This paper will appear in the FORTE'94 Proceedings.

Verifying Timing Properties of Concurrent Algorithms

Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch[a*]

[a]MIT Laboratory for Computer Science, Cambridge, MA 02139

This paper presents a method for computer-aided verification of timing properties of real-time systems. A timed automaton model, along with *invariant assertion* and *simulation* techniques for proving properties of real-time systems, is formalized within the Larch Shared Language. This framework is then used to prove time bounds for two sample algorithms—a simple counter and Fischer's mutual exclusion protocol. The proofs are checked using the Larch Prover. Keywords: I.3, I.8, I.6/II.12: Larch, III.1, IV.8

1. Introduction

Techniques based on *simulations* are widely accepted as useful for verifying the correctness of (untimed) concurrent systems. These methods involve describing both the problem specification and an implementation as state machines, establishing a correspondence known as a *simulation mapping* between their states, and proving that the mapping is preserved by all transitions of the implementation. Such methods are attractive because they provide insights into a system's behavior, appear to be scalable to systems of substantial size, and provide assistance in modifying system descriptions and proofs.

It is usually possible to describe the transitions of the specification, the transitions of the implementation, and the simulation relation, all as equations involving states. Then the proof that the simulation mapping is preserved is an exercise in equational deduction. Such deductions are natural candidates for partial automation. Proofs of this sort for untimed systems have already been automated, for example, using HOL [8], Isabelle [16], and the Larch Prover [21].

Recently, the simulation method has been extended to proofs of correctness and timing properties for timing-based systems [11, 13, 10]. The extended method is based on the *timed automaton* model of Merritt, Modugno and Tuttle [15]. Both the specification and implementation are described as timed automata, which include *timing conditions* in their states. The implementation's conditions represent timing assumptions, and the specification's conditions represent timing upper and lower bounds to be proved. As in the untimed case, a simulation mapping is defined between the states of the implementation and those of the specification; but now the mapping typically includes inequalities involving the timing conditions. The proof that the mapping is preserved by all transitions has a similar deductive flavor to the proofs in the untimed case, but now the deductions involve inequalities as well as equations.

*Research supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contracts N00014-92-J-1795 and N00014-92-J-4033, by the National Science Foundation under grants 9115797-CCR and 9225124-CCR, and by the Air Force Office of Scientific Research and the Office of Naval Research under contract F49620-94-1-0199.

The simulation method for timed systems has the same attractions as for untimed systems. Furthermore, it is capable of proving performance as well as correctness properties. Examples of proofs done by hand using this method appear in [11, 10, 20, 6, 9].

Just as in the untimed case, the timed proofs are amenable to automation. Specifically, the notions of timed automata, invariant assertions, and simulation mappings are formalized using the Larch Shared Language [5], and this formal infrastructure is used to specify, verify, and analyze two sample algorithms—a simple counter [11] and Fischer's mutual exclusion protocol. Fischer's algorithm has been verified many times by many people [1, 18, 19], including some with machine assistance [19]. But in addition to the usual correctness property of mutual exclusion, we prove a more difficult timing property—an upper bound on the time from when some process requires the resource until some process acquires it.

The rest of the paper proceeds as follows. We introduce our techniques by way of a simple example in Section 2. Then we use these techniques to verify Fischer's mutual exclusion protocol in Section 3.

2. A Simple Example

In this section, we verify the correctness and timing properties of a simple timed automaton. We present both manual and machine-checked proofs. Our model of timed automata is based on work by Merritt, Modugno, and Tuttle [15] and by Lynch and Attiya [11]. We describe this model by means of an example in this section and more formally in Appendix A.

Consider a counting automaton C_k, which decrements a counter with initial value k and issues a single report when the counter reaches 0. We will verify that C_k implements the specification given by another automaton R, which just issues a single report. We will also establish bounds a_1 and a_2 on how long it takes the specification automaton R to issue its report based on k and the time bounds c_1 and c_2 for the actions of the implementation automaton C_k. Figure 1 defines the two automata.

The untimed part of each automaton is a simple state-transition system. Actions are said to be *enabled* in the states satisfying their preconditions. Actions are classified as *external* or *internal* so that we may compare an implementation with its specification.

To describe timing properties, the actions are partitioned into *tasks*. A task is *enabled* when any of its actions are enabled. Lower and upper bounds, $lower(C)$ and $upper(C)$, on each task C specify how much time can pass after C becomes enabled before either one of its actions occurs or the task is disabled. The upper bound can be infinite.

The timed part of each automaton contains three additional state components: a real-valued variable *now* representing the current time, and two functions *first* and *last* representing the earliest and latest times that some action from each task can occur. All times are absolute, not incremental. All tasks that are not enabled have trivial *first* and *last* components (i.e., 0 and ∞). In a start state, $now = 0$, and $first(C) = lower(C)$ and $last(C) = upper(C)$ for each enabled task C.

A timed action is a pair associating either an untimed action or a special *time-passage* action with the time it occurs. The time-passage action (ν, t) modifies only the *now* component of the state, setting it equal to t; it cannot let time pass beyond any task's

Specification automaton: $R(a_1, a_2)$	Implementation automaton: $C_k(c_1, c_2)$
State	**State**
reported, initially *false*	*reported*, initially *false*
	count, initially $k \geq 0$
Actions	**Actions**
External *report*	**External** *report*
Pre: $\neg reported$	Pre: $count = 0 \wedge \neg reported$
Eff: $reported \leftarrow true$	Eff: $reported \leftarrow true$
	Internal *decrement*
	Pre: $count > 0$
	Eff: $count \leftarrow count - 1$
Tasks	**Tasks**
$\{report\}$: $[a_1, a_2]$	$\{report\}$: $[c_1, c_2]$
	$\{decrement\}$: $[c_1, c_2]$

Figure 1. A counting process and its specification

upper bound, i.e., $t \leq last(C)$ for all tasks C. Other actions (π, t) are viewed as happening instantaneously at time t. They must not occur before the lower bound for their tasks (i.e., $first(task(\pi)) \leq now$), and they do not modify *now*. They reset the values of *first* and *last* for their task and for any other tasks that are newly enabled or disabled as a result of their effect on the untimed part of the state. We write $s \xrightarrow{(\pi,t)} s'$ to denote a transition of the timed automaton.

An execution of a timed automaton is *admissible* if time increases without bound. A state is *reachable* if it appears in some execution. Properties that are true of every reachable state are *invariants*. The visible behavior of a timed automaton is characterized by its *admissible timed traces*, which are the sequences of external timed actions in admissible executions. We say that one timed automaton *implements* another if any admissible timed trace of the first is also an admissible timed trace of the second.

2.1. Manual Proofs

We seek to show that $C_k(c_1, c_2)$ implements $R(a_1, a_2)$ when $a_1 = (k+1)c_1$ and $a_2 = (k+1)c_2$. Note that our notion of correctness for timed automata incorporates both safety properties (e.g., that C_k issues no more than one report) and liveness properties (e.g., that it issues its report in time at most $(k+1)c_2$).

The key steps in the proof are (1) proving that the states of C_k satisfy an invariant and (2) defining a *simulation mapping* between the states of C_k and those of R. Given such a mapping f, a straightforward proof by induction shows that f maps any admissible execution of C_k to some admissible execution of R. We say that a binary relation f between states of C_k and states of R is a *simulation mapping* from C_k to R if it satisfies the following conditions:

1. If $f(s, u)$, then $u.now = s.now$.
2. If s is a start state of C_k, then there is a start state u of R such that $f(s, u)$.
3. If s and u are reachable states such that $f(s, u)$ and $s \xrightarrow{(\pi,t)} s'$, then there is a state

u' of R such that $f(s', u')$, and a sequence of timed actions that takes R from u to u' and has the same visible behavior as (π, t).

For the first step, we prove that C_k preserves the invariant $count > 0 \Rightarrow \neg reported$. This invariant is trivially true in C_k's initial state. Only the *report* action can make *reported* true, and that can happen only if $count = 0$. Thus, every action preserves the invariant.

For the second step, we define $f(s, u)$, where s is a state of C_k and u is a state of R, to hold if and only if the untimed components of the two states are the same and the timing components are properly related, i.e., if and only if

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

We prove that f is a simulation mapping from $C_k(c_1, c_2)$ to $R(a_1, a_2)$ when $a_1 = (k+1)c_1$ and $a_2 = (k+1)c_2$. If $f(s, u)$, then $u.now = s.now$ by definition. It is also easy to see that $f(s_0, u_0)$, where s_0 and u_0 are the start states of C_k and R. Finally, suppose s and u are reachable states of C_k and R such that $f(s, u)$ and that $s \xrightarrow{(\pi,t)} s'$. We show that there is a sequence of timed actions with the same visible behavior as (π, t) that takes R from u to some state u' such that $f(s', u')$. There are three possibilities for π.

1. If $\pi = report$, we show that R can take a *report* step, resulting in a state u' such that $f(s', u')$. Because $f(s, u)$ and $(report, t)$ is enabled in s, we have $u.reported = s.reported = \mathit{false}$, $s.count = 0$, and $u.\mathit{first}(report) \leq s.\mathit{first}(report) \leq t$. Hence $(report, t)$ is enabled in u and $f(s', u')$, because $u'.now = u.now = s.now = s'.now$.

2. If $\pi = decrement$, we show that R need not take any step. Since *decrement* is internal, it suffices to show that $f(s', u)$. Because $f(s, u)$ and *decrement* occurred, we have $u.now = s.now = s'.now$, $u.reported = s.reported = s'.reported$, $s.count > 0$, and $u.\mathit{first}(report) \leq s.\mathit{first}(decrement) + s.count \cdot c_1 \leq s.now + s.count \cdot c_1$. We consider two cases. If $s.count > 1$, then $u.\mathit{first}(report) \leq s.now + c_1 + (s.count - 1) \cdot c_1 = s'.\mathit{first}(decrement) + s'.count \cdot c_1$, because the time bound for *decrement* is reset. If $s.count = 1$, then $\neg s.reported$ by the invariant for C_k and $u.\mathit{first}(report) \leq s.now + c_1 = s'.\mathit{first}(report)$, because *report* is newly enabled. Similary, $u.last(report) \geq s'.last(decrement) + s'.count \cdot c_2$ if $s.count > 1$ and $u.last(report) \geq s'.last(report)$ if $s.count = 1$.

3. If $\pi = \nu$, we show that R can take a corresponding (ν, t) step, resulting in a state u' such that $f(s', u')$. Since $t \geq s.now = u.now$, to show that (ν, t) is enabled in u', we only need to check that $t \leq u.last(report)$. If $s.count > 0$, then $t \leq s.last(decrement) < u.last(report)$. Otherwise, $t \leq s.last(report) \leq u.last(report)$. Since time-passage actions modify only the *now* components of the states, and $u'.now = t = s'.now$, we have $f(s', u')$.

```
AutomatonCount (C, k): trait
  includes Automaton(C), CommonActionsRC, Natural
  States[C] tuple of count: N, reported: Bool
  introduces
    k                  : → N
    decrement, report  : → Actions[C]
  asserts
    sort Actions[C] generated by report, decrement
    sort Tasks[C] generated by task
    ∀ s, s': States[C], a, a': Actions[C]
      isExternal(report); isInternal(decrement); common(report) = report;
      task(a) = task(a')        ⇔ a = a';
      start(s)                  ⇔ ¬s.reported ∧ s.count = k;
      enabled(s, report)        ⇔ s.count = 0 ∧ ¬s.reported;
      effect(s, report, s')     ⇔ s'.count = s.count ∧ s'.reported;
      enabled(s, decrement)     ⇔ s.count > 0;
      effect(s, decrement, s')  ⇔ s'.count + 1 = s.count ∧ s'.reported = s.reported;
      inv(s)                    ⇔ s.count > 0 ⇒ ¬s.reported
  implies
    Invariants(C, inv)
    ∀ s: States[C], a: Actions[C]
      enabled(s, task(decrement)) ⇔ enabled(s, decrement);
      enabled(s, task(report)) ⇔ enabled(s, report);
      a = report ∨ a = decrement
```

Figure 2. LSL trait defining untimed part of automaton C_k

2.2. Machine-Checked Proofs

In order to check this simulation proof mechanically, we must first create machine-readable versions of the definitions and abstractions used in the manual proof, filling in details normally suppressed in careful, but not completely formal proofs. To this end, we use the Larch Shared Language (LSL), which provides suitable notational and parametrization facilities. Later, we use the Larch Prover (LP), which provides assistance for reasoning in first-order logic. The versions of these tools used for this paper are enhancements of the versions described in [5, 4]; the primary differences are that both tools now support full first-order logic, and that LP now has features for reasoning about linear inequalities [17] similar to those in the Boyer-Moore prover [2, 3] and in PVS [19].

2.3. Machine-Readable Definitions

Figure 2 contains an LSL definition of the untimed part of automaton C_k. This formal definition mimics the definition given in Figure 1. It builds upon a library of LSL specifications, shown in Appendix B, that defines general notions related to timed automata and that can be reused in simulation proofs like the ones in this paper.

The basic unit of specification in LSL is a *trait*, which introduces symbols for *sorts* (such as `Actions[C]` and `States[C]`) and *operators* (such as `decrement` and `enabled`), and which constrains their properties by axioms expressed in first-order logic. Sort symbols denote disjoint nonempty sets of values; operator symbols denote total mappings from

```
SimulationRC: trait
  includes
    TimedAutomaton(R, br, TR), AutomatonReport(R),
    TimedAutomaton(C, bc, TC), AutomatonCount(C, k)
  introduces
    a, c :                          → Bounds
    f    : States[TC], States[TR] → Bool
  asserts ∀ u: States[TR], s: States[TC], cr: Tasks[R], cc: Tasks[C]
    br(cr) = a;              % bounds [a1, a2] for tasks of R
    bc(cc) = c;              % bounds [c1, c2] for tasks of C
    c.bounded;
    a = (k+1)*c;
    f(s, u) ⇔
        u.now = s.now
      ∧ u.basic.reported = s.basic.reported
      ∧ (if s.basic.count > 0
          then s.bounds[task(decrement)] + (s.basic.count * c)
          else s.bounds[task(report)]) ⊆ u.bounds[task(report)]
  implies SimulationMap(TC, TR, f)
```

Figure 3. LSL trait defining the timed simulation of R by C_k

tuples of values to values. When a trait *includes* another, it inherits the other trait's symbols and axioms. Thus `AutomatonCount` inherits general properties of automata from the library trait `Automaton` and properties of the natural numbers from the trait `Natural` in the Larch handbook [5]. Because LSL requires sorts to represent disjoint nonempty sets, `AutomatonCount` also includes the following trait `CommonActionsRC`, and it defines a map `common` from the actions of `C` to a new sort `CommonActions` so that the traces of `C` (whose actions have sort `Actions[C]`) can be compared with those of `R` (whose actions have sort `Actions[R]`).

```
CommonActionsRC: trait
  introduces report: → CommonActions
  asserts CommonActions generated by report
```

When a trait *implies* another, its theory is claimed to include that of the other. The `implies` clause in `AutomatonCount` claims that the predicate `inv` satisfies the axioms of the library trait `Invariants`; Figure 4 contains an LP proof of this claim. The `implies` clause also lists several lemmas that are easy to verify with LP, but are not noticed automatically by the prover.

The specification of R's untimed part is similar to, but shorter than C_k's. The trait `SimulationRC` in Figure 3 uses the library trait `TimedAutomaton` to extend these two specifications to the timed parts of C_k and R. It also claims that a particular relation `f` is a simulation mapping, i.e., that `f` satisfies the properties of the library trait `SimulationMap`. Later we use LP to verify this claim.

```
execute AutomatonCount
set proof-methods ⇒, normalization
prove start(s) ⇒ inv(s)
  qed
prove inv(s) ∧ isStep(s, a, s') ⇒ inv(s') by cases on a
  qed
```

Figure 4. LP proof of invariance for automaton C_k

The most notable feature of the formalization process is that it is quite mechanical to move from definitions such as those in Figure 1 to LSL definitions. In fact, one could write a compiler to perform the translation.

2.4. Machine-Checkable Proofs

This section contains two entire LP proof scripts, one showing that automaton C_k preserves its invariant, and the other that `f` is indeed a timed forward simulation. LP's proof mechanisms include proofs by cases and induction, equational term rewriting (for simplifying hypotheses and conjectures), and decision procedures for proving linear inequalities.

The LP proof of invariance in Figure 4 is virtually identical to the manual proof. It begins with commands that load the axioms of the trait `AutomatonCount` and that set LP's proof methods. That the invariant holds in the initial state is proved without human guidance. That the invariant is preserved by all actions requires exactly the same guidance as in the manual proof: separate consideration of each action.

The proof that `f` is a simulation mapping in Figure 5 is considerably longer than the proof of invariance, but similar in length and structure to the manual proof.[1] The user guides the proof that each start state `s` of C_k corresponds to a start state `u` of R by producing an explicit description of `u` and showing LP why "it is easy to see that `f(s, u)`."[2] In the induction step of the proof, `s'c` and `sc` are fresh constants that LP generates and substitutes for the variables `s` and `s'` when it assumes the hypotheses of the implication it is trying to prove. In addition to suggesting separate consideration of each action, and to providing the simulating execution fragment for each action, the user provides guidance for the induction step of the proof using the `set immunity` and `instantiate` commands, which call LP's attention to instances of the hypotheses (and other facts) used by the decision procedure for linear arithmetic.

3. Fischer's Mutual Exclusion Algorithm

In this section, we use timed automata to model Fischer's well-known timing-based mutual exclusion algorithm, which uses a single shared read-write register [7]. We use

[1]Two periods `..` in this proof script mark the end of a multiline LP command; they do not indicate any elision of the script.

[2]While the length of this proof suggests room for improvement in LP, the need to consider the case $k = 0$ separately suggests room for clarification in the manual proof.

```
execute SimulationRC
set proof-methods ⇒, normalization
prove f(s, u) ⇒ u.now = s:States[TC].now
  qed
prove start(s:States[TC]) ⇒ ∃ u (start(u) ∧ f(s, u))
    resume by specializing u to [[false], 0, update({}, task(report), a)]
      instantiate c:Tasks[C] by task(report) in *Hyp
      instantiate c:Tasks[C] by task(decrement) in *Hyp
      resume by specializing a:Actions[R] to report
        resume by case k = 0
          resume by cases on c:Tasks[R]
          resume by cases on c:Tasks[R]
  qed
declare variables u: States[TR], alpha: StepSeq[TR]
set immunity ancestor
prove
  f(s, u) ∧ isStep(s:States[TC], a, s') ∧ inv(s:States[TC]) ∧ inv(u:States[TR])
    ⇒ ∃ alpha (execFrag(alpha) ∧ first(alpha) = u ∧ f(s', last(alpha))
                  ∧ trace(alpha) = trace(a:Actions[TC]))
  by cases on a:Actions[TC]
  ..
      resume by cases a1c = report, a1c = decrement
        % Case 1: simulate decrement action
        resume by specializing alpha to {uc}
          instantiate c:Tasks[C] by task(report) in *impliesHyp
          instantiate c:Tasks[C] by task(decrement) in *impliesHyp
          resume by case s'c.basic.count = 0
            instantiate t:Time by c.first, n by s'c.basic.count in Real
            instantiate t:Time by c.last, n by s'c.basic.count in Real
        % Case 2: simulate report action
        resume by specializing alpha to
          ({uc}) {addTime(report, uc.now),
                  [[true], uc.now, update(uc.bounds, task(report), [false,0,0])]}
          ..
          resume by cases on c:Tasks[R]
      % Case 3: simulate passage of time
      resume by specializing alpha to ({uc}) {nu(lc), [uc.basic, lc, uc.bounds]}
        resume by cases on c:Tasks[R]
          instantiate c:Tasks[C] by task(report) in *Hyp
          resume by case sc.basic.count = 0
            instantiate c:Tasks[R] by reportTask in *Hyp
            instantiate n by sc.basic.count in TimedAutomaton
            instantiate c:Tasks[C] by task(decrement) in *Hyp
  qed
```

Figure 5. LP proof that `f` is a simulation mapping

State
$region_i \in \{remainder, trying, critical, exit\}$ for $i \in I$, initially $remainder$

Actions

External try_i
Pre: $region_i = remainder$
Eff: $region_i \leftarrow trying$

External $exit_i$
Pre: $region_i = critical$
Eff: $region_i \leftarrow exit$

External $crit_i$
Pre: $region_i = trying$
for all j, $region_j \neq critical$
Eff: $region_i \leftarrow critical$

External rem_i
Pre: $region_i = exit$
Eff: $region_i \leftarrow remainder$

Tasks
$\{try_i\}$: $[0, \infty]$
$\{crit_i : i \in I\}$: $[0, 5a + 2c]$
$\{exit_i\}$: $[0, \infty]$
$\{rem_i\}$: $[0, 2a]$

Figure 6. Automaton M: a simple specification for mutual exclusion

simulations to prove not only mutual exclusion, but also an upper bound on the time to reach the critical region, which is much harder to prove than mutual exclusion. We believe that the use of simulations both gives insight into the algorithm and yields a convincing proof that can be checked using automated provers like LP.

3.1. A Specification for Mutual Exclusion

We begin with the specification in Figure 6 of a *mutex object* M described as a timed automaton that keeps track of the regions of all processes (with indices in I) and ensures that at most one process is in its critical region at any time.

Notice that all *crit* actions belong to the same task. Intuitively, this means that if one or more processes are trying to acquire the resource when it is free, then one will succeed within time $5a + 2c$. (The parameters a and c here are derived from the bounds we will impose on the tasks of Fischer's algorithm.)

3.2. Fischer's Timed Mutual Exclusion Algorithm

In this algorithm, shown in Figure 7, there is a single shared register. Intuitively, if some process has the resource, the register contains the index of that process; and if no process has, wants, or is releasing the resource, the register contains 0.[3] Each process trying to obtain the resource tests the register until its value is 0, and then sets it to its own index. Since several processes may be competing for the resource, the process waits for the register value to stabilize, and then checks the register again. The process whose index remains in the register (the last one to set it) gets the resource, and the others return to testing until the register is 0 again. When a process exits, it resets the register to 0.

One problem with this algorithm as described so far is that a fast process might not wait

[3]We assume $0 \notin I$.

State

$pc_i \in \{remainder, test, set, check, leave\text{-}trying, critical, reset, leave\text{-}exit\}$ for $i \in I$, initially $remainder$
$x \in I \cup \{0\}$, initially 0

Actions

External try_i
Pre: $pc_i = remainder$
Eff: $pc_i \leftarrow test$

Internal $test_i$
Pre: $pc_i = test$
Eff: if $x = 0$ then $pc_i \leftarrow set$

Internal set_i
Pre: $pc_i = set$
Eff: $x \leftarrow i$
$pc_i \leftarrow check$

Internal $check_i$
Pre: $pc_i = check$
Eff: if $x = i$
then $pc_i \leftarrow leave\text{-}trying$
else $pc_i \leftarrow test$

External $crit_i$
Pre: $pc_i = leave\text{-}trying$
Eff: $pc_i \leftarrow critical$

External $exit_i$
Pre: $pc_i = critical$
Eff: $pc_i \leftarrow reset$

Internal $reset_i$
Pre: $pc_i = reset$
Eff: $x \leftarrow 0$
$pc_i \leftarrow leave\text{-}exit$

External rem_i
Pre: $pc_i = leave\text{-}exit$
Eff: $pc_i \leftarrow remainder$

Tasks

Assume $a < b \leq c$

$\{try_i\}$: $[0, \infty]$
$\{test_i\}$: $[0, a]$
$\{set_i\}$: $[0, a]$
$\{check_i\}$: $[b, c]$
$\{crit_i\}$: $[0, a]$
$\{exit_i\}$: $[0, \infty]$
$\{reset_i\}$: $[0, a]$
$\{rem_i\}$: $[0, a]$

Figure 7. Automaton F: Fischer's algorithm

long enough, check the register before a slow process has managed to set it, and so proceed to its critical region. The slow process might then overwrite the register with its own index, which would remain there until the slow process checked it and entered its critical region as well, violating mutual exclusion. This situation can be avoided by a simple time restriction that requires every process to wait long enough for any other process to see the new value in the register, or else to overwrite it. Formally, $upper(set_i) < lower(check_j)$ for all $i, j \in I$.

Notice that every action is a task by itself, corresponding to our intuition that each process acts independently of the other processes. We define timing conditions for all the tasks other than try_i and $exit_i$ in order to prove the timing conditions for the specification.[4]

Finally, we use the following invariants in our proofs of the simulations. The last, which we call *strong mutual exclusion*, clearly implies mutual exclusion.

[4] We can show tight, slightly better bounds at the cost of additional complexity. See [9].

State

$region_i \in \{remainder, trying, critical, exit\}$ for $i \in I$, initially $remainder$
$status$, an element of $\{start, seized, stabilized\}$, initially $start$

Actions

External try_i
Pre: $region_i = remainder$
Eff: $region_i \leftarrow trying$

Internal $seize$
Pre: for some i, $region_i = trying$
$status = start$
for all i, $region_i \neq critical$
Eff: $status \leftarrow seized$

Internal $stabilize$
Pre: $status = seized$
Eff: $status \leftarrow stabilized$

External $crit_i$
Pre: $region_i = trying$
$status = stabilized$
Eff: $region_i \leftarrow critical$
$status \leftarrow start$

External $exit_i$
Pre: $region_i = critical$
Eff: $region_i \leftarrow exit$

External rem_i
Pre: $region_i = exit$
Eff: $region_i \leftarrow remainder$

Tasks

$\{try_i\}$: $[0, \infty]$
$\{seize\}$: $[0, 3a + c]$
$\{stabilize\}$: $[0, a]$
$\{crit_i : i \in I\}$: $[0, a + c]$
$\{exit_i\}$: $[0, \infty]$
$\{rem_i\}$: $[0, 2a]$

Figure 8. Automaton I: an intermediate milestone automaton

1. If $x = i$, then $pc_i \in \{check, leave\text{-}trying, critical, reset\}$.
2. If $x = i \neq 0$, $pc_i = check$, and $pc_j = set$ then $first(check_i) > last(set_j)$.
3. If $pc_i \in \{leave\text{-}trying, critical, reset\}$, then $x = i$ and $pc_j \neq set$ for all j.

3.3. Milestones: An Intermediate Abstraction

While we could give a simulation mapping directly from F to M, it seems useful to introduce an intermediate level of abstraction that we believe captures the intuition behind Fischer's algorithm. We then define two intuitive simulation mappings, one from the algorithm to the intermediate automaton, and one from the intermediate automaton to the specification, thereby proving that the algorithm implements the specification.

The intermediate automaton, shown in Figure 8, expresses two *milestones* toward the goal of some process reaching its critical region. The first occurs when a process sets the register from 0 to its index; we say that the register is *seized* at this point. After this, the register will have some non-zero value until some process reaches its critical region and resets the register as it exits. Thus only processes that have already tested the register will set it. The second milestone, a *stabilize* event, occurs when the last process sets the register, i.e., when no other process has $pc = set$.

We need one easy invariant for this automaton:

If $status \neq start$, then $region_i = trying$ for some i and $region_j \neq critical$ for all j.

3.4. Simulations

3.4.1. Simulation from Intermediate to Specification

We define a relation g between the states of I and M, where $g(s,u)$ if and only if:

- $u.now = s.now$
- $u.region_i = s.region_i$
- $u.last(crit) \geq \begin{cases} s.last(seize) + 2a + c & \text{if } seize \text{ is enabled in } s \\ s.last(stabilize) + a + c & \text{if } stabilize \text{ is enabled in } s \\ s.last(crit) & \text{if } crit_j \text{ is enabled in } s \text{ for some } j \end{cases}$
- $u.last(rem_i) \geq s.last(rem_i)$ if $s.region_i = exit$

It is straightforward to show that g is a simulation mapping. This simulation corresponds to the notion that *seiz*ing and *stabiliz*ing are just steps that need to be done before a process can enter its critical region. Note, however, that *seize* and *stabilize* are not actions of individual processes, but of the entire system.

3.4.2. Simulation from Algorithm to Intermediate

We define a relation f between the states of F and I, where $f(s,u)$ if and only if:

- $u.now = s.now$
- $u.region_i = \begin{cases} trying & \text{if } s.pc_i \in \{test, set, check, leave\text{-}trying\} \\ critical & \text{if } s.pc_i = critical \\ exit & \text{if } s.pc_i \in \{reset, leave\text{-}exit\} \\ remainder & \text{if } s.pc_i = remainder \end{cases}$
- $u.status = \begin{cases} start & \text{if } s.x = 0 \text{ or } s.pc_i \in \{critical, reset\} \text{ for some } i \\ seized & \text{if } s.x \neq 0,\ s.pc_i \notin \{critical, reset\} \text{ for all } i \text{, and} \\ & \quad s.pc_i = set \text{ for some } i \\ stabilized & \text{if } s.x \neq 0 \text{ and } s.pc_i \notin \{set, critical, reset\} \text{ for all } i \end{cases}$
- $u.last(seize) \geq \begin{cases} s.last(reset_i) + 2a + c & \text{if } s.pc_i = reset \\ \min_i\{w(i)\} & \text{if } s.x = 0 \text{ where } w(i) = \begin{cases} s.last(test_i) + a & \text{if } s.pc_i = test \\ s.last(set_i) & \text{if } s.pc_i = set \\ s.last(check_i) + 2a & \text{if } s.pc_i = check \\ \infty & \text{otherwise} \end{cases} \end{cases}$
- $u.last(stabilize) \geq s.last(set_i) \quad$ if $s.pc_i = set$
- $u.last(crit) \geq \begin{cases} s.last(check_i) + a & \text{if } s.pc_i = check \text{ and } s.x = i \\ s.last(crit_i) & \text{if } s.pc_i = leave\text{-}trying \end{cases}$
- $u.last(rem_i) \geq \begin{cases} s.last(reset_i) + a & \text{if } s.pc_i = reset \\ s.last(rem_i) & \text{if } s.pc_i = leave\text{-}exit \end{cases}$

The *now* and *region* correspondences are straightforward; that for *status* follows naturally from the intuition given earlier about the *seize* and *stabilize* milestones. The first inequality for *seize* says that if some process is about to *reset*, then the simulated state must allow the register to be seized at least up to $2a+c$ after the *reset* occurs. The second inequality for *seize* says that if $x = 0$ (so, by strong mutual exclusion, no process is about to *reset*) then the time until the register must be 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.

Most of the proof that f is a simulation mapping involves straightforward but tedious checking that each action of F preserves the mapping, since the corresponding behavior in I is easy to intuit. (It is the same action if it is external, and no action if not.) The one exception to this is the *set* action. Recall the intuition here is that, if it is the first time the register is set (i.e., it was previously 0), then there must be a corresponding *seize* action. If no other process is about to set the register (i.e., no other process has $pc = set$), then this is the last *set* before some process enters its critical region, and so there must be a corresponding *stabilize* action. We examine this case and its proof in more detail.

If s and u are reachable states of F and I such that $f(s,u)$ and $s \stackrel{(set_i,t)}{\longrightarrow} s'$, then $s.pc_i = set$, $s'.pc_i = check$, and $s'.x = i \neq 0$. By strong mutual exclusion, $s'.pc_j \notin \{critical, reset\}$ for all j. We have the following cases:

1. If $s.x = 0$, let u' be such that $u \stackrel{(seize,t)}{\longrightarrow} u'$. The state exists because $u.status = start$, $u.region_i = trying$, and $u.region_j \neq critical$ for all j.

   (a) If $s.pc_j \neq set$ for all $j \neq i$, then let u'' be such that $u' \stackrel{(stabilize,t)}{\longrightarrow} u''$, which is possible since $u'.status = seized$. So $u'' = u$ except that $u''.status = stabilized$, $u''.last(seize) = \infty$ and $u''.last(crit) = s.now + a + c$. Since $s.now + a + c$ is greater than any of the time bounds that occur in the condition for $last(crit)$, and $s'.pc_j \neq set$ for all j, we have $f(s', u'')$.

   (b) If $s.pc_j = set$ for some $j \neq i$, then we see that $f(s', u')$ since $s'.pc_j = set$ and $u' = u$ except that $u'.status = seized$, $u'.last(seize) = \infty$, and $u'.last(stabilize) = s.now + a \geq s'.last(set_{j'})$ for all j' such that $s'.pc_{j'} = set$.

2. If $s.x \neq 0$ and $s.pc_j \neq set$ for all $j \neq i$, then let u' be such that $u \stackrel{(stabilize,t)}{\longrightarrow} u'$. The state exists because $u.status = seized$, and $u' = u$ except that $u'.status = stabilized$, $u'.last(stabilize) = \infty$, and $u'.last(crit) = s.now + a + c$. Since $s.now + a + c$ is greater than any of the time bounds that occur in the condition for $last(crit)$, and $s'.pc_j \neq set$ for all j, we have $f(s', u')$.

3. If $s.x \neq 0$ and $s.pc_j = set$ for some $j \neq i$, then $f(s', u)$ since $u.status = seized$, and $s'.pc_j = set$.

Our method of proof uses old, familiar techniques (invariant assertions and simulation mappings) in a novel way (on timed automata) to provide rigorous proofs of timing properties. The time bounds established by this simulation are new; there was no clear, rigorous proof of them before. Furthermore, the bounds aren't completely obvious: the extra c is necessary; we can demonstrate executions that need this extra time. We used the same library of LSL traits that we used for the counting process to formalize these automata and simulations, and we used LP to check the entire proof.

4. Conclusions

We have defined, within the Larch Shared Language, a set of abstractions to support proofs of timing properties of timed systems. We have used these abstractions to carry out computer-aided proofs of time bounds for two sample algorithms—a simple counter and Fischer's mutual exclusion protocol—using invariant assertion and simulation techniques.

We see several advantages of this general approach. Because they can be used for proofs of timing properties in addition to ordinary correctness properties, invariants and simulations are very powerful in the real-time setting. The invariants and simulation mappings also serve as "documentation", expressing key insights about a system's behavior (including its timing). Our experience in going from the simple counter to Fischer's algorithm suggests that these methods are scalable to systems of realistic size. They also appear to provide assistance when modifying systems. When we modify a system or its specification only slightly, we expect that LP will be able to recheck most of the original proof automatically, thereby allowing us to concentrate our attention on what has truly changed without having to worry that we have overlooked some important detail.

The first proof we attempted, that of the counter, took many weeks. Making it work successfully required understanding the manual proof better (e.g., that it relied on an invariant of the automaton C), finding LSL formalizations that were easy to reason about using LP, and finding appropriate LP proof strategies (e.g., for dealing with transitivity before LP was enhanced with decision procedures for linear inequalities). As a result of our increased understanding, and of enhancements made to LP in response to our experience, the proof for Fischer's algorithm took much less time—about four days to fill in all the details of the last simulation, from F to I, which was the most difficult. This amount of time does not seem unreasonable, given that we get the added assurance of a machine-checked proof. But we would like to reduce further the amount of time and user guidance required for proofs of this sort. We expect this to happen as we refine our formalizations and our tools, and we believe that practical machine-checked proofs for real-time processes are not such a distant goal.

Finally, we expect to use our methods to prove timing properties for many more examples. We also expect to extend the timed automaton model used in this paper to encompass other timing-based systems that arise in practice. For example, work in [6] on the Generalized Railroad Crossing example uses a slightly more general timed automaton model [10, 13]; nevertheless, the proof uses simulation methods very similar to those in this paper.

REFERENCES

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2(82):253–284, 1992.
2. Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.
3. Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.
4. Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, DEC Systems Research Center, December 1991.
5. John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
6. 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.

7. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
8. P. Loewenstein and David L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, number 531 in LNCS, pages 302–311. Springer-Verlag, 1990.
9. Victor Luchangco. Using simulation techiniques to prove timing properties. Master's thesis, MIT Electrical Engineering and Computer Science, 1994. In progress.
10. Nancy Lynch. Simulation techniques for proving properties of real-time systems. Technical Memo MIT/LCS/TM-494, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, November 1993.
11. Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. Technical Memo MIT/LCS/TM-412.e, Lab for Computer Science, Massachusetts Institute Technology, Cambridge, MA, November 1991.
12. Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
13. Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In J. W. de Bakker, C. Huizing, and G. Rozenberg, editors, *Proceedings of REX Workshop "Real-Time: Theory in Practice"*, number 600 in LNCS, pages 397–446. Springer-Verlag, 1992.
14. Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part II: Timing-based systems. Technical Memo MIT/LCS/TM-487, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, April 1993.
15. Michael Merritt, F. Modugno, and Mark Tuttle. Time constrained automata. In *CONCUR'91 Proceedings of a Workshop on Theories of Concurrency: Unification and Extension*, Amsterdam, August 1991.
16. Tobias Nipkow. Formal verification of data type refinement. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, number 430 in LNCS, pages 561–589. Springer-Verlag, 1990.
17. Anna Pogosyants. Incorporating specialized theories into a general purpose theorem prover. Master's thesis, MIT Electrical Engineering and Computer Science, 1994. In progress.
18. F. B. Schneider, B. Bloom, and K. Marzullo. Putting time into proof outlines. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real Time: Theory and Practice*, Mook, The Netherlands, June 1991. Springer Verlag.
19. N. Shankar. Verification of real-time systems using PVS. In *Fourth Conference on Computer-Aided Verification*, pages 280–921, Elounda, Greece, June 1993. Springer-Verlag.
20. Jørgen Søgaard-Andersen. *Correctness of Protocols in Distributed Systems.* PhD thesis, Technical University of Denmark, Lyngby, Denmark, December 1993.
21. Jørgen Søgaard-Andersen, Stephen Garland, John Guttag, Nancy Lynch, and Anya Pogosyants. Computer-assisted simulation proofs. In *Fourth Conference on Computer-Aided Verification*, pages 305–319, Elounda, Greece, June 1993. Springer-Verlag.

A. Input/Output Automata and Simulations

A.1. The I/O Automaton Model

We use a slight variant of the standard *I/O automaton model* from [12]. 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.[5]
- a set $steps(A)$ of steps, which is a subset of $states(A) \times acts(A) \times states(A)$;
- a partition $tasks(A)$ of the actions into at most countably many equivalence classes.

We write $s \xrightarrow{\pi}_A s'$ or just $s \xrightarrow{\pi} s'$ as shorthand for $(s, \pi, s') \in steps(A)$.

An action π is said to be *enabled* in a state s provided that there exists a state s' such that $s \xrightarrow{\pi} s'$. A set of actions is said to be *enabled* in s if some action in the set is enabled in s.

An *execution fragment* is a finite or infinite alternating sequence $s_0\pi_1 s_1 \pi_2 s_2 \ldots$, where s_j is a state, π_j is an action, $s_{j-1} \xrightarrow{\pi_j} s_j$ for each j, and the sequence ends with a state if it is finite. An *execution* is an execution fragment with $s_0 \in start(A)$. A state of an I/O automaton is *reachable* if it is the final state of some finite execution of the automaton.

The *trace* of an execution is the sequence of external actions that occur in the execution. Often, we express requirements to be satisfied by an I/O automaton A by another I/O automaton B.

A.2. MMT Automata

MMT automata were originally defined by Merritt, Modugno and Tuttle [15]; we use a special case of their definition appearing in [11, 13]. An MMT automaton is an I/O automaton with only finitely many tasks together with lower and upper time bounds, $lower(C)$ and $upper(C)$, for each task C. We require that $0 \leq lower(C) < \infty$, $0 < upper(C) \leq \infty$, and $lower(C) \leq upper(C)$.

A *timed execution* of an MMT automaton is a sequence $s_0(\pi_1, t_1)s_1(\pi_2, t_2)s_2 \ldots$, where $s_0\pi_1 s_1 \pi_2 s_2 \ldots$ is an execution of the underlying I/O automaton, $t_i \leq t_{i+1}$, and t_i satisfies the given *lower* and *upper* bound requirements. Formally, define j to be an *initial index* for a task C provided that C is enabled in s_j, and $j = 0$, or C is not enabled in s_{j-1}, or $\pi_j \in C$; initial indices are the points at which the bounds for C begin to be measured. Then for every initial index j for a task C, the following conditions must hold:

1. (Upper bound) If $upper(C) \neq \infty$, 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 any $k > j$ with $t_k < t_j + lower(C)$ and $\pi_k \in C$.

Finally, if the execution is infinite, it must be *admissible*, i.e., the times associated with the actions must increase without bound. Each timed execution of an MMT automaton A gives rise to a *timed trace*, which is just the subsequence of external actions and their

---

[5] The external actions are usually further partitioned into *input* and *output* actions, thus the name "I/O automaton". This distinction is important for composition and fairness, which we do not consider in this paper. A more complete discussion is found in [12].

associated times. The *admissible timed traces* of an MMT automaton A are the timed traces that arise from the admissible timed executions of A.

A.3. Timed Automata

Lynch and Attiya [11] describe how to incorporate the timing information of an MMT automaton A into the state, yielding an I/O automaton A' of a special form. We call automata derived in this way *timed automata*.

Each state of A' is a record consisting of a component *basic*, which is a state of A, a component $now \in \mathsf{R}^{\geq 0}$, and, for each task C of A, components $first(C)$ in $\mathsf{R}^{\geq 0}$ and $last(C)$ in $\mathsf{R}^{\geq 0} \cup \{\infty\}$. For each start state s of A', $s.basic \in start(A)$ and $s.now = 0$. Also, if s is a start state and 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 pairs of an action of A or the *time-passage* action ν, and non-negative reals. Each non-time-passage action is classified as external or internal; time-passage actions are internal.

If $\pi \in acts(A)$, then $s \xrightarrow{(\pi,t)}_{A'} s'$ exactly if all the following conditions hold:

1. $s'.now = s.now = t$.
2. $s.basic \xrightarrow{\pi}_A s'.basic$.
3. For each $C \in tasks(A)$:
   (a) If $\pi \in C$ then $s.first(C) \leq t$.
   (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 $s'.first(C) = t + lower(C)$ and $s'.last(C) = t + upper(C)$. In this case, we say that C is *newly enabled* in s'.
   (d) If C is not enabled in s' then $s'.first(C) = 0$ and $s'.last(C) = \infty$.

On the other hand, $s \xrightarrow{(\nu,t)}_{A'} s'$ exactly if all the following conditions hold:

1. $s.now < t = s'.now$.
2. $s'.basic = s.basic$.
3. For each $C \in tasks(A)$:
   (a) $t \leq s.last(C)$.
   (b) $s'.first(C) = s.first(C)$ and $s'.last(C) = s.last(C)$.

We define the *admissible timed executions* of A' to be those in which the times associated with the time-passage actions increase without bound, and the *admissible timed traces* to be the traces of admissible timed executions. With this definition, the MMT automaton A and its corresponding timed automaton A' have exactly the same admissible timed traces.

We refer to the MMT automaton and its corresponding timed automaton interchangeably. Also, we often omit the *basic* part of the selector, writing *s.field* as a shorthand for *s.basic.field*, where *field* is a component of the MMT automaton's state.

Timed automata satisfy the following invariants:

Lemma 1 *In all reachable states of A', and for every task C:*

1. $now \leq last(C)$

2. $first(C) \leq now + lower(C)$

3. If C is enabled, then $last(C) \leq now + upper(C)$.

4. If C is not enabled, then first$(C) = 0$ and last$(C) = \infty$.
5. If upper$(C) = \infty$, then last$(C) = \infty$.

A.4. Invariants and Simulations

An *invariant* of a automaton is any property that is true in all reachable states. We usually establish an invariant I by proving that all start states satisfy it, and that all steps preserve it, i.e., $start(s) \Rightarrow I(s)$ and $I(s) \wedge (s \xrightarrow{\pi} s') \Rightarrow I(s')$.

The definition of a simulation mapping is paraphrased from [13, 14, 10]. If A and B are timed automata with invariants I_A and I_B, then a *simulation mapping* from A to B with respect to I_A and I_B is a relation f between $states(A)$ and $states(B)$ such that:

1. If $f(s,u)$, then $u.now = s.now$.
2. If $s \in start(A)$, then there exists some $u \in start(B)$ such that $f(s,u)$.
3. If $f(s,u)$ for states s and u of A and B satisfying I_A and I_B respectively, and $s \xrightarrow{(\pi,t)}_A s'$, then there exists some u' such that $f(s',u')$ and there is some execution fragment from u to u' with the same timed external actions as (π, t).

The most important fact about simulation mappings is that they imply admissible timed trace inclusion.

Theorem 1 *If there is a simulation mapping from A to B, with respect to some invariants, then every admissible timed trace of A is an admissible timed trace of B.*

B. Library of LSL Traits for Timed Automata

The trait `Automaton` (Figure 9) provides LSL definitions for terminology regarding untimed automata. For example, it defines the execution fragments of an automaton `A` to be those elements of sort `StepSeq[A]` that satisfy the predicate `execFrag`, which itself is defined inductively.

The trait `Invariants` (Figure 10) lists the proof obligations for showing that a property is an invariant of an automaton. The Larch tools provide support for checking that these properties hold.

The trait `Bounds` (Figure 11) describes intervals, which may be unbounded above, of time during which an action may occur. Time itself is modeled as a real number using the Larch handbook trait `Real`, upon which LP's decision procedure for linear inequalities is based.

The trait `TimedAutomaton` (Figures 12 and 13) associates time bounds `b(c)` with each task `c` of an untimed automaton `A`, defining a timed automaton `TA`. This corresponds directly to the transformation of an MMT automaton into a timed automaton described in the Appendix A.

Finally, the trait `SimulationMap` (Figure 14), which generated the proof obligations in Figure 5, defines what it means for one timed automaton to simulate another. This also corresponds directly to the definition of simulation mappings in Appendix A. Recall that timed automata are actually just untimed automata with special requirements; in particular they must have a *now* component. Thus we use the `NowExists` assumption to ensure that this definition is applied only to automata for which it is meaningful.

```
Automaton (A): trait
  introduces
    start       : States[A]                         → Bool
    enabled     : States[A], Actions[A]             → Bool
    effect      : States[A], Actions[A], States[A]  → Bool
    isExternal  : Actions[A]                        → Bool
    isInternal  : Actions[A]                        → Bool
    isStep      : States[A], Actions[A], States[A]  → Bool
    {__}        : States[A]                         → StepSeq[A]
    __{__,__}   : StepSeq[A], Actions[A], States[A] → StepSeq[A]
    execFrag    : StepSeq[A]                        → Bool
    first, last : StepSeq[A]                        → States[A]
    common      : Actions[A]                        → CommonActions
    empty       :                                   → Traces
    __ ^ __     : Traces, CommonActions             → Traces
    trace       : Actions[A]                        → Traces
    trace       : StepSeq[A]                        → Traces
    task        : Actions[A]                        → Tasks[A]
    enabled     : States[A], Tasks[A]               → Bool
    inv         : States[A]                         → Bool
  asserts
    sort StepSeq[A] generated by {__}, __{__,__}
    sort Traces generated by empty, ^
    ∀ s, s': States[A], a, a': Actions[A], ss: StepSeq[A], t: Tasks[A]
      isInternal(a) ⇔ ¬isExternal(a);
      isStep(s, a, s') ⇔ enabled(s, a) ∧ effect(s, a, s');
      execFrag({s});
      execFrag(({s}){a,s'}) ⇔ isStep(s, a, s');
      execFrag((ss{a,s}){a',s'}) ⇔ execFrag(ss{a,s}) ∧ isStep(s, a', s');
      first({s}) = s;
      last({s}) = s;
      first(ss{a,s}) = first(ss);
      last(ss{a,s}) = s;
      trace({s}) = empty;
      trace(ss{a,s}) = (if isExternal(a) then trace(ss) ^ common(a) else trace(ss));
      trace(a) = (if isExternal(a) then empty ^ common(a) else empty);
      enabled(s, t) ⇔ ∃ a (enabled(s, a) ∧ task(a) = t)
```

Figure 9. LSL trait defining untimed automata

```
Invariants (A, inv): trait
  assumes Automaton(A)
  asserts ∀ s, s': States[A], a: Actions[A]
    start(s) ⇒ inv(s);
    inv(s) ∧ isStep(s, a, s') ⇒ inv(s')
```

Figure 10. LSL trait defining proof obligations for proofs of invariance

```
Bounds: trait
  includes Real(Time)
  Bounds tuple of bounded: Bool, first, last: Time
  introduces
    __+__ : Bounds, Time   → Bounds
    __+__ : Bounds, Bounds → Bounds
    __*__ : N, Bounds      → Bounds
    __ ⊆ __ : Bounds, Bounds → Bool
    __ ∈ __ : Time, Bounds   → Bool
  asserts ∀ b, b1, b2: Bounds, t: Time, n: N
    0 ≤ b.first;
    b.first ≤ b.last;
    b + t = [b.bounded, b.first + t, b.last + t];
    b1 + b2 = [b1.bounded ∧ b2.bounded, b1.first + b2.first, b1.last + b2.last];
    n * b = [b.bounded, n * b.first, n * b.last];
    b1 ⊆ b2 ⇔
      b2.first ≤ b1.first
        ∧ ( (b1.bounded ∧ b2.bounded ∧ b1.last ≤ b2.last) ∨ ¬b2.bounded );
    t ∈ b ⇔ b.first ≤ t ∧ (t ≤ b.last ∨ ¬b.bounded)
```

Figure 11. LSL definition of time bounds for actions in an automaton

```
TimedAutomaton (A, b, TA): trait
  assumes Automaton(A)
  includes Automaton(TA), Bounds, FiniteMap(Bounds[A], Tasks[A], Bounds)
  States[TA] tuple of basic: States[A], now: Time, bounds: Bounds[A]
  introduces
    b       : Tasks[A]          → Bounds
    nu      : Time              → Actions[TA]
    addTime : Actions[A], Time → Actions[TA]
  asserts
    Actions[TA] generated by addTime, nu
    ∀ s, s': States[TA], c: Tasks[A], a: Actions[A], t: Time
      defined(s.bounds, c);
      isInternal(nu(t));
      isInternal(addTime(a, t)) ⇔ isInternal(a);
      start(s) ⇔
          start(s.basic) ∧ s.now = 0
        ∧ ∀ c (  ( enabled(s.basic, c) ⇒ s.bounds[c] = b(c))
                ∧ (¬enabled(s.basic, c) ⇒ ¬(s.bounds[c]).bounded));
      enabled(s, nu(t)) ⇔ s.now ≤ t ∧ ∀ c (t ∈ s.bounds[c]);
      effect(s, nu(t), s') ⇔
          s'.now = t ∧ s'.basic = s.basic ∧ s'.bounds = s.bounds;
```

Figure 12. LSL definition of timed I/O automata (part 1)

```
      enabled(s, addTime(a, t)) ⇔
          s.now = t ∧ enabled(s.basic, a) ∧ t ∈ s.bounds[task(a)];
      effect(s, addTime(a, t), s') ⇔
          s'.now = t ∧ effect(s.basic, a, s'.basic)
       ∧ ∀ c (  (enabled(s'.basic, c) ∧ enabled(s.basic, c) ∧ task(a) ≠ c
                      ⇒ s'.bounds[c] = s.bounds[c])
               ∧ (enabled(s'.basic, c) ∧ task(a) = c ⇒ s'.bounds[c] = b(c) + t)
               ∧ (enabled(s'.basic, c) ∧ ¬enabled(s.basic, c)
                      ⇒ s'.bounds[c] = b(c) + t)
               ∧ (¬enabled(s'.basic, c) ⇒ ¬(s'.bounds[c]).bounded));
      trace(addTime(a, t)) = trace(a);
      common(addTime(a, t)) = common(a);
      inv(s) ⇔
         ∀ c (  s.now ∈ s.bounds[c]
              ∧ (¬enabled(s.basic, c) ⇒ ¬(s.bounds[c]).bounded)
              ∧ (enabled(s.basic, c) ⇒ (s.bounds[c]).last ≤ (s.now + b(c).last))
              ∧ (s.bounds[c]).first ≤ (s.now + b(c).first)
              ∧ (¬b(c).bounded ⇒ ¬(s.bounds[c]).bounded)
              ∧ inv(s.basic) )
 implies
   Invariants(TA, inv)
   ∀ n: N, c: Tasks[A] (0 ≤ (n * b(c).last))
   ∀ s, s': States[TA], a: Actions[TA], c: Tasks[A]
      isStep(s, a, s') ∧ inv(s) ∧ enabled(s.basic, c)
        ⇒ (s.bounds[c]).last ≤ (s'.bounds[c]).last
```

Figure 13. LSL definition of timed I/O automata (part 2)

```
SimulationMap (A1, A2, f): trait
  assumes Automaton(A1), Automaton(A2), NowExists(A1), NowExists(A2)
  introduces f: States[A1], States[A2] → Bool
  asserts
    ∀ s, s': States[A1], u: States[A2], a: Actions[A1], alpha: StepSeq[A2]
      start(s) ⇒ ∃ u (start(u) ∧ f(s, u));
      f(s, u) ⇒ u.now = s.now;
      f(s, u) ∧ inv(s) ∧ inv(u) ∧ isStep(s, a, s') ⇒
        ∃ alpha (execFrag(alpha) ∧ first(alpha) = u
                   ∧ f(s', last(alpha)) ∧  trace(alpha) = trace(a))
```

Figure 14. LSL definition of simulation mapping