Automatic Verification of the Timing Properties of MMT Automata

by

Ekrem Sezer Söylemez

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1994

Author ....
Department of Electrical Engineering and Computer Science
February 4, 1994

Certified by ....
Nancy A. Lynch
Professor
'is Supervisor

Certified by ....
Stephen J. Garland
Principal Research Scientist
hesis Supervisor

Accepted by ....
Leonard A. Gould
Chairman, Departmental Committee on Graduate Students

Automatic Verification of the Timing Properties of MMT Automata

by

Ekrem Sezer Söylemez

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 1994, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis represents the first use of the Larch tools to verify the timing properties of distributed algorithms, as specified by MMT automata. It shows general methods to formalize and verify automata and timed forward simulations. Additionally, it describes a set of libraries to aid the axiomization of MMT automata and simulation relationships. It includes a sample verified forward simulation relationship. Finally, it details the difficulties that will face future users, and offers some solutions.

Thesis Supervisor: Nancy A. Lynch
Title: Professor

Thesis Supervisor: Stephen J. Garland
Title: Principal Research Scientist

Acknowledgments

I would like to thank all the people who made this possible. First and foremost, this means my thesis advisors, whose expertise in the material was invaluable in doing the research, and whose comments helped improve the quality of the document immeasurably. Stephen Garland in particular helped me throughout the proving process, and was subjected to many pages of unpolished document. I would also like to thank my family, who gave me the motivation I needed to sit down and write the first draft, and to keep writing until it was done. Without them, the thesis might never have been written.

Contents

List of Figures

Chapter 1

Introduction

The purpose of this thesis is to explore the idea of using the Larch facilities to verify hand proofs of distributed algorithms. It represents a first attempt at using these tools in this context, and opens avenues for future exploration in this area. This section informally discusses the mathematical model, previous research, and the problem the thesis addresses.

1.1 Automata

Algorithms can be described using automata. The most fundamental type of automaton is the I/O automaton [9]. This model is used to describe algorithms without any properties that involve time. Once an algorithm has been described using I/O automata, a variety of techniques can be used to prove the algorithm's correctness. In this paper, the most commonly used type of automata is the Merritt, Modugno, Tuttle (MMT) automata [12]. This model provides a simple way to describe basic timing properties of an algorithm. Unlike other models that can be used to describe more complex timing properties, MMT automata are very similar to I/O automata. It is helpful to think of an MMT automaton as an I/O automaton that is built out of an untimed base I/O automaton and some additional information about time.

1.2 Proving Properties of MMT Automata

Since MMT automata are very similar to I/O automata, timing properties of MMT automatoncan be proved in much the same way as other properties of I/O automata. The timing requirements become additional proof obligations.

1.2.1 Operational Proofs

The most obvious way to reason about a program is to discuss its possible executions. Thus, to prove a certain property of the automaton, one must show that every execution has that property. Such a proof is called an operational proof. Unfortunately, it is very difficult to create a rigorous operational proof. This is because there are many executions, and it is hard to be certain that all have been considered. Furthermore, it is at best very hard, and maybe impossible, to create a standard form for operational proofs, as they in general depend on the possible executions.

1.2.2 Invariant Proofs

Fortunately, there is an alternate proof technique, called invariant reasoning, that alleviates some of the difficulties of operational reasoning. In order to write an invariant proof of some property p of an automaton, one finds some logical statement that is true in every reachable of the automaton and that implies p. This logical statement, called the invariant, corresponds to the intuitive reason why the automaton has the property. Finding the invariant can be difficult, but once it is done, most of the rest of the proof follows a general pattern. If one makes sure to prove the invariant in the automaton's initial states, that every action preserves the invariant, and that the invariant implies the property, one can be sure the property is true of any execution of the automaton.

Frequently, it is more convenient to express the goal of the proof as a high level automaton, rather than as a set of properties. This high level automaton is referred to as the specification automaton, or sometimes simply the specification. Essentially, it expresses the task that the more detailed automaton is intended to implement.

Once one has a specification written, proving correctness amounts to showing that the implementation simulates the specification. In other words, anything the implementation does could also have been done by the specification on the same input. This relationship can be stated formally using an abstraction function from the states and actions of the implementation to the states and actions of the specification. More precisely, the function must map each state of the implementation to a set of states of the specification, and each action of the implementation to a sequence of actions of the specification, such that the state before the sequence is an element of the abstraction function of the state before the implementation's action, and the same condition holds for the states after the action. Thus, most invariant proofs of simulations have very similar obligations and structures.

1.3 Difficulties of Invariant Proofs

Although their similar structures makes invariant proofs easier to write and understand, there are still many difficulties with them. The primary difficulty is that it is hard to create a complete proof. There are several reasons for this, most of which involve the quantity of things to be proved.

1.3.1 Proof Obligation Difficulties

One major contributor to this excess of things to be proved is the quantity of things to be proved about each action performed by the automaton. One must show that each action of the implementation preserves the relationship defined by the abstraction function and preserves the invariants, both those associated with the base automaton and with the timing properties. Furthermore, one must show that for any action in the implementation there is a legal sequence of actions in the specification within the abstraction function of the implementation's action. Many of these proof obligations are extremely easy. The consequence of the quantity and relative ease of the proof obligations is that it is often tempting to omit certain portions of the proof that seem "obvious," but which may in fact be somewhat subtle. Worse still, it is sometimes easy to forget a proof obligation entirely, or to make an unjustified, but "obvious" assumption; and it is very difficult, when reading someone else's proof, to be sure that they cover all of their obligations and make no unjustified assumptions.

1.3.2 Uncertainty About Correctness

Thus, it is at worst difficult and at best extremely tedious to construct a completely rigorous hand proof of an I/O automaton's properties. Furthermore, the addition of timing information into MMT automàta dramatically increases the number of proof obligations for the easy steps. It also increases the number of actions by introducing an action that represents the passage of time. Therefore, although invariant proofs offer the possibility of complete rigor, it is difficult to achieve in practice.

Worse still, when one is examining someone else's proof, there is no way to be sure that it is completely correct other than by carefully examining each case to be sure that the proof correctly deals with each obligation. This can be almost as difficult as writing the proof in the first place, and is much more tedious.

1.4 Solution: Machine-Assisted Proofs

In some ways, the fact that most of the proof steps are uninteresting is a blessing in disguise. While it means that much of the proving is boring, it also means that a substantial amount of it can be automated with a machine assistant. The general concept of machine assisted proving is to get a computer to keep track of the proof obligations and to perform the easy steps of the proof. With a machine assistant, the user will only need to do the interesting steps and provide general guidelines. Interesting steps correspond to providing invariants, defining the abstraction function, deciding on important lemmas to prove, applying key facts, etc. Machine-assisted proving seems to have the potential to alleviate the two main difficulties of invariant proofs. It eliminates many of the tedious steps, while guaranteeing complete rigor.

1.4.1 Previous Research

The appeal of automatic proving is obvious. It allows anyone who comes in contact with the proof to be certain of the proof's correctness, to the extent that they can trust the proof assistant, which is usually more than they can trust any single proof. Furthermore, it alleviates some of difficulties of writing a proof by hand.

There are many different approaches to automatic proving. These range from fully automated proving to computer assistance. This section briefly discusses some research done in the area before my thesis.

Perhaps the most popular method is to use the Higher Order Logic (HOL) theorem prover [3] in proving simulations [11]. This prover provides mechanisms for very general systems of logic, which the user defines. Some attempts have been made to perform proofs of timed systems with it [1].

Another approach, taken by [5], is to describe a concurrent algorithm using two component finite state machines described with process algebra. This is significantly different, as it does not involve simulation relations at all.

Of the many approaches, the most similar work to mine is [13]. This is similar because it uses the Larch tools [4, 2] as a computer assistant. Furthermore, it proves a simulation relationship between I/O Automata. The primary difference is that the automata involve no timing properties. The absence of timing properties simplifies proofs enormously. It dramatically reduces the proof obligations in small examples. Furthermore, it greatly reduces the amount of algebraic and logical manipulations necessary. The proofs performed here are among the first to make heavy use of LP's quantifier facilities.

1.5 My Research

1.5.1 The Goal

Ideally, we would like to publish proofs that have been computer verified. Thus, whenever a proof is published, the reader would know that it is error free. Before this goal can be attained, however, several questions must be resolved. First of all, is machine-assisted proving of timing properties even possible, and if so how? Even if it is possible with current provers, it may be to difficult to be practical. Another issue is the readability of computer-verified proofs, which tend to be long and difficult to read. What is the best trade-off between clarity, brevity, and completeness?

1.5.2 Method

In order to answer these questions, I used the Larch Tools to verify an invariant proof of a simulation between MMT automata. This thesis answers questions of how the process works, including the difficulties encountered during the process of verifying a proof.

Tools

The Larch tools were originally developed for rigorous reasoning about sequential programs, and have been used on relatively easy untimed I/O automata simulations [13]. Using the Larch tools to develop a rigorous proof consists of two phases: formalizing the axioms and proof goals with the Larch Shared Language (LSL), and proving these goals using the Larch Prover (LP) computer assistant.

The LSL formalization of the model is much like the hand proof model. Each concept, such as I/O Automaton or forward simulation, is defined by a trait, in terms of lower level traits and primitive concepts. For example, the `TimedAutomaton` trait is built upon the `IOAutomaton` and `Time` traits, among others. The `TimedAutomaton` trait is, in turn, used to define the properties of specific automata. A simulation relationship is then defined between two specific automata. Thus, since the LSL traits for timed automata and simulation relationships have already been created, defining a new MMT automaton is as simple as writing an LSL description of the automaton's properties, and stating that it is an MMT automaton. Once the traits have been defined, the LSL checker is used to produce a set of axioms and proof obligations. For example, if the trait defining a specific automaton claims that a set of invariants hold throughout its operation, then running the LSL checker on that trait produces a proof obligation that states that the invariant holds in every reachable state; the user must verify this obligation with the Larch Prover.

The Larch Prover is the machine assistant for the proof itself. In many ways, it can be thought of as an interpreter for the proof. LP keeps track of known facts, and goals to be proved. The facts that LP remembers include axioms from the LSL traits, lemmas proved to make later sections easier, and context-dependent hypotheses. The job of the user is to provide significant proof steps that the prover cannot figure out itself. At present, there are many steps for which LP needs guidance, but may not ultimately. For example, many algebraic manipulations are difficult in LP and must be guided through specific steps (e.g., use transitivity here, add c to each side there, etc). Soon this difficulty will be overcome as LP's developers are working on a module to handle these proof steps without user guidance. Even in the long run, however, the user will need to provide an overall proof strategy for LP. This includes such things as whether to use induction or contradiction, providing and using lemmas, and dealing with quantifiers.

1.5.3 The Example

The proof I have verified is a relatively simple simulation relationship. The proof shows that an automaton that counts down at a certain speed and outputs a report when it gets to zero must output the report within a certain time range. This was chosen as the example for several reasons. First of all, it is a fairly easy proof, so that I could focus on the difficulties inherent to automatic verification rather than the difficulties specific to the proof. Secondly, the hand proof has been worked through many times in many variants. One appears in [8]. Finally, despite its relative ease, it captures many elements common to most simulation proofs.

Chapter 2

Background

This chapter presents a brief technical introduction to the concepts and methods used in hand proofs. To this end, it discusses a basic model of untimed algorithms and an approach to correctness proofs with it. The chapter then discusses an extension that adds some timing information to the model, and how this extension changes the proof techniques. This chapter presents only the tip of the iceberg. A more detailed understanding of the material would be helpful for continuing research in this area but is not necessary for the remainder of the thesis. For a more complete explanation of the untimed model and proof methods, see [9]. For a more complete explanation of the timed model and proof methods, see [8], [10], or [7].

2.1 Input/Output Automata

An Input/Output (I/O) automaton is a way of describing the asynchronous execution of a process. An automaton A has two components: *actions*(A) and *states*(A). An action is a transition from one state to another. The automaton is allowed to begin in a set of start states: $start(A) \subseteq states(A)$. The set *actions*$(A)$ is divided into *internal* and *external* subsets. The *external* subset is further split into *input* and *output* actions. Finally, the actions are partitioned into equivalence classes by *part*(A). These equivalence classes are referred to as the *classes* of the automaton, and are chosen by the creator of the automaton. The motivation for having classes will be seen in the timed setting. The ways in which an action may affect the state is defined by *steps*(A). This is a set of *(state, action, state)* triples. If there is a triple of the form (s, a, s'), then a is said to be *enabled* in s, and the effects of a in s are said to be s'. I/O automata are required to be *input enabled.* This means that every input action is enabled in every state. We represent an *execution* of the I/O automaton as a sequence

$$s_0, a_0, s_1, a_1, \ldots$$

such that $s_0 \in start(A)$ and $\forall i (s_i, a_i, s_{i+1}) \in steps(A)$. An execution may be either finite or infinite. An *execution fragment* is a similar sequence, but s_0 need not be a start state. The *trace* of an execution fragment is the list of all external actions in that fragment. An *extended step* is a triple (s, β, s') such that there is an execution fragment β that starts in s and ends with s'. A state s is said to be *reachable* if there is some execution that ends in s. Often, we wish to establish a property for all reachable states of an automaton. Such properties are called *invariants.* Since invariants are often used, certain techniques have developed for proving them. The most common is to prove the invariant in every start state, and then prove that every action preserves the invariant. In other words, if the invariant is true before the action, then it will be true after the action. These two steps ensure that the invariant is true in every reachable state s by induction on the length of the execution fragment necessary to reach s. Invariants are also discussed in [7].

2.2 Forward Simulation

Often, proving the correctness of an automaton that implements an algorithm is done by showing a *forward simulation* relationship between that *implementation* automaton and a *specification* automaton that provides a high-level description of what it means for the implementation to be correct. A forward simulation from an implementation automata to a specification automaton informally states that anything the implementation can do, the specification can also. This can be expressed somewhat more formally by saying that the traces of the implementation are a subset of the traces of the specification.

The formal definition of a forward simulation from A to A' relies on an abstraction function. This is a function f from the states of A (the implementation) to sets of states of A' (the specification). The requirements on f are

1. $\forall s \in start(A)\ \exists u \in start(A')$, such that $u \in f(s)$, and

2. If s is a reachable state of A, u is a reachable state of A', $u \in f(s)$, and $(s, a, s') \in steps(A)$, then there is an extended step (u, β, u') such that $u' \in f(s')$, and $trace(a) = trace(\beta)$.

This ensures the condition described informally above by induction on the length of an execution. In other words, the basis is the initial states, and the inductive step is an action of the implementation.

The definition above is expressed in terms of the reachable states of the automata. One can alternatively express the second obligation as

2'. If s is state of A such that $s \in I_A$, u is a state of A', $u \in f(s) \bigcap I_{A'}$, and $(s, a, s') \in steps(A)$, then there is an extended step (u, β, u') such that $u' \in f(s')$, and $trace(a) = trace(\beta)$.

where I_A and $I_{A'}$ are the sets of states that satisfy the invariants of A and A', respectively. This second phrasing is actually stronger than the first, but it is also easier to show. Section 2.5.1 discusses a similar transformation for the timing case in more detail.

2.3 Timed Automata

A timed automaton is an augmented I/O automaton that has additional information to allow discussion of timing properties. The additional information takes the form of a *boundmap* that expresses the time requirements for each class of the automaton. The timed automaton consisting of I/O automaton A and boundmap b is represented by (A, b). The boundmap associates a lower and upper bound with each class (sometimes referred to as $b_l(C)$ and $b_u(C)$, respectively). Informally, the bounds associated with a class signify the interval of time during which, if an action in the class is enabled, an action must occur. For example, consider a class c that has a lower bound of a and an upper bound of b, and that becomes enabled at time t (and remains enabled). In this case, some action in c must happen between the times $t + a$ and $t + b$. Just as I/O automata have executions, traces, and execution fragments, timed automata have *timed executions, timed traces* and *timed execution fragments* that have a time associated with each action. E.g.,

$$s_0, (a_1, t_1), s_1, (a_2, t_2), s_2, \ldots$$

In such a sequence, the subscripts are referred to as the *indices*. From any such sequence α of states, actions, and times, one can produce the *ordinary* sequence (signified $ord(\alpha)$) that contains the same states and actions, but omits the times.

The formal use of the boundmap is defined as follows (taken from [8]).

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

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

2.4 MMT Automata

In order to carry out assertational reasoning on timed systems in the same way as it is done on untimed systems, it is convenient to incorporate the time into the state of the automata. This section presents the MMT automaton, which can be created from a regular timed automaton.

The MMT automaton for a timed automaton (A, b) is denoted by $time(A, b)$ in [8] and [12], where a more thorough explanation of MMT automata can be found. An action of an MMT automaton created from (A, b) must be either an action of A, augmented by the time at which it occurred, or a special $NULL(t)$ action, which advances time to t. The input and output subsets of $time(A, b)$ are divided the same way as in A, and the internal actions subset is $internal(A) \bigcup \{NULL\}$.

The state of A is kept in a *basic* component of the state of $time(A, b)$. Furthermore, the state is augmented with a *now* component that represents the current time. Finally, each class C has values $first(C)$ and $last(C)$ in the state. We use record notation to signify values in the state. For example, in a state s, the *basic* component is represented as $s.basic$. If s is a start state, then for every enabled class C, $s.first(C) = b_l(C)$, and $s.last(C) = b_u(C)$. For the classes that are not enabled in s, $s.first(C) = 0$, and $s.last(C) = \infty$, as a default. The values of $first$ and $last$ outside of the start states, as well as the definition of a step $(s, (\pi, t), s')$, are defined as follows (taken from [8]).

1. if $\pi \in actions(A)$ then:
   (a) $s.now = s'.now$
   (b) $(s.basic, \pi, s'.basic) \in steps(A)$
   (c) i. If $\pi \in C$, then $s'.first(C) \leq t$.
      ii. If $s'.basic \in enabled(A, C)$, $\pi \notin C$, and $s.basic \in enabled(A, C)$, then $s'.first(C) = s.first(C)$ and $s'.last(C) = s.last(C)$.
      iii. If $s'.basic \in enabled$ and either $\pi \in C$ or $s.basic \in disabled(A, C)$, then $s'.first(C) = t + b_l(C)$ and $s'.last(C) = t + b_u(C)$.
      iv. If $s'.basic \in disabled(A, C)$, then $s'.first(C) = 0$ and $s'.last(C) = \infty$.

2. if $\pi = NULL$, then
   (a) $s.now \leq t = s'.now$.

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

(c) $\forall C \in part(A) t \leq s.last(C)$.

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

2.4.1 Invariants

The following invariants, taken from [8], hold for all reachable states of all MMT automata. They are easy to check, but are not proved here.

For every class C and any reachable state s of an MMT automaton $time(A, b)$:

1. $s.last(C) \geq s.now$
2. if $s.basic \in enabled(A, C)$, then $s.last(C) \leq s.now + b_u(C)$
3. if $s.basic \in disabled(A, C)$, then both $first(C) = 0$ and $last(C) = \infty$

2.5 Timed Forward Simulation

This is simply an extension of the normal forward simulation which ensures the timed correctness of the simulation. In other words, we want to guarantee that any time the implementation takes an action, the specification could have taken an equivalent action. It is also called a strong possibilities mapping in [8]. The formal definition listed below comes directly from [8].

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

1. For every start state s of $time(A, b)$ there is a start state u of $time(A', b')$ such that $u \in f(s)$.

2. If s is a reachable state of $time(A, b)$, $u \in f(s)$ is a reachable state of $time(A', b')$ and $(s, (\pi, t), s')$ is a step of $time(A, b)$, then there is an extended step (u, β, u') of $time(A', b')$ such $u' \in f(s')$ and the timed traces of π and β are equal.

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

2.5.1 Invariant Definition

The definition above of timed forward simulations is expressed in terms of reachable states. In this context, it is more convenient to use an equivalent definition that is instead expressed in terms of invariants of the system. This is because it is easier to express and verify invariants than reachability with the Larch tools. Consequently, we will use the following definition of timed forward mapping, taken loosely from [7], where it is referred to as "weak forward simulation."

Let (A, b) and (A', b') be timed automata with the same set Π of external actions. Let I_A and $I_{A'}$ be invariants of (A, b) and (A', b'), respectively. Let f be a mapping from the states of $time(A, b)$ to sets of states $time(A', b')$. Then f is a *strong possibilities mapping* from $time(A, b)$ to $time(A', b')$ provided that the following conditions hold:

1. For every start state s of $time(A, b)$ there is a start state u of $time(A', b')$ such that $u \in f(s)$.

2. If s is a state of $time(A, b)$ such that $s \in I_A$, $u \in f(s) \bigcap I_{A'}$ is a state of $time(A', b')$ and $(s, (\pi, t), s')$ is a step of $time(A, b)$, then there is an extended step (u, β, u') of $time(A', b')$ such $u' \in f(s')$ and the timed traces of π and β are equal.

3. If s and u are states of $time(A, b)$ and $time(A', b')$, respectively such that $s \in I_A$ and $u \in f(s) \bigcap I_{A'}$, then $u.now = s.now$.

In order to use this definition, we must first show that if a function satisfies the conditions of the invariant definition, then it will also satisfy the conditions of the reachability definition. This is true because each of the conditions of the invariant definition implies the same condition of the original definition:

1. This condition of the invariant definition clearly implies the same condition of the original definition, as the conditions are identical.

2. If this condition holds for the invariant definition, then something is true when-

ever the states s and u of the automata satisfy the invariants. However, if these states are reachable, then they must satisfy the invariants. Therefore, the same condition of the original definition must hold for s and u also.

3. This condition holds for the same reason as the prior one.

Therefore, the invariant definition is a stronger requirement than the original one, and if a function satisfies the invariant definition, then it must also satisfy the original definition.

Chapter 3

The Larch Tools

The Larch Tools support a method of systematically constructing and verifying proofs. The process consists of two phases. Informally, the first is describing the proof goals. This phase, commonly called formalizing the model, is done in the Larch Shared Language (LSL). The formalization provides the axioms and *proof obligations.* Proof obligations are facts that must be proved to show the correctness of a trait. Once the formalization is done, the proof itself is carried out using the Larch Prover (LP). Note that these phases may be intertwined, especially if one discovers an error in the formalization after having worked within LP for a while.

This chapter is intended to give an introduction to the Larch Tools, and to provide an understanding of the role each plays in constructing and verifying proofs. It provides a basis for the next chapter, which contains an actual proof, as an example of how the Larch Tools work when dealing with timed automata. To that end, this chapter is broken down into sections that

- provide an idea of what each tool is for, and a basic understanding of its use.
- describe the use of the Larch Shared Language in formalization, including the overall approach and some examples.
- describe the purpose and use of the Larch Prover, and explain an example proof.

3.1 Overview and Purpose of Each Tool

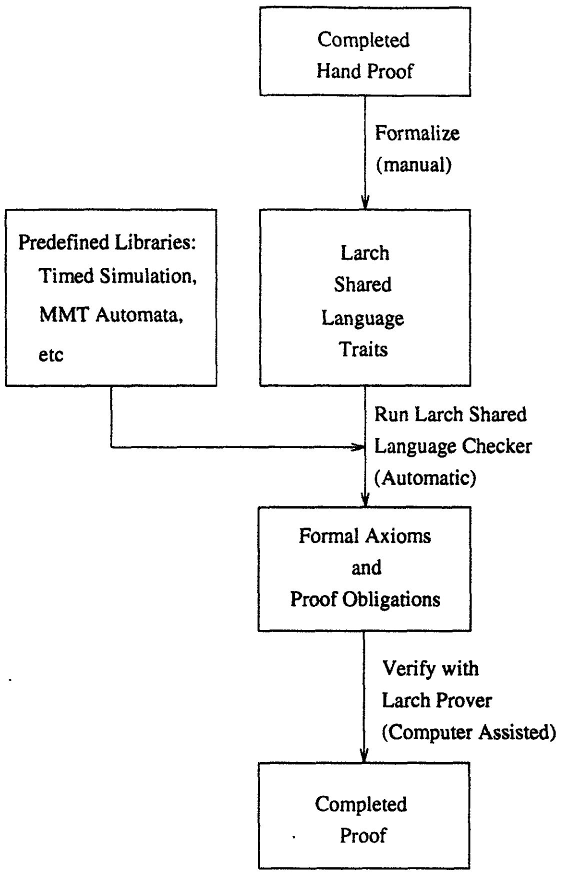
Completed Hand Proof
Formalize (manual)
Predefined Libraries: Timed Simulation, MMT Automata, etc
Larch Shared Language Traits
Run Larch Shared Language Checker (Automatic)
Formal Axioms and Proof Obligations
Verify with Larch Prover (Computer Assisted)
Completed Proof

Figure 3-1: Pictorial Overview of the Formalization Process.

The Larch Tools provide a method for formalizing an informal proof. This process consists of formalizing the specification, and performing the proof itself. Figure 3-1 provides an overview of the process and shows the role of each tool. The Larch Shared Language is used to define the axiomization and proof obligations, and the Larch Prover carries out the proof.

The Larch Shared Language provides a method for describing the automata and proof goals at a high level. This description must be done manually, and consists of creating a number of *traits*. Each of the traits describes a conceptual object. The tool itself takes the form of a compiler: once the traits have been entered, the LSL checker is run on them to produce the axioms and conjectures that need to be proved in a way that the Larch Prover can understand. By contrast, the Larch Prover is interactive.

Once the axioms and proof goals have been defined, LP attempts to perform the proof. However, if it comes to a proof requirement that it cannot perform automatically, it asks the user for input to help it with the proof. The user's input can be remembered and used again in the future to handle the same or similar obligations. Sequences of commands can be thought of as a program, and LP as their interpreter.

3.2 Formalizing the Model with LSL

As described in the previous section, LSL is a language to formally describe the model used and the theorem to be proved at a higher level than the Larch Prover. It produces a sequence of LP commands that assert the axioms and introduce the conjectures be proved. The process itself is referred to as *formalization* and the model used in the hand proof is called the *informal* model, in contrast to the *formal* model described in LSL.

3.2.1 Notation in Larch

In this thesis, Larch code or names are printed in `typewriter` font, in order to differentiate them from regular text. Furthermore, Larch code has often been printed with mathematical symbols where the real Larch code actually contain ascii strings. This section provides a guide for translating the symbolic definitions shown here to their ascii equivalents.

Printed Symbol	Ascii Equivalent
$\neg$	`~`
$\vee$	`\/`
$\wedge$	`/\`
$\rightarrow$	`->`
$\Leftrightarrow$	`<=>`
$\Rightarrow$	`=>`
$\cup$	`\U`

$\langle$	`\<`
$\rangle$	`\>`
$\forall$	`\A`
$\exists$	`\E`
$\leq$	`<=`
$\geq$	`>=`
$\in$	`\in`
$\ominus$	`\ominus`

There is one exception. The first $\forall$ in the `asserts` section uses the string `\forall` rather than `\A`.

3.2.2 Generic Background for LSL

This section presents a conceptual overview of the Larch Shared Language, without going into details related to automata. The goal is to give the reader a sufficient understanding of the issues to write most specifications, and to understand more advanced works on LSL if the information here is insufficient for the reader's needs. For one such treatment see [4]. An LSL formalization consists of a group of traits. Each trait represents a conceptual object. For example, a trait might describe the integers, or associativity. Each trait is allowed to be parameterized. Thus, you could say that `i` is an integer or that `+` is associative. In practice, each trait is kept in its own file. This allows users to easily find the traits they need. Each trait defines two different kinds of things: *sorts*, and *operators*. Sorts are basically types, in the programming sense. Operators are *functions* on zero or more arguments. (An operator with zero arguments is a constant).

When a trait has been completed, it can be tested for errors and compiled into an LP-compatible format using the LSL checker. This will produce proof obligations for the trait. The remainder of this section will describe traits at a high level, and discuss how they are mapped into LP.

Defining Sorts

One way to define sorts is while defining an operator. If the name of a previously undefined sort is specified in the signature of an operator (see the section on Defining Operators), then that name is taken to be the name of a new sort.

Another method of defining a sort within a trait is explicitly. For example, the trait that represents the bounds on a single class contains the following line:

```
Bounds tuple of first:Time, last: Time
```

This line introduces the sort `Bounds`.

The final way to define a sort is by referencing another trait. If, for example, you needed to use time in a trait, then you could simply refer to the time trait. Later in this section is a more detailed discussion of interrelating traits.

Defining Operators

The `introduces` section defines a list of operators (and, by extension, constants, since they are simply operators without arguments) for use in the trait. For example, the *introduces* section for the trait `Time` includes the following lines.

```
introduces
0: -> T
__ + __, __ - __ : T, T -> T
```

To understand these lines one must know that each one is divided into two sections by the colon. The first section describes the name and format of the operator (+ or - are names). Operators are allowed to be prefix, postfix, or mixfix. The __ symbol represents the location of arguments in this section. The second half of each line tells the *signature* of the operator, i.e., the sorts of the arguments and the result. Note that if the second half says there are arguments, but the first half does not show any locations for them, then they follow the name of the operator in function notation. The default form of named operators is `OPERATOR( ARG1, ARG2, ... , ARGN)`. Also, the line `0:-> T` says that `0` is a constant of sort T. In the middle line the + and - operators go in between two items of sort T, and the result of adding or subtracting two items of sort T is another item of sort T.

Facts

LSL provides two means of defining facts about a trait. The first is for axioms, which are true by definition. The second is for facts that can be proved from the other facts in the trait. When the LSL checker is run on a trait, the axioms become facts, and the provable facts become proof obligations. However, if the trait is referenced from another trait (how to do this is discussed in the next section), then both become facts.

The `asserts` section found in each trait states the axioms of the trait. There are two parts of the `asserts` section. The first tells how things are generated. This is approximately an enumeration of the possible values something can take. For example, in every automaton, there is a `generated by` statement such as

```
Actions generated by act1, act2, act3
```

where `act1`, `act2`, and `act3` would be replaced by a complete list of the actions of the automaton.

The rest of the `asserts` section describes general facts about the trait.

This section is fairly easy to read without any particular explanation. However, for an explanation of the details of this section, see [4]. The information in [4] is slightly out of date. Statements in the `asserts` section are now allowed to have existential and universal quantifiers. Also, the == operator has been eliminated. Ambiguities in the parsing of = should be resolved with parentheses instead. The major useful but non-intuitive fact about this section is that when you put in some fact such as `a = b`, the LSL checker takes this as a strong hint that `a` is defined as `b`. Thus, the rewrite rule that LP creates will probably be ordered as `a -> b`. Rewrite rules are discussed in more detail in section 3.3.

The `implies` section of a trait lists some useful, supposedly provable facts about that trait. This section differs from the `asserts` section only in that when the LSL checker is run on the trait, the `implies` section become proof obligations rather than facts. Thus, almost all the remarks in the previous paragraph hold true for this section as well.

Relating to Other Traits

Most often, a trait does not exist in a vacuum, but is instead defined in terms of other traits. There are three methods of doing this. The first two methods are the **`assumes`** and **`includes`** sections of a trait. The difference between these two is somewhat subtle. If another trait is included, then the trait included is a part of the definition of the new trait. However, if another trait is assumed, then the definition of the new trait does not make sense unless the assumed trait is included elsewhere. The best way to see this distinction is to see it in use. One example is the `Executions` trait, where the distinction is discussed in context (see page 32.) The third method is as a statement in the **`implies`** section. This requires the user to prove the properties in the trait. For example, when one wishes to prove a simulation relationship between two automata `A` and `B`, one usually uses a line such as `Simulation(A, B)` in the **`implies`** section of the complete trait for the system.

3.2.3 The Layered Approach to LSL Specification

The informal description of an MMT automaton is based on several other concepts, such as I/O automata, and time. Similarly, a specific automaton is described as an MMT automaton that has certain additional properties. In other words, the general approach is to build up the description in layers, so that each layer is defined in terms of the lower levels. Thus, understanding a concept necessitates understanding the lower level concepts. This same approach is used in LSL. In LSL each concept is defined in its own trait. Thus, the specific simulation relationship to be proved is expressed as a trait that is defined in terms of lower level traits such as the specific automata and simulation relationships in general. Figure 3-2 shows a module dependency diagram for the proof I carried out. One of the major benefits of this approach is the reusability of traits. This means that, in effect, there is a library of predefined traits which now includes things like MMT automata and timed simulation relationships. Because of this library, defining a new simulation will really only require describing the details of the specific automata and the relationship between them. The traits that need to be redefined when doing this are shown in dotted boxes in figure 3-2. This reusability cuts down on a lot of work, both in formalizing the simulation, and in the proving process, as the traits in the library have already been tested and honed. Furthermore, the use of predefined libraries increases the user's confidence in results proved, as the axioms will be less likely to have inconsistencies.

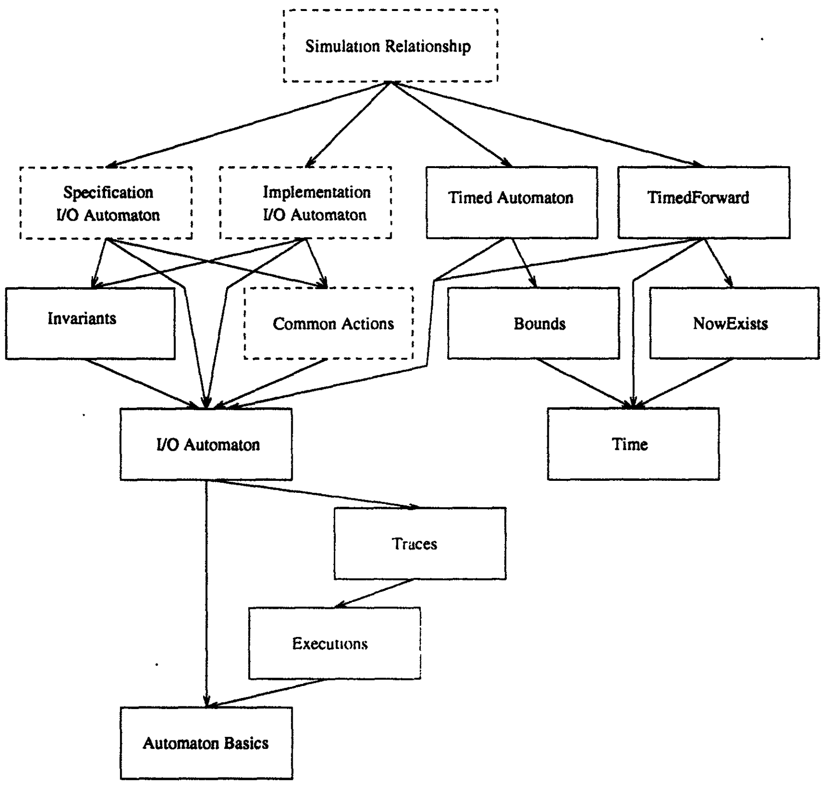

Figure 3-2: Module Dependency Diagram of a Timed Forward Simulation Relationship.

The rest of this section describes in detail each of the traits used in defining a specific automaton. In other words, this section describes the reusable library for automata. Traits for the automata used in the example can be found in the example chapter. This should be useful both as a reference and as an example of what real traits look like.

```
AutomatonBasics (A): trait
  introduces
    start      : A$States                      → Bool
    enabled    : A$States, A$Actions           → Bool
    effect     : A$States, A$Actions, A$States → Bool
    isExternal : A$Actions                     → Bool
    isInternal : A$Actions                     → Bool
    isInput    : A$Actions                     → Bool
    isOutput   : A$Actions                     → Bool
  asserts ∀ a: A$Actions, s: A$States
    isExternal(a) ⇔ isInput(a) ∨ isOutput(a);
    ¬(isInput(a) ∧ isOutput(a));
    isInternal(a) ⇔ ¬isExternal(a);
    isInput(a) ⇒ enabled(s, a)
```

Figure 3-3: LSL trait defining basics of input-enabled I/O automata

3.2.4 I/O Automata

This section shows the traits that specify an I/O Automaton. The first four of these collectively describe input—enabled I/O Automata in general. The fifth discusses the invariants of automata, and provides some proof obligations that are required of a trait for it to actually describe an I/O automaton.

The `AutomatonBasics` trait provides the basic vocabulary of automata, as well as some basic concepts that arise directly from the definitions. Note that since it is defined in terms of nothing else, but used in other definitions, it does not `include` or `assume` any other traits. Just as in the informal definition, the start states are a subset of the states. In the formal model this subset is defined by the boolean start operator. Thus,

$$\forall s \in \mathit{states}, s \in \mathit{start} \Leftrightarrow \texttt{start}(s) = \text{TRUE}$$

The constructs `isExternal`, `isInternal`, `isInput`, and `isOutput` express the notions of the external, internal, input, and output subsets of actions in the same way. Furthermore, the trait says that external actions are partitioned into input and output actions, and that actions in general are either input or output. Furthermore,

```
Executions (A): trait
  assumes AutomatonBasics(A)
  introduces
    isStep      : A$States, A$Actions, A$States  → Bool
    null        : A$States                       → A$StepSeq
    __⟨__,__⟩   : A$StepSeq, A$Actions, A$States → A$StepSeq
    execFrag    : A$StepSeq                      → Bool
    first, last : A$StepSeq                      → A$States
  asserts
    ∀ s, s': A$States, a, a': A$Actions, ss: A$StepSeq
      isStep(s, a, s') ⇔ enabled(s, a) ∧ effect(s, a, s');
      execFrag(null(s));
      execFrag(null(s)⟨a,s'⟩) ⇔ isStep(s, a, s');
      execFrag((ss⟨a,s⟩)⟨a',s'⟩) ⇔
        execFrag(ss⟨a,s⟩) ∧ isStep(s, a', s');
      first(null(s)) = s;
      last(null(s)) = s;
      first(ss⟨a,s⟩) = first(ss);
      last(ss⟨a,s⟩) = s;
```

Figure 3-4: LSL trait defining executions I/O automata

it expresses the idea that input actions are always enabled. The effects of an action are expressed as a subset of all possible pre/post pairs of states. Thus, the effects of an action of the automaton are not required to be deterministic, as an action can take the automaton from one pre-state to any of several post-states. This property is also true of the informal model, but is not often used in practice.

The `Executions` trait formalizes the informal notion of executions, and execution fragments. Within the context of this trait, `null(s)` refers to the empty sequence of actions beginning at the state s, rather than the passage of time action discussed later in the timed setting. The `Executions` trait lets one define a formal analog to the sequence $s_0, a_0, s_1, a_1, ...a_{n-1}, s_n$. To express this sequence in LSL, one would use

```
(...((null(s0)<a0, s1>)<a1,s2>)...<an-1,sn>
```

Note that this trait **`assumes`** `AutomatonBasics`, rather than including it. This is because the notion of executions is intended to make sense on top of the definition of an existing automaton, rather than on an automaton defined by this trait.

```
Traces (A): trait
  assumes AutomatonBasics(A), Executions(A)
  introduces
    common    : A$Actions               → CommonActions
    empty     :                         → Traces
    __ ^ __   : Traces, CommonActions → Traces
    trace     : A$Actions               → Traces
    trace     : A$StepSeq               → Traces
  asserts
    Traces generated by empty, ^
    ∀ s: A$States, a: A$Actions, ss: A$StepSeq
      trace(null(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)
```

Figure 3-5: LSL trait defining traces for I/O automata

The trait **Traces** formalizes the notion of the behavior of an execution. In this trait, the trace of an execution fragment is the sequence of all external actions in that fragment. It shows how to construct the trace of any finite sequence of actions.

We introduce a function **common** to map the actions of an automaton into a new sort **CommonActions**. This is necessary because LSL requires sorts to represent disjoint, non empty sets. This allows the traces of an automaton **A1** (which have actions of sort **A1$Actions**) to be compared with the traces of an automaton **A2** (which have actions of sort **A2$Actions**), via the common actions. The definition of **CommonActions** must be provided when specifying the particular automaton. This is usually a fairly mechanical process.

The **IOAutomaton** trait gathers these three traits and introduces the idea of equivalence classes of actions. This is the complete description of an input-enabled I/O automaton. It includes all the axioms. The **AutomatonBasics**, **Executions**, and **Traces** traits define most of these axioms, so the **IOAutomaton** trait merely needs to add a few finishing touches. These are the definition of **classes**, and the addition of a function **inv** that maps states to booleans, to express the invariant of the automaton.

Even though the **IOAutomaton** trait defines all the axioms about input enabled I/O

```
IOAutomaton (A): trait
  includes AutomatonBasics(A), Executions(A), Traces(A)
  introduces
    class   : A$Actions            → A$Classes
    enabled : A$States, A$Classes  → Bool
    inv     : A$States             → Bool
  asserts ∀ s: A$States, a: A$Actions, c:A$Classes
    enabled(s, c) ⇔ ∃ a (enabled(s, a) ∧ class(a) = c);
```

Figure 3-6: LSL trait bringing together I/O automata definition

```
Invariants (A): trait
  assumes IOAutomaton(A)
  asserts ∀ s, s': A$States, a: A$Actions
    start(s) ⇒ inv(s);
    inv(s) ∧ enabled(s, a) ∧ effect(s, a, s') ⇒ inv(s');
    enabled(s, a) ⇒ ∃ s' effect(s, a, s')
```

Figure 3-7: LSL trait defining requirements for I/O automata

automata, it alone is not sufficient for defining an I/O automaton. This is because I/O automata must satisfy certain requirements in order to be correct. The `Invariants` trait expresses these requirements. The first two statements in this trait deal with proving the correctness of the invariant. The final statement requires that an action may only be `enabled` in a certain state if there is a post-state for that action in that state.

Thus, defining an I/O automaton requires two interactions with other traits. First, one must `include` the `IOAutomaton` trait. and `imply` the `Invariants` trait. For a complete description of defining an automaton, see section 3.2.7.

3.2.5 MMT Automaton

Before showing how an MMT automaton is defined, we must first show three auxiliary traits. These are `Time`, `Bounds`, and `NowExists`.

The `Time` trait provides a basic definition of time. It has probably received more attention than most of the other traits in the formalization. This is because earlier

```
Time (T): trait
  includes TotalOrder(T), Natural(- for ⊖), AC(+, T)
  introduces
    0: → T
    __ + __, __ - __: T, T → T
    __ * __: N, T → T
  asserts ∀ t, t1, t2: T, n: N
    0 ≤ t;
    0 + t1 = t1;
    0 * t = 0;
    succ(n)*t = (n*t) + t;
    t ≤ (t + t1);
    (t + t1) ∸ t = t1;
    (t + t1) < (t + t2) ⇔ t1 < t2;
    (t + t1) ≤ (t + t2) ⇔ t1 ≤ t2;
    (t + t1) = (t + t2) ⇔ t1 = t2;
    0 < n ⇒ ¬((n*t) < t);
    n > 1 ⇒ ((n-1)*t) + t = n*t
```

Figure 3-8: LSL trait defining the properties of time

versions attempted to allow time to be infinite, in order to allow classes without upper bounds. This led to some inconsistencies in earlier versions of the axioms, as the requirement $t \neq \infty$ was omitted accidentally. When this requirement was inserted in the proper places, it became difficult to perform algebra involving time. Thus, to solve this problem, infinity was moved to the Bounds trait, and time is required to be finite.

One of the notable things about the Time trait is its heavy reliance on the library of traits. It should be noted that AC (associative/commutative) has special support in LP. Thus using them is more powerful than an equivalent formalization of the same properties.

The Bounds trait represents the concept of the period of time during which an action from a class must occur. The trait also includes a definition of what it means for a class's occurrence to be unbounded in time, and what it means to add a time to the bounds. Both of these are useful in the TimedAutomaton trait. As discussed in the previous paragraph, time must be finite, but the concept of an infinite upper bound is allowed by this trait.

```
Bounds: trait
  includes Time(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
    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 ⇔
      (b1.bounded ∧ b2.bounded ∧ b2.first ≤ b1.first ∧ b1.last ≤ b2.last)
         ∨ ¬b2.bounded;
    t ∈ b ⇔ (b.first ≤ t ∧ t ≤ b.last) ∨ ¬b.bounded
```

Figure 3-9: LSL trait defining a single class of the boundmap

```
NowExists (A): trait
  introduces __.now : A$States → Time
```

Figure 3-10: LSL trait for checking if the automaton has a time in its state

The `NowExists` trait is provided as support for the `TimedForward` trait, which formalizes the notion of timed simulation relationships described in Chapter 2. In the `TimedForward` trait (figure 3-12, described in more detail later), `NowExists` is used to check if the I/O automata involved in the simulation are actually MMT automata which always have a *now* component in their state. Thus, the `TimedForward` trait **`assumes`** `NowExists` for each of the two automata involved in order to make sure that they are MMT automata. Although the requirements of MMT automata are, in reality, more stringent than simply having a *now* component, it is not necessary for an I/O automaton to meet these other requirements for the `TimedForward` trait to make sense. Thus, using `NowExists` leaves open the possibility of using another type of automaton with timing properties in conjunction with the `TimedForward` trait.

The `TimedAutomaton` trait is the largest and most complex of the traits used in our formalization. It describes how an MMT automaton may be created from an untimed I/O automaton and a *boundmap*, which relates a bounds to every class. It follows the definition of MMT automata outlined in Chapter 2 and [8]. Because of this, it is much like the definition of most other I/O automata, only more complex. The next several paragraphs discuss the `TimedAutomaton` trait's various noteworthy and difficult aspects.

As with the informal model, the actions of an MMT automaton are the *null* action, along with the actions of the untimed I/O automaton, each associated with the time of execution. Much of the `TimedAutomaton` trait is devoted to explicitly stating the various facts associated with this. Some of the facts included are

- the boundmap must define bounds for every class,
- the *null* action is internal,
- the effect of the $null(t)$ action is to advance time to t,
- how far forward the $null(t)$ action can advance time,
- an action from the I/O automaton is enabled in the timed automaton if it is enabled in the same state of the untimed automaton,

```
TimedAutomaton (A, b, TA): trait
  assumes IOAutomaton(A)
  includes
     IOAutomaton(TA), Bounds,
     FiniteMap(A$Bounds, A$Classes, Bounds, __[__] for apply)
  TA$States tuple of basic: A$States, now: Time, bounds: A$Bounds
  introduces
    b       : A$Classes       → Bounds
    null    : Time            → TA$Actions
    addTime : A$Actions, Time → TA$Actions
  asserts
    TA$Actions generated by null, addTime
    ∀ s, s': TA$States, c: A$Classes, a: A$Actions, t: Time
      defined(s.bounds, c);
      isInternal(null(t));
      isInternal(addTime(a, t)) ⇔ isInternal(a);
      isInput(addTime(a, t)) ⇔ isInput(a);
      start(s) ⇔
          start(s.basic) ∧ s.now = 0
       ∧ ∀ c (   ( enabled(s.basic, c) ⇒ s.bounds[c] = b(c))
               ∧ (¬enabled(s.basic, c) ⇒ ¬(s.bounds[c]).bounded) );
      enabled(s, null(t)) ⇔
          s.now ≤ t ∧ ∀ c (t ∈ s.bounds[c]);
      effect(s, null(t), s') ⇔
          s'.now = t ∧ s'.basic = s.basic ∧ s'.bounds = s.bounds;
      enabled(s, addTime(a, t)) ⇔
          s.now = t ∧ enabled(s.basic, a)
       ∧ (¬isInput(a) ⇒ t ∈ s.bounds[class(a)]);
      effect(s, addTime(a, t), s') ⇔
          s'.now = t ∧ effect(s.basic, a, s'.basic)
       ∧ ∀ c (   (enabled(s'.basic, c) ∧ enabled(s.basic, c)
                     ∧ class(a) ¬= c ⇒ s'.bounds[c] = s.bounds[c])
               ∧ (enabled(s'.basic, c) ∧ class(a) = c
                     ⇒ s'.bounds[c] = b(c) + t)
               ∧ (enabled(s'.basic, c) ∧ ¬enabled(s.basic, c)
                     ⇒ s'.bounds[c] = b(c) + t)
               ∧ (¬enabled(s'.basic, c)
                     ⇒ ¬(s'.bounds[c]).bounded));
      trace(addTime(a, t)) = trace(a);
      common(addTime(a, t)) = common(a);
      inv(s) ⇔
          ∀ c (   s.now ∈ s.bounds[c]
               ∧ (¬enabled(s.basic, c) ⇒ ¬(s.bounds[c]).bounded))
 implies Invariants(TA)
   ∀ a: A$Actions, t: Time
      isOutput(addTime(a, t)) ⇔ isOutput(a:A$Actions)
```

Figure 3-11: LSL trait describing the creation of a Timed Automaton

- the effects of an action from the I/O automaton on the times at which other actions may occur,
- all other actions are internal or external just as they were in the I/O automaton,
- the trace of the timed automaton is the same as the trace of the I/O automaton, and
- the common actions of the timed automaton are the same as the common actions of the untimed automaton.

It is quite easy to recognize which line in the trait corresponds to each of these facts.

The boundmap is defined with the `FiniteMap` trait, which is taken from the LSL library of standard traits. It is used to introduce a function `b` from `classes` to time bounds. Note that the use of the `FiniteMap` trait restricts our definition of MMT automata to those which contain only a finite number of classes, as the `FiniteMap` trait only allows a finite number of entries. The boundmap associates a time bound `b(c)` with each class `c` in the untimed automaton. The values of each of the bounds must be defined with the trait defining the specifics of the simulation relationship.

Invariants that hold for all MMT automata are listed at the end of the `asserts` section. As with the definition of any I/O automaton, invariants must be set up as a function `inv` that maps states of the automaton to boolean values. This allows the `Invariants` trait to be used to verify the invariants. They will be useful for future users of the library, as they allow us to use these properties as needed from LP. The fact that they are preserved has been verified using LP. In fact, the proof of one of them is discussed in the LP section of this chapter as an example.

3.2.6 Simulations

This section describes the `TimedForward` trait used to define the notion of a timed simulation between two automata `A1` and `A2`. The requirements of a formally defined timed simulation are exactly the same as the requirements put forth in section 2.5.1.

```
TimedForward (A1, A2, f): trait
  assumes IOAutomaton(A1), IOAutomaton(A2), NowExists(A1), NowExists(A2)
  introduces f: A1$States, A2$States → Bool
  asserts ∀ s, s': A1$States, u: A2$States, a: A1$Actions,
                  alpha: A2$StepSeq
    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 3-12: LSL trait defining basics of input-enabled I/O automata

Note that the **assumes** line requires that the two automata be I/O automata that have a **now** component. We take this approach because we don't explicitly have access to the untimed versions of the automata and the boundmaps, which would be necessary in order to have an **assumes** section like

```
assumes
  TimedAutomaton(UntimedA1, Boundmap1, A1),
  TimedAutomaton(UntimedA2, Boundmap2, A2), ...
```

While the approach we take does not require the automata in the relationship to be MMT automata, it does guarantee everything necessary for the `TimedForward` trait to make sense. This allows the trait to be somewhat more general: any form of timed automata that contains a **now** component can use the trait.

3.2.7 The Specific Traits

Now that the library has been displayed, the natural question is, *what is left for the user to do?* The user needs to do the following to formalize a timed simulation relationship:

- Write formal descriptions of two untimed automata that **include** the `IOAutomaton` trait, and **imply** the `Invariant` trait.

- Write a `CommonActions` trait that has all the common actions of the two automata.
- Write a `simulation` trait that `includes` the traits defining the specific automata, defines the simulation relation f, and implies the `TimedForward` trait.

An example of this can be seen in the next chapter.

3.2.8 Using the LSL Checker

The LSL checker is run on a specific trait. It checks the syntax and static semantics of the trait and all its subtraits. Furthermore, if the *-lp* option is specified, the checker generates axioms and proof obligations for the trait. These are put in two files, as follows.

- *trait_axioms.lp* This file gives an LP description of all the axioms of the trait. Generally, it is run once, and a freeze file is produced. (See next section for a description of what this means.)
- *trait_checks.lp* This is a list of proof obligations for the trait. These generally result from the `assumes` and `implies` statements in that trait. This file generally is taken and edited extensively to prove the theorems.

One notable fact about the LSL Checker is that it only produces checks for the trait itself, and not for the subtraits. Thus, in order to prove a simulation relationship, you must verify the proof obligations in the `Simulation` trait and in the traits defining the two automata, which have their own proof obligations due to the implied `Invariants` trait. Only after checking all three traits can you be sure that the simulation actually holds, as otherwise the invariants of the automata may be left unproved. Usually, proving these invariants is quite easy, as they will normally follow quickly by an inductive argument.

3.3 Using LP

Once the formalization of the model has been carried out using LSL, the checks must be carried out using LP. This section provides a basic understanding of how to use LP to accomplish these goals. It begins with an introduction to some of the most important concepts of LP. Next, it talks briefly about a few important commands. Finally, it explains a relatively simple proof in detail. The proof comes from the verification of an MMT invariant. For a more technical and detailed coverage of the material here, see [2], or use LP's `help` facility.

3.3.1 LP Concepts

In many ways, LP can be thought of as an interactive interpreter, much like the old BASIC interpreters, or a LISP/SCHEME interpreter. In a typical session, LP loads the axioms and proof obligations, and verifies a proof script. If a problem occurs, it stops, and has the user continue the proof manually. LP is often called a computer proof assistant.

LP helps with the proving process in two major ways. It keeps track of the state of the proof, and it automatically carries out the easy proof steps. For the user, LP's ability to keep track of the state of the proof is crucial to knowing that a proof is correct. This ability ensures that no case can be glossed over. LP also remembers all the facts one has available at any given time. In LP terminology, the theorem you are trying to prove (which may be a proof obligation from LSL) is called a *conjecture.* To prove the conjecture, you need to prove various *goals* which may be divided into several *subgoals.* For proving these subgoals, you may have certain *hypotheses* which you have assumed during the course of your proof. These could be, for example, facts about the case you are in. The other capability that LP provides as a computer assistant is a completely automatic handling of the easy proof steps. This means that LP can automatically perform some reasoning. At other times, LP merely needs a pointer in the right direction, such as "prove this by induction." Sometimes, however, LP needs detailed guidance about how to perform a section of a proof. The section on LP commands gives a description of how to guide LP in these two ways.

For the most part, the reasoning that LP handles automatically is done by *normalization.* Under ideal conditions, every expression has a unique normal form. When LP encounters an expression, it attempts to rewrite it to a normal form by applying *rewrite rules.* Most facts in LP take the form of rewrite rules. For example, if a is defined to be $b+c$, then there probably is a rewrite rule of the form $a \rightarrow b+c$. (This is read "a rewrites to $b+c$.") If the rewriting process is going well, then conjectures will be normalized down to identities, so LP will be able to handle the proof automatically. However, the following is a list of some of the less pleasant properties of normalization.

- The rewriting process is non-deterministic. Consider, for example the system with the rules $a \rightarrow b$ and $a \rightarrow c$. Then, a can be rewritten to either b or c, depending on which rule gets applied. This could be a problem, for example, if one were trying to prove $a = b$. LP could rewrite the a to b, reducing this to the identity $b = b$, or it could rewrite the a to c, thus getting $c = b$, which is not an identity. This does not happen much in practice, however, as LP *internormalizes* its rewrite rules. This means that it uses its rewrite rules to normalize each other. In the example, it could use the rule $a \rightarrow b$ to normalize the rule $a \rightarrow c$ to $b \rightarrow c$. Now it would normalize $a = b$ to the identity $c = c$.

- The rewriting sometimes fails to prove things. Sometimes the situation can be helped by using the *critical-pairs* facility, described in the next section.

- Needed facts can be accidentally normalized away if they rewrite to identities.

- The rewriting process is not always guaranteed to terminate.

Because of these problems, LP provides the user with several commands to direct the normalization process. Although they are needed fairly rarely, it is certainly useful to be aware of them. They are discussed in the next section.

3.3.2 Commands

This section is just a brief overview of some of the most useful and often used commands. It is intended as a primer to be read before the tour of the simple proof in the next section. It will cover issues that are disjoint from the ones in the proof tour. To get more detailed help about any command while using LP, simply type `help` `⟨command name⟩`.

Dealing with Files LP has several commands that deal with files. They are

- **freeze** - remember the state of the proof.
- **thaw** - restore a previously frozen state.
- **execute** - perform a list of commands in a file.

Instantiation Frequently during the course of a proof, one needs a specific form of a general fact. For example, one may have the general fact $\forall x\ (f(x) = g(y))$, and need to use this fact for the case of $x = 4$. One LP command that can be used at these times is `instantiate`. In the example above, the precise command would be

```
instantiate x by 4 in theRule
```

where `theRule` is the name of the fact.

Critical Pairs Since normalization is LP's primary way of discovering the equivalence of expressions, most facts are kept in the form of rewrite rules. This has the advantage of being largely automatic, but it still sometimes needs direction to discover certain facts. Critical-Pairs is a command for helping the prover find new facts that result from considering rewrite rules as equations. For example, consider the situation of manually applying transitivity using the following rewrite rules:

```
Trans: a < b ∧ b < c → a < c
Rule1: x1 < x2 → TRUE
Rule2: x2 < x3 → TRUE
```

(Here, `a`, `b`, and `c` are variables and `x1`, `x2`, and `x3` are constants.) It is clear to a human that `x1 < x3` by using transitivity. For LP, however, it is necessary to perform two `critical-pairs` operations. The first:

```
Critical-pairs Trans with Rule1
```

yields the rewrite rules

```
NewRule1: x2 < c → x1 < c
NewRule2: a < x1 → a < x2
```

Now, the command

```
Critical-pairs NewRule1 with Rule2
```

yields the rewrite rule

```
 x1 < x3→true
```

This example is somewhat unusual in that one usually does not need to use the `critical-pairs` command twice to get the new fact.

Conceptually, then, critical pairs is an attempt to combine two rewrite rules in a way that can yield a valuable new fact. It is not necessary to remember the details of how it works, merely that it matches common sections of the rules to yield the new fact.

Directing Normalization Most of the time LP's normalization methods work very well. It serves as a solid basis for showing the equivalence of two expressions. Occasionally, however, there are problems. Sometimes, the difficulties described in section 3.3.1 occur. At other times, it is simply easier to read expressions in their unnormalized form. To deal with these situations, LP provides methods of directing the normalization process.

There are two commands to direct normalization. To prevent a particular rewrite rule from being used during normalization, one makes it `inactive`. To prevent a particular expression from being normalized, one sets its `immunity`. The activity of a rule is somewhat easier to understand, because there are only two modes: active and inactive. If a rewrite rule is active, it is used in normalization. If it is inactive, it is not. If a rewrite rule is inactive, it may still be used explicitly, with the command `rewrite <expression> with <rule>`.

Immunity has three modes: on, off, and ancestor. When an expression has immunity on, then it will not be normalized, except explicitly. When it has immunity off, it will be normalized by any active rewrite rules. When it has ancestor immunity, it will not be normalized by its ancestors. For example, if you have a rewrite rule such as $f(x) \rightarrow 4$, and you execute the command `instantiate x with 2 in rule`, then the expression produced, $f(2) \rightarrow 4$, would be a special case of the original rule. Thus, it would be normalized away if immunity were set off, but it would not be if immunity were set to ancestor.

Deduction Rules In addition to rewrite rules, LP also has *deduction rules.* These provide an operational semantics for logical implication. Deduction rules take the form

```
WHEN hyp YIELD conclusion
```

where `hyp` and `conclusion` are the hypotheses and the conclusion of the implication. To use a deduction rule explicitly, one can *apply* it to a formula or rewrite rule. However, LP makes deduction rules `active` by default, which means that they will be automatically applied to all formulas and rewrite rules that are not `immune`.

When a deduction rule is applied to a formula that matches its first hypothesis, the result is the rest of the deduction rule with the matching terms substituted in. For example, applying the rule

```
    WHEN a < b, b < c YIELD a < c
```

to the formula

```
x1 < x2
```

produces the deduction rule

```
when x2 < c yield x1 < c
```

Appyling this deduction rule to the formula

```
x2 < x3
```

produces the deduction rule

```
when true yield x1 < x3
```

which reduces to the formula

```
x1 < x3
```

Thus LP can sometimes apply the sample deduction rule automatically to derive facts about transitivity, without the user having to compute critical pairs.

LP also allows users to apply deduction rules to prove conjectures. For example, given the deduction rule

```
when x2 ≤ c yield x1 ≤ c
```

and the rewrite rule

```
a ≤ a → true
```

the user can finish the proof of `x1 ≤ x2` by typing

```
apply deduction-rule to conjecture
```

This causes LP to match the conclusion `x1 ≤ c` of the deduction rule to the conjecture `x1 ≤ x2` by letting `c` be `x2`, after which LP substitutes `x2` for `c` in the hypothesis of the deduction rule to get `x2 ≤ x2`, which reduces to `TRUE` and finishes the proof of the conjecture.

Informational Commands This section discusses informational commands, both because they are useful, and because they do not usually appear in completed proofs, as they are not needed in this context. The most useful informational command is `display`. Part of its usefulness comes from the fact that it can be used to display a number of things. For a complete list, see [2] or the LP help facilities.

There are three commonly used types of displays. The most useful is sets of rules. A set of rules can be defined by name, containing-operator, or a union or intersection of sets of rules. One common example is `display *hyp`, which displays all the current hypotheses. This is useful for establishing the context in which one is working. The second is `display proof`, which tells the state of the proof in gory detail. The final is `display symbols`, which provides a complete list of declared sorts, operators, and variables.

Another useful informational command is `show normal-form`. This command shows each of the steps involved in normalizing an expression. This can be useful when you can't seem to get a fact to appear, because it is unexpectedly normalized away.

Yet another often used informational command is `history`. This gives a list of the previous commands. It is usually given an argument, which tells how many previous commands to display. By default, it gives all the commands used to get to the present state. This is probably more commands than you want to know when working with I/O automata.

3.3.3 A Guided Tour of a Simple Proof

This section presents a simple proof from the verification of the `TimedAutomaton` trait. It corresponds to the fact that whenever a class of actions in an automaton is not enabled, then that class has has a lower bound of 0 and an upper bound of ∞.

The proof presented is a proof script. This means that LP's responses to the commands here are not listed. Instead, we supply a commentary that will tell both why the commands in the script were chosen, as well as what their effects were.

The proof script begins with some preliminary commands. Commands similar to these are found at the start of most proof scripts.

```
set script TimedAutomaton
set log TimedAutomaton

Thaw TimedAutomaton

declare variables
  s: TA$States
  s': TA$States
  a: A$Actions
  a: TA$Actions
  t: Time
  ..

set name InvariantsTheorem
```

The first two lines tell LP where to keep a record of the commands it executes and the output it gives, respectively. This is often useful as a reference later, particularly if the script is still being worked on. The third command tells LP to restore a previously saved state. In this case, the state restored contains the axioms necessary for the remainder of the proof. The next command creates a number of variables to be used later. Note that there are two variables with the name **a**. When the user wishes to refer to one of these, it is necessary to specify the sort of the one intended. The final command sets the name to call facts that LP discovers.

Most of these commands were automatically generated by the LSL Checker, when it produced a file to verify this invariant. The only exception to this rule is the command `Thaw TimedAutomaton`, which was originally `execute TimedAutomaton_Axioms`. In order to avoid having to wait for `TimedAutomaton_Axioms` to execute each time I worked on the invariant proof script, I simply executed it once, and saved the results.

```
prove (start(s:TA$States) => inv(s:TA$States))
```

This command was also generated by the LSL Checker. It is one of the two proof requirements the checker generates for this invariant. These requirements come from the implication of the `Invariants` trait by the`TimedAutomaton` trait. Notice that it is one of the two halves of a typical hand proof of an invariant.

Once this `prove` command has been parsed, LP begins attempting to prove it. By default, this means it tries to normalize the conjecture. Normalization alone fails to prove this conjecture, however. The next command provides the first step of user guidance for this proof.

```
resume by ⇒
```

The goal of this proof in an implication. To LP's normalization process, however, an implication is just like any other statement. That is to say, it is only true if it normalizes to true. Consequently, there is a special command to tell LP to break up the conjecture and treat it as an implication. This command is

```
resume by ⇒
```

This command causes LP to try to use the hypotheses of the implication to try to prove the conclusion. Since the conjecture we are working on is an implication, the script uses this command to tell LP to treat it as one.

```
  <> 1 subgoal for proof of ⇒
    [] ⇒ subgoal
  [] conjecture
```

Lines that begin with the symbols <> and [] show progress towards proving a conjecture. These lines are added by LP to the script file. Users do not have to type them. The <> symbol (called a diamond) signifies the start of work towards a new goal. Similarly, the symbol [] (commonly called a box) signifies the completion of a proof goal. Therefore, these lines show that LP treated the conjecture as an implication, and automatically proved the implication subgoal, thus proving the conjecture. For more about diamonds and boxes, see section 5.3.

```
qed
```

This command asks LP to confirm the completion of a proof. When executed in a script, it causes the script to stop running if the proof is incomplete. If the proof is complete, LP continues on to the next command.

```
set proof-method ⇒, normalization
```

In the previous proof, the user had to tell LP to treat the conjecture as an implication. Since it is usually desirable to do this when the conjecture is an implication, LP provides a *proof method* to automatically break up the conjecture when it is an implication. Proof methods are approaches to proving that LP can automatically apply to all conjectures. The default proof method (as mentioned earlier without the name) is `normalization`. The command above simply tells LP to use the implication proof method as well as normalization.

```
prove
  ((inv(s:TA$States) ∧ enabled(s:TA$States, a:TA$Actions)
      ∧ effect(s:TA$States, a:TA$Actions, s':TA$States)) ⇒ inv(s':TA$States))
        by induction on a:TA$Actions
  ..
```

This is the second and final proof obligation for proving the correctness of the invariant. It corresponds to proving that any action that every action preserves the invariant. This, when combined by the previous line, is sufficient to show that the invariant holds in any reachable state.

The line that reads `by induction on a:TA$Actions` gives LP a hint for performing the proof. As discussed earlier, `induction` is the method used to tell LP to consider each action separately. Once LP has been given this hint, it is able to carry out the rest of the proof by itself, as shown by the following diamonds and boxes that were automatically generated.

```
  <> 2 subgoals for proof by induction on 'a:TA$Actions'
    <> 1 subgoal for proof of ⇒
      [] ⇒ subgoal
    [] basis subgoal
    <> 1 subgoal for proof of ⇒
      [] ⇒ subgoal
    [] basis subgoal
  [] conjecture
qed
```

Chapter 4

Example - Counting Automaton

This chapter covers the proof that I verified using Larch.

This automaton proof comes from [8]. In this paper, it is used as an example of an MMT Automaton simulation proof. The proof given there is similar to the hand proof here, but the bounds shown here are slightly tighter.

The first section gives the hand proof of the simulation relationship. Some of the equations in it are labelled for reference from the explanation of the LSL traits and the LP script, which are in later sections of the chapter.

4.1 Hand Proof

The general goal of this proof is to show a timed forward simulation relationship between an automaton that issues a single external report action, and an automaton that counts down internally before issuing a single external report action.

4.1.1 Automata Definitions

Counting Automaton: $Count(c_1, c_2, k)$:

This automaton, which will be referred to as *Count*, is the implementation. It is parameterized on three variables. The first two are the common bounds for the classes, and the third is the number from which it counts down. The definition itself is given as a timed automaton. Within the proof, it is treated as an MMT automaton. The transformation is carried out in the standard way from the definition given here. See Chapter 2 for more details.

The definition of the automaton contains first a definition of the state variables, and the start states. The definition next lists and describes the actions. Finally, it gives the classes and their bounds in terms of the parameters.

After this section, the state variables and actions of the automaton will be referred to using record notation. When record notation is used to refer to an action, the pre-state of the action will be on the left, and the action's name the right.

State:

count, init $k \geq 0$

reported, Boolean, init $false$.

Actions:

report: Output

Precondition: $count = 0 \wedge reported = false$

Effect: $reported := true$

decrement: Internal

Precondition: $count > 0$

Effect: $count := count - 1$

Classes:

$\{report\}$ $[c_1, c_2]$

$\{decrement\}$ $[c_1, c_2]$

Invariants: I_C

The only invariant specific to this automaton is $count > 0 \rightarrow \neg reported$. However, we may use some of the invariants of all MMT automata discussed in Chapter 2.

A state s is said to be in I_C if these invariants are satisfied.

Proof: by induction on the length of execution.

Basis: Trivial, since $reported = false$.

Inductive step: Clearly, because *reported* may only become true through the *report* action, which may only happen if $count = 0$. Thus, every action preserves the invariant.

Specification Automaton: *Report(lb, ub)*

This automaton, which will be referred to as *Report*, is the specification. It is described in much the same way as *Count* was.

State:

reported, Boolean, init $false$.

Actions:

report: Output

Precondition: $reported = false$

Effect: $reported := true$

Classes:

$\{report\}$ $[lb, ub]$

Invariants: I_R

No invariants specific to this automaton are needed, although I_R still requires the invariants of all MMT automata discussed in Chapter 2.

4.1.2 Mapping

This section offers a definition of the abstraction function from *Count* to *Report*.

Let s be a state of *Count* and u be a state of *Report*. Define $(s, u) \in f$ provided that:

I. $u.now = s.now$

II. $u.basic.reported = s.basic.reported$

III.

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

IV.

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

4.1.3 Theorem

f is a Forward simulation from *Count* to *Report* if $lb \leq (k+1)c_1$ and $ub \geq (k+1)c_2$.

The precise requirements for a forward simulation are included with the proof. However, for review of the details of the definition, see Chapter 2.

4.1.4 Proof

f is a Forward simulation from *Count* to *Report* if all three conditions of the definition are satisfied.

Condition one: For every start state s of *Count* there is a start state u of *Report* such that $u \in f(s)$.

Since there is only one start state s of *Count*, it suffices to show that there is a start state u of *Report* such that $u \in f(s)$.

Define u as follows:

- $u.now = 0$
- $u.basic.reported = false$
- $u.first(report) = lb$
- $u.last(report) = ub$

Since this is the start state of *Report*, condition one is satisfied if $u \in f(s)$. Conditions I and II of the mapping are clearly satisfied:

$$u.now = 0 = s.now,$$

$$u.basic.reported = false.$$

Conditions III and IV will be proved by cases:

case 1: $k > 0$, condition III

Because *u.report* is enabled (since u is a start state and the *report* action is enabled in the start state of the *Report* automaton)

$$u.first(report) = lb \tag{4.1}$$

(by definition of timed automata's initial bounds),

$$lb \leq (k+1)c_1$$

(by definition of *lb*), and therefore,

$$u.first(report) = lb \leq (k+1)c_1$$

Furthermore, since $k > 0$, *s.decrement* is enabled. Hence, by definition of timed automata's initial bounds,

$$s.first(decrement) = c_1. \tag{4.2}$$

Because *s.basic.count* $= k$ initially,

$$s.first(decrement) + s.basic.count \cdot c_1 = c_1 + k \cdot c_1 = (k+1)c_1,$$

$$s.first(decrement) + s.basic.count \cdot c_1 \geq u.first(report).$$

Thus, III is satisfied for $k > 0$.

case 1: $k > 0$, condition IV: Similarly,

$$u.last(report) = ub \geq (k+1)c_2 \tag{4.3}$$

since *u.report* is initially enabled. Since *s.decrement* is also enabled,

$$s.last(decrement) = c_2. \tag{4.4}$$

Therefore,

$$s.last(decrement) + s.basic.count \cdot c_2 = c_2 + k \cdot c_2 = (k+1)c_2,$$

$$u.last(report) = s.last(decrement) + s.basic.count \cdot c_2 = c_2 + k \cdot c_2.$$

Therefore condition IV is satisfied for $k > 0$.

case 2: $k = 0$

Because *u.report* is enabled initially,

$$u.first(report) = lb,$$

by definition of the timed automaton. Also,

$$lb \leq (k+1)c_1,$$

by definition of *lb*.

$$u.first(report) \leq (k+1)c_1 = c_1.$$

Since $k = 0$. Furthermore, since $s.basic.count = k = 0$, *s.report* is enabled. Thus,

$$s.first(report) = c_1. \tag{4.5}$$

$$u.first(report) \leq s.first(report)$$

Thus condition III of the mapping is satisfied for $k = 0$.

Similarly, since *u.report* is enabled and $k = 0$,

$$u.last(report) = lb \geq c_2. \tag{4.6}$$

Additionally, $s.basic.count = k = 0$, so $s.report$ is enabled and

$$s.last(report) = c_2. \tag{4.7}$$

$$u.last(report) \geq s.last(report)$$

Thus condition IV of the mapping is satisfied for $k = 0$.

Condition two: If s' is a state of *Count* such that $s \in I_C$, $u' \in f(s') \bigcap I_R$ is a state of *Report* and $(s', (\pi, t), s)$ is a step of *Count*, then there is an extended step (u', β, u) of *Report* such $u \in f(s)$ and $\beta|(\Pi \times \Re) = (\pi, t)|(\Pi \times \Re)$.

By case of π:

Case 1: $\pi = decrement$

Let $\beta = NULL$, where $u.now = u'.now$. First we must show that (u', β, u) is an extended step of *Report*. Since $u'.now = u.now$, and $\forall C \in part(A'), u'.now \leq s'.last(C)$, $\forall C \in part(A'), u.now \leq s'.last(C)$.

Now, to show $u \in f(s)$. The first two conditions of the mapping are easy:

$$u.now = u'.now = s'.now = s.now, \text{ and}$$

$$u.basic.reported = u'.basic.reported = s'.basic.reported = s.basic.reported$$

These two facts result because neither π nor β modifies *now* or *reported*. Consequently, the equality before the actions implies equality after it.

The second two conditions of the mapping are proved by case of $s'.basic.count$:
Case 1: $s'.basic.count > 1$

$$u.first(report) = u'.first(report) \tag{4.8}$$

$$u'.first(report) \leq s'.first(decrement) + s'.basic.count \cdot c_1 \tag{4.9}$$

$$s'.first(decrement) + s'.basic.count \cdot c_1 \leq s'.now + c_1 + (s'.basic.count - 1)c_1 \tag{4.10}$$

$$s'.now + c_1 + (s'.basic.count - 1)c_1 = s.first(decrement) + s.basic.count \cdot c_1 \tag{4.11}$$

$$u.first(report) \leq s.first(decrement) + s.basic.count \cdot c_1 \tag{4.12}$$

Equation 4.8 derives because the *NULL* action does not change anything in u's state. Equation 4.9 is the precondition given by the mapping. Equation 4.10 arises from the fact that $s'.first(decrement) \leq s'.now$ in order for *decrement* to be executed. Equation 4.11 is the rule for the new lower bound of an action which has been executed, and remains enabled. Equation 4.12 inequality provides a restatement, verifying that the mapping is maintained.

$$u.last(report) = u'.last(report) \tag{4.13}$$

$$u'.last(report) \geq s'.last(decrement) + s'.basic.count \cdot c_2 \tag{4.14}$$

$$s'.last(decrement) + s'.basic.count \cdot c_2 \geq s'.now + c_2 + (s'.basic.count - 1)c_2 \tag{4.15}$$

$$s'.now + c_2 + (s'.basic.count - 1)c_2 = s.last(decrement) + s.basic.count \cdot c_2 \tag{4.16}$$

$$u.last(report) \geq s.last(decrement) + s.basic.count \cdot c_2 \tag{4.17}$$

These equations are precisely analogous to the lower bound condition.

Case 2: $s'.basic.count = 1$

$$u.first(report) = u'.first(report) \tag{4.18}$$

$$u'.first(report) \leq s'.first(decrement) + c_1 \tag{4.19}$$

$$s'.first(decrement) + c_1 \leq s.now + c_1 \tag{4.20}$$

$$s.now + c_1 = s.first(report) \tag{4.21}$$

$$u.first(report) \leq s.first(report) \tag{4.22}$$

Equation 4.18 derives because the *NULL* action does not change anything in u's state. Equation 4.19 is the precondition, with 1 substituted for *s.basic.count*. Equation 4.20 is because $s'.first(decrement) \leq s'.now$ in order to execute *decrement*. Equation 4.21 is the lower bound imposed on report when it becomes enabled. Equation 4.22 shows that the goal has been attained.

$$u.last(report) = u'.last(report) \tag{4.23}$$

$$u'.last(report) \geq s'.last(decrement) + c_2 \tag{4.24}$$

$$s'.last(decrement) + c_2 \geq s.now + c_2 \tag{4.25}$$

$$s.now + c_2 = s.last(report) \tag{4.26}$$

$$u.last(report) \geq s.last(report) \tag{4.27}$$

These equations are precisely analogous to the lower bound condition.

Since there are no external actions to either *decrement* or *NULL*, $\beta|(\Pi \times \Re) = (\pi, t)|(\Pi \times \Re)$.

Case 2: $\pi = report$

When $\pi = report, \beta = report$. First we must show that (u', β, u) is an extended step of $time(A', b')$. For this to be true, the following conditions must be upheld:

$$\beta \text{ must be enabled in } u'.basic$$

$$u'.first(report) \leq u'.now$$

The first of these will be true if $u'.basic.reported$ = FALSE which is the case because $s'.basic.reported$ = FALSE, and $u' \in f(s')$.The second is true because $u'.first(report) \leq s'.first(report) \leq s'.now = u'.now$ since $s.basic.count = 0$.

Now, to show $u \in f(s)$:

$$u.basic.reported = \text{TRUE} = s.basic.reported$$

Since both u and s have reported, *reported* is TRUE by definition.

$$u.now = u'.now = s'.now = s.now$$

Neither automaton changes the time during the report action.

$$u.first(report) = 0 = s.first(report)$$

$$u.first(report) = \infty = s.first(report)$$

Both automata have the *report* action disabled, so the values *first* and *last* are the defaults.

In both cases, the only external action is *report*. Thus, $\beta|(\Pi \times \Re) = (\pi, t)|(\Pi \times \Re)$.

Case 3: $\pi = NULL$

When $\pi = NULL$, $\beta = NULL$. Additionally, the β *NULL* should advance time to the same point as the π *NULL*. (ie: t) First we must show that (u', β, u) is an extended step of $time(A', b')$. In order to show this, we must show that

$$t \leq u'.last(report). \tag{4.28}$$

since report is the only class. If $u'.basic.reported$ =TRUE then $u'.last(report) = \infty$, so this condition is upheld. If $u'.basic.reported$ = FALSE, then

$$u'.last(report) \geq \begin{cases} s'.last(decrement) + s'.basic.count \cdot c_2 & \text{if } s'.basic.count > 0, \\ s'.last(report) & \text{otherwise.} \end{cases} \tag{4.29}$$

Thus, if $s'.basic.count = 0$, then

$$t \leq s'.last(report) \leq u'.last(report) \tag{4.30}$$

$$t \leq s.last(decrement) = s'.last(decrement) \tag{4.31}$$

$$s'.last(decrement) \leq s'.last(decrement) + s'.basic.count \cdot c_2 \leq u'.last(reported) \tag{4.32}$$

Thus since $(s', (\pi, t), s)$ is a legal step, (u', β, u) is also.

Now, to show $u \in f(s)$:

$$u.now = t = s.now$$

Since both have advanced to the time t.

Since the $NULL$ action does not change the *basic* state, the second condition of the mapping is held. Furthermore, the $NULL$ action does not change any $first$ or $last$ values. Thus, conditions two and three are unaffected by the action, and remain valid after it.

Since there are no external actions to $NULL$, $\beta|(\Pi \times \Re) = (\pi, t)|(\Pi \times \Re)$.

Thus, the mapping upholds condition two for any step π.

Condition three: If s and u are states of *Count* and *Report*, respectively such that $s \in I_C$, and $u \in f(s) \bigcap I_R$, then $u.now = s.now$.

This condition is upheld by the map by definition.

$\mathcal{QED}$

4.2 LSL Formalization

This section displays and discusses the four traits necessary to formalize the informal model presented in section 4.1.1. These four traits rely heavily on the library of traits described in section 3.2. To understand the precise dependencies, see figure 3-2.

```
CommonActions (A): trait
  assumes IOAutomaton(A)
  introduces
    report  :  → A$Actions
    report  :  → CommonActions
    reportC :  → A$Classes
  asserts ∀ a: A$Actions
    common(report) = report;
    isOutput(report);
    class(report) = reportC
  implies equations
    ¬isInput(report)
```

Figure 4-1: LSL trait defining automata's common actions

```
AutomatonReport (R): trait
  includes IOAutomaton(R), CommonActions(R)
  R$States tuple of reported: Bool
  asserts
    R$Actions generated by report
    R$Classes generated by reportC
    ∀ s, s': R$States
      start(s)              ⇔ ¬s.reported;
      enabled(s, report)    ⇔ ¬s.reported;
      effect(s, report, s') ⇔  s'.reported;
      inv(s)
  implies Invariants(R)
```

Figure 4-2: LSL trait defining the specification automaton

The CommonActions trait defines the actions and classes common to both of the automata in the simulation relationship. Since the automata this trait relates are the untimed ones, we can see that the only common action should be the report action, and the only common class the report class. In order to alleviate human confusion, a class of the automata with only a single action has been conventionally named the same as the action except with a trailing C. This convention is followed in the other automaton as well.

AutomatonReport is the specification automata described in section 4.1.1. Just as in the informal definition, it has one class and one action: report. Notice that

```
AutomatonCount (C, k): trait
  includes IOAutomaton(C), CommonActions(C), Natural(- for ⊖)
  C$States tuple of count: N, reported: Bool
  introduces
    k          : → N
    decrement  : → C$Actions
    decrementC : → C$Classes
  asserts
    C$Actions generated by report, decrement
    C$Classes generated by reportC, decrementC
    ∀ s, s': C$States
      ¬isExternal(decrement);
      class(decrement) = decrementC;
      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 = s.count - 1
                                    ∧ s'.reported = s.reported;
      inv(s)                    ⇔ s.count > 0 ⇒ ¬s.reported
  implies
    Invariants(C)
    ∀ s: C$States
      enabled(s, decrementC) ⇔ enabled(s, decrement);
      enabled(s, reportC) ⇔ enabled(s, report)
```

Figure 4-3: LSL trait defining the implementation automaton

this specification also conveys the information about when the action is enabled, and the effects of the action on the state. The line `inv(s)` means that the automaton has the trivial invariant. This is in contrast to the counting automaton, which has a non-trivial invariant.

`AutomatonCount` is the formal model of the automaton described in 4.1.1. Just as the specification trait, it defines each action and class, and states how they relate. Furthermore, it defines the state and the requirements and effects of each action. The invariant described is that the automaton cannot have the report state variable true before the count has reached zero. This is verified using LP in section 4.3.1.

The `Simulation` trait pulls together all the lower level traits and defines the

```
Simulation: trait
  includes
    AutomatonReport(R), AutomatonCount(C, k),
    TimedAutomaton(R, br, TR), TimedAutomaton(C, bc, TC)
  introduces
    a, c: → Bounds
    f: TC$States, TR$States → Bool
  asserts ∀ u: TR$States, s: TC$States, cr: R$Classes, cc: C$Classes
    % Assign time bounds to the classes of TR and TC.
    br(cr) = a;
    bc(cc) = c;
    c.bounded;
    % Describe the relation between these time bounds.
    a = (k+1)*c;
    % Define the simulation relation.
    f(s, u) ⇔
        u.now = s.now
      ∧ u.basic.reported = s.basic.reported
      ∧ (if s.basic.count > 0
         then s.bounds[class(decrement)] + (s.basic.count * c)
         else s.bounds[class(report)])
                ⊆ u.bounds[class(report)]
  implies TimedForward(TC, TR, f)
```

Figure 4-4: LSL trait defining the simulation relationship

mapping between the states of the two automata. Again, this is much like the informal model of the system.

The most notable thing about the formalization process is that it is quite mechanical to move from the informal definitions to the formal definitions. Simply do the following things once for each automaton:

- define the state as a tuple.
- introduce each unique class and action.
- list each action in a `generated by` statement.
- define the class of each action.
- define the requirements and effects of each action.
- define the invariant, if any.

Next define the common actions in the `CommonActions` trait. Finally, define the `Simulation` trait. This requires the user to

- include each I/O Automaton trait, and map them to boundmaps to define timed automata.
- define the boundmaps
- define the simulation relation
- `imply` the `TimedForward` trait

4.3 Commented Proof Scripts

This section contains all the script files run by LP to prove the simulation relationship. They are accompanied by comments that help describe what each section of the script is doing, and relate it to the hand proof. Whenever possible, I have tried to refer to the specific equations and proof sections in the hand proof. In this way, I hope the formal proof becomes clear.

There are two proof scripts. The first one merely proves the invariant of the implementation automaton. The second proves the simulation itself.

4.3.1 Verification of the Implementation Automaton's Invariant

The following proof script verifies the invariant of the Counting Automaton. It works by initially verifying it in the initial states, and subsequently showing that the invariant cannot be broken by any action from a state where the invariant holds. This proof is quite similar to the example in Chapter 3, and the style is common to all invariant proofs.

```
execute AutomatonCount_Axioms

set name theorem
set proof-methods ⇒, normalization
```

This first section loads in the axioms and sets the proof methods. These proof methods will enable most of the rest of the proof to be performed without human intervention. These two proof methods are almost always used, as they are usually appropriate when they are applicable. For more about proof methods see Chapter 5.

```
prove start(s) ⇒ inv(s)
  qed
```

The initial conditions are proved solely with normalization and implication. No human guidance is needed.

```
prove
  inv(s) ∧ enabled(s, a) ∧ effect(s, a, s') ⇒ inv(s')
  by induction on a
  ..
  qed
```

The proof that the invariant is preserved by every action requires a little guidance: the `induction` command. This command tells LP to consider each action that *a* might be separately. It requires `induction` rather than a `resume by cases` command because the actions were listed in a `generated by` statement in the LSL formalization.

The similarity between this proof and the proof shown in Chapter 3 is striking: they require exactly the same guidance from the user. This demonstrates some of the power of the prover, as future users will have a very good idea about what is required to prove invariants in general.

4.3.2 Verification of the Simulation Relationship

The proof of the simulation relationship is considerably more complex. It is very similar in structure to the hand proof. It follows, along with heavy commenting.

```
set script simulation
set log simulation

thaw Simulation

set name theorem
set proof-methods ⇒, normalization
```

The previous commands were very similar to earlier proofs. They set up the environment in which the proof operates.

```
% First proof obligation

prove start(s: TC$States) ⇒ ∃ u (start(u) ∧ f(s, u))
```

This is the first of the three proof obligations necessary to prove a forward simulation. It corresponds to condition one in the hand proof. In the statement, the s is universally quantified. Thus, this statement requires that there is a start state u of the specification (the variable u always refers to a state of the specification in this proof) that is in the mapping from s.

```
  resume by specializing u to [[false], 0, update({}, class(report), a)]
```

At the time this command was executed, the conjecture was that there is, in fact, such a state u. This command offers to LP an explanation of what the state is. It is exactly the same as the state supplied for u in the same condition of the hand proof: `reported = FALSE, time = 0`, the `report` class has its initial bounds a.

From here on, the rest of this case serves only to show that this state is a start state, and satisfies the mapping. LP now rewrites these two conditions by substituting in definitions. This turns the conjecture into an enormous conjunction. This is fairly common in LP proofs. Happily many of the conjuncts are proved automatically by normalization. This is also fairly common.

```
    instantiate c:C$Classes by class(report) in *hyp
    instantiate c:C$Classes by class(decrement) in *Hyp
```

This tells LP that the hypotheses that are true of classes in general are also true of these two classes in specific. In terms of the hand proof, this gives equations 4.1, through 4.7.

```
    resume by case k = 0
      resume by induction on c:R$Classes
        resume by specializing a:R$Actions to report
      resume by induction on c:R$Classes
        resume by specializing a:R$Actions to report
qed
```

Just as the hand proof does, the LP proof splits its proof into two case: $k = 0$ and $k > 0$. The four commands following the case statement perform each of the cases. In both cases, the goal is the same: to show that the `report` action of the simulation is enabled. The rest of the proof is handled automatically. In each case, the induction command tells LP to look at each class separately. Fortunately, there is only a single class of this automaton, so there is only one class to consider. The specific subgoal that the `resume by specializing` commands address is:

```
    ∃ a:R$Actions(enabled([false], a) ∧ class(a):R$Classes = reportC)
```

Once this subgoal is proved, the case is complete, and proving these two cases is sufficient to finish the proof of the first obligation.

```
% Second proof obligation
```

```
prove f(s, u) ⇒ u.now = s:TC$States.now
qed
```

Just as this requirement is trivial by the hand proof, it is carried out without any guidance by LP.

The next several sections contain proofs of lemmas for the final proof obligation.

```
% The first two lemmas enable LP to deduce ordering relations based on
% transitivity.  These lemmas may become unnecessary when work is
% completed to supply LP with a decision procedure for the ordering of
% the nationals.

set name Transitivity
prove when x:Time ≤ y:Time, y:Time ≤ z:Time yield x:Time ≤ z:Time
  critical-pairs *Hyp with IsTO
  critical-pairs *Hyp with Transitivity
qed

prove when y:Time ≤ z:Time, x:Time ≤ y:Time yield x:Time ≤ z:Time
qed
```

These are a proofs of deduction rules that enable transitivity to be handled more easily than by the two critical pairs commands described in section 3.3.2. Deduction rules in general are also discussed in section 3.3.2.

```
% The next lemma supplies a fact that should be included in, or implied
% by, the LSL Handbook trait for the natural numbers.

set name natLemma
prove n = 0 ∨ n = 1 ∨ n > 1
  resume by induction
qed
```

This lemma is useful for a certain case statement in the proof. It follows the case breakdown of the hand proof of this condition. Unlike most other induction commands, the `resume by induction` here is actually induction in the usual sense.

```
% The following lemmas establish some simple facts about the classes of
% actions in C and R.  They might be unnecessary if LP provided special
% handling for singleton sorts.

declare variables
  rc: R$Classes
  ra: R$Actions
  rs: TR$States
  ..

set name lemma
prove class(ra) = class(report) ⇔ ra = report by induction
qed

prove a:C$Actions = report ∨ a:C$Actions = decrement by induction
qed

prove ¬(enabled([true], a) ∧ a:R$Actions = report) by induction
qed
```

The first of these lemmas states that the only action in the `report` class is the `report` action. Similarly, the second states that the only actions in the implementation automaton are `decrement` and `report`. The final lemma states that the `report` action of the specification automaton cannot be enabled if the `reported` state variable is true.

Now for the proof of the final condition itself.

```
% Third proof obligation

declare variables
  a: TC$Actions
  u: TR$States
  s: TC$States
  alpha: TR$StepSeq
  s': TC$States
  ..

set name theorem
set immunity ancestor
prove
  f(s, u)
   ∧ isStep(s: TC$States, a, s')
   ∧ inv(s:TC$States)
   ∧ inv(u:TR$States)
   ∧ inv(s.basic:C$States)
  ⇒ ∃ alpha (execFrag(alpha)
                ∧ first(alpha) = u
                ∧ f(s', last(alpha))
                ∧ trace(alpha) = trace(a:TC$Actions))
  by induction on a:TC$Actions
  ..
```

This proof obligation is the same as Condition 2 in the hand proof. Its form is definitely noteworthy. In the hypotheses, we assume the invariants of all the automata (except the untimed specification, which has only a trivial invariant). This is sound because they have already been proved for all reachable states (in section 3.3.3 and section 4.3.1). This is essential to the proof, as it is not possible without the invariants. Furthermore, including the invariants in the hypotheses has been explicitly put in the `TimedForward` trait, from which the proof obligations are generated.

As in most situations described, the `induction` statement tells LP to consider each action separately during this proof.

```
  register top s'c
```

This is a command that helps direct LP's rewriting. It serves to tell LP how we want the normal form. In this case, it tells LP that it should rewrite formulas expressed in terms of post-states to pre-states, since `s'c` is a post-state variable.

```
  % Case 1: simulate passage of time
  resume by specializing
```

```
    alpha to null(uc) ⟨null(tc), [uc.basic, tc, uc.bounds]⟩
    ..
```

Just as in the hand proof, we must specify what the sequence of actions by the specification is. This command is somewhat difficult to read because the two nulls mean different things. The first indicates the start state of α: the pre-state of the specification. The second indicates the time passage action that advances time to tc. The section in brackets defines the post-state: the basic component of the state and the boundmap are unchanged, and now is tc.

```
    resume by induction on c:R$Classes
```

Just as the proof of condition 3 for the NULL action in the hand proof begins with a proof that the sequence is legal, we must do so for the null action as well. In the hand proof, this amounts to showing 4.28. We arrived at this equation because reportC is the only class with bounds to check. Thus, the first command reminds LP that reportC is the only class of the specification.

```
      resume by case sc.basic.count = 0
        resume by case (uc.bounds[class(report)]).bounded
```

The first command tells LP to break the proof down into cases as equation 4.29 does. The second command handles the cases of *u.basic.reported*, as the report class will be unbounded only when the report action is disabled, and this can only happen when the automaton has already reported. Notice that in the hand proof, these cases are done in the other order.

```
          instantiate c:C$Classes by class(report) in *Hyp
```

This command proves that the sequence is valid if *s.basic.count* = 0, and *u.basic.reported* =FALSE, this is the same as equation 4.30.

```
        % The remainder of this case guides LP to apply transitivity.
        resume by case (uc.bounds[class(report)]).bounded
          instantiate c:C$Classes by class(decrement) in *hyp
          set immunity on
```

```
prove
  (s'c.bounds[class(decrement)]).last ≤
  ((s'c.bounds[class(decrement)]).last
     + (s'c.basic.count * c:Bounds.last))
  ..
set immunity ancestor
critical-pairs *CaseHyp with *ImpliesHyp
apply Transitivity to theorem
```

This entire sequence of commands serves only to prove 4.31 and 4.32. This difficulty is somewhat characteristic of LP's difficulties with algebra. Fortunately, this problem is well known to LP's programmers, who are working on ways to improve it. This is discussed more in Chapter 6.

On the bright side, proving that this α is a legal step is sufficient to prove the entire NULL case, unlike the hand proof, where considerably more work must be put in to prove it.

Next, we must prove the simulation for the actions of the untimed implementation automata, augmented with time.

```
resume by cases c6c = report, c6c = decrement
```

This command tells LP to consider the different actions of untimed automata separately.

```
% Case 2A: simulate report action
resume by specializing
  alpha to
  null(uc) ⟨addTime(report, uc.now),
             [[true], uc.now,
              update(uc.bounds, class(report), [false, 0, 0])]
           ⟩
  ..
  resume by induction on c:R$Classes
```

These commands offer LP general guidelines for this section of proof by supplying it with a sequence α, and reminding it that the only class of the specification is `report`. This case shows the power of LP when there is only simple algebra: these hints alone suffice to guide LP through the case.

```
% Case 2B: simulate decrement action
resume by specializing alpha to null(uc)
```

Once again, we must supply LP with a sequence α for the specification automaton.

```
    resume by case
       sc.basic.count = 0,
       sc.basic.count = 1,
       sc.basic.count > 1
       ..
      instantiate n by sc.basic.count in natLemma
```

The `resume by case` command parallels the case breakdown in the hand proof. However, it also includes a 0 case, since LP does not yet know that the decrement action must occur when $s.basic.count > 0$ (although it works out that portion of the proof without guidance). The second line offers justification of why these are all the natural numbers, by relying on a lemma.

```
instantiate c:C$Classes by class(report) in *impliesHyp
resume by case (uc.bounds[class(report)]).bounded
  critical-pairs *ImpliesHyp with *ImpliesHyp
  resume by ∧-method
    apply Transitivity to conjecture
    apply Transitivity to conjecture
```

These commands work out equations 4.18 through 4.27, thus solving the case of $s.basic.count = 1$. Although these commands are not as nice as they might be, in that LP requires considerable guidance to work out this algebra, they are still considerably shorter than the same section of the hand proof.

```
        prove sc.basic.count - 1 ≠ 0 by contradiction
            critical-pairs *contraHyp with *
```

These commands prove a useful lemma for the *s.basic.count* > 1 case.

```
        instantiate c:C$Classes by class(decrement) in *impliesHyp
        define-class $timeLemma Time / contains-operator(-:N,N→N)
        instantiate t by c.first, n by sc.basic.count in $timeLemma
        instantiate t by c.last, n by sc.basic.count in $timeLemma
        resume by case (uc.bounds[class(report)]).bounded
          resume by ∧-method
            apply Transitivity to conjecture
            apply Transitivity to conjecture
```

These commands guide LP through equations 4.8 through 4.17 in order to complete the case of *s.basic.count* > 1. Again, the prover needs much guidance, but the proof is still much shorter than the hand proof.

```
qed
```

This completes the LP proof of the simulation relationship. Notice that it is about the same length as the hand proof, even with all the commentary.

Chapter 5

Techniques for Using LP

Using the Larch Prover varies a great deal in ease. There are times when it is a joy, and LP is able to prove things quite easily, often considerably more easily than the hand proof. There are also times when it is very frustrating, as there seems to be no way to make it accept a proof for an "obvious" conjecture.

When I first started, the major problem was my lack of expertise. I would waste entire days because I simply did not know how to drive LP effectively. This section is intended to work with sections 3.3.1 and section 4.3 in explaining the most confusing issues to a new LP user.

Later, the problems began to come from other corners. Most of the severe stumbling blocks resulted from omissions in the axioms. These have now been worked out of the standard traits. Furthermore, considerable effort has been put to honing the LSL traits so they provide more powerful support for the LP user. This means that the new user starts out with a much better environment than I did. For example, when I first tried to prove the `TimedAutomata` trait's invariants several months ago, I could not prove them in several days of work, even though I was a fairly skilled LP user at the time. Now, however, it is easy to prove them in about fifteen minutes.

Occasionally, LP would lack a very useful capability, and I was forced to work without it. At times, this presented a significant delay. Even now, there are still many improvements that would make LP much more useful. However, as time goes on, more and more powerful abilities are being added to LP. Consequently, it's always becoming easier to use.

This section presents techniques for using LP that can help eliminate many of the frustrating times, and alleviate them when they do come. The section begins with a description of an overall approach to writing a proof script. Next, it discusses pitfalls in LP use. It continues with hints for times when LP is stuck. Finally, it concludes with a general approach to polishing a completed proof.

Another description of the approach, and more of these sorts of hints, can be found in [2], although the hints found there are usually more applicable in a setting where there are fewer facts than in the automata environment described here. Furthermore, there are several new developments in the Larch Prover that may render some of the more specific hints less useful. In general, however, I believe that the hints will be applicable in most situations where the user is stuck.

5.1 Hand Proof

When I did my proof, I first wrote a hand proof of the simulation relationship. During the course of the machine proof, I found this to be very helpful, as it is often difficult to think about the proof itself while using LP. This was particularly true of times when I was doing some relatively complex algebra. This is because I was able to use my hand proof to see both the steps that were needed and which hypotheses would be useful to perform the steps I made. It became something of a checklist of facts needed to prove my conjecture.

In general, when doing the hand proof beforehand, it is best to go into as much detail as possible, especially as to *why* a fact is known. This may pay significant dividends later on, when doing the proof in LP.

On the other hand, since there are times when LP can accomplish difficult sections of proof with little or no guidance, it does not really make sense to work out every section in extreme detail beforehand. One possible approach is to make a sketch of the proof by hand, and then begin LP. When LP gets stuck, stop working on it, and work out the section in detail by hand before returning to LP.

5.2 Breaking Down Proofs

In general, any significant proof cannot be carried out in one step, but must be broken down into subproofs, each of which is easier that the whole proof. This section discusses how to perform this breaking down process. It consists of three parts. The first discusses the concept of breaking down a proof and describes a general approach, the second discusses proof trees, and the third proof methods.

5.2.1 Top Down Proof Design

The analogy between proving theorems and software engineering is sometimes made. This analogy holds to a certain degree. The LP user must be able to break down a proof into more manageable pieces, just as a software engineer must break down a program. Essentially, the process that the LP user goes through parallels top-down design of the program. This section details how this process works, and the tools which LP provides the user to perform the break down process. In general the act of breaking down the proof consists of two parts. The first is splitting a large proof obligation into two or more smaller obligations, and the second is proving the smaller obligations. There are several ways of performing the first part, which may be more or less appropriate, depending on the precise nature of the proof obligation to be broken down. The next several paragraphs deal with some of the possible methods that may be used on a proof obligation.

Conjunctions In many cases the proof obligation will be a conjunction. For example, $a \wedge b$. In these cases, the **resume by /\** command (discussed in section 4.3.2 as well as in [2]) may be appropriate. While this is definitely useful when initially doing a proof, it often makes the proof script longer than it need be, as it often produces several very similar cases. Thus, it is sometimes a useful tool for doing the initial proof, but may be eliminated in the polishing phases.

If-Then If the conjecture is of the form `if a then b else c`, then it is frequently useful to employ the `resume by if` command. This breaks down the conjecture into two subgoals: `b` given `a` and `c` given `¬a`. It is much like the **`resume`** `by /\` command, in that it is useful when initially doing the proof, yet can often be eliminated in the polishing stages. For more information about this command, see [2].

Separation into cases If a hand proof has been written, it is usually pretty easy to tell when to employ this technique. It will, in general, follow the hand proof. Unlike conjunction, it will almost always remain in the proof even after polishing. There are two forms of breakdown into cases. The first is **`resume`** `by` **`case`** `a`, which separates the proof into cases a and $\neg a$; and the second is **`resume`** `by` **`case`** `a,b,c` which produces four subgoals: the conjecture with a, b, or c as a hypothesis, and that a, b, and c are all the possible cases (the justification subgoal). Breakdown into cases is also discussed in section 3.3 and [2].

Induction Induction is normally pretty clear. Use it when you use it in the hand proof. However, there are some times when it is necessary to use it when you used a case breakdown in the hand proof. For example, I ran into this difficulty when I wanted to say something about all the classes of an automaton. In general, it is appropriate to use **`resume`** `by induction` rather than cases whenever the LSL formalization shows that the case breakdown comes from a **`generated by`** statement.

Using lemmas Probably the most important aspect of the process of breaking down proofs is using lemmas. In order to know when a lemma may be needed, it is often helpful to have a thorough hand proof. This will show you many of the times when a particular lemma is helpful. For example, in algebraic manipulations, it is good to know some of the intermediary steps in the manipulations in order to know what to make a lemma. If your chain of reasoning went $eqn_1 ... eqn_i ... eqn_n$, then it may be helpful to use eqn_i as a lemma in the proof of eqn_n. Often, however, it is not clear from the hand proof what lemma will be needed. At these times, I usually pick something that I believe to be a reasonable lemma, and assert it. I then try to use the lemma to prove the conjecture. If I can do so, then I subsequently try to prove the lemma. If not, I look for another lemma.

Once it's split

When a proof goal has been split into easier subgoals, there are several places where work can be done, namely on each the subgoals, or using the subgoals to prove the main goal. This last section (integrating the subgoals to prove the main goal) may not always be present, either because it can be viewed as just another subgoal or because it is trivial. Since there are several places to work, the user will generally have a choice to make about what to work on. I recommend asserting each of the subgoals temporarily. In this way, the process of using LP to prove a theorem amounts to repeatedly taking an assertion, breaking it down into subgoals, temporarily asserting these in order to use them to prove the goal, and eliminating assertions that can be proved. The proving process ends when there are no more assertions left to eliminate.

5.2.2 Proof Tree

Essentially you can think of a proof as a tree of subproofs, each of which proves some portion of the general conjecture. In practice, this means that an incomplete proof will include several sections of proof that are not complete, but that may be almost done.

Proof Methods

When I first came to understand the idea of using `resume by /\` and `resume by =>`, I found that most of my work using LP consisted of me typing a command based solely on the structure of the conjecture. While this was easy, and made me feel that I was accomplishing a lot without too much mental strain, there is a more efficient way of doing things: *proof methods.* A proof method is a technique that LP automatically attempts to apply to any conjecture before giving up and getting a command from the user (or the script that it is running). For example, if the proof method is /\, then LP will always break down conjunctions and attempt to prove each conjunct separately before giving up and surrendering control to the user. The main use of proof methods arises from the fact that you can have multiple proof methods.

For example, you could have `/\, =>, normalization` as a proof method. With this method, all conjectures would be broken down by all of these methods before the user was asked for input. The methods are applied in the order provided. In many cases this is irrelevant, as a conjecture cannot be both a conjunction and an implication, but it is definitely relevant in the instance of normalization. In general, I like to apply normalization last, as this will make the proof log somewhat easier to read. While I was working on the draft of the proof, I used `/\, =>, if, normalization` as my proof method. Although this usually produces more cases than necessary, it means that in any situation where input was needed, I knew that I would only be dealing with a simple proof obligation (in the sense of it not being a union of two other obligations). This means that I did not have to deal with the issue of proving both halves of a conjunct at once or similar issues. Although this led to a longer proof script, it meant the precise goal I was working on was somewhat more clear in my mind than it would otherwise have been. Proof methods are also discussed in [2].

5.3 Pitfalls in LP Use

This section deals with some of the issues that caused me enduring trouble when I was working on my proof. Almost universally, these difficulties resulted from missing axioms. For these problems, I found that I could never quite manage to prove any properties involving the axioms. I would leave several assertions that seemed very similar to me in the proof script, because I did not know how to eliminate them. It turned out that they were multiple instances of the same missing axiom. The easiest way to deal with problems like this is to try to modify each assertion in the proof script that you cannot otherwise eliminate so that it is a statement that is clearly true in general rather than one that relies on the specific details of your system. For example, in my proof I was missing the axiom:

$$a \leq b \Leftrightarrow a + c \leq b + c$$

In all instances when I needed this in my algebra, I could not get LP to accept the proof as complete, but asserting the specific fact derived from this would be sufficient to perform the proof. For example, when I needed to get equation 4.10 I could never get the proof to go through, because I kept trying to prove this statement was true because one could simply "add $s'.basic.count \cdot c_1$ to both sides of the inequality," but LP refused to believe this could be done, because it was missing the axiom.

Another difficulty, in the same vein, that I had resulted from the lack of invariants in the initial formalization of I/O Automata. This kept me stuck in various places for several weeks. I would have in my hand proof "⟨something⟩ is obviously true," but I could not seem to prove it in LP. This was because the "⟨something⟩" was an invariant of the automaton which was true in any reachable state. However, the formalization we had at the time did not contain the concept of an invariant, much less the specific invariant I needed. So we were forced to add invariants to the formalization. As a result of this, I would heartily recommend that during the hand proof the writer pay careful attention to invariants that are being used. Then, when it is complete, a list may be drawn up and incorporated into the formalization of the automaton. In general, it is better to include an invariant than not to. Remember that to get the proof obligations for invariants, you must run the LSL checker on the automaton's LSL file. The LP output produced will include the necessary obligations and the axioms with which to prove them.

A final, very different difficulty resulted from working with long proof scripts. Every notable proof has many subgoals, many of which require commands to prove them. To the user, it is obvious that a specific command is intended to solve a particular goal, rather than necessarily being applicable to all of the goals in the proof. To LP, however, the proof script is simply a list of commands. Thus, it is sometimes possible for the proof script to get "out of sync" with the proof goals. For example, say a change in the axioms is made, which causes a sequence of commands that used to prove a particular subgoal to fail to do so. Now LP will continue to perform commands, but the commands it is doing were not intended for this goal. LP will continue to execute the script, until it stops for some other reason, such as referring to a hypothesis that does not exist. At this time, it will tell the user why it stopped, but the reason it stopped will have nothing to do with the real problem. Worse still, it is very difficult to find the original problem, as LP has died in a completely different section of proof.

Fortunately, LP provides a mechanism to get around this difficulty. It is called *Box-Checking.* If box-checking is on, then whenever LP completes the proof of a subgoal, it looks for a line that begins with the symbol `[]`. Furthermore, whenever it starts working on a new subgoal, it looks for a line that starts with `<>`. If it fails to find the line it is looking for, then it stops immediately. Furthermore, if it sees a line that starts with either of these symbols at any other time, it immediately stops. Thus, one merely needs a proof script that contains these lines. Fortunately, LP automatically inserts these lines into its record of the commands it executed in the ".lpscr" file.

Unfortunately for box-checking, it can be tedious to work with while developing a proof. For example, if you add in some new commands that prove a lemma rather than simply asserting it, there is a good chance that you will forget to add the necessary `<>` and `[]` lines. If so, then LP will needlessly stop execution to tell you this. Despite its problems, however, using box-checking will save you a tremendous amount of time, as you will always be informed if the commands you thought would prove a subgoal did not do so.

5.4 Getting Stuck

The most frustrating time during the course of a proof is when you are trying to prove something that seems atomic. In other words, the goal to be proved cannot seem to be broken down any more, and seems to follow immediately from known facts. It seems to you that a conjecture should have been proved, but for some reason LP just does not accept the proof. At times like this, I often found myself staring at the screen in frustration and not actually doing anything. This section presents things to do, so that you can learn why LP does not understand the proof you have given it.

At this time, the best thing to do is to think about why the proof is true. From what facts does it follow? Try to find each of these facts. For this step, it is often helpful to use LP's display command and class facilities. For example, you might use a command such as

```
display cont-op(a) / cont-op(b)
```

where a and b are two constants found in the formulae you are interested in. If you cannot find one of the known facts, consider where in the LSL formalization it should be. Check that file for it. If LP has all the facts but you still cannot prove the goal, try to piece together the facts as LP does (i.e.: using rewrite rules) and prove the goal.

Some other useful things to do or tools to use:

- **show normal-form** of a crucial fact that never seems to appear. Perhaps it is normalized away.
- **set immunity on** particularly if some facts are being normalized (see section 3.3.1 for more about this).
- **set trace-level** x, where x is some level between 5 and 8. This will give you many facts about what exactly is going on, such as why the facts you are looking for cannot be proved. Make sure you have a very long scrollbar!

5.5 Slogans

Here is a small set of slogans that I think are good to follow, and will make your life as an LP user somewhat easier. Most of them are mentioned elsewhere in this chapter, but I have gathered them here for convenience.

- a careful hand proof pays big dividends later, especially in sections where you expect LP to have difficulties.
- use box-checking.

- break it down (as much as possible, and merge the cases later).
- assert first and prove later.
- Check your axioms if it stays broken.

5.6 Polishing the Proof

The idea of polishing a proof goes back to the original motivation for using automatic verification. Ideally, we would like to be able to publish an LP proof rather than a hand proof. To this end, we need to make the proof as lucid as possible, yet have it not be too long, or incomplete on any details. The ideal medium for this would be a hypertext format, as this would allow the proof to be complete and detailed, yet not require users to read through any details they did not need. Furthermore, the tree structure of the proof allows a mechanical method for deciding where one can click to see more information. At any particular level, each goal would be displayed, and clicking on it would display all the subgoals (and how they collectively prove the goal), but not how the subgoals themselves are proved. Additionally, clicking on any rewrite rule's name could display that rule. Happily, all this information is available in the proof log. Thus, it is possible to write a "compiler" from the proof log to a hypertext viewer format.

For the interim, however, we have taken the attitude that "brevity is the soul of wit." Often proof scripts seem to run very long, and still not provide a very clear picture of what is happening in the proof. Thus, we have attempted to shorten the proof script as much as possible. In this way, we can still afford to insert manual comments, as I did in chapters 3 and 4. In general, we have been very successful in shortening proofs. In the case of the example proof, the original was 616 lines and the polished form was 173. Many techniques were brought to bear on the proof, many by Steve Garland rather than me. Here I will discuss some of the most powerful ones that I am familiar with.

One of the easiest techniques is eliminating box-checking. In our case, this saved about 25 percent of the length of the full proof. This increases the responsibility of the user for making the proof lucid, as the user must now give a context for each command, but the user will be able to do this much more coherently and in less space than the boxes and diamonds.

Another technique, which I have spoken of before, is combining conjuncts or cases. This can be very useful if the two cases or conjuncts are very similar. The following sections come from the example proof. They show an example of how space can be saved by polishing the proof.

The following is the original script to prove condition 1 of the forward simulation.

```
set proof-method  ∧ , ⇒, if, normalization
set name ForwardTheorem
prove start(s: TC$States) ⇒  ∃ u (start(u) ∧ f(s, u))
  resume by specializing
    u to [[false], 0, update({},class(report), a.first),
                      update({},class(report), a.last)]
    ..
    resume by case 0 < k
       set immunity ancestor
        instantiate c:C$Classes by decrementC in *Hyp
        instantiate c:C$Classes by decrementC in *Hyp
        crit *ImpliesHyp with AutomatonReport
        crit *ifhyp with lemma
        crit *ImpliesHyp with AutomatonReport
        crit *ifhyp with lemma
        crit *ifhyp with lemma
        crit *ifhyp with lemma
%%now do k = 0 case:
        instant c:C$Classes by reportC in *hyp
        instant c:C$Classes by reportC in *hyp
        crit *ImpliesHyp with AutomatonReport
        crit *ifhyp with lemma
        crit *ImpliesHyp with AutomatonReport
        crit *ifhyp with lemma
        crit *ifhyp with lemma
        crit *ifhyp with lemma
  qed
```

The remainder is the polished version of the above script. It proves the same condition, but in about half the number of lines. Most of the savings here come from combining conjunctions and "if-then-else"s that had similar requirements and moving some operations to before a case statement, so that both cases can see the results without performing the operations individually.

```
set name theorem
set proof-methods ⇒, normalization
prove start(s: TC$States) ⇒ ∃ u (start(u) ∧ f(s, u))
    make immune conjecture % NOTE: Optimization to save time.
    resume by specializing
      u to [[false], 0, update({},class(report), a.first),
                        update({},class(report), a.last)]
      ..
      instantiate c:C$Classes by reportC in *hyp
      instantiate c:C$Classes by decrementC in *Hyp
      resume by case k = 0
        resume by induction on c:R$Classes
            critical-pairs *hyp with AutomatonReport
        resume by induction on c:R$Classes
            critical-pairs *hyp with AutomatonReport
qed
```

Chapter 6

Conclusions

Initially, we wished to explore three questions about automatic verification of proofs involving the timing properties of MMT automata. First, is it possible? Second, is it feasible to do easily? Third, can we find a way to create a proof that is lucid enough to publish? The answer to the first is certainly yes, although we always expected that it would be. The answer to the second is that it is somewhat feasible at present. The environment has improved immensely during the course of my work with it, and it will continue to do so as the tools develop. The remaining sections discuss directions of future development. Finally, publishing these proofs appears to be feasible. We will now need some tools to automate this process, as well.

Perhaps the most telling question is "how does computer aided proving compare with conventional proving by hand?" Both have their strengths and weaknesses. The major strength of LP is that, with it, the user is virtually guaranteed a correct proof, whereas a hand proof has a very high chance of having some omission. Furthermore, most sections are much easier to prove with LP than by hand. However, with the present state of the tools there are also some sections that are much harder to do with LP than by hand, particularly if the user is inexperienced. Also, commenting an LP proof for a reader is much easier if there is a hand proof to refer to. As the tools develop, however, the balance will increasingly swing in favor of LP, as it will retain its advantages, and the problems will become increasingly minor.

6.1 Future Work

Most of the future work I see comes in two forms: using the tools and improving the tools. At the present time, the environment we have described here is certainly usable, but contains many annoyances that make it overall somewhat harder to use than simply writing a hand proof. Still, some hardy souls are pressing forward with newer and more difficult proofs. Currently there is an effort to verify a correctness proof of Fischer's mutual exclusion algorithm [6]. In order to alleviate the difficulties currently faced in proof verification, work is being done to improve the tools. Here are some examples of ideas for future work on support.

To aid with formalization, we have considered an I/O automaton to LSL compiler. This would enable the user to suppress most of the details of using LSL, as it would no longer be necessary to write traits directly. However, there are certainly problems with this approach. The primary problem is that it would need very complete error checking, as any error messages generated by the LSL checker due to problems with the traits generated by the compiler would leave the user bewildered, as error messages would not make much sense. Perhaps a better first step is some sort of macro facility that would eliminate the necessity of entering the entire trait.

This sort of macro facility might, for example, offer the ability to automatically insert a new action. At present, inserting a new action and its class requires the addition of seven lines in the trait describing the automaton. However, only two of these require information beyond the action and its class. Thus, the macro facility could automatically create the other five lines. This would save considerable time in the formalization of automata with many actions.

However, most of the benefit to be gained from improvements to the environment is in the proving process, as the user spends much more time proving than formalizing. These improvements fall into two categories. The first is developing more powerful traits, which will generate better axioms. This process has certainly begun with my example, but the next few proofs carried out may also find rough edges in the axioms to be polished. The second category of improvements to the proving process is improvements to LP. Some possible improvements (in various degrees of readiness) are

- to offer better support for arithmetic and algebra
- to improve the front end.
- to extend the critical pairs command to statements containing quantifiers. This would be useful in many places where dealing with quantifiers is currently necessary, as these require much more expertise.
- to offer support for using a case statement rather than induction to signify all the classes or actions of an automaton.

There are also many more improvements that will be made to LP in the future, as it has a dedicated set of programmers.

For presentation, there are two kinds of improvements that might be useful. For paper proofs, there could be some sort of customizable facility that allows the user to automatically generate a reasonable paper proof from an on-line proof. For viewing proofs on-line, one might use a hypertext proof viewer such as the one described in Chapter 3.

6.2 Vision of Paradise

The way I see it, the ultimate goal of all of this is to make an environment where the entire proving process can be carried out on-line. Thus, the necessity for writing a hand proof will be eliminated. Furthermore, proof assistants will be able to take care of everything besides the highest level strategy. This is currently very close to reality when the proof is very easy, but the difficult cases are still much harder to handle in LP than by hand.

Bibliography

[1] R. Cardell-Oliver, R. Hale, and J. Herbert. An embedding of timed transition systems in HOL. *IFIP Transactions A [Computer Science and Technology]*, A20:263–78, 1993.

[2] Stephen J Garland and John V. Guttag. *A Guide to LP, The Larch Prover.* Digital SRC, 1991.

[3] M. J. C. Gordon. HOL: a proof generating system for Higher-Order-Logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification , Verification and Synthesis*, Kluwer, 1988.

[4] John V Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[5] P. Loewenstein and D.L. Dill. Formal verification of cache systems using refinement relations. *Proceedings of the 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 228–33, 1992.

[6] Victor Luchango and Nancy Lynch. Correctness proof of Fischer's mutual exclusion algorithm. Technical report, Lab for Computer Science, LCS, Cambridge, MA, November 1993.

[7] 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, 02139, November 1993.

[8] 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, 02139, November 1991.

[9] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1988.

[10] 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, 02139, April 1993.

[11] W. Mao and G.J Milne. An automated proof technique for finite-state machine equivalence. *Proceedings of Computer Aided Verification, 3rd International Workshop CAV '91*, pages 233–43, 1992.

[12] M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In *CONCUR'91 Proceedings Workshop on Theories of Concurrency: Unification and Extension*, Amsterdam, August 1991.

[13] J. Søgaard-Andersen, S. Garland, J. Guttag, N. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In *Fourth Conference on Computer-Aided Verification*, pages 305–319, Elounda, Greece, June 1993. Springer-Verlag.