized algorithm for finding a spanning tree in a network based on an unknown undirected graph G [2]. Processes are assumed to be identical (e.g., they do not have unique identifiers), and so the only way to break symmetry in the construction of the tree is using randomness. The algorithm is too complicated to explain quickly, but the basic idea is that each process tries to form a spanning tree with itself as the root, using a broadcast-convergecast strategy. In doing this, each process uses a randomly-chosen identifier, and if it discovers the existence of another root (by discovering a different identifier) it takes steps to combine with the other root.

---

[1] Again, ℓ is an upper bound on process step time.

The problem is that there might be more than one root with the same identifier, preventing them from discovering each other's existence. To address this problem, the roots make repeated random choices of new identifiers, according to a careful discipline, and propagate these throughout their trees. Then it turns out that within time proportional to the diameter of the network, and with at least a nonzero constant probability, a unique spanning tree will result.

This algorithm is a randomized distributed algorithm of typical difficulty; the state of the art in the distributed algorithms area has till now not permitted careful proofs of such algorithms. Nevertheless, in [1] this bound is proved, using Markov rule and coin lemma methods. There are two interesting points to be noted about the proof. First, as is common when careful proofs are carried out for complicated algorithms, a mistake was found in the original code. (It was easy to fix.) Second, although the final proof is fairly lengthy, the parts of it that deal with probabilities constitute only a few pages. One effect of our work on systematizing such proofs has been the reduction of reasoning about probabilistic systems to reasoning about non-probabilistic systems.

Discussion. Systematic proofs of probabilistic time bounds for probabilistic algorithms are possible. The methods we have used so far – Markov rules and coin lemmas – are somewhat different from those used for non-probabilistic algorithms. Mechanical assistance is clearly possible for the non-probabilistic parts of the reasoning; we do not know if additional assistance is possible for the probabilistic parts.

Intuitively speaking, there seems to be some redundancy in the methods we have applied so far to probabilistic systems. Both the strategy of combining Markov rules and the techniques for proving individual Markov rules involve measuring progress toward a goal. Perhaps there is a way of unifying these methods.

It should also be possible to extend simulation methods directly to probabilistic systems, without first removing the probabilistic choices. The simulation relations themselves might be probabilistic, corresponding states with probability distributions on states rather than with single states. It remains to define such correspondences, verify their soundness, and see how they can be used in verification.

5 Conclusions

There are now a wide range of practical, systematic methods for verifying time performance properties for concurrent systems. These include timed automata, simulations, invariants, probabilistic timed automata, Markov rules and coin lemmas. Computer assistance is possible for most of these, using equational theorem provers such as Larch.

Remaining work includes developing the formal methods for probabilistic systems further, and unifying them with the simulation and invariant methods. It also includes further attempts to automate these methods, and to apply them to many more examples.

References

[1] Sudhanshu Aggarwal. Time optimal self-stabilizing spanning tree algorithms. Master's thesis, MIT Electrical Engineering and Computer Science, May 1994.

[2] Sudhanshu Aggarwal and Shay Kutten. Time optimal self stabilizing spanning tree algorithms. In R.K. Shyamasundar, editor, *13th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 400–410, Bombay, India., December 1993. Springer-Verlag.

[3] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

[4] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[5] 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.

[6] 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.

[7] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc., Tenth Intern. Workshop on Real-Time Operating Systems and Software*, May 1993.

[8] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of the 15th IEEE Real-Time Systems Symposium.*, San Juan, Puerto Rico, December 1994. To appear.

[9] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. Technical Memo MIT/LCS/TM-511, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1994. To appear.

[10] G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, Toronto, 1977.

[11] Victor Luchangco. Using simulation techiniques to prove timing properties. Master's thesis, MIT Electrical Engineering and Computer Science, 1994. In progress.

[12] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In *Proceedings of the Seventh International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE'94)*, IFIP Transactions, Berne, Switzerland, October 1994. IFIP WG6.1, Elsevier Science Publishers B. V. (North Holland).

[13] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[14] Nancy Lynch. Simulation techniques for proving properties of real-time systems. In *A Decade of Concurrency: Reflections and Perspectives*, Lecture Notes in Computer Science, pages 375–424, REX School/Symposium, Noordwijkerhout, the Netherlands, June 1993. Springer-Verlag.

[15] Nancy Lynch. Simulation techniques for proving properties of real-time systems. In Sang H. Son, editor, *Principles of Real-Time Systems.* Prentice Hall, 1994. To appear.

[16] Nancy Lynch, Isaac Saias, and Roberto Segala. Proving time bounds for randomized distributed algorithms. In *Thirteenth Annual ACM Symposium on the Principles of Distributed Computing*, Los Angeles, CA, August 1994.

[17] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part II: Timing-based systems. Submitted for publication.

[18] 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.

[19] Nancy A. Lynch and Hagit Attiya. Using mappings to prove timing properties. *Distrib. Comput.*, 6:121–139, 1992.

[20] Michael Merritt, Francemary Modugno, and Mark R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, editors, *CONCUR'91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423, Amsterdam, The Netherlands, August 1991. Springer-Verlag.

[21] Roberto Segala. PhD thesis, MIT Dept. of Electrical Engineering and Computer Science, 1993. In progress.

[22] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. In *Proceedings of the 5th International Conference on Concurrency Theory - CONCUR'94*, Lecture Notes in Computer Science, Uppsala, Sweden, August 1994. Springer-Verlag.