Using Simulation Techniques to Prove Timing Properties

by

Victor Luchangco

S.B. Mathematics (1992)
Massachusetts Institute of Technology

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

Master of Science in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author ....
Department of Electrical Engineering and Computer Science
May 19, 1995

Certified by ....
Nancy A. Lynch
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Using Simulation Techniques to Prove Timing Properties

by

Victor Luchangco

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 1995, in partial fulfillment of the
requirements for the degrees of
Master of Science in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis presents a methodology based on simulations and invariants for proving timing properties of real-time, distributed systems. This methodology is used to prove tight time bounds for two systems, a leader election protocol for a ring of processes, and Fischer's timing-based mutual exclusion algorithm. A framework for verifying these proofs using the Larch tools is also developed, and the proof for Fischer's algorithm is checked within this framework.

Many formal methods have been developed for proving the correctness of untimed distributed systems. However, real-time systems often have subtle timing dependencies that are difficult to analyze and reason about. Furthermore, for many real-time systems, correctness is insufficient; it is important to satisfy certain performance requirements. It is necessary, therefore, to extend the formal models and techniques to the timed setting.

We use a timed automaton model, together with simulations which establish that one automaton implements another. The methodology presented here exploits the strengths of simulation-based techniques, and is demonstrated to be applicable to real-time systems, for proving performance, as well as correctness, properties. The resulting proofs are rigorous and systematic, and have a hierarchical structure that appears to scale reasonably to large systems. In addition, they are amenable to automated verification.

Thesis Supervisor: Nancy A. Lynch
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to acknowledge the assistance and support of many people, without whom this thesis would not have been possible. Nancy Lynch did much more than supervise this thesis; her patience and direction was the foundation for my research, and I consider myself extremely fortunate to have such an enthusiastic and encouraging supervisor. The Theory of Distributed Systems group she leads is a wonderful environment in which to do research. The people in the group have been a source of both intellectual stimulation and encouragement, especially Rainer Gawlick and Roberto Segala, who continually prodded me to complete this thesis. Steve Garland, who maintains the Larch tools, oversaw the work in Chapter 6, and helped me a lot in writing my first conference paper and giving my first talk.

Outside of this immediate research community, I have been lucky to have many friends who kept me sane, even when things seemed very bleak. Tony Eng, Nicole Lazo, and Luis Sarmenta listened patiently to (lengthy) ramblings about my research and related interests, and helped make this thesis intelligible. Even more than their intellectual assistance though, I am grateful for their friendship. To Gene Osgood with whom I did all my undergraduate work, to my roommates, to the people on Burton 5 when I was an undergraduate, and to those in UCF, I am thankful for the endless distraction that made MIT such an enjoyable, as well as enriching, experience.

My family, of course, made it possible for me to come to MIT in the first place. I still marvel at my parents' courage and trust in allowing me to go to the other side of the world, and their patience with my delinquency in writing.

For all these, and so much more, I thank God, Who keeps me and blesses me so abundantly. I couldn't have planned all the wonder and joy I have had, and I am grateful to have been so pleasantly surprised.

Contents

Chapter 1

Introduction

A variety of formal methods have been developed to analyze distributed systems, and prove they satisfy basic correctness properties. For real-time systems, it is also often important to establish timing properties, which guarantee the performance of these systems. Sometimes, even the correctness of the system may depend on these timing properties. Unfortunately, real-time systems often have subtle timing dependencies that are difficult to analyze, and proofs of these properties have typically been ad hoc. (See [LS93], which contains several examples of such proofs.) This thesis develops a methodology for analyzing real-time systems.

One family of methods that has been used successfully for analyzing untimed, or asynchronous, systems is based on the notion of a *simulation*, which establishes a correspondence between the given system and a more abstract system that specifies the allowable behavior. Specifically, a system is described as a state machine, and its *behavior* is the externally observable aspects of the sequence of steps it performs in a particular execution. A *simulation* is a correspondence between the states of a system and the states of its *specification* such that any step of the system appears identical to some sequence of steps of the specification that preserves the simulation. In this case, every behavior of the system is also a possible behavior of its specification, and we say that the system *implements* its specification.

Together with *invariance assertions*, simulations have been used by many researchers to verify the correctness of a wide variety of asynchronous systems, some quite complex and subtle. They have used many different specification methods, including temporal logic, automata, CCS, and UNITY [LS84, LT87, CM88, WLL88, Lyn89b, Mil89, Nip89, Gaw92, AL91, SLL93a, KMP93].

Techniques based on simulations yield rigorous, formal proofs, which typically have a systematic decomposition into independent pieces. This structure makes proofs easier to read and to check, both directly and using automated tools. In addition, because the system and its specification are expressed in the same way, as an abstract program or machine, simulation-based techniques allow *layered* or *hierarchical* proofs, where a system is proven to have certain properties using a sequence of abstract machines beginning with one that describes the given system and ending with one that describes the desired properties, in which every abstract machine implements its successor.

More recently, simulation-based techniques have been extended to timed systems, providing a formal and systematic approach to proving timing properties [LA92, AL92].[1] These techniques have been demonstrated on small examples, and the style and difficulty of the proofs are comparable to those of typical inductive assertional arguments. The local nature of the checks suggests that this method may scale well to more complex systems. However, it is important to test this hypothesis, by providing concrete evidence with larger examples, and to develop a framework that exploits the strengths of simulation-based techniques, and mitigates its shortcomings.

In particular, a methodology that exploits the strengths of simulation-based techniques and indicates possible simulations would be very helpful. For example, Lynch and Attiya make explicit their strategy for finding simulations with their definition of *progress function collections*. Guidelines for picking appropriate intermediate specifications would also be very useful. A good methodology will yield modular, hierarchical proofs, with intermediate specifications and simulations that capture the intuition behind the algorithm.

Also, the proofs in [LA92] are rather lengthy and involve much tedious checking. This seems to be an inevitable consequence of the demand for more formal reasoning. It is important to understand how the length and complexity of the proof increases with the system being studied. It is desirable, of course, to minimize this as much as possible without sacrificing rigor. General theorems can capture common arguments, and eliminate repetitive work, as well as expose additional structure. For common system components, such as channels, standard transformations may produce simpler intermediate specifications. If simulations are given in a standard form, this too can be exploited.

[1] The *strong possibilities mappings* of [LA92] define a simulation.

Another promising prospect is the development of automated tools. Automated tools can be used to mechanically verify a proof, which provides added confidence in the correctness of a proof by eliminating the possibility of human error in the manipulation of formal expressions. Some tools can infer "trivial" steps, and so reduce the need to check by hand the straightforward, uninteresting parts of the proof. These parts are usually omitted, anyway, from the discussion of the proof, since they are not instructive. Nonetheless, to have a complete proof, it is important to check that they are correct. If the tool is "smart" enough, it may even deduce the desired claim without any guidance. If it cannot, it may provide information indicating its difficulty, which may be helpful in constructing the proof.

This thesis builds primarily on the work by Lynch and Attiya [LA92]. To describe timed systems, we use a variation of the *timed automaton* of Merritt, Modugno, and Tuttle [MMT91], which we call the *MMT automaton*. An MMT automaton consists of an *I/O automaton* [LT89, LT87], together with a *boundmap*, which specifies the timing assumptions for the system. Following Lynch and Attiya, we incorporate the timing conditions into the state, to yield an I/O automaton of a particular form, which we call a *timed automaton*.[2] We then define a class of simulations, the *timed forward simulations*, which only considers the *admissible* behaviors, those in which time is unbounded, which correspond to the real behaviors of the system.

We use these simulations, together with invariant assertions, to prove correctness and timing properties of two systems, a simple message-passing protocol due to LeLann, Chang, and Roberts [LeL77, CR79] for *leader election* on a ring of processes, and Fischer's timing-based *mutual exclusion* algorithm using a single shared read-write register [Lam87, Fis85]. For both algorithms, we use intermediate specifications to obtain hierarchical proofs, and we extract general heuristics for finding these intermediate specifications.

We also use the Larch tools [GH93] to verify the proof for Fischer's algorithm, building on the work by Söylemez [Söy94], formalizing the basic model and techniques in the Larch Shared Language (LSL), and verifying the proofs using an enhanced version of the Larch Prover (LP) [GG91, Pog95]. This elaborates on work described in [LSGL94]. In this proof, we also try to encapsulate commonly used arguments in lemmas, to make the proof more modular, and to expose some general principles that may be useful in other proofs.

[2] This is different from the timed automaton of [MMT91]. Our terminology also differs from that of [LA92], and reflects later usage, such as in [Lyn93, LSGL94].

Related Work

Other models and techniques for handling time and proving timing properties have been developed. Lynch and Vaandrager have developed a very general notion of a timed automaton, and they describe a wide variety of simulation-based techniques for this model [LV91, LVarb]. Abadi and Lamport [AL92] demonstrate how timing properties can be expressed using Lamport's *Temporal Logic of Actions* (TLA) [Lam91], and thus, the methods developed for TLA, including simulations, can be immediately applied. Their use of simulations, however, is more restricted, and they did not address how timing properties, specifically, can be approached systematically. Many others, including Haase [Haa81], Tel [Tel88], Shaw [Sha89], Harel, Lichtenstein, and Pnueli [HLP90], Alur and Dill [AD90], and Shankar and Lam [SL87, Sha92], also use models that incorporate timing information into the state, but none of them use simulations in their proofs. Shankar uses a model almost identical to ours, except that there are no explicit time passage steps.

Other methods for modelling timed distributed systems include temporal logic [AL92], process algebras [DS89, Wan91, NS91], and Petri nets [CR83].

Several different approaches to proving timing properties have also been proposed, many of them based on augmented temporal logics. The earliest work used *bounded temporal operators* [BH81, KVdR83], but scattered examples of an *explicit clock* approach, presented systematically by Ostroff [Ost89], also can be found. Henzinger, Manna, and Pnueli [HMP94] compare these two styles. More recently, Alur and Henzinger [AH89] presented an approach called *temporal quantification*, embodied by their new logic, TPTL.

Automatic verification motivated much of the design of recent temporal logics, so an important consideration was that they be decidable. Harel, Lichtenstein, and Pnueli [HLP90] presented a decidable restricted explicit clock logic. Alur and Henzinger's TPTL is also decidable, and in another paper [AH90], they explore the trade-off between complexity and expressiveness. All these logics use discrete time (i.e., the natural numbers), since extensions to dense time domains are undecidable. Alur, Courcoubetis, and Dill [ACD90], however, present a logic based on "branching time" computation tree logic (CTL), rather than on linear time, with a dense time domain. These logics are all intended to be used with *model-checking* verification procedures, introduced by Clarke and Emerson [CE81], in which a system is modelled by a finite state machine, and every reachable state is verified to satisfy the desired property.

There are many other approaches to automatic verification. For example, Wang, Pettersson, and Daniels [WPD94] present a method based on solving a system of constraints on the clock variables of a process algebraic specification. Our approach is to follow as closely as possible the formal reasoning we already use to convince ourselves, and use a general purpose theorem prover to verify our steps [SGG+93, Söy94, LSGL94]. Engberg, Grønning, and Lamport [EGL92], and Shankar [Sha93] also take this approach, though Shankar uses PVS rather than LP.

Outline of the Thesis

The rest of the thesis is organized as follows. Chapter 2 contains some background and motivation for those unfamiliar with formal reasoning for distributed systems in general, and simulations in particular. The theoretical foundations, the models and techniques used in this thesis, together with some very simple illustrative examples, are developed in Chapter 3. The next two chapters explore in detail some larger, more interesting examples. Particular attention should be paid to how these proofs are organized, as this illustrates the methodology. In Chapter 4, a timing analysis for LeLann, Chang, and Roberts election protocol for asynchronous ring networks is presented. Chapter 5 examines Fischer's timing-based mutual exclusion algorithm, a nontrivial test case which illustrates techniques for reasoning about time. In addition to correctness, an upper bound on the time to reach the critical region is proved. The timing analyses in these chapters provide the only rigorous proofs we know of for the time bounds of these algorithms. Chapter 6 considers the use of the Larch tools to develop and verify simulation proofs, and evaluates their use in verifying the proof of the Fischer algorithm. Chapter 7 concludes with some discussion about our experiences, and future directions.

Chapter 2

Background and Motivation

This chapter introduces and attempts to motivate the choices made in this thesis, in terms of models, methods, and tools.

2.1 Formal Reasoning for Distributed Systems

2.1.1 What is a Distributed System?

As computers have become more widespread, distributed systems have become the standard computation environment. A distributed system is a collection of sequential processes running concurrently, which must coordinate with each other in order to solve a problem. Unlike a sequential computer, a distributed system cannot easily be described by specifying its output on a particular set of input; the interaction between the processes must also be considered. Thus we characterize distributed systems by the behaviors they can exhibit.

Coordination necessitates interprocess communication, which is typically expensive compared with steps taken locally by individual processes. Distributed systems have different mechanisms for communicating, and there is usually some uncertainty, such as message delay, or possible loss or damage of data, that is associated with communication. Since there are multiple independent processes, the possibility of failure of one or more of the processes must also be accounted for, and it is important to understand how the system as a whole behaves in the presence of such failures.

2.1.2 The Need for Formal Reasoning

Distributed systems, and the problems we wish them to solve, can and often are described informally. They may arise in the real world, or be an abstraction of some observed phenomenon, and thus have no *a priori* formal specification. Reasoning informally about these systems is often helpful, but to make clear, precise claims about them, it is important to have a formal model in which to express and reason about them.

There are two basic reasons for using formal methods. First, good formal descriptions make any assumptions about the system explicit, and any claims about it precise. Formal proofs also make our reasoning very precise, indicating which facts are needed to deduce each step in the proof. Thus, formal methods distinguish the essential features of a system from the details of a particular implementation. This can be especially useful for extending or generalizing the models and claims, and is helpful in understanding the system better.

Second, formal proofs can be checked more easily. For very simple systems, informal reasoning may be clear enough, but when the reasoning is subtle or very complex, it is difficult to argue convincingly without some formal notation that can be checked carefully. A good formal proof can be examined in small chunks, which can be verified individually, and then pieced together to get the desired result. It can also be made progressively more detailed and explicit as the need arises. This makes it easier to localize and understand any difficulties.

These reasons are especially compelling for distributed systems, for which we lack a reliable intuitive understanding. Even apparently simple systems may exhibit a complex variety of behaviors, some unanticipated.[1] Good formal models help develop our intuition about how distributed systems work.

2.1.3 Operational vs. Assertional Reasoning

Since distributed systems are characterized by their behavior, a *property* of the system is a statement that is true of any behavior exhibited by that system. It is possible to reason directly about the behavior of a system *operationally*, that is, as it unfolds. This often corresponds naturally with how we might reason informally. However, since a distributed system can exhibit a wide variety of behaviors with little natural structure, it is difficult to

[1] This is an empirical observation supported by many researchers in the field [CM88, LL90, Sch93].

check that all behaviors have been considered.[2] Furthermore, minor modifications to the algorithm often result in some vastly different behaviors, requiring proofs to be substantially rewritten. This results in proofs that tend to be *ad hoc*, and difficult to follow carefully, and which cannot easily be modified to prove properties about similar systems.

Assertional methods attempt to overcome these problems by focusing on how the system is affected by individual steps that it can take. That is, the system is described by a state machine, and properties are expressed as assertions about states rather than about behaviors. The states are often viewed as a collection of *state variables* modified by the *actions* of the machine, which captures the intuition that individual steps often only affect part of the system.

The most important assertional technique is *invariance reasoning.* An *invariant* is an assertion that is true of every state reachable by some execution of the system. This is usually proved inductively on the length, that is, the number of steps, of an execution, by showing that the initial states satisfy the invariant, and that every action maintains it. This typically allows invariance proofs to be decomposed straightforwardly into several simpler pieces which, taken together, establish the invariant. Thus, the difficult, and interesting, aspect of invariance proofs is discovering the "right" invariant. This requires insight about the system, and usually, once expressed, the invariant is helpful in understanding the system.

Safety and Liveness

When using assertional reasoning, we distinguish properties as expressing either *safety* or *liveness*. Intuitively, a *safety property* is a claim that nothing bad has happened, while a *liveness property* is a claim that something good eventually happens.[3] Safety depends upon the history of an execution; liveness, on the future.

Some properties have both safety and liveness aspects. For example, in the *leader election* problem, where a single leader must be selected from a collection of processes, that "at most one leader is selected" is a safety property, and that "some leader is selected" is a liveness property. At any point in an execution, it is easy to see whether safety has been violated or liveness has been satisfied, but not so easy to determine that safety will not be violated, nor that liveness will not be satisfied. Safety properties can be expressed as

[2] Again, this is a subjective empirical claim supported by many researchers [LL90, CM88, Lam93b].

[3] Alpern and Schneider define these precisely in [AS85].

assertions about the states,[4] but liveness properties cannot be. Thus, it is generally much easier to prove safety than liveness.

2.2 Models for Distributed Systems

In the study of computation, it is useful to distinguish between the machine, that is, the computer or network of computers, and the program executing on the machine. The abstract, formal description of the machine is the *model*, while the abstract program is the *algorithm*. A complete formal description of the system consists of the algorithm expressed in the formal model.

2.2.1 Desiderata

A good model for any class of objects should be expressive, accurate, and tractable. That is, it should be easy to describe any object in that class in terms of the model; conclusions derived from the model should reflect truths about the object; and it should be possible to derive interesting properties from the model. Thus, a good model should expose important attributes, and conceal irrelevant details. Different models do not necessarily represent different objects; they may present different views of the same object. So there is not one correct model, but rather, the choice of model depends on the questions one asks.

In particular, since distributed systems are characterized by their behaviors, this should be reflected by the model; the externally observable aspects of the model should be distinguishable. Since distributed systems are often constructed from subsystems, the model should support some notion of *composition*, which allows systems to be put together to form larger systems, in a way that corresponds with our intuition. The model must also reflect the communication mechanism, and the cost and uncertainties associated with it, as well as the behavior of the system in the presence of failures.

2.2.2 Modelling Time in Distributed Systems

A fundamental issue in distributed computing is modelling timing uncertainties, which become important when the processes need to coordinate their actions with each other and,

[4] In some cases, it is necessary to augment the state with *auxiliary variables* which record the history of the execution.

for real-time systems, with the environment. These uncertainties arise from many factors, such as the current load on the computer, the medium for interprocess communication, the reliability of this medium, and the distance separating the computers, and can affect not only the time to communicate, but also the local step times of individual processes.

Synchronous and Asynchronous Models

The simplest possibility, the *completely synchronous* model, ignores these uncertainties, and assumes the processes all take steps together, in distinct rounds, where processes simply wait until all the processes have had a chance to take a step before proceeding to the next round. This greatly simplifies the analysis of distributed systems, and many problems have been studied in this context [LS93]. In these models, the time complexity of an algorithm is usually measured by the number of rounds of communication it takes.

On the other extreme, *asynchronous* models make no timing assumptions, forbidding protocols from using any timing information. This provides a robust model, where algorithms do not depend on any timing conditions that a particular system may not satisfy. Also, for most systems we are interested in, where communication is expensive compared with taking local steps, the timing uncertainty for message delivery is also likely to be large, making it difficult, if not impossible, to synchronize computation using communication. Thus, this model is quite realistic as well, and a lot of research on distributed algorithms has been done in this context [Dij65, Lam74, LT87, Gaw92, LS93]. In this case, the time complexity an algorithm is measured by the number of steps it takes.

However, these models cannot be used for systems which use timing restrictions to rule out certain behaviors, and thus achieve simpler or chaaper solutions for some problems. Moreover, research in the asynchronous setting has yielded many impossibility results, usually giving lower bounds on the resources required to solve certain problems [Lyn89a, BL93]. Many of these results arise in the context of fault-tolerance, where the system is required to solve the problem, even though components may fail in some specified fashion.

Partially Synchronous Models

In recent years, there has been an increased interest in introducing a formal notion of time into distributed models (e.g., [BH81, SL87, DS89, Ost89, Wan91]), and a methodology for proving timing properties. By taking advantage of timing restrictions, a distributed system designer may opt for simpler, more efficient protocols, which would not be possible in the asynchronous setting. And many systems do have some timing assumptions, and can tolerate seriously degraded behavior in the absence of these assumptions (e.g., the *time-out* mechanism in many communication protocols). And even in the asynchronous setting, where performance arguments are largely ad hoc, a good methodology for reasoning about time would be useful.

Modelling timed distributed systems presents several difficulties. In state transition systems, for example, modelling time requires a real valued state variable that varies continuously, or some similar mechanism. This greatly increases the number of states, and requires states to change continuously rather than in discrete steps, as in conventional systems. It is possible, however, to restrict attention to an interesting subset of systems that are modelled sufficiently well by discrete state transition systems.

We would also like our model to capture certain characteristics of time, such as its monotonicity, and exclude from consideration executions that do not correspond to real possibilities. Thus, our model ought to ignore executions in which time reaches a limit or goes backward.

Furthermore, protocols using time often rely on implicit relationships among their timing assumptions to guarantee not only performance, but also correctness, making them difficult to decompose into modular pieces. Even small changes to these assumptions can cause drastic changes in the behavior of a system. Because of this, reasoning about even rather simple systems can be surprisingly difficult, and usually involves checking a lot of details.

However, *liveness properties* play a smaller role in timed systems than they do in untimed systems. Claims that certain events eventually occur are often replaced with stronger claims that they occur within a given amount of time, that is, timing or performance claims. But since time increases without bound, timing properties are merely safety properties. This suggests that assertional reasoning may be especially useful in this setting.

2.2.3 Models for Communication

Since interprocess communication represents the dominant cost in distributed algorithms, the mechanism for communication is typically an important aspect of any distributed model. While there are many different models for communication, most can be viewed as either *shared memory* or *message passing* models.

As implied by the name, processes in a shared memory model communicate via memory registers that can be read and written by multiple processes. This was one of the earliest models used [Dij65]. Because the processes are executing concurrently, it is important to define what happens when two or more processes attempt to write into a register simultaneously. The simplest, and strongest, interpretation is to assume that reading and writing the shared registers happens *atomically*, which means that every read and write operation can conceptually be ordered, even though they may happen at the same time. In some shared memory models, each shared variable can be written only by one process, though it can be read by others, and there are many other variations of this model.

A message passing model more closely matches the intuition of a network of computers, where a process sends a message by placing it in a *message channel*, and another process receives it by removing it from the channel. These models vary in the topology of the network they model (i.e., which other processes a process may send messages to), reliability of the channel (e.g., whether the channel may lose messages, or deliver them out of order) and message delivery delay (i.e., the time between when a message is sent and when it is received).

2.2.4 Other Issues: What we do not model

This thesis is primarily concerned with analyzing the timing behavior of distributed systems, and particularly with developing a methodology to approach proving timing properties using simulations. So, while other issues such as *composition* and *fault-tolerance* are important considerations in our choice of model, we shall not present them here, but rather merely remark on them as they arise.

2.3 Simulations

Simulations form the basis for a powerful class of assertional techniques in which both the system and its specification are modelled by abstract programs or state machines. A system is shown to satisfy or implement its specification by establishing a correspondence, the simulation, between the states of the two machines, such that any step of the system appears identical to some steps of the specification that preserve the simulation. Simulation-based methods have been widely used for verifying safety properties of asynchronous systems, and their value is well established. Lamport [Lam93a] and Lampson [Lam93b] have argued that simulation-based and related techniques are the most practical methods available for verifying concurrent systems.

2.3.1 Advantages of Simulation Proofs

Like invariants, simulations are usually established by induction on the length of an execution, and the induction step is proved for every possible action. This gives simulation proofs a modular decomposition similar to invariance proofs.

If the state is expressed as a collection of state variables, the simulation is often the conjunction of conditions on these variables, which can be checked almost independently. This provides further structure and modularity to the proofs. It also typically makes them more robust. Minor changes in the system or its requirements affect only a few of the conditions, so little additional work needs to be done to accommodate them. This is especially useful in the design of distributed systems.

A simulation proof typically indicates, for every possible step of the implementation, the corresponding step, or sequence of steps, of the specification. Thus, this captures, in a way, some of the intuition used in operational reasoning, expressing more abstractly how a system executes. So while invariants capture the static aspects of the system, simulations express a more dynamic view.

In addition, because there is no syntactical distinction between a system and its specification, it is possible to introduce an intermediate system specification which can be viewed either as a specification for the original system, or an implementation of the original specification. A good intermediate specification highlights the essential features of an execution, and abstracts away the details used to implement these features. A sequence of intermediate specifications can be used to construct a *layered* or *hierarchical* proof, in which each intermediate specification implements its successor, with the original system at the bottom of this "hierarchy" and the original specification at the top.

An intermediate specification may also describe many systems, possibly by capturing some common structure exploited by these systems to solve a problem. This not only is useful for understanding the problem, but also allows the upper layers of the hierarchy to be reused in several proofs.

One apparent disadvantage of simulation proofs is that they tend to be long compared with similar operational arguments. However, this is because they are very complete and explicit, whereas operational proofs often omit the analysis of "trivial" details. Of course, in any proof, suppression of detail is often necessary to make the structure of the proof clear, especially when checking these details is straightforward and unilluminating. It is still possible, indeed desirable, to omit these details in the discussion of the proof, though they should still be checked, lest the proof be incomplete or wrong. That it is easy to detect the omission of details that need to be checked is really an advantage, rather than a handicap.

2.3.2 Proving Timing Properties with Simulations

As noted earlier, assertional techniques are especially promising in the timed setting, because liveness properties, which are difficult to prove, are replaced by timing properties, which express safety rather than liveness. Several researchers have extended their models and methods to handle time and timing conditions, and many argue that the techniques developed for the asynchronous setting carry over into their extended models, so that entirely new techniques need not be invented.

However, most examples in literature have analyzed smaller systems, not using simulations, and not proving timing properties—just correctness.

This thesis builds chiefly on the work by Lynch and Attiya [LA92], applying their techniques to systems with more actions, and which exhibit a greater variety of behaviors. We are interested in any general methods or heuristics to control the complexity of the proofs, particularly for timing properties, taking advantage of the hierarchical structure of simulation proofs. In particular, we discover that it is helpful to look for markers of definite progress (e.g., loop termination), which we call *milestones*, and define intermediate specifications with internal actions that represent reaching these milestones.

2.4 Automated Tools

Automated tools represent an important factor in determining the extent to which techniques to reason formally about distributed systems can be applied. These tools can be used to verify the formal proofs, and detect logical gaps or errors in symbol manipulation that are likely to arise in lengthy proofs. They can also be developed to fill in "trivial" steps, or carry out "similar reasoning" repeatedly in several cases, and thus greatly reduce the boring, repetitive, and tedious work often required for a truly complete proof. This is especially useful for simulation-based proofs, because of their length and the amount of tedious detail that needs to be checked.

2.4.1 Choosing a Tool

There is a wide range of possible useful automated tools, ranging from model-checkers [CG87], which exhaustively search all possible states to verify properties without any human guidance, to programs that simply check the validity of each step in a detailed proof, from specialized provers optimized for a particular domain of applications, to general purpose theorem provers, from provers that halt and request guidance at difficult places in the proof, to those that search silently for solutions.

There are clearly trade-offs in the various choices. Model-checking and decision procedures, for example, work well when the state space is small. However, since theorem proving in general is undecidable, and even when it can be decided, is often computationally intractable, these approaches that attempt to find proofs without guidance may run indefinitely; even deciding when to terminate the search is difficult. Thus, for general theorem proving, some human guidance is necessary. On the other hand, reducing the need for detailed human guidance is one of the major motivations for pursuing automated assistance. A restricted language may allow for increased automation at the expense of expressive power, and thus preclude its use in many contexts. Highly specialized tools may better meet the needs of the users for which it was was designed, but are less likely to find wide applicability.

In addition, automatic provers should be able to reproduce for a human user the reasoning used to derive theorems, ideally in a form that lends insight about the proof. The language understood by the prover must also be reasonably comprehensible to people, so that there is some assurance that what the prover verifies is in fact what was intended. Moreover, because initial attempts often have mistakes, it is important to provide meaningful feedback when a proof cannot be derived, or a claim being verified is in fact not true. When the entire proof is provided, it may be sufficient to simply point out the particular step that fails, but as the program derives more of the proof for itself, it becomes less clear what is useful to a human reader in order to correct the proof.

In this thesis, we use the Larch tools [GH93] to formalize and verify proofs that correspond closely with proofs that we do without automated tools. The system is formalized in the Larch Shared Language, and checked using LP, the Larch Prover [GG91], both enhanced to handle full first-order logic.

2.4.2 The Implications of Time

Since time is a continuous quantity, adding time to the formal model presents additional challenges for automated tools. Tools designed specifically for finite state machines, for example, must cope with the uncountable possibilities introduced by time. In any system, the ability to reason about continuous quantities also needs to be added, preferably in a way that will easily deduce the elementary properties of real numbers. Time bounds also increase the uncertainty in the system, and inequalities are more difficult to handle than equations, especially for tools, such as LP, which rely on rewriting terms into canonical forms based on equations.

2.4.3 Using Automated Tools

Unfortunately, only very modest problems have been analyzed completely using automated tools, and there is a need to evaluate whether the tools can cope with the increased complexity of larger systems, especially when timing information is introduced. One of the major goals of this thesis is to understand how proofs can be designed, and what sort of tools should be developed, in order to improve automated assistance for proving timing properties.

Chapter 3

Models and Methods

In this chapter, we introduce the theoretical models and methods used in this thesis. To simplify the discussion, we omit some structure in the model typically introduced to address some important issues for distributed systems that are not considered in this thesis.

3.1 The I/O Automaton Model

All the work in this thesis is done in the context of *I/O automata*, introduced by Lynch and Tuttle [LT87] to describe asynchronous systems. This model is a simple state transition system, where actions of the system label the transitions between states. The actions may be either *external* or *internal*. An *execution* (or *run*) of the system is a sequence of transitions from one state to another, and a *behavior* of the system is a sequence of external actions labelling transitions in an execution.

One of the primary motivations for using this model is the notion of *composition*, which allows us to build an automaton from smaller automata in a way that corresponds to our intuition, i.e., the resulting automaton behaves as we expect. This leads to the notion of an *action signature*, which describes the interface of an automaton, i.e., how it can be composed with other automata. The action signature distinguishes *input*, *output*, and *internal* actions, where an automaton must always be able to accept any input action (though it may simply ignore it). Composition allows us to model complex distributed systems by building them up out of smaller systems. This provides modularity in our descriptions and our proofs.

In this thesis, however, we are concerned primarily with the issues introduced by timing, rather than by composition, and to simplify the discussion, we do not distinguish between input and output actions, considering them both simply as *external* actions.

Formally, an *I/O automaton* A consists of:[1]

- a set $states(A)$ of states;
- a nonempty subset $start(A)$ of start states;
- a set $acts(A)$ of actions, partitioned into *external* and *internal* actions;
- a set $steps(A)$ of steps, which is a subset of $states(A) \times acts(A) \times states(A)$.

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

There are no restrictions on $steps(A)$. This allows us to easily model nondeterminism and the fact that not all actions may be possible from any state.[2] We say that an action π is *enabled* in a state s if there is a state s' such that $s \xrightarrow{\pi} s'$.

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

Intuitively, an execution represents the entire computation done by the system, up to a certain point, if it is finite. Assertions about the system only need to be true of reachable states, since only reachable states can occur in an execution. Traces correspond to the visible behavior of the automaton, and executions that have the same trace (even on different systems) cannot be distinguished externally. System requirements can only restrict the traces, not the executions themselves, as these may depend on how we choose to model the system.

3.2 MMT Automata

In order to reason about time in concurrent systems, Merritt, Modugno, and Tuttle extended the I/O automaton model by defining *timed executions* and allowing an automaton to

[1] Readers familiar with I/O automata will notice that the *fairness partition* is not included in this definition. This is because fairness is not considered in this thesis. This notion is revived as *tasks* in the section on adding timing information, but with a rather different interpretation.

[2] If we distinguished between input and output actions, we would require that the automaton be *input-enabled*, that is, for every state s and every input action π, there exists a state s' such that $(s, \pi, s') \in states(A)$.

specify some restrictions on these timed executions [MMT91]; we use a special case of their definition [LA92, LV91], which we call *MMT automata*. An MMT automaton partitions the actions into *tasks*,[3] and defines upper and lower bounds on the time it may take to perform each task. A task is considered to be performed by any action in that task. *Timed executions* are simply executions of I/O automata, with each action paired with the time at which it occurs, where time is allowed to be any nonnegative number, and executions are assumed to start at time 0. These times must satisfy the bounds for the tasks, as well as some other conditions that we consider natural for time, i.e., they should not decrease along an execution, nor should infinitely many events happen in a finite interval.

Formally, an *MMT automaton M* consists of:

- an I/O automaton A;
- a finite partition $tasks(M)$ of $acts(A)$;
- two functions, $lower_M : tasks(M) \rightarrow [0, \infty)$ and
  $upper_M : tasks(M) \rightarrow (0, \infty]$
  that satisfy $lower_M(C) \leq upper_M(C)$ for all $C \in tasks(M)$.

We often omit the subscripts when the automaton is clear from context. The states, actions, and steps of M are the same as those of A, i.e., $states(M) = states(A)$, etc. We say that a task C is enabled in a state s if any action in C is enabled in s, i.e., if π is enabled in s for some $\pi \in C$.

A *timed execution* of an MMT automaton is a sequence $s_0(\pi_1, t_1)s_1(\pi_2, t_2)s_2 \ldots$ where $s_0\pi_1 s_1 \pi_2 s_2 \ldots$ is an execution of the underlying I/O automaton, $t_i \leq t_{i+1}$, and t_i satisfies the given *lower* and *upper* bound requirements. Since execution starts at time 0, we define $t_0 = 0$. Formally, if a task C is enabled in s_j, we say that it is *newly enabled* by $s_{j-1} \xrightarrow{\pi_j} s_j$ if π_j is not enabled in s_{j-1} (or $j = 0$), or $\pi_j \in C$. In this case, the following conditions must hold:

Upper bound: If there exists $k > j$ with $t_k > t_j + upper(C)$, then there exists $k' > j$ with $t_{k'} \leq t_j + upper(C)$ such that either $\pi_{k'} \in C$ or C is not enabled in $s_{k'}$.

Lower bound: There does not exist any $k > j$ with $t_k < t_j + lower(C)$ and $\pi_k \in C$.

[3] In the more general theory of I/O automata, these were introduced to address fairness, which does not concern us here. In this context, it is easier to have tasks only in the timed setting.

Intuitively, the upper bound condition says that, whenever a task C is scheduled to be done (i.e., it is enabled), if the time passes beyond its upper bound, then in the interim, either the task is done (i.e., some action $\pi \in C$ occurs), or it was disabled. The lower bound condition says that the task cannot be done before the specified lower bound from the time it was newly enabled. Both of these conditions express safety properties. Like executions for I/O automata, timed executions correspond to the computation done by the system, up to a certain point.

A timed execution is *admissible* if it is infinite and the times associated with the actions increase without bound,[4] or if it is finite and every task C enabled in the final state has no upper bound, i.e., $upper(C) = \infty$. Each timed execution of an MMT automaton M gives rise to a *timed trace*, which is just the subsequence of external actions paired with their associated times. The *admissible timed traces* of an MMT automaton are the timed traces that arise from the admissible timed executions.

An admissible timed execution corresponds to an execution in which time increases without bound; if it is finite, the only tasks which may be enabled in the final state are those which are not required to occur in a bounded amount of time. Thus, admissibility expresses a liveness property. Admissible timed traces describe the visible behavior of the system.

3.3 Timed Automata

The MMT automaton models timing constraints by imposing extra conditions on the executions of I/O automata. The main motivation for this is to provide a clean notion of composition. However, this makes it difficult to use some of the methods developed for proving properties of I/O automata; in particular, it is not obvious how to use simulations to prove timing properties for MMT automata.

Lynch and Attiya [LA92] describe how to incorporate the timing information of an MMT automaton M into the state, yielding an equivalent I/O automaton T of a special form. We call automata derived in this way *timed automata.* This transformation is useful because all the techniques developed for I/O automata can be immediately applied to the timed

[4] Infinite timed executions which only allow a finite amount of time to pass are called *Zeno* executions, after Zeno's paradox, in which to reach his goal, Achilles must take an infinite number of steps, each half the length of the remaining distance, approaching closer but never reaching it each time.

automaton corresponding to an MMT automaton.

The idea is to augment the state with a variable *now* to represent current time, as well as variables $\mathit{first}(C)$ and $\mathit{last}(C)$ for each task C to represent the earliest and latest times when the task must be done. All these variables represent time in absolute, not incremental, terms. A special *time-passage* action is added to allow time to increase, but not beyond any of the deadlines set by the upper bounds. To guarantee the lower bounds, a constraint is added to each action of the MMT automaton.

Formally, each state of T consists of the following components:

$\mathit{basic} \in \mathit{states}(M)$, initially a start state of M

$\mathit{now} \in [0, \infty)$, initially 0, representing the current time

for each task C of M:

$\mathit{first}(C) \in [0, \infty)$, initially $\mathit{lower}_M(C)$ if C is enabled in *basic*, 0 otherwise.

$\mathit{last}(C) \in (0, \infty]$, initially $\mathit{upper}_M(C)$ if C is enabled in *basic*, ∞ otherwise.

The actions of T are the actions of M and a special *time-passage action* ν. The time-passage action is internal, and the other actions are classified as internal or external according to their classification in M.

If $\pi \in \mathit{acts}(M)$, then $s \xrightarrow{\pi}_T s'$ exactly if all the following conditions hold:

- $s.\mathit{basic} \xrightarrow{\pi}_M s'.\mathit{basic}$.
- $s'.\mathit{now} = s.\mathit{now}$.
- For each $C \in \mathit{tasks}(M)$:
  - If $\pi \in C$ then $s.\mathit{first}(C) \le s.\mathit{now}$.
  - $$s'.\mathit{first}(C) = \begin{cases} s.\mathit{first}(C) & \text{if } C \text{ is enabled in both } s.\mathit{basic} \text{ and } s'.\mathit{basic} \\ & \quad \text{and } \pi \notin C \\ s.\mathit{now} + \mathit{lower}_M(C) & \text{if } C \text{ is newly enabled by } s.\mathit{basic} \xrightarrow{\pi}_M s'.\mathit{basic} \\ 0 & \text{if } C \text{ is not enabled in } s'.\mathit{basic} \end{cases}$$
  - $$s'.\mathit{last}(C) = \begin{cases} s.\mathit{last}(C) & \text{if } C \text{ is enabled in both } s.\mathit{basic} \text{ and } s'.\mathit{basic} \\ & \quad \text{and } \pi \notin C \\ s.\mathit{now} + \mathit{upper}_M(C) & \text{if } C \text{ is newly enabled by } s.\mathit{basic} \xrightarrow{\pi}_M s'.\mathit{basic} \\ \infty & \text{if } C \text{ is not enabled in } s'.\mathit{basic} \end{cases}$$

Notice that the three cases for $s'.\mathit{first}(C)$ and $s'.\mathit{last}(C)$ above are mutually exclusive, and cover all the possibilities.

On the other hand, $s \xrightarrow{\nu}_T s'$ exactly if all the following conditions hold:

- $s'.\mathit{basic} = s.\mathit{basic}$.
- $s.\mathit{now} < s'.\mathit{now}$.
- For each $C \in \mathit{tasks}(M)$:
  - $s'.\mathit{now} \leq s.\mathit{last}(C)$.
  - $s'.\mathit{first}(C) = s.\mathit{first}(C)$
  - $s'.\mathit{last}(C) = s.\mathit{last}(C)$.

For notational convenience, we often refer to the tasks of M as tasks of T; we say a task is enabled in $s \in \mathit{states}(T)$ if it is enabled in $s.\mathit{basic}$, and that it is newly enabled by $s \xrightarrow{\pi}_T s'$ (for $\pi \neq \nu$) if it is newly enabled by $s.\mathit{basic} \xrightarrow{\pi}_M s'.\mathit{basic}$. The following lemma gives us necessary and sufficient conditions for an action π to be enabled in a state s of T.

Lemma 3.1 If T is a timed automaton and $s \in \mathit{states}(T)$ then

- $\pi \neq \nu$ is enabled in s if and only if π is enabled in $s.\mathit{basic}$ and $s.\mathit{now} \geq s.\mathit{first}(C)$, where C is the task of π.[5]
- ν is enabled in s if and only if $s.\mathit{now} < s.\mathit{last}(C)$ for all tasks C.

Proof: This follows directly from the definition of the transformation from MMT automata to timed automata. ■

A *timed execution* of a timed automaton is a sequence $s_0(\pi_1, t_1)s_1(\pi_2, t_2)s_2 \ldots$ where $s_0\pi_1 s_1\pi_2 s_2 \ldots$ is an execution and $t_i = s_i.\mathit{now}$ for all i. The *admissible timed executions* are those in which the times associated with the actions increase without bound,[6] and the *admissible timed traces* are the traces of admissible timed executions. Lynch and Attiya prove the following theorem:

Theorem 3.2 An MMT automaton and its corresponding timed automaton have the same admissible timed traces.

[5]If its lower bound has not been reached, a task of a timed automaton may be enabled even though none of its actions are. This is equivalent, but notationally more convenient, to the presentation in [LA92], which checks whether tasks of the underlying MMT automaton are enabled.

[6]This forces there to be an infinite number of time-passage actions in an admissible timed execution.

State
reported $\in \{true, false\}$, initially *false*
countdown $\in \mathbb{N}$, initially k

Actions

External report
Pre: $countdown = 0 \wedge \neg reported$
Eff: $reported \leftarrow true$

Internal decrement
Pre: $countdown > 0$
Eff: $countdown \leftarrow countdown - 1$

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

Figure 3-1: Automaton *Counter*: A Simple Counter

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

Notice also that lower bounds of 0 and upper bounds of ∞ impose no restrictions on the automaton, making the corresponding *first* and *last* variables superfluous. In these cases, we simply omit these variables from the automaton.[7]

3.4 Notational Conventions and an Example

The model is described abstractly, rather than in terms of a particular language or semantics, to allow flexibility and generality. However, we usually describe the state as a collection of state variables, which are modified by the actions. We typically write the actions in *precondition-effect* form, making it easy to determine whether an action is enabled, and if it is, how the new state differs from the old. For the timing information, we simply list the tasks, and the time bounds associated with each. When there is only one action in a task, we often abuse notation by using the name of the action to denote the task as well.

A simple example, given in Figure 3-1, describes an automaton *Counter* which counts down from k, and issues a report when it reaches 0. It has two state variables, *reported* and *countdown*, and two actions, an internal decrement action and an external report action, each in a separate task. It has lower and upper bounds of c_1 and c_2 on the time it can take to make a step, either to decrement its counter if it is not yet 0, or to report if it is already 0.

[7]Formally, we need to prove that this does not change the behavior of the automaton, which follows from Lemma 3.3 proved later in this chapter.

3.5 Invariants

An *invariant* of an automaton is any property that is true in all reachable states. We usually establish an invariant I by proving that all start states satisfy it, and that all steps preserve it, i.e., $start(s) \Rightarrow I(s)$ and $I(s) \wedge (s \xrightarrow{\pi} s') \Rightarrow I(s')$. Often to establish the induction step, we use properties already proven to be invariant, i.e., $start(s) \Rightarrow I'(s)$ and $I(s) \wedge I'(s) \wedge (s \xrightarrow{\pi} s') \Rightarrow I'(s')$, allowing us to break the proof into more manageable pieces.

Timed automata satisfy the following invariants:

Lemma 3.3 In all reachable states of a timed automaton T, and for every task C:

1. $now \leq last(C)$.
2. $first(C) \leq now + lower(C)$.
3. If C is enabled then $last(C) \leq now + upper(C)$.
4. If C is not enabled then $first(C) = 0$ and $last(C) = \infty$.
5. If $upper(C) = \infty$ then $last(C) = \infty$.

Proof: We only give the proof of the first; the rest are similar. We proceed, as indicated above, by induction.

Base Case: In the start state, $now = 0$ and $last(C) = upper(C) > 0$ if task C is enabled, and $last(C) = \infty$ if not. So $now \leq last(C)$ for any task C.

Induction Step: Assuming $s.now \leq s.last(C)$ and $s \xrightarrow{\pi} s'$, we show that $s'.now \leq s'.last(C)$. We consider separately when π is the time-passage action, and when it is not.

Case 1 ($\pi = \nu$): By construction, $s'.now \leq s.last(C) = s'.last(C)$.

Case 2 ($\pi \neq \nu$): By construction, $s'.now = s.now$, and we have the following cases:

Case a (C is enabled in both $s.basic$ and $s'.basic$ and $\pi \notin C$)**:**

$s'.last(C) = s.last(C) \geq s.now = s'.now.$

Case b (C is newly enabled by $s.basic \xrightarrow{\pi} s'.basic$)**:**

$s'.last(C) = s.now + upper(C) = s'.now + upper(C) > s'.now.$

Case c (C is not enabled in $s'.basic$)**:** $s'.last(C) = \infty \geq s'.now$.

■

We can also prove a simple invariant about *Counter*, which says that $reported = false$ unless $countdown = 0$.

Invariant 3.4 For *Counter*: If $reported = true$ then $countdown = 0$.

Proof: (By induction)

Base Case: In the start state, $reported = false$, so this holds vacuously.

Induction Step: Assume that the invariant holds in s and that $s \xrightarrow{\pi} s'$.

Case 1 ($\pi = \mathsf{report}$): $s'.countdown = s.countdown = 0$, so the invariant holds in s'.

Case 2 ($\pi = \mathsf{decrement}$): $s'.reported = s.reported = false$, so the invariant holds vacuously in s'.

Case 3 ($\pi = \nu$): $s'.basic = s.basic$ so this invariant holds inductively.

■

3.6 Simulations

We often express the requirements of a system with a timed automaton that exhibits the allowed behaviors. In this case, a system meets its specification if every behavior exhibited by the system can also be exhibited by the specification. Formally, we say that a timed automaton A *implements* another timed automaton B if every admissible timed trace of A is an admissible timed trace of B. Thus we cannot distinguish A from B simply by observing its behavior. Note that this relationship is not symmetric; B may allow behaviors that A will not exhibit. Two automata are equivalent if each implements the other.

Simulations provide a powerful method to prove that one automaton implements another. There are many variations of simulations [LVara, LVarb], but in this thesis, we only need one of the simplest, the *timed forward simulation*.[8]

Formally, if A and B are timed automata then a *timed forward simulation* from A to B is a relation f between $states(A)$ and $states(B)$ such that:

Time: If $f(s, u)$ then $u.now = s.now$.

Start: If $s \in start(A)$ then there exists some $u \in start(B)$ such that $f(s, u)$.

[8]This is called a *weak timed forward simulation* in [LV91, LVarb, Lyn93].

Step: If $f(s,u)$ for reachable states s and u of A and B, and $s \xrightarrow{\pi}_A s'$, then there exists some u' such that $f(s',u')$ and there is some execution fragment of B from u to u' with the same timed external behavior as $(\pi, s'.now)$.

Notice that the last condition applies only to reachable states of A and B, so we may use any invariants proved for A or B in our proof of the simulation.

We denote $\{u : f(s,u)\}$ by $f[s]$; we usually write $u \in f[s]$ for $f(s,u)$, and we say that u is simulated by s. The key fact about timed forward simulations is expressed in the following theorem:

Theorem 3.5 If there is a timed forward simulation from A to B then A implements B.

Proof: Every admissible timed trace is the trace of some admissible timed execution. Suppose that $\alpha = s_0\pi_1 s_1 \pi_2 s_2 \ldots$ is an admissible timed execution of A. We show that if there is a simulation f from A to B then there is an admissible timed execution α' of B that has the same trace as α.

Let $\alpha_i = s_0\pi_1 s_2 \pi_2 \ldots \pi_i s_i$ for $i = 0, 1, \ldots$. By the start condition, there is a start state u_0 of B such that $f(s_0, u_0)$. We shall construct executions α'_i of B such that α'_i has the same timed trace as α_i. Define $\alpha'_0 = u_0$. Since $s_0 \xrightarrow{\pi_1}_A s_1$, by the step condition, there is some state u_1 such that $f(s_1, u_1)$ and there is some execution fragment α'_1 of B from u_0 to u_1 with the same timed external behavior as $(\pi_1, s_1.now)$. Since u_0 is a start state, α'_1 is an execution of B, and it has the same timed trace as $\alpha_1 = s_0\pi_1 s_1$.

Given an execution α'_{i-1} starting with u_0 and ending in some state $u_{i-1} \in f[s_{i-1}]$ with the same timed trace as α_{i-1}, we recursively define u_i and α'_i as follows: Since s_{i-1} and u_{i-1} are reachable, by the step condition, there is an execution fragment of B starting with u_{i-1} and ending in some state $u_i \in f(s_i, u_i)$ with the same external timed behavior as $(\pi_i, s_i.now)$. We use this execution fragment to extend α'_{i-1} to an execution α'_i that ends in u_i. Thus α'_i has the same timed trace as $\alpha_i = \alpha_{i-1}\pi_i s_i$.

If α is finite, that is $\alpha = \alpha_n$ for some n, then we are done since α'_n has the same timed trace as α. Otherwise, $\alpha' = \lim_{i\to\infty} \alpha'_i$ is an execution of B with the same timed trace as α. Thus A implements B.

$\blacksquare$

State
$reported \in \{true, false\}$, initially $false$

Actions

External report
Pre: $\neg reported$
Eff: $reported \leftarrow true$

Tasks
$\{$report$\}$: $[a_1, a_2]$

Figure 3-2: Automaton *Report*

f is a relation between the states of *Counter* and of *Report*, where $u \in f[s]$ if and only if:

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

Figure 3-3: A Simulation from *Counter* to *Report*

3.7 A Simulation Proof

We conclude this chapter with a very simple simulation proof that illustrates many of the common ideas used with this technique, including some general observations and heuristics, and "natural" interpretations. We prove that the counter automaton from Section 3.4 implements a simpler automaton *Report*, shown in Figure 3-2, that has only a single report action, which must occur within a specified time interval. We show that *Counter* implements *Report* if $a_1 \leq (k+1)c_1$ and $a_2 \geq (k+1)c_2$. This proof is a slightly revised version of proofs in [LA92, Söy94, LSGL94].

We begin by defining a relation f between states of the implementation automaton, in this case *Counter*, and states of the specification automaton, in this case *Report*. This is shown in Figure 3-3. With appropriate assumptions about the timing bounds, we prove that this relation is a simulation from *Counter* to *Report*.

Simulations are often defined like this one, as a list of conditions, one for each state variable of the specification, including one that guarantees the timing condition in the simulation definition. The conditions for the untimed state variables are usually functions of the untimed state variables of the implementation, while the conditions on the time bounds are usually inequalities that guarantee that the specification allows enough time for the implementation to take the steps necessary to simulate the task. We assume here that $k > 0$, so report is not enabled in the start state of *Counter*.

Proof that f is a simulation from *Counter* to *Report* if $[(k+1)c_1, (k+1)c_2] \subseteq [a_1, a_2]$:

Time: By definition of f.

Start: In the start states u_0 and s_0 of *Report* and *Counter*,

$$u_0.now = 0 = s_0.now$$
$$u_0.reported = false = s_0.reported$$
$$u_0.first(\mathsf{report}) = a_1 \leq (k+1) \cdot c_1 = s_0.first(\mathsf{decrement}) + s_0.countdown \cdot c_1$$
$$u_0.last(\mathsf{report}) = a_2 \geq (k+1) \cdot c_2 = s_0.last(\mathsf{decrement}) + s_0.countdown \cdot c_2$$

so $u_0 \in f[s_0]$.

Step: Suppose s and $u \in f[s]$ are reachable states and that $s \xrightarrow{\pi} s'$.

Case 1 ($\pi =$ report)**:** This simulates $u \xrightarrow{\mathsf{report}} u'$.

Since report is enabled in s, we have $s.reported = false$ and $s.countdown = 0$. Thus, $u.reported = false$ and $u.first(\mathsf{report}) \leq s.first(\mathsf{report})$ since $u \in f[s]$, so report is enabled in u, and this is a step of *Report*.

Thus $u'.now = u.now = s.now = s'.now$ and $u'.reported = s'.reported = true$, so $u' \in f[s']$.

Case 2 ($\pi =$ decrement)**:** There is no corresponding step in *Report*. Since decrement is internal, we show $u \in f[s']$.

Because decrement is enabled in s, we have $s.reported = false$ and $s.countdown > 0$. Thus $u.now = s.now = s'.now$, $u.reported = s.reported = s'.reported = false$, and

$$
\begin{aligned}
u.first(\mathsf{report}) &\leq s.first(\mathsf{decrement}) + s.countdown \cdot c_1 && \text{since } u \in f[s], \\
&\leq s.now + s.countdown \cdot c_1 && \text{since decrement occured,} \\
&= s'.first(\mathsf{decrement}) + (s.countdown - 1) \cdot c_1 && \text{by construction,} \\
&= s'.first(\mathsf{decrement}) + s'.countdown \cdot c_1 && \text{by the effect of decrement,} \\
u.last(\mathsf{report}) &\geq s.last(\mathsf{decrement}) + s.countdown \cdot c_2 && \text{since } u \in f[s], \\
&\geq s.now + s.countdown \cdot c_2 && \text{by Lemma 3.3,} \\
&= s'.last(\mathsf{decrement}) + (s.countdown - 1) \cdot c_2 && \text{by construction,} \\
&= s'.last(\mathsf{decrement}) + s'.countdown \cdot c_2 && \text{by the effect of decrement.}
\end{aligned}
$$

Case 3 ($\pi = \nu$)**:** This simulates $u \xrightarrow{\nu} u'$, where $u'.now = s'.now$.

We know $u' \in f[s']$ since $u \in f[s]$, so we only need to verify that $s'.now \leq u.last(\textsf{report})$. If $s.reported = u.reported = true$ then $u.last(\textsf{report}) = \infty$. Otherwise,

$$u.last(\textsf{report}) \geq \left\{ \begin{array}{ll} s.last(\textsf{report}) & \text{if } s.countdown = 0 \\ s.last(\textsf{decrement}) + s.countdown \cdot c_2 & \text{if } s.countdown > 0 \end{array} \right\} \geq s'.now$$

■

Chapter 4

LeLann-Chang-Roberts Election Algorithm

We now consider a simple asynchronous algorithm by LeLann, Chang and Roberts [LeL77, CR79], which solves the *election* problem for a ring network in the message passing model. Although the algorithm is asynchronous, we assume bounds on the communication delay and the local step times in order to prove an upper bound on the time to election. This does not rule out any possible behaviors, and it is commonly done in the timing analysis of asynchronous distributed systems.

4.1 The Election Specification

In the election problem, several essentially similar processes in a network elect a single process from amongst themselves to be the *leader*. This is important when dissimilar tasks need to be performed by the processes, and so the tasks must be distributed among the processes. Once a process has been elected, it can assign the tasks to the other processes.

We index the processes for notational convenience, but the processes do not have access to these indices. This problem is specified by the automaton *Election* in Figure 4-1. Notice that all the leader_i actions constitute a single task, so this is not a distributed description of the system. Exactly one leader action occurs within time t_{report}, after which the automaton takes no further (visible) actions.

State
reported $\in \{true, false\}$, initially *false*

Actions

External leader_i
Pre: $\neg reported$
Eff: $reported \leftarrow true$

Tasks
$\mathsf{leader} = \{\mathsf{leader}_i\}$: $[0, t_{\mathsf{report}}]$

Figure 4-1: Automaton *Election*: A Simple Specification for Leader Election

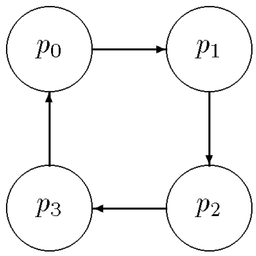

Figure 4-2: A Four Process Unidirectional Communication Ring

4.2 Some Preliminary Definitions

We only consider this problem for the simple case of an asynchronous ring network with unidirectional communication. The processes are arranged in a circle, and each process sends messages only to the process immediately clockwise from it, and receives messages only from the process immediately counterclockwise. We index the processes in an n-process ring by $\mathbb{Z}_n$ (the integers modulo n) increasing clockwise around the ring (see Figure 4-2). The channels are indexed by the processes they link, delivering messages clockwise around the ring.

Although the processes do not know their indices, each process p_i has a unique identifier $\mathrm{UID}_i \in \mathcal{I}$, where $\mathcal{I}$ is totally ordered. Thus, if $i \neq j \pmod{n}$ then either $\mathrm{UID}_i < \mathrm{UID}_j$ or $\mathrm{UID}_j < \mathrm{UID}_i$. We assume each process knows only its own identifier, and nothing about the identifiers of other processes, except that they are different from its own. In particular, the identifiers do not necessarily increase or decrease around the ring. We also assume that p_0 has the maximum identifier, i.e., $\mathrm{UID}_0 = \max\{\mathrm{UID}_j : j \in \mathbb{Z}_n\}$. This last assumption does not change the problem since the processes do not know their own indices, which are

State
For each $i \in \mathbb{Z}_n$
  $status_i \in \{unknown, chosen, reported\}$, initially $unknown$
  $pending_i \in Queues(\mathcal{I})$, initially a queue with only UID_i
  $channel_{i,i+1} \in Queues(\mathcal{I})$, initially empty

Actions

External leader_i
  Pre: $status_i = chosen$
  Eff: $status_i = reported$

Internal $\mathsf{send}_i(m) : m \in \mathcal{I}$
  Pre: m is at the front of $pending_i$
  Eff: m is removed from the front of $pending_i$
    m is added to the back of $channel_{i,i+1}$

Internal $\mathsf{receive}_i(m) : m \in \mathcal{I}$
  Pre: m is at the front of $channel_{i-1,i}$
  Eff: m is removed from the front of $channel_{i-1,i}$
    if $m > \mathrm{UID}_i$ then m is added to the back of $pending_i$
    if $m = \mathrm{UID}_i$ then $status_i \leftarrow chosen$

Tasks
  $\{\mathsf{leader}_i\}$: $[0, l]$
  $\mathsf{send}_i = \{\mathsf{send}_i(m) : m \in \mathcal{I}\}$: $[0, l]$
  $\mathsf{receive}_i = \{\mathsf{receive}_i(m) : m \in \mathcal{I}\}$: $[0, d]$

Figure 4-3: Automaton *LCR*: LeLann-Chang-Roberts Algorithm

used for notational convenience only.

We also assume the channels are reliable and FIFO, that is, on any channel. We model these channels by queues, adding messages sent to the back of the queue, and removing messages received from the front.

For any set M, we denote the set of queues containing elements of M by $Queues(M)$. We write queues as sequences from front to back. In the queue m_1, m_2, m_3, for example, m_1 is at the front, and will be removed next, while m_3 is at the back, and was added last. Given two queues q_1 and q_2, their concatenation $q_1 \circ q_2$ is the queue beginning with the elements from q_1 followed by the elements of q_2. Notice that if an element is removed from the front of q_2 and added to the back of q_1, their concatenation $q_1 \circ q_2$ is unchanged, and that if it is merely removed from q_2, their new concatenation is a subsequence of their original concatenation. We will sometimes treat a queue as the set of its elements, using appropriate notation.

4.3 LeLann-Chang-Roberts Algorithm

The automaton *LCR* in Figure 4-3 expresses a simple protocol proposed by LeLann [LeL77], and improved by Chang and Roberts [CR79], to elect the process with the maximum iden-

State
For each $i \in \mathbb{Z}_n$
$status_i \in \{unknown, chosen, reported\}$, initially $unknown$
$pending_i \in Queues(\mathcal{I})$, initially a queue with only UID_i

Actions

External leader_i
Pre: $status_i = chosen$
Eff: $status_i = reported$

Internal $\mathsf{deliver}_{i-1,i}(m) : m \in \mathcal{I}$
Pre: m is at the front of $pending_{i-1}$
Eff: m is removed from the front of $pending_{i-1}$
if $m > \mathrm{UID}_i$ then m is added to the back of $pending_i$
if $m = \mathrm{UID}_i$ then $status_i \leftarrow chosen$

Tasks
$\{\mathsf{leader}_i\}$: $[0, l]$
$\mathsf{deliver}_{i-1,i} = \{\mathsf{deliver}_{i-1,i}(m) : m \in \mathcal{I}\}$: $[0, l+d]$

Figure 4-4: Automaton *NoChannel*: LeLann-Chang-Roberts Algorithm Without Channels

tifier. Every process sends out its identifier, and waits for it to return around the ring. However, an identifier is discarded by any process with a higher identifier, and a message must be received by every process before returning to its originator. So only the maximum identifier will not be discarded; it will return to its originator, which will then declare itself the leader.

It is easy to see that this algorithm elects a single leader, but it is less clear how long this protocol could take. If the processes sent messages synchronously, it would take n rounds of communication. However, if some processes and channels are slower than others, the messages may "pile-up" at these bottlenecks. Thus a single process may have up to n messages pending, and the last message to be received would be delayed until all the earlier messages are sent. This does not, however, slow down the entire system; in particular, we show that *LCR* implements *Election* if $t_{\mathsf{report}} \geq n(l+d)+l$.

We can simplify our analysis of this algorithm by noticing that from an abstract point of view, there is little distinction between messages about to be sent, i.e., messages in *pending*, and messages already sent but not yet received, i.e., messages in the outgoing *channel*. This suggests a useful intermediate specification, which we develop in the next section

4.4 Eliminating Channels

We define a new automaton *NoChannel*, shown in Figure 4-4, which does not distinguish between messages about to be sent, and messages already sent but not yet received. The two queues of messages are simply concatenated, and the send and corresponding $\mathsf{receive}$ actions are replaced by a deliver action, with bounds to allow enough time to do both actions.

4.4.1 Observations and Invariants

We begin with some informal observations about the system. Recall that we are assuming UID_0 is the maximum identifier, i.e., $\mathrm{UID}_0 > \mathrm{UID}_j$ for all $j \neq 0$. Thus, p_0 will discard every identifier it receives, and no other process will discard UID_0. Because the queues are FIFO, the identifiers are not re-ordered in the queues; each will be received by p_0 according to the position of its process in the ring, unless it is discarded by some other process. In particular, when UID_0 is received by p_0, all other identifiers must already have been discarded. Also, no process other than p_0 will change its *status*, i.e., all other processes will always have $status = unknown$.

To formally state and prove this intuition as invariants of *NoChannel*, let $messages = pending_{n-1} \circ pending_{n-2} \circ \cdots \circ pending_0$, the queue of identifiers that have not been discarded, beginning with the next one to be delivered to p_0. We begin by showing that the identifiers in *messages* may be discarded, but not re-ordered or duplicated, i.e., *messages* is always a subsequence of *messages* in the start state. This is a special case of the following invariant:

Invariant 4.1 For *NoChannel*:
For all k, $pending_k \circ pending_{k-1} \circ \cdots \circ pending_0$ is a subsequence of $\mathrm{UID}_k, \mathrm{UID}_{k-1}, \ldots, \mathrm{UID}_0$.

Proof: (By induction)

Base Case: In the start state, $pending_k = \mathrm{UID}_k$ for all k, so $pending_k \circ \cdots \circ pending_0 = \mathrm{UID}_k, \ldots, \mathrm{UID}_0$.

Induction Step: Assume this holds for s and $s \xrightarrow{\pi} s'$.

Case 1 ($\pi = \mathsf{leader}_i$ or $\pi = \nu$)**:** $s'.pending_k = s.pending_k$ for all k so this holds inductively.

Case 2 ($\pi = \mathsf{deliver}_{i,i+1}(\mathrm{UID}_j)$)**:** The *pending* queues change only in that UID_j is removed from the front of $pending_i$ and possibly added to the back of $pending_{i+1}$, so $s'.pending_k \circ \cdots \circ s'.pending_0$ is a subsequence of $s.pending_k \circ \cdots \circ s.pending_0$, which by the inductive hypothesis, is a subsequence of $\mathrm{UID}_k, \ldots, \mathrm{UID}_0$ for all k.

■

We have the following useful corollary:

Invariant 4.2 For *NoChannel*:
If UID_k is at the front of $pending_{k-1}$ then $k = 0$ and $messages = \mathrm{UID}_0$.

Proof: This follows immediately from the previous invariant, since $\mathrm{UID}_k \notin pending_{k-1} \circ \cdots \circ pending_0 \subseteq \mathrm{UID}_{k-1}, \ldots, \mathrm{UID}_0$, unless $k = 0$, and thus $k - 1 = n - 1$. But then $messages = pending_{n-1} \circ \cdots \circ pending_0$ is a subsequence of $\mathrm{UID}_{n-1}, \ldots, \mathrm{UID}_0$, with UID_0 in front, so $messages = \mathrm{UID}_0$. ■

The next invariant asserts that only p_0 will ever change its status, that is, every other process will always have $status = unknown$.

Invariant 4.3 For *NoChannel*: $status_k = unknown$ for all $k \neq 0$.

Proof: (By induction)

Base Case: In the start state, $status_k = unknown$ for all k.

Induction Step: Assume this holds in s, and that $s \xrightarrow{\pi} s'$.

Case 1 ($\pi = \mathsf{deliver}_{i-1,i}(\mathrm{UID}_i)$)**:** Since UID_i is at the front of $s.pending_{i-1}$, by Invariant 4.2, $i = 0$. So $s'.status_k = s.status_k = unknown$ for all $k \neq i = 0$, by the inductive hypothesis.

Case 2 ($\pi = \mathsf{deliver}_{i-1,i}(\mathrm{UID}_j)$ for $j \neq i$ or $\pi = \mathsf{leader}_i$ or $\pi = \nu$)**:** $s'.status_k = s.status_k$ for all k, so this holds inductively.

■

Finally, we prove that p_0 will know it is the leader only after it has discarded its own identifier, which will be the last one to be discarded.

Invariant 4.4 For *NoChannel*: $status_0 = unknown \Leftrightarrow \mathrm{UID}_0 \in messages \Leftrightarrow messages \neq \emptyset$.

Proof: (By induction)

Base Case: In the start state, $status_i = unknown$ for all i and $\mathrm{UID}_0 \in messages$.

Induction Step: Assume that this holds in s, and that $s \xrightarrow{\pi} s'$.

Case 1 ($s.status_0 \neq unknown$)**:** By the inductive hypothesis, $s.messages = \emptyset$, and so $s'.status_0 \neq unknown$ and $s'.messages = \emptyset$.

f is a relation between states of *LCR* and of *NoChannel*, where $u \in f[s]$ if and only if:

- $u.now = s.now$
- $u.status_i = s.status_i$ for all $i \in \mathbb{Z}_n$
- $u.pending_i = s.channel_{i,i+1} \circ s.pending_i$ for all $i \in \mathbb{Z}_n$
- $u.last(\mathsf{deliver}_{i,i+1}) \geq \begin{cases} s.last(\mathsf{receive}_{i+1}) & \text{if } s.channel_{i,i+1} \text{ is not empty} \\ s.last(\mathsf{send}_i) + d & \text{otherwise} \end{cases}$ for all $i \in \mathbb{Z}_n$
- $u.last(\mathsf{leader}_i) \geq s.last(\mathsf{leader}_i)$ for all $i \in \mathbb{Z}_n$

Figure 4-5: A Simulation from *LCR* to *NoChannel*

Case 2 ($s.status_0 = unknown$)**:** By Invariant 4.3, $s.status_i = unknown$ for all i, so $\pi \neq \mathsf{leader}_i$.

Case a ($\pi = \mathsf{deliver}_{n-1,0}(\mathrm{UID}_0)$)**:** Since UID_0 is at the front of $s.pending_{n-1}$, by Invariant 4.2, $s.messages = \mathrm{UID}_0$. So $s'.status_0 = chosen$ and $s'.messages = \emptyset$.

Case b ($\pi = \mathsf{deliver}_{i-1,i}(\mathrm{UID}_0)$ for $i \neq 0$)**:** Since the identifiers are unique and UID_0 is the maximum, $\mathrm{UID}_0 > \mathrm{UID}_i$. So $\mathrm{UID}_0 \in s'.pending_i \subseteq s'.messages$ and $s'.status_0 = s.status_0 = unknown$.

Case c ($\pi = \mathsf{deliver}_{n-1,0}(\mathrm{UID}_j)$ for $j \neq 0$)**:** By the inductive hypothesis, $\mathrm{UID}_0 \in s.messages$, so $\mathrm{UID}_0 \in s'.messages$ and since the identifiers are unique, $\mathrm{UID}_0 \neq \mathrm{UID}_j$, so $s'.status_0 = s.status_0 = unknown$.

Case d ($\pi = \mathsf{deliver}_{i-1,i}(\mathrm{UID}_j)$ for $i, j \neq 0$)**:** By the inductive hypothesis, $\mathrm{UID}_0 \in s.messages$, both of which are unchanged by this action.

Case e ($\pi = \nu$)**:** $s'.basic = s.basic$ so this holds inductively.

■

4.4.2 *LCR* Implements *NoChannel*

We now show that *LCR* implements *NoChannel* by proving that the relation f defined in Figure 4-5 is a simulation from *LCR* to *NoChannel*. Notice that the timing condition for the $\mathsf{deliver}_{i,i+1}$ task only requires the upper bound to allow enough time for the next send_i action and its corresponding $\mathsf{receive}_{i+1}$, if there are no messages already sent but not yet received, i.e., if $channel_{i,i+1}$ is empty. The following proof is straightforward, and does not even require the invariants proved above.

Proof that f is a simulation from LCR to $NoChannel$:

Time: By definition of f.

Start: If s_0 and u_0 are start states of LCR and $NoChannel$ then $u_0.now = 0 = s_0.now$ and for all $i \in \mathbb{Z}_n$:

$u_0.status_i = unknown = s_0.status_i$

$s_0.channel_{i,i+1}$ is empty

$u_0.pending_i$ and $s_0.pending_i$ both contain only UID_i

$u_0.last(\mathsf{deliver}_{i,i+1}) = l + d = s_0.last(\mathsf{send}_i) + d$

$u_0.last(\mathsf{leader}_i) = \infty$

So $u_0 \in f[s_0]$.

Step: Suppose that s and $u \in f[s]$ are reachable states and that $s \xrightarrow{\pi} s'$.

Case 1 **($\pi = \mathsf{leader}_i$):** This step is simulates $u \xrightarrow{\mathsf{leader}_i} u'$, which has the same external behavior.

a. To see that leader_i is enabled in u, notice

$$\begin{aligned} u.status_i &= s.status_i && \text{since } u \in f[s] \\ &= chosen && \text{since } \mathsf{leader}_i \text{ is enabled in } s \end{aligned}$$

b. We have $u' \in f[s']$, since $u \in f[s]$ and all variables are unchanged except

$u'.status_i = reported = s'.status_i$ and

$u'.last(\mathsf{leader}_i) = \infty = s'.last(\mathsf{leader}_i)$

Case 2 **($\pi = \mathsf{send}_i(\text{UID}_j)$):** There is no corresponding action in $NoChannel$. This has the same external behavior since $\mathsf{send}_i(\text{UID}_j)$ is internal.

We have $u \in f[s']$ since $u \in f[s]$ and $s' = s$ except for $s'.pending_i$, $s'.channel_{i,i+1}$, $s'.last(\mathsf{send}_i)$, and possibly $s'.last(\mathsf{receive}_{i+1})$, and:

a. Since UID_j is removed from the front of $pending_i$ and added to the back of $channel_{i,i+1}$, we have $s'.channel_{i,i+1} \circ s'.pending_i = s.channel_{i,i+1} \circ s.pending_i$.

b. If $s.channel_{i,i+1}$ is empty, i.e., $\mathsf{receive}_{i+1}$ is not enabled in s, then

$$\begin{aligned} u.last(\mathsf{deliver}_{i,i+1}) &\geq s.last(\mathsf{send}_i) + d && \text{since } u \in f[s], \\ &\geq s.now + d && \text{by Lemma 3.3,} \\ &= s'.last(\mathsf{receive}_{i+1}) && \text{since } \mathsf{receive}_{i+1} \text{ is newly enabled.} \end{aligned}$$

c. If $s.channel_{i,i+1}$ is not empty, then $s'.last(\mathsf{receive}_{i+1}) = s.last(\mathsf{receive}_{i+1})$.

Case 3 **($\pi = \mathsf{receive}_i(\text{UID}_j)$):** This simulates $u \xrightarrow{\mathsf{deliver}_{i-1,i}(\text{UID}_j)} u'$, which has the same external behavior because both are internal.

a. Since UID_j is at the front of $s.channel_{i-1,i}$, then UID_j is at the front of $u.pending_{i-1} = s.channel_{i-1,i} \circ s.pending_{i-1}$, so $\mathsf{deliver}_{i-1,i}(\text{UID}_j)$ is enabled in u.

b. We need to check that $u' \in f[s']$. Notice that $s' = s$ except for $channel_{i-1,i}$, $last(\mathsf{receive}_i)$, and possibly either $pending_i$ and $last(\mathsf{send}_i)$ or $status_i$ and $last(\mathsf{leader}_i)$, and $u' = u$ except for $pending_{i-1}$ and $last(\mathsf{deliver}_{i-1,i})$, and possibly either $pending_i$ and $last(\mathsf{deliver}_{i,i+1})$ or $status_i$ and $last(\mathsf{leader}_i)$. UID_j is removed from the front of both $s.channel_{i-1,i}$ and $u.pending_{i-1}$, so $u'.pending_{i-1} = s'.channel_{i-1,i} \circ s'.pending_{i-1}$. If this is not empty, then $u'.last(\mathsf{deliver}_{i-1,i}) = u.now + l + d$, which, by Lemma 3.3, is greater than $s'.last(\mathsf{receive}_i)$ if $s'.channel_{i-1,i}$ is empty, and greater than $s'.last(\mathsf{send}_{i-1}) + d$ otherwise.

If $\mathrm{UID}_j < \mathrm{UID}_i$, this is all that needs to be checked. If $\mathrm{UID}_j = \mathrm{UID}_i$ then $u'.status_i = s'.status_i = chosen$, and $u'.last(\mathsf{leader}_i) = s'.last(\mathsf{leader}_i) = s'.now + l$. If $\mathrm{UID}_j > \mathrm{UID}_i$ then UID_j is added to the back of $pending_i$ (in both s and u), so $u'.pending_i = s'.channel_{i,i+1} \circ s'.pending_i$. If $u.pending_i$ is empty, then $u'.last(\mathsf{deliver}_{i,i+1}) = s'.now + l + d = s'.last(\mathsf{send}_i) + d$. Otherwise, $u'.last(\mathsf{deliver}_{i,i+1}) = u.last(\mathsf{deliver}_{i,i+1})$ which is greater than $s'.last(\mathsf{receive}_{i+1}) = s.last(\mathsf{receive}_{i+1})$ if $s'.channel_{i,i+1} = s.channel_{i,i+1}$ is not empty, and greater than $s'.last(\mathsf{send}_i) + d = s.last(\mathsf{send}_i) + d$ if it is.

Case 4 ($\pi = \nu$)**:** This simulates $u \xrightarrow{\nu} u'$ such that $u'.now = s'.now$, which has the same external behavior.

a. We know $u \xrightarrow{\nu} u'$ because for all i, $s'.now \leq s.last(\mathsf{leader}_i) \leq u.last(\mathsf{leader}_i)$ and

$$s'.now \leq \left\{ \begin{array}{ll} s.last(\mathsf{receive}_{i+1}) & \text{if } channel_{i,i+1} \neq \emptyset \\ s.last(\mathsf{send}_i) + d & \text{if } channel = \emptyset \end{array} \right\} \leq u.last(\mathsf{deliver}_{i,i+1})$$

b. Since $u \in f[s]$ and all variables except now are unchanged, $u' \in f[s']$.

■

4.5 A Template for Synchronous n-round Algorithms

The invariants in the previous section prove that *NoChannel* will never elect any leader other than p_0, but they do not establish that p_0 will actually be elected. For the rest of this chapter, we establish not only that it will be elected, but also an upper bound on the time to election. We do this by viewing the automaton as though it were a synchronous n-round algorithm. We capture this by defining a simulation between *NoChannel* and a

State
  $reported \in \{true, false\}$, initially $false$
  $round \in \mathbb{N}$, initially 0

Actions

External leader_i	**Internal** $\mathsf{increment}$
Pre: $round = n \wedge \neg reported$	Pre: $round < n$
Eff: $reported \leftarrow true$	Eff: $round \leftarrow round + 1$

Tasks

$\mathsf{leader} = \{\mathsf{leader}_i\}$: $[0, l]$	$\{\mathsf{increment}\}$: $[0, \rho]$

Figure 4-6: Automaton *Rounds*: A Template for n-Round Algorithms

simple automaton *Rounds*, shown in Figure 4-6, which keeps track of the rounds already completed.

The increment action signifies the end of an abstract round. At the end of the nth round, the leader task becomes enabled. Notice that, as in *Election*, all the leader actions are in a single task. *Rounds* and *Election* are essentially the same as *Counter* and *Report* in Chapter 3, except that the automaton now counts up to n, rather than down to 0. Also, the single report action is replaced by a set of leader actions, and the bounds on the two tasks of *Rounds* are different. However, the proof that *Rounds* implements *Election* when $t_{\mathsf{report}} \geq n\rho + l$ is identical in form to the proof that *Counter* implements *Report* in Section 3.7, so we do not reproduce it here.

4.6 *NoChannel* Implements *Rounds*

In this section, we show how to simulate the rounds of a synchronous execution. Intutively, one round of communication, which may take up to $l + d$ time to complete, corresponds to a deliver action for every pair of connected processes with a message pending. Because of the asynchrony, one "round" may start before the previous one, even several previous ones, ends. We are only interested in the number of rounds that have completed, which we determine by the "distance" the "slowest" messages have travelled.

4.6.1 Preliminary Definitions and Lemmas

To capture this intuition formally, we introduce the notion of the *reach* of a process, which corresponds to the distance its identifier has travelled. This can be determined by the *pending* queue it appears in, unless it has already been discarded, in which case the reach is defined to be n. The "slowest" identifiers are then those of processes with the minimum reach, and the processes which contain these identifiers in their *pending* queues are the *bottlenecks*. Formally, for any reachable state of *NoChannel*, we define:

- $\mathit{reach}(j)^1 = \begin{cases} i - j \in \{0, 1, \ldots, n-1\} & \text{if } \mathrm{UID}_j \in \mathit{pending}_i \\ n & \text{if } \mathrm{UID}_j \notin \mathit{messages} \end{cases}$

  This is well-defined because, by Invariant 4.1, $\mathrm{UID}_j \in \mathit{pending}_i$ for at most one $i \in \mathbb{Z}_n$.

- $\mathit{minreach} = \min\{\mathit{reach}(j) : j \in \mathbb{Z}_n\}$ is the minimum reach of any process.

- $\mathit{Slowest} = \{j \in \mathbb{Z}_n : \mathit{reach}(j) = \mathit{minreach}\}$ is the set of indices of processes with the "slowest" identifiers.

- $\mathit{Bottlenecks} = \{i \in \mathbb{Z}_n : \mathrm{UID}_j \in \mathit{pending}_i \text{ for some } j \in \mathit{Slowest}\}$ is the set of indices of processes holding the "slowest" identifiers.

Lemma 4.5 The following are true:

- In any state of *NoChannel*, $\mathit{reach}(j) = n$ if and only if $\mathrm{UID}_j \notin \mathit{messages}$ and $\mathit{minreach} = n$ if and only if $\mathit{messages} = \emptyset$ if and only if $\mathit{Bottlenecks} = \emptyset$.

- If $s \xrightarrow{\mathsf{deliver}_{i,i+1}(\mathrm{UID}_j)} s'$ then $s'.\mathit{reach}(j) > s.\mathit{reach}(j)$ and $s'.\mathit{minreach} \geq s.\mathit{minreach}$.

Proof: This follows directly from the definitions of *reach* and *Bottlenecks*. ∎

4.6.2 The Simulation

We now show that *NoChannel* implements *Rounds* by proving that the relation g defined in Figure 4-7 is a simulation from *NoChannel* to *Rounds*. Recall that p_0 has the maximum identifier, so that it will eventually report that it is the leader. Following the intuition above, the round is determined by the minimum reach of any process. The upper bound on the increment action must allow enough time for all of the slowest identifiers to be delivered. By definition, these identifiers are in the *pending* queues of the bottleneck processes.

[1] Note $\mathit{reach}\colon \mathbb{Z}_n \to \mathbb{Z}$ for each state of *NoChannel*.

g is a relation between states of *NoChannel* and of *Rounds*, where $u \in g[s]$ if and only if:

- $u.now = s.now$
- $u.reported \iff s.status_0 = reported$
- $u.round = s.minreach$
- $u.last(\mathsf{leader}) \geq s.last(\mathsf{leader}_0)$ if $s.status_0 = chosen$.
- $u.last(\mathsf{increment}) \geq s.last(\mathsf{deliver}_{i,i+1})$ if $i \in s.Bottlenecks$

Figure 4-7: A Simulation from *NoChannel* to *Rounds*

Proof that g is a simulation from *NoChannel* to *Rounds* if $\rho \geq l + d$:

Time: By definition of g.

Start: In the start states u_0 and s_0 of *Rounds* and *NoChannel*,

$$u_0.now = 0 = s_0.now$$
$$u_0.reported = false \text{ and } s_0.status_0 = unknown \neq reported$$
$$u_0.round = 0 = s_0.minreach \text{ since } reach(j) = 0 \text{ for all } j$$
$$u_0.last(\mathsf{increment}) = \rho \geq l + d = s_0.last(\mathsf{deliver}_{i,i+1}) \text{ for all } i$$

so $u_0 \in g[s_0]$.

Step: Assume s and $u \in g[s]$ are reachable states and that $s \xrightarrow{\pi} s'$.

Case 1 ($\pi = \mathsf{leader}_i$)**:** Since $s.status_i = chosen$, by Invariant 4.3, $i = 0$. This step simulates $u \xrightarrow{\mathsf{leader}_0} u'$ for some u', which has the same external behavior.

a. To see that $leader_0$ is enabled in u, note

$s.messages = \emptyset$	by Invariant 4.4
$s.minreach = n$	by Lemma 4.5
$u.round = n$	since $u \in g[s]$
$\neg u.reported$	since $u \in g[s]$ and $s.status_0 = chosen$

b. We have $u' \in g[s']$ since $u \in g[s]$ and all variables are unchanged except $s'.status = reported$ and $u'.reported = true$, and $u'.last(\mathsf{leader}) = \infty = s'.last(\mathsf{leader}_i)$.

Case 2 ($\pi = \mathsf{deliver}_{i,i+1}(\mathrm{UID}_j)$)**:** Since this is internal, it must simulate internal actions, i.e., $\mathsf{increment}$.

Case a ($j = i + 1$)**:** This step simulates $u \xrightarrow{\mathsf{increment}} u'$ for some u'.

By Invariant 4.2, $j = 0$ and $s.messages = \mathrm{UID}_0$, since UID_j is at the front of $s.pending_{j-1}$. Thus, $s.minreach = n - 1$ and $s'.messages = \emptyset$, $s'.status_0 =$

chosen, $s'.minreach = n$, and $s'.last(\textsf{leader}_0) = s.now + l$. So $\textsf{increment}$ is enabled in u.

To see that $u' \in g[s']$, we check $u'.round = n$ and $u'.reported = false$ since $u \in g[s]$, and $u'.last(\textsf{leader}) = u.now + l = s.now + l$ since $\textsf{leader}$ is newly enabled.

Case b ($s.Slowest = \{j\}$ and $i + 1 \neq j$)**:** This simulates the execution fragment $u \xrightarrow{\textsf{increment}} \cdots \xrightarrow{\textsf{increment}} u'$, where $u'.round = s'.minreach$.

Such an execution fragment exists because $u.reported = false$, and this is not changed by $\textsf{increment}$ actions, and $u.round = s.minreach \leq s'.minreach \leq n$ by Lemma 4.5. Moreover, $s'.reach(j) > s.reach(j)$ by Lemma 4.5, and for all $j' \neq j$, $s'.reach(j') = s.reach(j') > s.reach(j)$, so we have $s'.minreach > s.minreach$, and there is at least one $\textsf{increment}$ in the execution fragment. Since $s.messages \neq \emptyset$, by Invariant 4.4, $\mathrm{UID}_0 \in s.messages$. But UID_0 is not discarded by any action except $\textsf{deliver}_{n-1,0}(\mathrm{UID}_0)$, so $\mathrm{UID}_0 \in s'.messages$, and $s'.minreach < n$.

We have $u' \in g[s']$ because $u' = u$ except $u'.round = s'.minreach$ and $u'.last(\textsf{increment}) = u.now + \rho \geq s'.now + (l + d) \geq s'.last(\textsf{deliver}_{i',i'+1})$ for any i' such that $s'.pending_{i'} \neq \emptyset$.

Case c ($j \in s.Slowest$ but $s.Slowest \neq \{j\}$)**:** There is no corresponding action in *Rounds*. We have $s'.reach(j') = s.reach(j') = s.minreach$ for some $j' \neq j$, so $s'.minreach = s.minreach$. Thus, $s'.Slowest = s.Slowest - \{j\}$, and $s'.Bottlenecks = s.Bottlenecks - \{i\}$, so $u \in g[s']$.

Case d ($j \notin s.Slowest$)**:** Again, there is no corresponding action in *Rounds*. Since $s'.reach(j') = s.reach(j')$ for all $j' \neq j$, we have $s'.Slowest = s.Slowest$. So $s'.Bottlenecks = s.Bottlenecks$ and $u \in g[s']$.

Case 3 ($\pi = \nu$)**:** This simulates $u \xrightarrow{\nu} u'$ where $u'.now = s'.now$, which has the same external behavior.

If $\textsf{leader}$ is enabled in u then $s.status_0 \neq reported$ and $s.minreach = n$ since $u \in g[s]$. By Lemma 4.5, $s.messages = \emptyset$, so by Invariant 4.4, $s.status_0 \neq unknown$, so $s.status_0 = chosen$. Thus $u.last(\textsf{leader}) \geq s.last(\textsf{leader}_0) \geq s'.now$.

If $\textsf{increment}$ is enabled in u then $s.minreach < n$, so $i \in s.Bottlenecks$ for some i by Lemma 4.5, and $u.last(\textsf{increment}) \geq s.last(\textsf{deliver}_{i,i+1}) \geq s'.now$.

And $u' \in g[s']$ since $u \in g[s]$ and all variables except *now* are unchanged.

■

4.7 Discussion

We conclude this chapter with a few remarks about the approach used in this proof. First, we note that the formal details of the proof were much easier to handle because of the hierarchical structure. For example, by introducing *NoChannel*, the difficulty in the proof was isolated mostly to the consideration of the deliver actions in the simulation from *NoChannel* to *Rounds*. Without this intermediate abstraction, much of that proof would need to be repeated for both the send and the receive actions of *LCR*. In addition, because there is no conceptual distinction between messages about to be sent, and messages sent but not yet received, this also simplified the statements of the invariants.

Second, each of the intermediate automata introduced in this example illustrate an important idea that may be useful in many other simulation proofs. The reduction of the buffered FIFO channels to a single queue representing both the *pending* buffer and the *channel* should be applicable to any automaton that has this mechanism. As mentioned, this allows us to reason about a simpler automaton, and makes the proofs clearer.

The *Rounds* automaton illustrated the idea of milestones, which we will see again in the next example. In this case, we view an asynchronous algorithm as running synchronously by specifying the "round" corresponding to each state. To do this, it is useful to track the progress of every message. The current round then corresponds to the "distance" the "slowest" message has travelled. Although the rounds here all have the same time bounds, this is not necessary. The important point is that it must be possible to partition the states so that the automaton never returns to one class of states once it has left it, before reaching the desired goal.

Chapter 5

Fischer's Mutual Exclusion Algorithm

We now consider the *mutual exclusion problem*, in which several processes compete for a *critical resource*. The mutual exclusion requirement demands that at most one process has the resource at any time. Burns and Lynch [BL93] proved that in the asynchronous shared memory model, the mutual exclusion problem for n processes requires at least n atomic read/write shared variables. In this chapter, we examine Fischer's timing-based mutual exclusion algorithm [Fis85, Lam87], using only a single shared variable. In addition to mutual exclusion, we prove an upper bound on the time any process must wait to acquire the resource while it remains unused. Our primary interest in this algorithm is as a test case for the methods we have developed for proving timing properties.

5.1 The Mutual Exclusion Problem Specification

In the mutual exclusion problem, several processes, called *users*, are competing for a *critical resource*, which cannot be used simultaneously by two processes. When a process is using the resource, we say that it is *critical*, or that it is in its *critical* region. When it does not need the resource, it is in its *remainder* region. To manage the resource, the users may have to take additional steps to acquire or release the resource, during which we say a user is in its *trying* or *exit* regions respectively. We assume that once a user has the resource, it will not be interrupted, i.e., it may continue to use the resource until it releases it.

The timed automaton *Mutex* in Figure 5-1 is the specification for a system that not

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

Actions

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

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

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

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

Tasks
$\{\mathsf{try}_i\}$: $[0, \infty]$
$\mathsf{crit} = \{\mathsf{crit}_i : i \in I\}$: $[0, t_{\mathsf{crit}}]$
$\{\mathsf{exit}_i\}$: $[0, \infty]$
$\{\mathsf{rem}_i\}$: $[0, t_{\mathsf{rem}}]$

Figure 5-1: Automaton *Mutex*: A Simple Specification for Mutual Exclusion

only guarantees mutual exclusion, but also upper bounds on the time that any user must spend in its trying region before some user is in its critical region, and the time any user must spend in its exit region. This automaton keeps track of the regions of each of the users (with indices in I), and ensures that at most one user is in its critical region at any time. Notice that all crit actions belong to the same task. Intuitively, this means that if some users are trying to acquire the resource when it is free, then one will succeed within the specified upper bound. This specification is not truly distributed, that is, it does not describe a truly distributed system, because the users can access each other's state.

5.2 Fischer's Mutual Exclusion Algorithm

Fischer proposed a simple timing-based algorithm using only a single $n+1$-valued atomic variable x that can be read and written by all the users. This register can contain any of the users' names, or a special *free* value. For simplicity, we will use a user's index as its name, and 0 as the free value. Intuitively, if a user is critical, the register contains its name, and if every user is in its remainder region, the register is free, i.e., it contains 0. Figure 5-2 contains A familiar pseudocode-style listing of the program executed by each user is given in Figure 5-2, and the corresponding timed automaton *Fischer* is shown in Figure 5-3. User i is in its trying region if $pc_i \in \{testing, set, checking, leave\text{-}trying\}$, and in its exit region if $pc_i \in \{reset, leave\text{-}exit\}$.

Each user trying to obtain the resource first tests the register until it is free, and when

Shared variable: $x \in I \cup \{0\}$, initially 0.

(pc_i)

remainder	*** Remainder Region ***
	try_i
testing	wait until $x = 0$
set	$x \leftarrow i$
	pause
checking	if $x \neq i$ then goto test
leave-trying	crit_i
critical	*** Critical Region ***
	exit_i
reset	$x \leftarrow 0$
leave-exit	rem_i (goto *remainder*)

Figure 5-2: Pseudocode for User i

State

$pc_i \in \{remainder, testing, set, checking, leave\text{-}trying, critical, reset, leave\text{-}exit\}$ for $i \in I$, initially *remainder*

$x \in I \cup \{0\}$, initially 0

Actions

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

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

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

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

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

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

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

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

Tasks

Assume $a < b \leq c$

$\{\mathsf{try}_i\}$: $[0, \infty]$
$\{\mathsf{test}_i\}$: $[0, a]$
$\{\mathsf{set}_i\}$: $[0, a]$
$\{\mathsf{check}_i\}$: $[b, c]$
$\{\mathsf{crit}_i\}$: $[0, a]$
$\{\mathsf{exit}_i\}$: $[0, \infty]$
$\{\mathsf{reset}_i\}$: $[0, a]$
$\{\mathsf{rem}_i\}$: $[0, a]$

Figure 5-3: Automaton *Fischer*: Fischer's Algorithm it is, sets it to its own name. Since several users may be competing for the resource, the user pauses long enough to give every user a chance to set the register. Then, when the register value has stabilized, it checks if the register still contains its name. If it does, the user takes the resource. Otherwise, it returns to testing until the register is free again. The last user to set the register gets the resource, and upon exiting, resets the register to 0.

To maintain mutual exclusion, every user must allow enough time for the register to stabilize before checking it. Otherwise, two users could both test the register and find it free. The faster one could then set and check the register, and enter its critical region before the slower one even manages to set the register. The slow user would then overwrite the register, and find its name still there when it checks. Thus, it would also enter its critical region, violating mutual exclusion. This is avoided by a simple timing restriction that requires each user to allow enough time before checking the register for any other user to set it. Formally, $upper(\mathsf{set}_i) < lower(\mathsf{check}_j)$ for all $i, j \in I$.

Notice that every action is a task by itself, and no user can access the state of any other user, corresponding to the intuition that each user acts independently of the other users. Also, each action reads or writes the shared register at most once, and external actions do not access it at all. This corresponds to the intuition that in one step an atomic read/write register can only be either read or written by a single process. We define time bounds for all the tasks other than try_i and exit_i in order to prove the time bounds for the specification.[1] We wish to prove that *Fischer* implements *Mutex* if $t_{\mathsf{crit}} \geq 5a + 2c$ and $t_{\mathsf{rem}} \geq 2a$.

5.3 Fischer's Algorithm Satisfies Mutual Exclusion

In this section, we demonstrate that Fischer's algorithm satisfies mutual exclusion. This is done entirely by proving some invariants about *Fischer*. But first we note the following useful fact:

Lemma 5.1 For *Fischer*: If $s \xrightarrow{\pi} s'$ and $s'.x \neq 0$ then $\pi = \mathsf{set}_{s'.x}$ or $s.x = s'.x$.

Proof: (By inspection)
Only the set and reset actions modify x, but the reset actions set x to 0, and $s'.x \neq 0$. ■

Next we establish the following easy invariant:

[1]We can show tight, slightly better bounds; see Section 5.8.

Invariant 5.2 For *Fischer*: If $x \neq 0$ then $pc_x \in \{checking, leave\text{-}trying, critical, reset\}$.

Proof: (By induction)

Base Case: In the initial state, $x = 0$, so this holds vacuously.

Induction Step: Assume that s satisfies the invariant, and that $s \xrightarrow{\pi} s'$.

Assume: $s'.x = i \neq 0$.

Prove: $s'.pc_i \in \{checking, leave\text{-}trying, critical, reset\}$

Case 1 ($\pi = \mathsf{set}_i$): $s'.pc_i = checking$

Case 2 ($\pi \neq \mathsf{set}_i$): By Lemma 5.1, $s.x = i$.

By the inductive hypothesis, $s.pc_i \in \{checking, leave\text{-}trying, critical, reset\}$.

But $\pi \neq \mathsf{reset}_i$, so $s'.pc_i \in \{checking, leave\text{-}trying, critical, reset\}$ also.

■

Recall that for this algorithm to work, every user must delay checking the register until all other users have a chance to set it. We only need to show this for the user whose index is currently in the register, since only it can successfully complete the check action and proceed to its critical region. The following invariant captures this crucial intuition.

Invariant 5.3 (Sufficient Confirmation Delay) For *Fischer*:
If $x \neq 0$ and $pc_x = checking$ then $\mathit{first}(\mathsf{check}_x) > \mathit{last}(\mathsf{set}_j)$ for all j such that $pc_j = set$.

Proof: (By induction)

Base Case: This holds vacuously in the initial state.

Induction Step: Assume that it holds in some reachable state s, and that $s \xrightarrow{\pi} s'$.

Assume: $s'.x = i \neq 0$ and $s'.pc_i = checking$.

Prove: $s'.\mathit{first}(\mathsf{check}_i) > s'.\mathit{last}(\mathsf{set}_j)$ for all j such that $s'.pc_j = set$.

Case 1 ($\pi = \mathsf{set}_i$): By the timing restriction and Lemma 3.3, if $s'.pc_j = set$ (i.e., set_j is enabled in $s'.basic$) then $s'.\mathit{first}(\mathsf{check}_i) = s'.now + lower(\mathsf{check}_i) > s'.now + upper(\mathsf{set}_j) \geq s'.\mathit{last}(\mathsf{set}_j)$.

Case 2 ($\pi \neq \mathsf{set}_i$): By Lemma 5.1, $s.x = i$, so $\pi \neq \mathsf{test}_j$ for any j.

If $s'.pc_j = set$ then $s.pc_j = set$ also, so set_j is not newly enabled by $s \xrightarrow{\pi} s'$. Also, $s.pc_i = checking$, so by the inductive hypothesis, $s'.\mathit{first}(\mathsf{check}_i) = s.\mathit{first}(\mathsf{check}_i) > s.\mathit{last}(set_j) = s'.\mathit{last}(set_j)$.

■

We can now prove the invariant that demonstrates mutual exclusion. It says that if any user is in the critical region, or immediately before or after it, then its index is in the variable x and no other user is about to overwrite it. Notice that there is no timing information in the statement of this invariant, though timing information is used in its proof. This invariant is slightly stronger than mutual exclusion, and clearly implies it:

Invariant 5.4 (Strong Mutual Exclusion) For *Fischer*:
If $pc_i \in \{\mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$, then $x = i$ and $pc_j \notin \{\mathit{set}, \mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ for all $j \neq i$.

Proof: (By induction)

Base Case: This holds vacuously in the initial state.

Induction Step: Assume that this holds in some reachable state s, and that $s \xrightarrow{\pi} s'$. If $\pi = \nu$, then $s'.\mathit{basic} = s.\mathit{basic}$, and this holds inductively. So assume that $\pi \neq \nu$.

Case 1 ($s.pc_i \in \{\mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ for some i)**:** $s.x = i$ and for all $j \neq i$, we have $s.pc_j \notin \{\mathit{set}, \mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$.

Case a ($\pi \in \{\mathsf{crit}_i, \mathsf{exit}_i\}$)**:** $s'.pc_j = s.pc_j \notin \{\mathit{set}, \mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ for all $j \neq i$ and $s'.x = s.x = i$, so the invariant holds.

Case b ($\pi = \mathsf{reset}_i$)**:** $s'.pc_j \notin \{\mathit{set}, \mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ for all j (including i), so the invariant holds vacuously.

Case c ($\pi \in \{\mathsf{try}_j, \mathsf{test}_j, \mathsf{check}_j, \mathsf{rem}_j\}$ for some $j \neq i$)**:** $s'.x = s.x = i$, and for all $j' \notin \{i, j\}$, $s'.pc_{j'} = s.pc_{j'} \notin \{\mathit{set}, \mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$, and $s'.pc_j \notin \{\mathit{set}, \mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ since $s.x \notin \{i, j\}$, so the invariant holds in s'.

From the possible values of the pc variables in s, no other actions are enabled.

Case 2 ($s.pc_j \notin \{\mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ for all j)**:**

Case a ($\pi = \mathsf{check}_{s.x}$)**:** Since $s.\mathit{first}(\mathsf{check}_{s.x}) \leq s.\mathit{now} \leq s.\mathit{last}(\mathsf{set}_j)$ for all j, by Invariant 5.3, $s.pc_j \neq \mathit{set}$ for all j. So this holds because $s'.x = s.x$, and $s'.pc_j \notin \{\mathit{set}, \mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ for all $j \neq s.x$.

Case b ($\pi \neq \mathsf{check}_{s.x}$)**:** $s'.pc_j \notin \{\mathit{leave\text{-}trying}, \mathit{critical}, \mathit{reset}\}$ for all j, so this holds vacuously.

■

State

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

Actions

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

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

Internal $\mathsf{stabilize}$
Pre: $status = seized$
Eff: $status \leftarrow stable$

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

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

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

Tasks

$\{\mathsf{try}_i\}$: $[0, \infty]$
$\{\mathsf{seize}\}$: $[0, 3a + c]$
$\{\mathsf{stabilize}\}$: $[0, a]$
$\mathsf{crit} = \{\mathsf{crit}_i : i \in I\}$: $[0, a + c]$
$\{\mathsf{exit}_i\}$: $[0, \infty]$
$\{\mathsf{rem}_i\}$: $[0, 2a]$

Figure 5-4: Automaton *Milestone*: An Intermediate Milestone Automaton

5.4 Milestones: An Intermediate Abstraction

Although Invariant 5.4 guarantees mutual exclusion, it does not bound the time a user may be in its trying region before some user (not necessarily the same one) enters its critical region. Intuitively, it can not be too long, since once any user sets the register, only users that have already tested the register and found it free will overwrite it. Each such user will set the register only once until the register becomes free again, and the last user that sets it will enter its critical region after waiting an appropriate amount of time, and its name will remain in the register until it resets it as it exits.

While we could construct a simulation directly from *Fischer* to *Mutex*, we find it useful to introduce an intermediate level of abstraction which captures this intuition. We define an automaton *Milestone*, shown in Figure 5-4, with actions that correspond to *milestones* toward the goal of some user entering its critical region. We then construct two intuitive simulations, one from *Fischer* to *Milestone*, and one from *Milestone* to *Mutex*, which together establish that every admissible timed trace of *Fischer* is an admissible timed trace of *Mutex*.

We say that the register is *seized* when a user sets it from 0 to its name. This is the first milestone; the register will not be free again until after some user enters its critical region, and resets the register as it exits. Thus, all the users that will set the register must have already tested it. The second milestone occurs when the last user sets the register. At this point we say the register is *stable*; no user will set it again until it has been reset by this user when it exits its critical region. If only one user wants the resource, then when it sets the register, it is both seized and $\mathsf{stabilize}$d. Notice that seize and $\mathsf{stabilize}$ are not actions of individual users, but of the entire system.

Informally, we might reason that a user entering its trying region will seize the register, if it is free, within time $2a$, i.e., enough time to do both test and set the register. Then every user that has already tested the register must set it within time a, after which the register will be stable. Finally, the last user to set the register will check it, and then enter its critical region within time $a+c$. However, this does not take into account the possibility that a user could already be in its trying region when the critical user exits. In this case, the register may not be free for additional time a, after which any user waiting to test the register will do so within time $2a$ as above, for total of time $3a$. But if the only users trying are still waiting to check the register, then it may take an additional time c before any discover their names have been overwritten, and are ready to test the register again. Thus, the upper bound for the seize action is $3a + c$.[2]

We first prove the following easy invariant, which simply states formally that the register is only *seized* or *stable* if some process is making progress towards its critical region, that is, that it is in its trying region, and no other process is critical.

Invariant 5.5 For *Milestone*:
If $status \neq start$ then $region_i = trying$ for some i and $region_j \neq critical$ for all j.

Proof: (By induction)

Base Case: This holds vacuously in the initial state.

Induction Step: Assume that this holds in some reachable state s, and that $s \xrightarrow{\pi} s'$.

Case 1 $(s.status = start)$**:** $\pi \neq \mathsf{stabilize}$ or crit_i for any i.

Case a $(\pi = \mathsf{seize})$**:** $s'.region_i = s.region_i = trying$ for some i, and $s'.region_j = s.region_j \neq critical$ for all j.

[2]This is not tight, and will be improved in Section 5.8.

g is a relation between the states of *Milestone* and of *Mutex*, where $u \in g[s]$ if and only if:

- $u.now = s.now$
- $u.region_i = s.region_i$
- $$u.last(\mathsf{crit}) \geq \begin{cases} s.last(\mathsf{seize}) + 2a + c & \text{if } s.region_i = trying \text{ for some } i, \text{ and} \\ & \quad s.status = start, \text{ and} \\ & \quad s.region_j \neq critical \text{ for all } j, \\ s.last(\mathsf{stabilize}) + a + c & \text{if } s.status = seized, \\ s.last(\mathsf{crit}) & \text{if } s.region_i = trying \text{ for some } i, \text{ and} \\ & \quad s.status = stable. \end{cases}$$
- $u.last(\mathsf{rem}_i) \geq s.last(\mathsf{rem}_i) \quad$ if $s.region_i = exit$

Figure 5-5: A Simulation from *Milestone* to *Mutex*

Case b ($\pi \in \{\mathsf{try}_j, \mathsf{exit}_j, \mathsf{rem}_j\}$ for some j)**:** $s'.status = s.status = start$, so this holds vacuously.

Case 2 ($s.status \neq start$)**:** $s.region_i = trying$ for some i and $s.region_j \neq critical$ for all j, so $\pi \notin \{\mathsf{seize}, \mathsf{try}_i, \mathsf{exit}_i, \mathsf{rem}_i\}$.

Case a ($\pi \in \{\mathsf{try}_j, \mathsf{exit}_j, \mathsf{rem}_j\}$ for some $j \neq i$)**:** $s'.region_i = s.region_i = trying$, $s'.region_j \neq critical$, and for all $j' \neq j$, $s'.region_{j'} = s.region_{j'} \neq critical$. ($\pi$ cannot be exit_j, but we deal with it here rather than make a separate case.)

Case b ($\pi = \mathsf{crit}_j$ for some j)**:** $s'.status = start$, so this holds vacuously.

Case c ($\pi = \mathsf{stabilize}$)**:** $s'.region_j = s.region_j$ for all j so this holds inductively.

■

5.5 *Milestone* Implements *Mutex*

Intuitively, the *seize* and *stabilize* actions are steps the system must take before any user can enter its critical region. We capture this with a relation g in Figure 5-5. The *now*, *region*, and rem conditions are all straightforward. Notice that qualification on the conditions involving seize, stabilize, and crit are their respective enabling conditions. Thus, for example, $u.last(\mathsf{crit}) \geq s.last(\mathsf{seize}) + 2a + c$ requires the crit deadline in *Mutex* to allow enough time for the seize action in *Milestone* plus an additional $2a + c$ time to take the remaining steps necessary to enter the critical region.

Proof that g is a simulation from *Milestone* to *Mutex* if $t_{\mathsf{crit}} \geq 5a+2c$ and $t_{\mathsf{rem}} \geq 2a$:

Time: By the definition of g.

Start: If u_0 and s_0 are start states of *Mutex* and *Milestone*, then $u_0.now = s_0.now = 0$ and for all i, $u_0.region_i = s_0.region_i = trying$ and $u_0.last(\mathsf{crit}) = u_0.last(\mathsf{rem}_i) = \infty$, so $u_0 \in g[s_0]$.

Step: Suppose s and $u \in g[s]$ are reachable states, and $s \xrightarrow{\pi} s'$:

Case 1 ($\pi = \mathsf{try}_i$)**:** This step simulates $u \xrightarrow{\mathsf{try}_i} u'$.

try_i is enabled in u, since $u.region_i = s.region_i = remainder$.

Case a ($u.region_j = s.region_j = trying$ for some j)**:**

Since $s' = s$ and $u' = u$ except that $u'.region_i = trying = s'.region_i$, we have $u' \in g[s']$.

Case b ($u.region_j = s.region_j \neq trying$ for all j)**:**

By Invariant 5.5, $s.status = start$.

Case i ($u'.region_j = u.region_j = s.region_j = critical$ for some j)**:**

$u'.last(\mathsf{crit}) = \infty$ and all other conditions continue to hold.

Case ii ($u.region_j = s.region_j \neq critical$ for all j)**:**

crit is newly enabled in u' and seize is newly enabled in s', so $u'.last(\mathsf{crit}) = u.now + t_{\mathsf{crit}} \geq (s.now + 3a + c) + 2a + c = s'.last(\mathsf{seize}) + 2a + c$.

Case 2 ($\pi = \mathsf{seize}$)**:** There is no corresponding step for *Mutex*.

$s' = s$ except that $s'.status = seized$, $s'.last(\mathsf{seize}) = \infty$, and $s'.last(\mathsf{stabilize}) = s.now + a$. Since $u.last(\mathsf{crit}) \geq s.last(\mathsf{seize}) + 2a + c \geq s.now + 2a + c = s'.last(\mathsf{stabilize}) + a + c$, we have $u \in g[s']$.

Case 3 ($\pi = \mathsf{stabilize}$)**:** Again, there is no corresponding step for *Mutex*.

By Invariant 5.5, $s.region_i = trying$ for some i. Thus, $s' = s$ except that $s'.status = stable$, $s'.last(\mathsf{stabilize}) = \infty$, and $s'.last(\mathsf{crit}) = s.now + a + c$. Since $u.last(\mathsf{crit}) \geq s.last(\mathsf{stabilize}) + a + c \geq s.now + a + c = s'.last(\mathsf{crit})$, we have $u \in g[s']$.

Case 4 ($\pi = \mathsf{crit}_i$)**:** This simulates $u \xrightarrow{\mathsf{crit}_i} u'$.

a. crit_i is enabled in u since $u.region_i = s.region_i = trying$, and $u.region_j = s.region_j \neq critical$ for all j by Invariant 5.5.

b. We have $u' = u$ except $u'.region_i = critical = s'.region_i$ and $u'.last(\mathsf{crit}) = \infty$, so $u' \in g[s']$.

Case 5 ($\pi = \mathsf{exit}_i$)**:** This simulates $u \xrightarrow{\mathsf{exit}_i} u'$.

a. exit_i is enabled in u since $u.region_i = s.region_i = critical$.

b. $u'.region_i = exit = s'.region_i$ and since rem_i is newly enabled in s' and u', $u'.last(\mathsf{rem}_i) = u.now + t_{\mathsf{rem}} \geq s.now + 2a = s'.last(\mathsf{rem}_i)$.

$s'.status = s.status = start$ by Invariant 5.5, so if crit is (newly) enabled in u', then seize is enabled in s' and $u'.last(\mathsf{crit}) = u.now + t_{\mathsf{crit}} \geq (s.now + 3a + c) + 2a + c \geq s'.last(\mathsf{seize}) + 2a + c$ by Lemma 3.3. Otherwise, $u'.last(\mathsf{crit}) = \infty$. So $u' \in g[s']$.

Case 6 ($\pi = \mathsf{rem}_i$)**:** This simulates $u \xrightarrow{\mathsf{rem}_i} u'$.

rem_i is enabled in u since $u.region_i = s.region_i = exit$. Since $u' = u$ except $u'.region_i = remainder = s'.region_i$, and $u'.last(\mathsf{rem}_i) = \infty$, we have $u' \in g[s']$.

Case 7 ($\pi = \nu$)**:** This simulates $u \xrightarrow{\nu} u'$, where $u'.now = s'.now$.

We know $s'.now \leq s.last(C)$ for any task C of *Milestone*, and we show that $s'.now \leq u.last(C)$ for any task C of *Mutex*. If C is not enabled in $u.basic$, or $C \in \{\mathsf{try}_i, \mathsf{exit}_i\}$ for some i, then $u.last(C) = \infty$. Otherwise, we have the following cases:

Case a ($C = \mathsf{crit}$)**:** $s.region_i = u.region_i = trying$ for some i, and $s.region_j = u.region_j \neq critical$ for all j.

Case i ($s.status = start$)**:** $u.last(\mathsf{crit}) \geq s.last(\mathsf{seize}) + 2a + c > s'.now$.

Case ii ($s.status = seized$)**:** $u.last(\mathsf{crit}) \geq s.last(\mathsf{stabilize}) + a + c > s'.now$.

Case iii ($s.status = stable$)**:** $u.last(\mathsf{crit}) \geq s.last(\mathsf{crit}) \geq s'.now$.

Case b ($C = \mathsf{rem}_i$ for some i)**:** $s.region_i = u.region_i = exit$, so $u.last(\mathsf{rem}_i) \geq s.last(\mathsf{rem}_i) \geq s'.now$.

Thus, if $u' = u$ except that $u'.now = s'.now$, then $u \xrightarrow{\nu} u'$ and $u' \in g[s']$.

■

5.6 *Fischer* implements *Milestone*

Recall the intuition we used to define the milestone automaton: The first time the register is set before some user gets the resource corresponds to a seize action, and the last time corresponds to a $\mathsf{stabilize}$ action. We denote by $w(i)$, an upper bound on the time before user i will set the register if it remains free. So if some users are trying to get the resource,

f is a relation between states of *Fischer* and of *Milestone*, where $u \in f[s]$ if and only if:

- $u.now = s.now$
- $u.region_i = s.region_i = \begin{cases} trying & \text{if } s.pc_i \in \{testing, set, checking, leave\text{-}trying\}, \\ critical & \text{if } s.pc_i = critical, \\ exit & \text{if } s.pc_i \in \{reset, leave\text{-}exit\}, \\ remainder & \text{if } s.pc_i = remainder. \end{cases}$
- $u.status = \begin{cases} start & \text{if } s.x = 0 \text{ or} \\ & \quad \text{for some } i,\ s.pc_i \in \{critical, reset\}, \\ seized & \text{if } s.x \neq 0, \text{ and} \\ & \quad \text{for all } i,\ s.pc_i \notin \{critical, reset\}, \text{ and} \\ & \quad \text{for some } i,\ s.pc_i = set, \\ stable & \text{if } s.x \neq 0 \text{ and} \\ & \quad \text{for all } i,\ s.pc_i \notin \{set, critical, reset\}. \end{cases}$
- $u.last(\mathsf{seize}) \geq s.last(\mathsf{reset}_i) + 2a + c \quad \text{if } s.pc_i = reset.$
- $u.last(\mathsf{seize}) \geq s.w(i)$ for some i $\quad$ if $s.x = 0$,

  where $s.w(i) = \begin{cases} s.last(\mathsf{test}_i) + a & \text{if } s.pc_i = testing, \\ s.last(\mathsf{set}_i) & \text{if } s.pc_i = set, \\ s.last(\mathsf{check}_i) + 2a & \text{if } s.pc_i = checking, \\ \infty & \text{otherwise.} \end{cases}$
- $u.last(\mathsf{stabilize}) \geq s.last(\mathsf{set}_i) \quad \text{if } s.pc_i = set.$
- $u.last(\mathsf{crit}) \geq \begin{cases} s.last(\mathsf{check}_x) + a & \text{if } s.pc_x = checking, \\ s.last(\mathsf{crit}_i) & \text{if } s.pc_i = leave\text{-}trying. \end{cases}$
- $u.last(\mathsf{rem}_i) \geq \begin{cases} s.last(\mathsf{reset}_i) + a & \text{if } s.pc_i = reset, \\ s.last(\mathsf{rem}_i) & \text{if } s.pc_i = leave\text{-}exit. \end{cases}$

Figure 5-6: A Simulation from *Fischer* to *Milestone*

and the register is free, the upper bound for seize in a simulated state must allow enough time for some user to set the register. If some user is exiting, but has not yet reset the register, then the simulated state must allow enough time for the user to reset the register, and then for some other user to seize it. Once the register has been seized, we only need to allow enough time for each user that is still going to set the register to do so, and once the register is stable, we only need to wait for the user that wrote last to check the register and then enter its critical region. When a user is exiting, we need to allow enough time for it to reset the register and then leave its exit region. For convenience, we often refer to the *region* of a user in a state of *Fischer*. We capture this intuition with the simulation f in Figure 5-6.

Proof that f is a simulation from *Fischer* to *Milestone*:

Time: By definition of f.

Start: In the start states u_0 and s_0 of of *Milestone* and *Fischer*, $u_0.now = s_0.now = 0$, $u_0.region_i = s_0.pc_i = remainder$ for all i, $u_0.status = start$ and $s_0.x = 0$, and $u_0.last(C) = \infty$ for tasks C of *Milestone*, so $u_0 \in f[s_0]$.

Step: Suppose that s and $u \in f[s]$ are reachable states of *Fischer* and *Milestone* respectively, and that $s \xrightarrow{\pi} s'$:

Case 1 ($\pi = \mathsf{try}_i$)**:** This step simulates $u \xrightarrow{\mathsf{try}_i} u'$.

try_i is enabled in u since $u.region_i = s.region_i = remainder$, and $s' = s$ except that $s'.pc_i = testing$ (and so $s'.region_i = trying$), and $s'.last(\mathsf{test}_i) = s.now + a$. We show that $u' \in f[s']$:

Case a (seize is newly enabled)**:** $u' = u$ except $u'.region_i = trying$ and

$$\begin{aligned} u'.last(\mathsf{seize}) &= u'.now + 3a + c \\ &= s'.now + a + 2a + c \\ &\geq \begin{cases} s'.last(\mathsf{reset}_j) + 2a + c & \text{if } s'.pc_j = reset. \\ s'.last(\mathsf{test}_i) + a \end{cases} \end{aligned}$$

Case b (seize is not newly enabled)**:** $u' = u$ except that $u'.region_i = s.region_i = trying$, and since $s.w(i) = \infty > s'.last(\mathsf{test}_i) + a = s'.w(i)$,

$$\begin{aligned} u'.last(\mathsf{seize}) &= u.last(\mathsf{seize}) \\ &\geq \begin{cases} s'.last(\mathsf{reset}_j) + 2a + c & \text{if } s'.pc_j = reset. \\ s'.w(j) \text{ for some } j & \text{if } s'.x = 0. \end{cases} \end{aligned}$$

Case 2 ($\pi = \mathsf{test}_i$)**:** There is no corresponding step in *Milestone*.

We show that $u \in f[s']$:

Case a ($s.x \neq 0$)**:** $s' = s$ except that $s'.last(\mathsf{test}_i) = s.now + a$. So $f[s'] = f[s]$ since $s'.x = s.x \neq 0$.

Case b ($s.x = 0$)**:** $s' = s$ except that $s'.pc_i = set$, $s'.w(i) = s'.last(\mathsf{set}_i) = s.now + a \leq s.last(\mathsf{test}_i) + a = s.w(i)$, and $s'.last(\mathsf{test}_i) = \infty$. Since $u.status = start$, $\mathsf{stabilize}$ is not enabled in $u.basic$, and the condition for $last(\mathsf{seize})$ is satisfied since for some j, $u.last(\mathsf{seize}) \geq s.w(j) \geq s'.w(j)$.

Case 3 $(\pi = \mathsf{set}_i)$**:** $s.pc_i = set$, $s'.pc_i = checking$, and $s'.x = i \neq 0$, so by strong mutual exclusion, $s'.pc_j \notin \{critical, reset\}$ for all j. We have the following cases:

Case a $(s.x = 0)$**:** seize is enabled in u since $u.status = start$, $u.region_i = s.region_i = trying$, and $u.region_j = s.region_j \neq critical$ for all j; suppose $u \xrightarrow{\mathsf{seize}} u'$, so $u'.status = seized$.

Case i $(s.pc_j \neq set$ for all $j \neq i)$**:** This step simulates $u \xrightarrow{\mathsf{seize}} u' \xrightarrow{\mathsf{stabilize}} u''$.

$u'' = u$ except that $u''.status = stable$, $u''.last(\mathsf{crit}) = s.now + a + c$, and $u''.last(\mathsf{seize}) = \infty$. Since $s.now + a + c$ is greater than any of the time bounds in the condition for $last(\mathsf{crit})$, and $s'.pc_j \neq set$ for all j, we have $u'' \in f[s']$.

Case ii $(s.pc_j = set$ for some $j \neq i)$**:** This step simulates $u \xrightarrow{\mathsf{seize}} u'$.

$u' \in f[s']$ since $s'.pc_j = set$ and $u' = u$ except that $u'.status = seized$, $u'.last(\mathsf{seize}) = \infty$, and $u'.last(\mathsf{stabilize}) = s.now + a \geq s'.last(\mathsf{set}_{j'})$ for all j' such that $s'.pc_{j'} = set$.

Case b $(s.x \neq 0$ and for all $j \neq i$, $s.pc_j \neq set)$**:** This step simulates $u \xrightarrow{\mathsf{stabilize}} u'$.

$\mathsf{stabilize}$ is enabled in u since $u.status = seized$. $u' = u$ except that $u'.status = stable$, $u'.last(\mathsf{stabilize}) = \infty$, and $u'.last(\mathsf{crit}) = s.now + a + c$. Since $s.now + a + c$ is greater than any of the time bounds in the condition for $last(\mathsf{crit})$, and $s'.pc_j \neq set$ for all j, we have $u' \in f[s']$.

Case c $(s.x \neq 0$ and $s.pc_j = set$ for some $j \neq i)$**:** There is no corresponding step in *Milestone*.

$u \in f[s']$ since $u.status = seized$, and $s'.pc_j = set$.

Case 4 $(\pi = \mathsf{check}_i)$**:** There is no corresponding step in *Milestone*.

We show that $u \in f[s']$ in three easy cases:

Case a $(s.x = i)$**:** $s' = s$ except that $s'.pc_i = leave\text{-}trying$, $s'.last(\mathsf{check}_i) = \infty$, and $s'.last(\mathsf{crit}_i) = s.now + a \leq s.last(\mathsf{check}_i) + a \leq u.last(\mathsf{crit})$.

Case b $(s.x = 0)$**:** $s' = s$ except that $s'.pc_i = testing$, $s'.last(\mathsf{check}_i) = \infty$, and $s'.last(\mathsf{test}_i) = s.now + a$, so $s'.w(i) = s.now + 2a \leq s.last(\mathsf{check}_i) + 2a = s.w(i)$. Thus, for some j, $u.last(\mathsf{seize}) \geq s.w(j) \geq s'.w(j)$.

Case c $(s.x \notin \{0, i\})$**:** There is nothing even to check. (In this case, $f[s] \subset f[s']$.)

Case 5 $(\pi = \mathsf{crit}_i)$**:** This simulates $u \xrightarrow{\mathsf{crit}_i} u'$.

$s.pc_i = \textit{leave-trying}$, so by strong mutual exclusion, $s.x = i$ and for all j, $s.pc_j \notin \{set, critical, reset\}$. Thus, $u.status = stable$ and $u.region_i = trying$, so crit_i is enabled in u.

We have $s' = s$ except that $s'.pc_i = critical$ and $s'.last(\mathsf{crit}) = \infty$, and $u' = u$ except that $u'.region_i = critical$, $u'.status = start$, and $u'.last(\mathsf{crit}) = \infty$, so $u' \in f[s']$ since seize, stabilize, and crit are all disabled in $u'.basic$.

Case 6 ($\pi = \mathsf{exit}_i$): This simulates $u \xrightarrow{\mathsf{exit}_i} u'$.

$u.region_i = s.region_i = critical$, so exit_i is enabled in u, $u.status = start$, and seize is not enabled in $u.basic$. We have $s' = s$ except that $s'.pc_i = reset$ and $s'.last(\mathsf{reset}_i) = s.now + a$.

$u' \in f[s']$ since $u' = u$ except that $u'.region_i = exit$, $u'.last(\mathsf{rem}_i) = u.now + 2a = s.now + a + a = s'.last(\mathsf{reset}_i) + a$, and if seize is enabled in $u'.basic$, $u'.last(\mathsf{seize}) = u.now + 3a + c = s.now + a + 2a + c = s'.last(\mathsf{reset}_i) + 2a + c$.

Case 7 ($\pi = \mathsf{reset}_i$): There is no corresponding step in *Milestone*.

$s.pc_i = reset$, so $u.status = start$, and $s' = s$ except that $s'.pc_i = \textit{leave-exit}$, $s'.x = 0$, $s'.last(\mathsf{reset}_i) = \infty$, and $s'.last(\mathsf{rem}_i) = s.now + a \leq s.last(\mathsf{reset}_i) + a \leq u.last(\mathsf{rem}_i)$. If seize is enabled in $u.basic$ then $s.region_j = trying$ for some $j \neq i$ and by strong mutual exclusion, $s.pc_j \neq \textit{leave-trying}$, so $s'.pc_j = s.pc_j \in \{testing, set, checking\}$, and $u.last(\mathsf{seize}) \geq s.last(\mathsf{reset}_i) + 2a + c \geq s.now + 2a + c \geq s'.w(j)$. Otherwise, $u.last(\mathsf{seize}) = \infty$. So $u \in f[s']$.

Case 8 ($\pi = \mathsf{rem}_i$): This simulates $u \xrightarrow{\mathsf{rem}_i} u'$.

rem_i is enabled in u since $u.region_i = s.region_i = exit$. If $u \xrightarrow{\mathsf{rem}_i} u'$ then $u' \in f[s']$ since $u' = u$ and $s' = s$ except that $u'.region_i = remainder = s'.pc_i$ and $u'.last(\mathsf{rem}_i) = \infty = s'.last(\mathsf{rem}_i)$.

Case 9 ($\pi = \nu$): This simulates $u \xrightarrow{\nu} u'$ where $u'.now = s'.now$.

We know $s'.now \leq s.last(C)$ for any task C of *Fischer*. We show that $s'.now \leq u.last(C)$ for any task C of *Milestone*,

If C is not enabled in $u.basic$, or $C \in \{\mathsf{try}_i, \mathsf{exit}_i\}$ for some i, then $u.last(C) = \infty$. Otherwise, we have the following cases:

Case a (C = seize): $u.status = start$ and $u.region_j \neq critical$ for all j, so $s.x = 0$ or $s.pc_i = reset$ for some i.

Case i ($s.pc_i = reset$)**:** $u.last(\mathsf{seize}) \geq s.last(\mathsf{reset}_i) + 2a + c > s'.now$.

Case ii ($s.x = 0$)**:** For some i,

$$u.last(\mathsf{seize}) \geq s.w(i) = \left\{ \begin{array}{ll} s.last(\mathsf{test}_i) + a & \text{if } s.pc_i = testing \\ s.last(\mathsf{set}_i) & \text{if } s.pc_i = set \\ s.last(\mathsf{check}_i) + 2a & \text{if } s.pc_i = checking \\ \infty & \text{otherwise} \end{array} \right\} \geq s'.now$$

Case b (C = $\mathsf{stabilize}$)**:** $u.status = seized$, so $s.pc_i = set$ for some i, and $u.last(\mathsf{stabilize}) \geq s.last(set_i) \geq s'.now$.

Case c ($C = \mathsf{crit}$)**:** $u.status = stable$, so $s.x \neq 0$ and by Invariant 5.2, $s.pc_{s.x} \in \{checking, leave\text{-}trying, critical, reset\}$. But $s.pc_i \notin \{critical, reset\}$ for all i and

$$u.last(\mathsf{crit}) \geq \left\{ \begin{array}{ll} s.last(\mathsf{check}_{s.x}) + a & \text{if } s.pc_{s.x} = checking \\ s.last(\mathsf{crit}_{s.x}) & \text{if } s.pc_{s.x} = leave\text{-}trying \end{array} \right\} \geq s'.now$$

Case d ($C = \mathsf{rem}_i$ for some i)**:** $u.region_i = exit$, so $s.pc_i \in \{reset, leave\text{-}trying\}$ and

$$u.last(\mathsf{rem}_i) \geq \left\{ \begin{array}{ll} s.last(\mathsf{reset}_i) + a & \text{if } s.pc_i = reset \\ s.last(\mathsf{rem}_i) & \text{if } s.pc_i = leave\text{-}exit \end{array} \right\} \geq s'.now$$

■

5.7 Discussion

The intermediate automaton in this example can also be viewed as introducing three "rounds" to entering the critical region. These rounds, corresponding to seizing the register, stabilizing the register, and entering the critical region, have different time bounds, but the key point is that once seized, the register will remain seized until it has been stabilized, and then it will remain stable until some process enters the critical region. These milestones allow us to track the progress of the system, and by bounding the time for each milestone, we can bound the total time to enter the critical region.

More generally, there need not be only one set of milestones, all of which need to be passed. Rather, there could be several alternative paths, each with its own set of milestones. This is similar to the *decrementing function* method of Floyd [Flo67], with milestones corresponding to decrementing the function. Using the milestones as actions of an intermediate automaton allows us to construct hierarchical proofs that are rigorous, modular, and intuitive.

5.8 Achieving Optimal Time Bounds

The upper bound proved for the seize action of *Milestone* is not tight, and thus neither is the bound on t_{crit}. In this section, we give a simulation that establishes an upper bound of $\max(2a+c-b, 3a)$ for the seize action. This yields an upper bound of $\max(4a+2c-b, 5a+c)$ for the time for some user to enter its critical region. This bound is tight because it is possible to construct executions of *Fischer* that reach each of these upper limits. To our knowledge, this bound was not known before.

To establish this tight bound on the seize task, only a few conditions of the simulation need to be modified. The proof that this is still a simulation follows the structure of the original proof, and only a few cases are affected, because our methodology produces a very modular proof.[3] Thus proving the improved bound was very simple, and did not involve any intricate reasoning, but was straightforward to derive from the original proof.

5.8.1 The Slack in the Time Bounds

We can see how the slack in the time bound arises by examining the informal reasoning given in Section 5.4, or the proof of the simulation from *Fischer* to *Milestone*. Recall that the bound for seize was not $2a$ because a user might already be in the trying region. After the critical user exits, the trying user may still not be able to successfully test the register, either because it is very slow and has not yet even checked the register after setting it earlier, or because the register has not yet been reset. The upper bound for seize of $3a + c$ allowed enough additional time both for a trying user to finish checking, and for the exiting user to reset the register.

In the formal proof, the simulation requires that enough time be left after an exiting user resets the register to allow a user still waiting to check the register enough time to do so, as though it had just set the register.

This is not tight for two reasons. First, these effects are not additive, since the users make progress concurrently. Thus, the bound should allow enough time for either possibility, but not for both, i.e., $\max(2a+c, 2a+a)$. Second, a user waiting to check the register must have set the register before the exiting user, which was the last to set the register. Meanwhile,

[3]The simulation from the intermediate automaton with the improved bounds to the mutual exclusion specification automaton also needs to refect the new bounds, but this change is trivial.

the exiting user checked the register, waiting at least time b before doing so, and entered and exited the critical region. Thus, the user waiting to check has already been waiting for at least time b, and thus will wait an additional time of at most $c - b$. Combining these yields the upper bound of $\max(2a + c - b, 3a)$, instead of $3a + c$, for the seize action.

This bound is tight, because there are executions that achieve it. For example, suppose two processes trying to acquire the critical resource set the register at the same time (i.e., there is no time-passage action between the two set actions), and the process which sets the register last waits exactly b time before checking it and proceeding to its critical section. If this process then immediately exits and resets the register, the other process may still take up to $c - b$ time before it checks the register and finds its name overwritten, and an additional a time to test whether the register is free.[4] Then it may take a time to set the register, for a total of $2a + c - b$ to seize the register after the critical resource became available. The $3a$ bound is easily achieved by an exiting process taking a time to reset the register after exiting, and then a trying process taking the full $2a$ time to test and then set the register after it has been reset.

5.8.2 A Proof Sketch of the Improved Bound

To establish the tight upper bound, we need to decouple of the two sources of delay in the simulation, and also prove an invariant that limits the time a user may take to check the register after some other user exits the critical region.

The new simulation, in Figure 5-7, is identical to the one in Figure 5-6, except in the conditions involving the seize actions. When some user is about to reset the register, the upper bound for seize is only required to allow enough time to test and set the register after it is reset. For the other condition, only the qualifier is different, extended to include any case that seize might be enabled in u.

With this change alone, we can prove an upper bound of $2a + c$ for the seize action. However, we can prove the tight upper bound of $\max(2a + c - b, 3a)$ with the following invariant, which says that a user still waiting to check the register while some other user is in its critical region, must have already waited at least time b. Note that if $pc_j = \mathit{checking}$ then $\mathit{last}(\mathsf{check}_j) - c$ represents the time that user j set the register before reaching its

[4]This suggests that if the register is free when a process checks it, it might immediately try to set it, rather than testing it again. This will in fact reduce the upper bound to seize the register to $\max(a + c - b, 3a)$.

f' is a relation between states of *Fischer* and of *Milestone'*, where $u \in f'[s]$ if and only if:

- $u.now = s.now$
- $u.region_i = s.region_i = \begin{cases} trying & \text{if } s.pc_i \in \{testing, set, checking, leave\text{-}trying\}, \\ critical & \text{if } s.pc_i = critical, \\ exit & \text{if } s.pc_i \in \{reset, leave\text{-}exit\}, \\ remainder & \text{if } s.pc_i = remainder. \end{cases}$
- $u.status = \begin{cases} start & \text{if } s.x = 0 \text{ or} \\ & \quad \text{for some } i,\ s.pc_i \in \{critical, reset\}, \\ seized & \text{if } s.x \neq 0, \text{ and} \\ & \quad \text{for all } i,\ s.pc_i \notin \{critical, reset\}, \text{ and} \\ & \quad \text{for some } i,\ s.pc_i = set, \\ stable & \text{if } s.x \neq 0 \text{ and} \\ & \quad \text{for all } i,\ s.pc_i \notin \{set, critical, reset\}. \end{cases}$
- $u.last(\mathsf{seize}) \geq s.last(\mathsf{reset}_i) + 2a \quad \text{if } s.pc_i = reset.$
- $u.last(\mathsf{seize}) \geq s.w(i)$ for some i $\quad$ if $s.x = 0$ or for some j, $s.pc_j = reset$.
- $u.last(\mathsf{stabilize}) \geq s.last(\mathsf{set}_i) \quad \text{if } s.pc_i = set.$
- $u.last(\mathsf{crit}) \geq \begin{cases} s.last(\mathsf{check}_x) + a & \text{if } s.pc_x = checking, \\ s.last(\mathsf{crit}_i) & \text{if } s.pc_i = leave\text{-}trying. \end{cases}$
- $u.last(\mathsf{rem}_i) \geq \begin{cases} s.last(\mathsf{reset}_i) + a & \text{if } s.pc_i = reset, \\ s.last(\mathsf{rem}_i) & \text{if } s.pc_i = leave\text{-}exit. \end{cases}$

Figure 5-7: A Simulation for Proving a Tight Bound for the seize Action

current state.

Invariant 5.6 For *Fischer*:
If $pc_i = critical$ and $pc_j = checking$ then $now \geq last(\mathsf{check}_j) - c + b$.

Proof sketch: Rather than prove this formally here,[5] we sketch a proof following the intuition described at the beginning of this section. If $pc_i = critical$ then by strong mutual exclusion, $x = i$, so user i was the last to set the register. This must have been at least time b earlier, since check_i has a lower bound of b, and if $pc_j = checking$, user j must have set the register before then. That is, if t is the time user j set the register then $t \leq now - b$, and $last(\mathsf{check}_j) = t + c$, yielding $last(\mathsf{check}_j) \leq now - b + c$ as required. ∎

We can now prove that *Fischer* implements *Milestone'*, where *Milestone'* is exactly the same as *Milestone*, except for an upper bound of $2a + \max(c - b, a)$ for the seize task. The

[5] We cannot prove this directly by induction. We first need to prove that if $pc_x = checking = pc_j$ for some $j \neq x$, then $last(\mathsf{check}_j) - c \leq first(\mathsf{check}_x) - b$. Then we strengthen the original invariant: If $pc_i \in \{leave\text{-}trying, critical\}$ then $now \geq last(\mathsf{check}_j) - c + b$.

only substantial differences from the previous proof are in the consideration of the exit and reset actions, but the case when $x \neq 0$ and $pc_j = reset$ for some j also need to be handled for the test and check actions. Other very minor changes are also needed, but these follow in an obvious way from the changes to these conditions and to the upper bound of seize.

Proof sketch that f' is a simulation from *Fischer* to *Milestone'*: We only consider the four cases mentioned above, where this proof differs significantly from the proof in Section 5.6. The changes for the test, check, and exit actions arise from the requirement that $u.last(\mathsf{seize}) \geq s.w(i)$ when $s.pc_j = reset$ for some j, even if $s.x \neq 0$. This simplifies the analysis for the reset action. The case for the exit action is identical to the original proof except for the last line, where Invariant 5.6 is used to establish $s'.w(j) = s'.last(\mathsf{check}_j) + 2a \leq s'.now + 2a + c - b$ when $s'.pc_j = checking$.

1. If $\pi = \mathsf{test}_i$ and $s.x \neq 0$ and $s.pc_j = reset$ for some j then $u.last(\mathsf{seize}) \geq s.last(\mathsf{reset}_j) + 2a \geq s.now + 2a = s'w(i)$.

2. If $\pi = \mathsf{check}_i$ and $s.x \notin \{0, i\}$ and $s.pc_j = reset$ for some j then $f'[s] \subset f'[s']$ as before, since for all j', $s'.w(j') \leq s.w(j')$.

3. If $\pi = \mathsf{exit}_i$ then $u.region_i = s.region_i = critical$, so exit_i is enabled in u, $u.status = start$, and seize is not enabled in $u.basic$. We have $s' = s$ except that $s'.pc_i = reset$ and $s'.last(\mathsf{reset}_i) = s.now + a$. If $u \xrightarrow{\mathsf{exit}_i} u'$, then $u' \in f'[s']$ since $u' = u$ except that $u'.region_i = exit$, $u'.last(\mathsf{rem}_i) = u.now + 2a = s.now + a + a = s'.last(\mathsf{reset}_i) + a$, and if seize is enabled in $u'.basic$,

$$
\begin{aligned}
u'.last(\mathsf{seize}) &= u.now + 2a + \max(c - b, a) \\
&\geq \begin{cases} s'.last(\mathsf{reset}_i) + 2a \\ s'.w(j) \text{ for any } j \text{ such that } s'.pc_j \in \{testing, set, checking\} \end{cases}
\end{aligned}
$$

4. If $\pi = \mathsf{reset}_i$ then $s.pc_i = reset$, so $u.status = start$, and $s' = s$ except that $s'.pc_i = leave\text{-}exit$, $s'.x = 0$, $s'.last(\mathsf{reset}_i) = \infty$, and $s'.last(\mathsf{rem}_i) = s.now + a \leq s.last(\mathsf{reset}_i) + a \leq u.last(\mathsf{rem}_i)$. For some j, $u.last(\mathsf{seize}) \geq s.w(j) = s'.w(j)$, So $u \in f'[s']$.

■

Chapter 6

Automated Proof Assistance

In this chapter, we explore how automated tools can be used to assist in simulation proofs in the style of the previous chapters. This builds mainly on work done by Söylemez [Söy94] and Søgaard-Andersen, Garland, Guttag, Lynch, and Pogosyants [SGG+93]. In particular, we formally verify the proof in Chapter 5 using LP. As the proof is very lengthy and repetitive, we consider only the salient features of the proof and the automation process; the full proof can be found in the appendix.

6.1 The Larch Tools

Larch is a family of tools intended to support formal specification and verification in programming. Since we are verifying abstract systems, rather than particular programs, we use only two tools from this family, the Larch Shared Language (LSL) and the Larch Prover (LP). In LSL, we write machine-readable definitions of our model and the abstract systems we are modelling, including the requirements they are expected to satisfy. We then use LP to reason about these systems and to prove that the required properties are guaranteed by the system.

6.1.1 The Larch Shared Language

The basic unit of specification in LSL is a *trait*, which introduces types, called *sorts*, and functions, called *operators*, that act on the sorts. Properties of these sorts and operators are expressed by assertions in the trait, using first-order logic. Typically, a trait defines a single concept or data type. Complex traits are often built using simpler traits, introducing a hierarchy of traits, which matches our mathematical understanding of the concepts. The Larch tools include a library of LSL traits formalizing many common concepts in discrete mathematics.

Two important characteristics of LSL are that sorts are disjoint, and that operators always represent total functions, though the value of the function may not be constrained over the whole domain. Also, the domain and range of operators are always sorts, specified when the operators are introduced. Thus, any trait can be checked for syntactic correctness in much the same way type-checking is done in many programming languages. This catches many of the simple mistakes made when writing these traits. We use an LSL checker to do this, as well as to generate input for LP, which will be discussed further in the next section.

Assertions are typically either logical expressions that are always true, or equations expressing the equivalence of two expressions of the same sort. It is also possible to make two other types of assertions about sorts. A sort is *generated by* a set of operators if every element of that sort can be derived by a finite application of the operators. This justifies structural induction, which cannot be expressed otherwise by a finite set of assertions in first-order logic. We may further specify that the sort is generated *freely* if every element generated by the operators is distinct. A sort is *partitioned by* a set of operators if distinct elements can always be distinguished by at least one of the operators.

A trait may define a data structure by declaring a new sort and asserting appropriate axioms. The sorts and traits may be parameterized, supporting a form of polymorphism. The LSL checker also understands shorthands for a few common data structures in computer science, such as records (called *tuples*) and enumerations, and it automatically generates the appropriate axioms.

A trait may also list useful consequences of its axioms in a special section called the *implies* clause. The intent is that these can be derived from the axioms. However, it is useful to list them explicitly since they often express desired properties of a trait. The LSL checker generates *proof obligations* for these implications.

6.1.2 The Larch Prover

The LSL checker generates input files for LP from the LSL traits, including a file of proof obligations. This translation is straightforward because LP understands equations, as well as "generated by" and "partitioned by" assertions. Logical assertions are interpreted as equations where the expression equals *true*.

Unlike the LSL checker, LP is an interactive tool; it processes each command before reading the next one. In addition to assertions, which declare new facts to LP, there are commands to introduce proof obligations, to provide guidance to LP when doing a proof, to query LP about its state, and to control the way in which LP works automatically.

LP is a general purpose proof assistant, which attempts to rewrite terms into canonical forms, so that logically equivalent expressions become syntactically identical. This is called *normalization*. LP converts the equations given into rewrite rules. To prevent these rules from being applied endlessly, LP defines a partial order, called the *registry*, on the operators, and rewrites higher operators to lower ones. Finding an appropriate partial order for the operators and rewriting the assertions comprise the bulk of the automatic work that LP does.

In contrast to LSL, LP views all facts equally—there is no hierarchy of traits. Instead, each assertion is given the name of the trait it was derived from, appended with a number to distinguish it from the other assertions of that trait. The statements to be proven are typically given a different name, to distinguish them from the other assertions. Whenever LP derives a new fact from an old one, it appends another number to its name.

Proof obligations in LP are called *conjectures*. LP considers a conjecture proved if it can normalize it to *true*. Rather than searching for a proof, LP normalizes the conjecture and all the facts it knows, and then relies on guidance from the user. It may be enough to point out particular facts or instances of general facts that can be used to further rewrite the conjecture. Often, however, it is necessary to direct LP to consider several cases, or to proceed by induction, or when trying to prove an implication, to assume the hypothesis and attempt to prove the conclusion. These are called *proof methods*.

It is also possible to direct LP to automatically try these proof methods before prompting the user for guidance. The trade-off here is that if the wrong methods are chosen, the proof may evolve in some totally inappropriate fashion before LP discovers that it cannot continue; the proof must then be backed out to some earlier stage, where the appropriate proof methods can be applied. It may also be much more difficult, at that point, to even understand what has gone wrong. This is much like compiling programs in languages that lack sufficient redundancy for the compiler to discover the error before it is significantly past the point the error occured.

Some proof methods, such as case analysis, introduce additional assumptions for parts of the proof, which need not even be consistent with the other assertions. These assumptions are often particularly relevant to the proof, and LP names each with a suffix of `XxxxHyp`, where the `Xxxx` indicates the proof method that introduced it.

LP can be run in "batch mode" by recording all the commands in a *script* file, which is then executed. Each command is processed in turn, exactly as if it had been entered interactively by a user. (An error, however, stops the execution of a script.) The current state of LP can also be saved in *freeze* files, which can later be *thawed*, so that some common work can be reused in several proofs. When executing a script, LP can also do *box checking*, in which every time a proof method is invoked to prove the current conjecture, LP checks that the file has certain marks that indicate that this was in fact intended, and every time the current conjecture is established, LP checks the file for other marks that indicate that this too was expected.

6.2 Machine-Readable Definitions

6.2.1 General Traits for Timed Automata

We begin by developing a library of traits that define the general notions used in timed simulation proofs. This is analogous to the the development of the model in Chapter 3. These traits can be reused in proofs similar to the ones in this thesis.

We begin with the definition of an I/O automaton `A` in Figure 6-1. The `enabled` and `effect` predicates are intended to support the use of precondition-effect form for specifying the transition relation of the automaton. Because sorts are determined by syntax alone, execution fragments are defined as those "step sequences" that satisfy the `execFrag` predicate. The start states and execution fragments are defined using predicates (`start` and `execFrag`) rather than sets, since predicates are easier to handle in LP. States, actions, and step sequences are parameterized by the automaton. However, traces represent external behavior, and thus must be comparable between automata. Since sorts are disjoint, the `common` operator is necessary to map the external actions to a common sort `CommonActions`.

We also define invariants in Figure 6-2. Notice that the operator `inv` is introduced in the `Automaton` trait, but it is not used in that trait. This is because invariants vary among automata. However, each automaton can define its own invariant, and then use the

```
Automaton (A): trait
  introduces
    start       : States[A]                         → Bool
    enabled     : States[A], Actions[A]             → Bool
    effect      : States[A], Actions[A], States[A]  → Bool
    isStep      : States[A], Actions[A], States[A]  → Bool
    isExternal  : Actions[A]                        → Bool
    isInternal  : Actions[A]                        → Bool
    {__}        : States[A]                         → StepSeq[A]
    __{__,__}   : StepSeq[A], Actions[A], States[A] → StepSeq[A]
    first, last : StepSeq[A]                        → States[A]
    execFrag    : StepSeq[A]                        → Bool
    task        : Actions[A]                        → Tasks[A]
    enabled     : States[A], Tasks[A]               → Bool
    inv         : States[A]                         → Bool
    common      : Actions[A]                        → CommonActions
    empty       :                                   → Traces
    __ ^ __     : Traces, CommonActions             → Traces
    trace       : Actions[A]                        → Traces
    trace       : StepSeq[A]                        → Traces
  asserts
    sort Traces generated by empty, ^
    ∀ a: Actions[A], s, s': States[A], ss: StepSeq[A], c: Tasks[A]
      isInternal(a)    ⇔ ¬isExternal(a);
      isStep(s, a, s') ⇔ enabled(s, a) ∧ effect(s, a, s');
      enabled(s, c)    ⇔ ∃ a (enabled(s, a) ∧ task(a) = c);

      first({s}) = s; first(ss{a,s}) = first(ss);
      last({s}) = s;  last(ss{a,s}) = s;
      execFrag({s});  execFrag(ss{a,s'}) ⇔ execFrag(ss) ∧ isStep(last(ss), a, s');
      trace({s}) = empty;
      trace(ss{a,s}) = (if isExternal(a) then trace(ss) ^ common(a) else trace(ss));
      trace(a) = (if isExternal(a) then empty ^ common(a) else empty);
```

Figure 6-1: Larch Trait Defining Untimed I/O Automata

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

Figure 6-2: Larch Trait Defining Invariants

```
Time (T): trait
  includes TotalOrder(T), Natural(- for ⊖), AC(+, T)
  introduces
    0, infinity :        → T
    __ + __     : T, T → T
    __ * __     : N, T → T
  asserts
    ∀ t, t1, t2: T, n: N
      0 ≤ t; t ≤ infinity;
      0 + t = t;
      t1 + t2 ≠ infinity ⇔ t1 ≠ infinity ∧ t2 ≠ infinity;
      0 * t = 0;
      succ(n) * t = (n * t) + t;
      t ≤ (t + t1);
      t ≠ infinity ⇒ ((t + t1) < (t + t2) ⇔ t1 < t2);
      t ≠ infinity ⇒ (t + t1 = t + t2 ⇔ t1 = t2);
  implies
    ∀ t, t1, t2: T, b: Bool
      infinity + t = infinity;
      t < infinity ⇔ t ≠ infinity;
      t ≠ infinity ⇒ ((t + t1) ≤ (t + t2) ⇔ t1 ≤ t2);
      (if b then t1 else t2) = t ⇔ (if b then t1 = t else t2 = t);
      (if b then t1 else t2) < t ⇔ (if b then t1 < t else t2 < t);
      (if b then t1 else t2) > t ⇔ (if b then t1 > t else t2 > t);
      (if b then t1 else t2) ≤ t ⇔ (if b then t1 ≤ t else t2 ≤ t);
      (if b then t1 else t2) ≥ t ⇔ (if b then t1 ≥ t else t2 ≥ t);
```

Figure 6-3: Larch Trait for Time

`Invariants` trait to express that it is in fact an invariant.

We then axiomatize time and boundmaps using three traits. We model time as non-negative reals extended with infinity. The `Time` trait in Figure 6-3 captures the properties desired.[1] The `Bounds` trait in Figure 6-4 is a tuple of lower and upper bounds, with some convenient operators. The `BoundMap` trait in Figure 6-5 specifies the mapping that assigns time bounds to each task of an automaton. Recall that an MMT automaton is merely an I/O automaton together with an appropriate boundmap.

The `TimedAutomaton` trait in Figure 6-6 defines the timed automaton `TA` corresponding to a I/O automaton `A` and a boundmap `b`. This straightforwardly expresses the transformation from MMT automata to timed automata described in Chapter 3.

Finally, the `TimedForward` trait in Figure 6-7 captures the definition of a timed forward simulation from one automaton to another. For this to be meaningful, the automata are required to have a `now` component in their state. This trait is also parameterized by in-

[1] We are doing concurrent research to use decision procedures to handle time more easily [Pog95]; this was presented in [LSGL94].

```
Bounds: trait
  includes Time(Time)
  Bounds tuple of first: Time, last: Time
  introduces
    __ + __   : Bounds, Time → Bounds
    unbounded :              → Bounds
  asserts
    ∀ b: Bounds, t: Time
      b + t = [b.first + t, b.last + t];
      unbounded = [0, infinity];
```

Figure 6-4: Larch Trait for Expressing Lower and Upper Bounds

```
BoundMap(A,b): trait
  includes Bounds
  introduces
    b : Tasks[A] → Bounds
  asserts
    ∀ c: Tasks[A]
      b(c).first < infinity;
      b(c).first ≤ b(c).last;
```

Figure 6-5: Larch Trait Defining a Boundmap for an Automaton

variants for each automaton, since the step condition only needs to be proved for reachable states.

6.2.2 The Automata and Simulations

We can now specialize these general traits to define the particular automata and simulations we used to verify Fischer's algorithm. Each automaton is defined by two traits, the first specifying the untimed components, and the second, adding the timed aspects. We begin by listing in Figure 6-8 the common actions that appear in the traces. An action is indexed by the process that performs it.

The `AutomatonMutex` trait in Figure 6-9 specifies the untimed behavior required for any mutual exclusion algorithm. An action is specified by its type and the index of its process, and the transition relation is specified by `enabled` and `effect` predicates for each action. The `unchanged` predicate is used to specify that any action changes only the state of its process. Notice that the `crit` actions are in a single task, and all the other actions are in classes by themselves. The `TimedMutex` trait in Figure 6-10 gives the time bounds on each of the tasks.

```
TimedAutomaton (A, b, TA): trait
  assumes Automaton(A), BoundMap(A,b)
  includes Automaton(TA), Bounds, FiniteMap(Bounds[A], Tasks[A], Bounds, __[__] for apply)
  States[TA] tuple of basic: States[A], now: Time, bounds: Bounds[A]
  introduces
    nu      : Time       → Actions[TA]
    addTime : Actions[A] → Actions[TA]
  asserts
    sort Actions[TA] generated freely by nu, addTime
    ∀ s, s': States[TA], c: Tasks[A], a: Actions[A], t: Time
      defined(s.bounds, c);

      isInternal(nu(t));
      isInternal(addTime(a)) ⇔ isInternal(a);
      common(addTime(a)) = common(a);

      start(s) ⇔ start(s.basic) ∧ s.now = 0
                    ∧ ∀ c (  (¬enabled(s.basic, c) ⇒ s.bounds[c] = unbounded)
                            ∧ (enabled(s.basic, c) ⇒ s.bounds[c] = b(c)));

      enabled(s, nu(t))    ⇔ s.now ≤ t ∧ t < infinity ∧ ∀ c (t ≤ (s.bounds[c]).last);
      effect(s, nu(t), s') ⇔ s'.now = t ∧ s'.basic = s.basic ∧ s'.bounds = s.bounds;

      enabled(s, addTime(a)) ⇔ enabled(s.basic, a) ∧ (s.bounds[task(a)]).first ≤ s.now;
      effect(s, addTime(a), s') ⇔
        s'.now = s.now
         ∧ effect(s.basic, a, s'.basic)
         ∧ ∀ c (s'.bounds[c] =
                  (if ¬enabled(s'.basic, c) then unbounded
                    else if enabled(s.basic, c) ∧ task(a) ≠ c then s.bounds[c]
                    else b(c) + s.now)
                 );

      inv(s) ⇔
          ∀ c (  s.now ≤ (s.bounds[c]).last
               ∧ (¬enabled(s.basic, c) ⇒ s.bounds[c] = unbounded)
               ∧ (enabled(s.basic, c) ⇒ (s.bounds[c]).last ≤ (s.now + b(c).last))
               ∧ (s.bounds[c]).first ≤ (s.now + b(c).first)
               ∧ (b(c).last = infinity ⇒ (s.bounds[c]).last = infinity))
           ∧ s.now < infinity
           ∧ inv(s.basic);
 implies
   Invariants(TA, inv)
   ∀ s, s': States[TA], a:Actions[TA], c:Tasks[A]
      isStep(s, a, s') ∧ inv(s)
        ⇒ (enabled(s.basic, c) ⇒ (s.bounds[c]).last ≤ (s'.bounds[c]).last);
```

Figure 6-6: Larch Trait for Generating Timed Automata from MMT Automata

```
TimedForward (A1, A2, f, I1, I2): trait
  assumes Automaton(A1), Automaton(A2), NowExists(A1), NowExists(A2),
          Invariants(A1, I1), Invariants(A2, I2)
  introduces
    f  : States[A1], States[A2] → Bool
    I1 : States[A1]             → Bool
    I2 : States[A2]             → Bool
  asserts
    ∀ s, s': States[A1], u: States[A2], a: Actions[A1], alpha: StepSeq[A2]
      f(s, u) ⇒ u.now = s.now;
      start(s) ⇒ ∃ u (start(u) ∧ f(s, u));
      f(s, u) ∧ inv(s) ∧ inv(u) ∧ I1(s) ∧ I2(u) ∧ isStep(s, a, s')
       ⇒ ∃ alpha (execFrag(alpha) ∧ first(alpha) = u ∧ f(s', last(alpha))
                  ∧ trace(alpha) = trace(a))
```

Figure 6-7: Larch Trait Defining Timed Forward Simulations

```
CommonActions: trait
  CommonActionTypes enumeration of try, crit, exit, rem
  introduces
    __[__] : CommonActionTypes, UID → CommonActions
  asserts sort CommonActions generated freely by __[__]
```

Figure 6-8: Larch Trait Listing the Common Actions

Likewise, the `AutomatonFischer` and `TimedFischer` traits in Figures 6-11 and 6-12 specify the untimed and timed aspects of Fischer's mutual exclusion algorithm. That each of the actions is in a class by itself can be derived from the assertion that `Tasks[F]` is generated freely by `task`. The `TimedFischer` trait also defines the *sufficient confirmation delay* and *strong mutual exclusion* invariants used in the proof of the simulation.

The automaton expressing the milestones in the algorithm is defined in Figures 6-13 and 6-14. Notice that `seize` and `stabilize` generate actions without any process index, since they are actions of the whole system. The `implies` clause lists some trivial but useful lemmas that LP does not automatically recognize as true.

Finally, the timed forward simulation from the milestone automaton to the mutual exclusion specification is defined in Figure 6-15 and the one from Fischer's algorithm to the milestone automaton in Figure 6-16. Except for the `STEP` operator in each, both are straightforward translations of the simulations in Chapter 5. The `STEP` operator allows LP to exploit the fact that the `effect` relation defines a total function.

```
AutomatonMutex (M): trait
  includes Automaton(M), Array1(Region, UID, Regions), CommonActions
  Region enumeration of rem, try, crit, exit
  States[M] tuple of region: Regions
  ActionTypes[M] enumeration of try, crit, exit, rem
  introduces
    __[__]    : ActionTypes[M], UID       → Actions[M]
    unchanged : States[M], States[M], UID → Bool
  asserts
    sort Actions[M] generated freely by __[__]
    sort Tasks[M] generated by task

    ∀ i: UID
      common(try[i]) = try[i];       common(exit[i]) = exit[i];
      common(crit[i]) = crit[i];     common(rem[i]) = rem[i];

      isExternal(try[i]);           isExternal(exit[i]);
      isExternal(crit[i]);          isExternal(rem[i]);

    ∀ a, a': Actions[M], i, i':UID
      task(a) = task(a') ⇔ a = a'  ∨   (∃ i (a = crit[i]) ∧ ∃ i' (a' = crit[i']));

    ∀ s, s': States[M], i, j: UID
      start(s)               ⇔ ∀ i (s.region[i] = rem);
      unchanged(s, s', i)    ⇔ ∀ j (j ≠ i ⇒ s'.region[j] = s.region[j]);
      enabled(s, try[i])     ⇔ s.region[i] = rem;
      effect(s, try[i], s')  ⇔ s'.region[i] = try  ∧ unchanged(s, s', i);
      enabled(s, crit[i])    ⇔ s.region[i] = try ∧ ∀ j (s.region[j] ≠ crit);
      effect(s, crit[i], s') ⇔ s'.region[i] = crit ∧ unchanged(s, s', i);
      enabled(s, exit[i])    ⇔ s.region[i] = crit;
      effect(s, exit[i], s') ⇔ s'.region[i] = exit ∧ unchanged(s, s', i);
      enabled(s, rem[i])     ⇔ s.region[i] = exit;
      effect(s, rem[i], s')  ⇔ s'.region[i] = rem  ∧ unchanged(s, s', i);

      inv(s) ⇔ ∀ j (i ≠ j ⇒ s.region[i] ≠ crit ∨ s.region[j] ≠ crit);

    implies
      Invariants(M, inv)
      ∀ s, s': States[M], at: ActionTypes[M], i: UID
        isStep(s, at[i], s') ⇒ unchanged(s, s', i);
```

Figure 6-9: Larch Trait Specifying the Untimed Mutual Exclusion Problem

6.3 Machine-Checkable Proofs

In this section, we examine parts of the proof that were checked mechanically. The entire proof, presented in the appendix, is too long to examine in detail, and much of it involves handling rather low-level details. However, as we shall see, most of the reasoning follows the same structure as the hand proof.

We will look at two proof scripts. The first is the proof of the sufficient confirmation

```
TimedMutex(TM): trait
  includes AutomatonMutex(M), TimedAutomaton(M, bdmap, TM)
  introduces
    a, b, c : → Time
  asserts
    ∀ i: UID
      bdmap(task(try[i])) = unbounded;            bdmap(task(exit[i])) = unbounded;
      bdmap(task(crit[i])) = [0, (4*a)+(2*c)];    bdmap(task(rem[i])) = [0, 2*a];
  implies
    ∀ s, s', s'': States[TM], a: Actions[TM]
      effect(s, a, s') ∧ effect(s, a, s'') ⇒ s' = s'';
```

Figure 6-10: Larch Trait Specifying Time Bounds for Mutual Exclusion

delay invariant; the second is a fragment of the proof of the simulation from Fischer's algorithm to the milestone automaton. Finally, we discuss how the proof was modified to establish the improved bounds in Section 5.8.

6.3.1 The Sufficient Confirmation Delay Proof

We shall examine in detail the entire script, shown in Figure 6-17, for the proof of Invariant 5.3, which established that processes "waited long enough" after setting the register before checking it. Recall that a script is just a file with commands that are executed by LP in order. This script, as well as the one in the next section, is presented without the box checking marks that are found in the full proof scripts in the appendix. The indentation indicates the structure of the proof, and where this is inadequate, comments have been added.

The first command tells LP to restore the work saved in a freeze file, which resulted from processing the axioms produced by the LSL checker from the `TimedFischer` trait.[2] The `set immunity` command sets a parameter which prevents instantiations from being normalized away by their parents. The `set name` command indicates the name for the axioms to be derived.

The next command specifies which proof methods LP will attempt automatically. If the conjecture is an implication, the first method directs LP to assume the hypothesis and attempt to prove the conclusion. Otherwise, LP will normalize the conjecture.[3]

[2]A few transitivity rules were also added to help LP in reasoning about inequalities.

[3]Unless directed otherwise, LP will always normalize the facts it has assumed; however, it will only normalize the conjecture if this is among its automatic proof methods, or it is explicitly instructed to do so.

```
AutomatonFischer (F): trait
  includes Automaton(F), Array1(PC, UID, PCs), CommonActions, SetShorthand(PC)
  PC enumeration of rem, test, set, check, lvtry, crit, reset, lvexit
  Reg tuple of free: Bool, owner: UID                  % owner only relevant if ¬free
  States[F] tuple of pc: PCs, x: Reg
  ActionTypes[F] enumeration of try, test, set, check, crit, exit, reset, rem
  introduces
    __[__]   : ActionTypes[F], UID       → Actions[F]
   unchanged : States[F], States[F], UID → Bool
  asserts
    sort Actions[F] generated freely by __[__]
    sort Tasks[F] generated freely by task

    ∀ i: UID
      common(try[i]) = try[i];      common(exit[i]) = exit[i];
      common(crit[i]) = crit[i];    common(rem[i]) = rem[i];

      isExternal(try[i]);            isExternal(crit[i]);
      isInternal(test[i]);           isExternal(exit[i]);
      isInternal(set[i]);            isInternal(reset[i]);
      isInternal(check[i]);          isExternal(rem[i]);

    ∀ s, s': States[F], i, j : UID
      start(s)                ⇔ ∀ i (s.pc[i] = rem) ∧ s.x.free;
      unchanged(s, s', i)     ⇔ ∀ j (i ≠ j ⇒ s'.pc[j] = s.pc[j]);
      enabled(s, try[i])      ⇔ s.pc[i] = rem;
      effect(s, try[i], s')   ⇔ s'.pc[i] = test ∧ s'.x = s.x ∧ unchanged(s, s', i);
      enabled(s, test[i])     ⇔ s.pc[i] = test;
      effect(s, test[i], s')  ⇔ s'.pc[i] = (if s.x.free then set else s.pc[i])
                                  ∧ s'.x = s.x ∧ unchanged(s, s', i);
      enabled(s, set[i])      ⇔ s.pc[i] = set;
      effect(s, set[i], s')   ⇔ s'.pc[i] = check ∧ unchanged(s, s', i)
                                  ∧ ¬s'.x.free ∧ s'.x.owner = i;
      enabled(s, check[i])    ⇔ s.pc[i] = check;
      effect(s, check[i], s') ⇔ s'.pc[i] = (if ¬s.x.free ∧ s.x.owner = i then lvtry
                                                                          else test)
                                  ∧ s'.x = s.x ∧ unchanged(s, s', i);
      enabled(s, crit[i])     ⇔ s.pc[i] = lvtry;
      effect(s, crit[i], s')  ⇔ s'.pc[i] = crit ∧ s'.x = s.x ∧ unchanged(s, s', i);
      enabled(s, exit[i])     ⇔ s.pc[i] = crit;
      effect(s, exit[i], s')  ⇔ s'.pc[i] = reset ∧ s'.x = s.x ∧ unchanged(s, s', i);
      enabled(s, reset[i])    ⇔ s.pc[i] = reset;
      effect(s, reset[i], s') ⇔ s'.pc[i] = lvexit ∧ s'.x.free ∧ unchanged(s, s', i);
      enabled(s, rem[i])      ⇔ s.pc[i] = lvexit;
      effect(s, rem[i], s')   ⇔ s'.pc[i] = rem ∧ s'.x = s.x ∧ unchanged(s, s', i);

      inv(s) ⇔ ¬s.x.free ⇒ (s.pc[s.x.owner] ∈ { check, lvtry, crit, reset });
    implies
      Invariants(F, inv)
      ∀ s, s':States[F], at: ActionTypes[F], i, j: UID, b: Bool, p, p1, p2: PC
        effect(s, at[i], s') ⇒ unchanged(s, s', i);
        s'.pc[j] = s.pc[j] ⇒ (enabled(s', at[j]) ⇔ enabled(s, at[j]));
        (if b then p1 else p2) = p ⇔ (if b then p1 = p else p2 = p);
        isStep(s, at[i], s') ∧ ¬s'.x.free ⇒ at[i] = set[s'.x.owner] ∨ s.x = s'.x;
        isStep(s, at[i], s') ∧ ¬s'.x.free ∧ s'.pc[j] = set ⇒ s.pc[j] = set;
        isStep(s, at[i], s') ∧ s'.pc[j] = check ⇒ s.pc[j] = check ∨ at[i] = set[j];
```

Figure 6-11: Larch Trait Specifying Untimed Aspects of Fischer's Algorithm

```
TimedFischer(TF): trait
  includes AutomatonFischer(F), TimedAutomaton(F, bdmap, TF), SetShorthand(PC)
  introduces
    a, b, c                          :              → Time
    SCD, StrongMutex, Mutex, Inv : States[TF] → Bool
  asserts
    ∀ i: UID
      a < b;
      bdmap(task(try[i])) = unbounded;      bdmap(task(crit[i])) = [0, a];
      bdmap(task(test[i])) = [0, a];        bdmap(task(exit[i])) = unbounded;
      bdmap(task(set[i])) = [0, a];         bdmap(task(reset[i])) = [0, a];
      bdmap(task(check[i])) = [b, c];       bdmap(task(rem[i])) = [0, a];

    ∀ s: States[TF], i, j: UID
      SCD(s) ⇔ ¬s.basic.x.free ∧ s.basic.pc[s.basic.x.owner] = check
                  ⇒ ∀ j (s.basic.pc[j] = set
                          ⇒ (s.bounds[task(check[s.basic.x.owner])]).first
                               > (s.bounds[task(set[j])]).last);

      StrongMutex(s) ⇔ ∀ i (s.basic.pc[i] ∈ { lvtry, crit, reset }
                              ⇒ ¬s.basic.x.free ∧ s.basic.x.owner = i
                                  ∧ ∀ j (s.basic.pc[j] ≠ set));

      Inv(s) ⇔ SCD(s) ∧ StrongMutex(s);

      Mutex(s) ⇔ ∀ i (s.basic.pc[i] ∈ { lvtry, crit, reset }
                        ⇒ ∀ j (j ≠ i ⇒ s.basic.pc[j] ∉ { lvtry, crit, reset }));

  implies
    Invariants(TF, Inv)
    ∀ s, s', s'': States[TF], a: Actions[TF], at, at': ActionTypes[F], i, j: UID, t: Time
      inv(s) ∧ isStep(s, addTime(at[i]), s')
       ⇒ ∀ j (j ≠ i ⇒ s'.bounds[task(at'[j])] = s.bounds[task(at'[j])]);
      a < c; a ≠ infinity;
```

Figure 6-12: Larch Trait Specifying Timed Aspects of Fischer's Algorithm

The invariant is proven with three conjectures. The first establishes the base case for the invariant, and the second proves that the invariant is preserved by the time passage action. They are proved automatically by LP. The `qed` asks LP to verify that the conjecture has been proven.

The last conjecture proves the invariant is preserved by all the other actions. Because this is an implication, LP first assumes its hypothesis, and attempts to prove `SCD(s')`. This is also an implication with hypothesis $s'.x \neq 0 \wedge s'.pc_{s'x} = \mathit{checking}$ (expressed in Larch by `¬s'.basic.x.free ∧ s'.basic.pc[s'.basic.x.owner] = check`). LP also assumes this hypothesis, as in the proof in Chapter 5.

LP then strips the universal quantifier off the conclusion of the implication defining

```
AutomatonIntermediate (I): trait
  includes Automaton(I), Array1(Region, UID, Regions), CommonActions
  States[I] tuple of region: Regions, status: Status
  Status enumeration of start, seized, stable
  Region enumeration of rem, try, crit, exit
  ActionTypes[I] enumeration of try, crit, exit, rem
  introduces
    __[__]           : ActionTypes[I], UID       → Actions[I]
    seize, stabilize :                          → Actions[I]
    unchanged        : States[I], States[I], UID → Bool

  asserts
    sort Actions[I] generated freely by __[__], seize, stabilize
    sort Tasks[I] generated by task
    ∀ i: UID
      common(try[i]) = try[i];       common(exit[i]) = exit[i];
      common(crit[i]) = crit[i];     common(rem[i]) = rem[i];

      isExternal(try[i]);           isExternal(crit[i]);
      isInternal(seize);            isExternal(exit[i]);
      isInternal(stabilize);        isExternal(rem[i]);

    ∀ a, a': Actions[I], i, i':UID
      task(a) = task(a') ⇔ a = a' ∨ (∃ i (a = crit[i]) ∧ ∃ i' (a' = crit[i']));

    ∀ s, s': States[I], i, j: UID
      start(s)                ⇔ ∀ i (s.region[i] = rem) ∧ s.status = start;
      unchanged(s, s', i)     ⇔ ∀ j (j ≠ i ⇒ s'.region[j] = s.region[j]);
      enabled(s, try[i])      ⇔ s.region[i] = rem;
      effect(s, try[i], s')   ⇔ s'.region[i] = try
                                   ∧ s'.status = s.status ∧ unchanged(s, s', i);
      enabled(s, seize)       ⇔ ∃ i (s.region[i] = try) ∧ ∀ j (s.region[j] ≠ crit)
                                   ∧ s.status = start;
      effect(s, seize, s')    ⇔ ∀ j (s'.region[j] = s.region[j]) ∧ s'.status = seized;
      enabled(s, stabilize)   ⇔ s.status = seized;
      effect(s, stabilize, s') ⇔ ∀ j (s'.region[j] = s.region[j]) ∧ s'.status = stable;
      enabled(s, crit[i])     ⇔ s.region[i] = try ∧ s.status = stable;
      effect(s, crit[i], s')  ⇔ s'.region[i] = crit
                                   ∧ s'.status = start ∧ unchanged(s, s', i);
      enabled(s, exit[i])     ⇔ s.region[i] = crit;
      effect(s, exit[i], s')  ⇔ s'.region[i] = exit
                                   ∧ s'.status = s.status ∧ unchanged(s, s', i);
      enabled(s, rem[i])      ⇔ s.region[i] = exit;
      effect(s, rem[i], s')   ⇔ s'.region[i] = rem
                                   ∧ s'.status = s.status ∧ unchanged(s, s', i);
      inv(s) ⇔ s.status ≠ start ⇒ (∃ i (s.region[i] = try) ∧ ∀ j (s.region[j] ≠ crit));
    implies
      Invariants(I, inv)
      ∀ s, s': States[I], at: ActionTypes[I], i: UID
        enabled(s, task(crit[i])) ⇔ ∃ i enabled(s, crit[i]);
        isStep(s, at[i], s') ⇒ unchanged(s, s', i);
      ∀ stat:Status
        stat = start ∨ stat = seized ∨ stat = stable;
```

Figure 6-13: Larch Trait Expressing Milestones for Fischer's Algorithm

```
TimedIntermediate(TI): trait
  includes AutomatonIntermediate(I), TimedAutomaton(I, bdmap, TI)
  introduces
    a, b, c : → Time
  asserts
    ∀ i: UID
      bdmap(task(try[i])) = unbounded;       bdmap(task(crit[i])) = [0, a+c];
      bdmap(task(seize)) = [0, (2*a)+c];     bdmap(task(exit[i])) = unbounded;
      bdmap(task(stabilize)) = [0, a];       bdmap(task(rem[i])) = [0, 2*a];
  implies
    ∀ s, s': States[TI], a: Actions[I], at: ActionTypes[I], t: Time, i, j: UID
      inv(s) ∧ isStep(s, addTime(a), s')
       ⇒ s'.bounds[task(rem[j])] = s.bounds[task(rem[j])]
           ∨ a = exit[j] ∨ a = rem[j];
      inv(s) ∧ isStep(s, addTime(at[i]), s') ∧ at ≠ crit
       ⇒ s'.bounds[task(stabilize)] = s.bounds[task(stabilize)]
           ∧ s'.bounds[task(crit[j])] = s.bounds[task(crit[j])];
    ∀ s, s', s'': States[TI], a: Actions[TI]
      effect(s, a, s') ∧ effect(s, a, s'') ⇒ s' = s'';
```

Figure 6-14: Larch Trait Expressing Time Bounds for the Milestones

```
I2M: trait
  includes
    TimedIntermediate(TI), TimedMutex(TM)
  introduces
    g    : States[TI], States[TM]  → Bool
    STEP : States[TM], Actions[TM] → States[TM]
  asserts
    ∀ u, u': States[TM], a: Actions[TM]
      STEP(u, a) = u' ⇔ effect(u, a, u');

    % The simulation relation.
    ∀ s:States[TI], u:States[TM], i:UID
      g(s, u) ⇔
        u.now = s.now
          ∧ ∀ i (  u.basic.region[i] = s.basic.region[i]
                 ∧ (enabled(s.basic, task(seize))
                     ⇒ (u.bounds[task(crit[i])]).last
                          ≥ ((s.bounds[task(seize)]).last + (2*a) + c) )
                 ∧ (enabled(s.basic, task(stabilize))
                     ⇒ (u.bounds[task(crit[i])]).last
                          ≥ ((s.bounds[task(stabilize)]).last + a + c) )
                 ∧ (enabled(s.basic, task(crit[i]))
                     ⇒ (u.bounds[task(crit[i])]).last ≥ (s.bounds[task(crit[i])]).last )
                 ∧ (enabled(s.basic, task(rem[i]))
                     ⇒ (u.bounds[task(rem[i])]).last ≥ (s.bounds[task(rem[i])]).last )
                );
  implies TimedForward(TI, TM, g, inv, inv)
```

Figure 6-15: Larch Trait for the Simulation from the Milestones to the Specification

```
F2I: trait
  includes TimedIntermediate(TI), TimedFischer(TF)
  introduces
    f    : States[TF], States[TI]  → Bool
    STEP : States[TI], Actions[TI] → States[TI]
    w    : States[TF], UID         → Time
  asserts
    ∀ u, u': States[TI], a: Actions[TI]
      STEP(u, a) = u' ⇔ effect(u, a, u');

    ∀ s:States[TF], i:UID
      s.basic.pc[i] = test  ⇒ w(s,i) = (s.bounds[task(test[i])]).last + a;
      s.basic.pc[i] = set   ⇒ w(s,i) = (s.bounds[task(set[i])]).last;
      s.basic.pc[i] = check ⇒ w(s,i) = (s.bounds[task(check[i])]).last + a + a;
      ¬(s.basic.pc[i] ∈ { test, set, check }) ⇒ w(s,i) = infinity;

    % The simulation relation.
    ∀ s: States[TF], u: States[TI], i: UID
      f(s, u) ⇔
        u.now = s.now
         ∧ ∀ i (  (u.basic.region[i] = rem ⇔ s.basic.pc[i] = rem)
                 ∧  (u.basic.region[i] = try ⇔
                        s.basic.pc[i] ∈ { test, set, check, lvtry })
                 ∧  (u.basic.region[i] = crit ⇔ s.basic.pc[i] = crit)
                 ∧  (u.basic.region[i] = exit ⇔
                        s.basic.pc[i] ∈ { reset, lvexit } )
                 ∧  (u.basic.status = start ⇔
                        s.basic.x.free ∨ ∃ i (s.basic.pc[i] ∈ { crit, reset }))
                 ∧  (u.basic.status = seized ⇔
                        ¬s.basic.x.free
                         ∧ ∀ i (s.basic.pc[i] ∉ { crit, reset })
                         ∧ ∃ i (s.basic.pc[i] = set))
                 ∧  (u.basic.status = stable ⇔
                        (¬s.basic.x.free ∧ ∀ i (s.basic.pc[i] ∉ { crit, reset, set })))
                 ∧  (enabled(s.basic, task(reset[i]))
                       ⇒ (u.bounds[task(seize)]).last
                            ≥ ((s.bounds[task(reset[i])]).last + a + a))
                 ∧  ∃ i: UID ((u.bounds[task(seize)]).last ≥ w(s, i))
                 ∧  (enabled(s.basic, task(set[i]))
                       ⇒ (u.bounds[task(stabilize)]).last ≥ (s.bounds[task(set[i])]).last)
                 ∧  (enabled(s.basic, task(check[i]))
                          ∧  ¬s.basic.x.free ∧ s.basic.x.owner = i
                       ⇒ (u.bounds[task(crit[i])]).last ≥ ((s.bounds[task(check[i])]).last + a))
                 ∧  (enabled(s.basic, task(crit[i]))
                       ⇒ (u.bounds[task(crit[i])]).last ≥ (s.bounds[task(crit[i])]).last)
                 ∧  (enabled(s.basic, task(reset[i]))
                       ⇒ (u.bounds[task(rem[i])]).last ≥ ((s.bounds[task(reset[i])]).last + a))
                 ∧  (enabled(s.basic, task(rem[i]))
                       ⇒ (u.bounds[task(rem[i])]).last ≥ (s.bounds[task(rem[i])]).last));
  implies TimedForward(TF, TI, f, StrongMutex, inv)
```

Figure 6-16: Larch Trait for the Simulation from Fischer's Algorithm to the Milestones

```
thaw TimedFischer
set immunity ancestor
set name SCD
set proof-methods ⇒, normalization

prove start(s:States[TF]) ⇒ SCD(s)
qed

prove SCD(s) ∧ isStep(s, nu(t), s') ⇒ SCD(s')
qed

prove SCD(s) ∧ inv(s':States[TF]) ∧ isStep(s, addTime(at[i]), s') ⇒ SCD(s')
        resume by case atc[ic] = set[s'c.basic.x.owner]
          % CASE 1: atc[ic] = set[s'c.basic.x.owner]
          prove (s'c.now + a) < (s'c.now + b)
            instantiate t by s'c.now, t1 by a, t2 by b in Time
          instantiate c by task(set[jc]) in *impliesHyp

          % CASE 2: ¬(atc[ic] = set[s'c.basic.x.owner])
          instantiate s by sc.basic, s' by s'c.basic, at by atc, i by ic in AutomatonFischer
          instantiate j by jc in AutomatonFischer
          instantiate j by sc.basic.x. owner in AutomatonFischer
          prove atc[ic] ≠ check[sc.basic.x.owner] by contradiction
          prove atc[ic] ≠ set[jc] by contradiction
          instantiate j:UID by jc in *hyp
qed
```

Figure 6-17: Larch Proof of Invariant 5.3: Sufficient Confirmation Delay

`SCD`,[4] and assumes the $s'.pc_j$ = `set` hypothesis of the resulting implication. This is also done, though not explicitly, in each of the cases of the hand proof.

LP generates fresh constants and substitutes them for the variables in the hypotheses it assumes.[5] These are the `sc`, `s'c`, `atc`, `ic`, and `jc` that appear in the proof.

When the conjecture is no longer an implication, LP normalizes it, and awaits further guidance, supplied by the remainder of this script. First, we instruct LP to consider two cases as in the hand proof. Note that `ic` represents the index of the process that took a step, which need not be `s'c.basic.x.owner` (i.e., $s'.x$).

If the action is $\mathsf{set}_{s'.x}$, then we prove $s'.now + a < s'.now + b$. Unfortunately, LP is not very good at even simple arithmetic, and a bit of further guidance is necessary for it to recognize this.[6] The `instantiate` command calls LP's attention to instances of general facts that are useful for establishing the current conjecture. The second instantiation, for

[4]All unbound variables are implicitly universally quantified in LP.

[5]This is justified by the *universal generalization* rule of logic.

[6]LP is being enhanced with decision procedures that will greatly improve its ability to deal with arithmetic.

```
% CASE 3: set[u1]
resume by ⇒
  instantiate j:UID by s'c.basic.x.owner in *impliesHyp    % StrongMutex
  prove s'c.basic.pc[i] ∉ { crit, reset }
    resume by case i = s'c.basic.x.owner
      instantiate j:UID by ic in *impliesHyp

  resume by case sc.basic.x.free
```

Figure 6-18: Larch Proof that set_i Preserves the Simulation: Part 1

example, causes LP to recognize that the conjecture follows from Lemma 3.3. This is enough for LP to derive the rest of the proof for that case.

If the action is not $\mathsf{set}_{s'.x}$, then Lemma 5.1, along with the other two lemmas, is instantiated with the relevant constants that LP generated. Then, though this was not done explicitly in the hand proof, it is proven (by contradiction) that $\mathsf{check}_{s'.x}$ and set_j are not newly enabled. The final instantiation uses the inductive assumption to finish the proof.

6.3.2 Preserving the Simulation under the set_i Action

We now present the script for the proof that the simulation is preserved by the set_i action. This script, shown in Figures 6-18, 6-19, and 6-20, does not stand alone, but rather is extracted from the larger proof that the simulation is preserved by any action of *Fischer*. However, as in the hand proof, this is the most complex and interesting part of the proof. Again, the script follows the structure of the hand proof closely.

Because this is only a fragment of a script, we begin by setting the context in which it occurs. The conjecture being proved is

```
f(s,u) ∧ isStep(s, a, s') ∧ inv(s) ∧ inv(u) ∧ StrongMutex(s)
   ⇒ ∃ alpha (execFrag(alpha) ∧ first(alpha) = u ∧ f(s', last(alpha))
                ∧ trace(alpha) = trace(a))
```

and this script verifies the case when `a = addTime(set[u1])`. Unlike in the previous script, the only proof method that LP applies by default is normalization. Thus, the conjecture is still an implication.

The `resume by =>` command causes LP to assume the hypothesis of the conjecture, and attempt to prove the conclusion, as was done automatically in the previous proof script. Following the proof in Chapter 5, we prove $\forall j,\ s.pc_j \notin \{critical, reset\}$, and then consider

```
% CASE (a): sc.basic.x.free
prove ∃ i:UID (uc.basic.region[i] = try)
  resume by specializing i:UID to s'c.basic.x.owner

resume by case  ∀ j:UID (s'c.basic.pc[j] ≠ set)

  % CASE i. ∀ j:UID (s'c.basic.pc[j] ≠ set)
  assert (ac = seize); u'c = STEP(uc, addTime(ac))
  assert a'c = stabilize; u''c = STEP(u'c, addTime(a'c))
  resume by specializing
    alpha to (({uc}) { addTime(ac), u'c }) { addTime(a'c), u''c }
    ..
    instantiate c:Tasks[I] by task(ac) in *impliesHyp / c-op(.first)

    resume by case i = s'c.basic.x.owner
      % CASE: ¬(i = s'c.basic.x.owner);  First case was automatic.
      instantiate j:UID by ic in *Hyp
      resume by ⇒
        instantiate i:UID by ic in *impliesHyp

  % CASE ii. ¬∀ j:UID (s'c.basic.pc[j] ≠ set)
  assert (ac = seize); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    instantiate c:Tasks[I] by task(ac) in *impliesHyp / c-op(.first)
    prove ∃ i:UID (s'c.basic.pc[i] = set)
      declare op ic:→UID
      fix j:UID as ic in *caseHyp
      resume by specializing i:UID to ic

    resume by case i = s'c.basic.x.owner
      % CASE: ¬(i = s'c.basic.x.owner);  First case was automatic.
      instantiate j:UID by ic in *impliesHyp
      resume by ∧
        resume by ⇒
          instantiate i:UID by ic in *impliesHyp
        resume by ⇒
          instantiate c:Tasks[F] by task(set[ic]) in *impliesHyp
```

Figure 6-19: Larch Proof that set_i Preserves the Simulation: Part 2

three cases. These cases correspond exactly to the cases in the hand proof, and they are numbered accordingly. Figure 6-19 contains the first case, proven in two subcases, and Figure 6-20 contains the later two cases.

Each case introduces constants to name the simulated actions and the resulting states. These form the simulated execution fragment, which LP attempts to verify meets the step condition. After specializing the conjecture, we first direct LP to consider that the lower bounds for the actions are met, with the `instantiate ... in *impliesHyp / c-op(.first)` command. This step was not in the hand proof because of our convention of omitting *first* components for trivial lower bounds. However, the LP traits derive

```
resume by case ∀ j:UID (s'c.basic.pc[j] ≠ set)

  % CASE (b): ¬sc.basic.x.free ∧ ∀ j:UID (s'c.basic.pc[j] ≠ set)
  assert (ac = stabilize); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    instantiate c:Tasks[I] by task(ac) in *impliesHyp / c-op(.first)
    prove ∃ i:UID (sc.basic.pc[i] = set)
      resume by specializing i:UID to s'c.basic.x.owner

    resume by case i = s'c.basic.x.owner
      % CASE: ic = s'c.basic.x.owner
      prove ¬∀ i:UID ¬(sc.basic.pc[i] = set) by contradiction
      % CASE: ¬(ic = s'c.basic.x.owner)
      instantiate j:UID by ic in *Hyp
      resume by ⇒
        instantiate i:UID by ic in *impliesHyp

  % CASE (c): ¬sc.basic.x.free ∧  ¬∀ j:UID (s'c.basic.pc[j] ≠ set)
  resume by specializing alpha to {uc}
    prove uc.bounds[task(seize)].last = infinity
      instantiate c:Tasks[I] by task(seize) in *hyp
    prove ¬∀ i:UID ¬(sc.basic.pc[i] = set) by contradiction
    prove ∃ i:UID (sc.basic.pc[i] = set)
      resume by specializing i:UID to s'c.basic.x.owner
    prove ∃ i:UID (s'c.basic.pc[i] = set)
      declare op ic:→UID
      fix j:UID as ic in *caseHyp
      resume by specializing i:UID to ic

    resume by case i = s'c.basic.x.owner
      % CASE: ic = s'c.basic.x.owner
      instantiate c:Tasks[I] by task(crit[ic]) in *impliesHyp
      % CASE: ¬(ic = s'c.basic.x.owner)
      instantiate j:UID by ic in *impliesHyp
      instantiate i:UID by ic in *impliesHyp
      resume by ∧
        resume by ⇒
        resume by ⇒
```

Figure 6-20: Larch Proof that set_i Preserves the Simulation: Part 3

the timed automaton systematically, so these conditions still need to be checked.

Within each case, we also prove some simple lemmas, usually with existential quantifiers. These are all straightforward, but the quantifier prevents LP from recognizing them automatically, and so they must be proved explicitly.

Finally, each case contains an additional case split not found in the hand proof. Actually, this split is necessary, but the case when $i = s'.x$ is so straightforward that we don't mention it explicitly. Notice that LP needs little or no guidance for that case; however, it does need to be directed to make the case split. The guidance provided to LP in the other case ($i \neq s'.x$)

merely directs LP to prove each condition of the simulation separately by assuming the hypothesis and establishing the conclusion.

6.3.3 The Improved Bounds

The proof presented in Section 5.8 was not carefully and systematically checked. Instead, it is informal, and appeals strongly to the similarity to the proof with the weaker bounds, claiming that any changes, other than those explicitly noted, are straightforward. Without automated verification, we must either be content with such informality, or else check every tedious step of the proof again.

One of the important advantages of using automated tools is that the computer can re-do these checks for us. If the changes really are straightforward, then the scripts should require little modification. This is also part of the motivation for choosing a tool that follows our conventional reasoning: if the structure of the hand proof does not change, neither will the structure of the automated proof.

The proof presented in the appendix is, in fact, not the original proof, but one which establishes a time bound of $2a + c$ for the seize task.[7] Other than the obvious changes in the traits reflecting the improved time bounds, the only changes required corresponded to those described in Section 5.8.

6.4 Discussion

The simplest but most significant observation to make is that we succeeded in verifying this proof using the Larch tools. This indicates that automating such proofs is *not* intractable, but in fact realistic. The proof follows the hand proof closely, which makes it easy to understand. It is also quite general, because it uses parameters, rather than specific values, for the time bounds, and is valid for any number of processes.

Though it is difficult to accurately quantify the development time, the main simulation proof, from *Fischer* to *Milestone*, took about a week to formalize and verify using LP. Since this proof was intended largely as a test case on which to tune LP better to accomodate simulation proofs, especially those involving time, many changes were made for readability

[7] The tight time bound was also verified using LP, but this required subtraction to be axiomatized, as well as the additional invariant to be proved.

and to reduce the running time. These changes, made over a period of about a year, reflect a better understanding of how LP verified the proof, as well as improvements made to LP suggested by the difficulties encountered in doing this proof.

Minor changes made in the hand proof seem to be easy to transfer to the automated proof. To establish the improved time bound for the `seize` task, for example, required only changes in the LSL traits reflecting the new bound, and a few minor changes to the proof scripts, which corresponded to the changes in the hand proof.

Another important consideration is the amount of computation LP needs to verify the proof. The current version of the proof takes about an hour of CPU time running on a DEC 3000 AXP Model 500 at 150MHz to process all the traits, and run all the proof scripts, a little under half of which is spent on the proof of the simulation from *Fischer* to *Milestone*. This is significantly reduced from our earlier proofs, mostly by assisting LP in ordering operators in the registry, and by choosing formalizations that LP handles more effectively. We believe there is still leeway to improve this further. For example, in a smaller test case, we achieved a 30% speed-up by using decision procedures for arithmetic and boolean algebra.

The main danger for Larch proofs is that the traits may define an inconsistent theory. We encountered this problem with our initial `Bounds` trait, which introduced a subtle inconsistency. Because we never directed LP to use this inconsistency, we only discovered it when we tried to tune the traits so that LP would do more of the proof without guidance. This problem was easily fixed, requiring only simple modifications to the traits.

Because determining consistency is undecidable, some theorem provers impose greater restrictions on axiomatizations which guarantee consistency.[8] However, such restrictions make it more difficult and awkward to express some concepts, and thus make proofs more complicated and less intuitive. This is especially problematic for proof development, since the high level structure may be obscured. A possible compromise approach is to allow greater flexibility at first, and then use a checker that accepts a restricted language once the structure of the proof has emerged. Our experience indicates that adapting a proof to use a new formalism is not difficult, if the fundamental concepts remain the same. This seems to stem, again, from the similarity of the automated proofs to our standard hand proofs.

[8] Assuming number theory is consistent.

We have found that adhering to the reasoning we employ in our hand proofs generally results in proof scripts that are clearer, more succinct, and easier to modify if necessary. This is especially true at the high level; once the structure of the proof is defined, it is convenient to set LP to do more work automatically, so that it can fill in the details with little guidance.

One difficulty with this approach, however, is that some "obvious" facts used are proved "by inspection," usually involving a simple check over many cases. In fact, in hand proofs, we often simply use such facts without explicitly mentioning them. LP, however, needs some guidance to derive the appropriate statement. This is often best handled by stating the required facts as lemmas, which can be verified by LP with little difficulty.

Also, as mentioned earlier, simple arithmetic and boolean algebra require more guidance in LP than corresponds to the hand proof, and this problem is exacerbated by first-order quantifiers. A new version of LP which uses specialized decision procedures to do arithmetic and boolean algebra is being tested, and we expect that this will improve both the readability and the speed of the proofs. Although first-order logic is undecidable, we are also considering ways to handle quantifiers better.

One of the most significant trade-offs in Larch is between usability and efficiency. Since LP is intended for developing proofs, not merely verifying them, it is designed to interact with the user. Thus, for example, LP attempts to retain the form of assertions as much as possible, so that the user can recognize where the various facts arose from. This means, however, that two semantically identical facts may retain syntactically different forms, and thus not be recognized as equivalent by LP.

Also, Larch does restrict its language to first-order logic, and forbids subtyping by requiring all sorts to be disjoint. This limits the expressive power of Larch, but simplifies its semantics, and allows greater syntactic checks on the input.

We are still trying to learn how to approach these proofs better, so that it will usually be easy to automate proofs of this sort. Some of this work, such as enhancing LP with decision procedures, is already done and simply needs to be exploited in the Fischer proof. Other work still needs to be done. We are also thinking about ways to improve the interaction between LP and the user, to assist in proof development, as well as ways to isolate the user from details conceptually unrelated to the proof, such as the ordering of the operators in the registry. That this proof has been entirely verified using Larch is very encouraging, but we can still see many ways in which we can improve.

Chapter 7

Conclusions and Future Work

We have presented a methodology based on simulations and invariants for analyzing real-time distributed systems and establishing bounds on the time to accomplish certain tasks. We have demonstrated it on some small but nontrivial examples, which previously had no rigorous timing analysis. In particular, the tight upper bound on the time to reach the critical region in Fischer's mutual exclusion algorithm was not, to our knowledge, known before. We have also verified the proof of Fischer's algorithm using the Larch tools.

This methodology involves specifying both the system and its requirements as automata, and establishing a relationship between them that proves that the system satisfies its requirements. Because both are specified as automata, it is possible to introduce intermediate specifications, which express some intuition about how the system unfolds. We have also proposed an approach to defining these intermediate automata, using milestones, to assist in proving timing properties.

This methodology leads to well-structured hierarchical proofs that are rigorous, systematic, and amenable to automatic verification. Also, invariants and simulations serve as "documentation", expressing key insights about a system's behavior, including its timing. Invariants capture the unchanging aspect of the system, while simulations characterize changes in the system, as reflected in the requirements. In this sense, simulations replace operational arguments with an assertional framework.

It also appears that this methodology will scale reasonably to realistic systems. This is because, although the length of the proofs increases as the systems grow, they do not become too complicated. Rather than large, intricate proofs, they typically consist of many small checks that can usually be done independently. Furthermore, many of the checks are trivial, and can be done automatically.

For an estimate of the complexity of simulation proofs, we can characterize the size and complexity of a system by the number of state variables and actions[1] of an automaton for that system. Proving an invariant typically requires separate consideration of each action. Thus, the proof for an invariant is roughly proportional to the size of the system.

A simulation, on the other hand, involves two automata, one for the implementation, and one for the specification. Again, separate consideration is usually required for each action of the implementation, to verify that the simulation can be preserved by that action. However, the simulation also grows more complex with the systems. In the examples we have done, the number of conditions defining the simulation is proportional to the number of state variables in the specification automaton, including the timing variables, or alternatively, the number of variables in the untimed state plus the number of actions. Each of these conditions must be preserved by every action in the implementation, so the proof of a simulation grows as the product of the sizes of the two systems.

Moreover, most of the cases in these proofs are trivial, and thus these proofs are amenable to automatic verification. We have defined a library of abstractions for the Larch tools, which we have used to verify Fischer's algorithm in a way that corresponds closely with the human reasoning we employ to convince ourselves. This provides added confidence that the proof is indeed correct, and that every case has been properly checked.

Using automated verification tools also promises to be helpful when modifying systems. When we modify a system or its specification only slightly, we expect that LP will be able to check most of the original proof automatically, allowing us to concentrate our attention on what has truly changed, without worrying that some important detail has been overlooked. This was, in fact, our experience when we proved the improved bounds for Fischer's algorithm.

More work is still necessary in applying these techniques to larger systems, to test both the methodology and the automated tools. A natural starting point is to verify other mutual exclusion algorithms. In particular, a detailed proof of the simulation given by Lynch

[1]Parameterized actions and variables technically correspond to many actions and variables, but can usually be treated uniformly, and thus can be considered a single action or variable for this analysis. For example, the mutual exclusion automata in Chapter 5 have a set of state variables and actions for each process, but the proof does not depend on the number of processes.

[Lyn93] for Dijkstra's mutual exclusion algorithm [Dij65] should be an informative test of the enhancements made to the Larch tools. The hybrid algorithm of Lynch and Shavit [LS92] is another possibly instructive example to examine, as are Lamport's "bakery" algorithm [Lam74], and algorithms proposed by Peterson and Fischer [PF77].

A much more ambitious study would be to attempt to analyze a more complex and subtle practical algorithm such as the distributed *minimum spanning tree* algorithm of Gallager, Humblet, and Spira [GHS83]. Welch, Lamport, and Lynch gave a rigorous and detailed, and very lengthy, analysis of this algorithm [WLL88], but it did not include a performance analysis. A timing analysis of this algorithm, accompanied by a simple, concise proof, would be relevant for practical systems, and also serve as an interesting case study of the methods developed here.

Operating systems, especially distributed operating systems, provide another rich domain for problems and protocols, such as synchronization [KR93] and scheduling [Jef92, Zho92] with hard real-time constraints, that might be analyzed using this methodology. For these, and other problems, it is important to characterize not only correctness but also timeliness.

Perhaps the most useful application of these techniques lies in the analysis of communication protocols [CAZ92, MSST93], which generally have only informal claims of efficiency and even correctness. For many of these, especially the distributed group communication protocols, the correctness guarantees are not always clear, and only recently have there been attempts at stating these more formally [HT93, FKL95, FvR95, MBRS94]. Unfortunately, these usually lack performance guarantees, which are essential for communication systems. Søgaard-Andersen, Lynch, and Lampson have recently done a lengthy case study applying simulation methods to communication protocols [SLL93a, SLL93b], but this does not include an analysis of the timing. Instead, the performance is typically determined empirically (e.g., [vRHB94]).

Performance guarantees, however, are often difficult to characterize, especially "soft" time bounds, that is, bounds that hold in "typical" cases. The MMT automaton model used in this thesis is adequate only for expressing "hard" time bounds which usually characterize real implementations. Lynch and Vaandrager have defined a more general timed automaton model [LVarb] to specify systems which have a more complex relationship between timing and state. However, little work has been done to develop a methodology for these more general automata, and in particular, this has not yet been used to model any systems with soft time bounds. Furthermore, it is unclear how much additional complexity in the proofs will result with this increased dependency between timing and state.

Another important class of systems that cannot be handled within the framework of this thesis are randomized algorithms. Segala [Seg95] has developed a general probabilistic automaton model, which Pogosyants and Segala [PS95] have specialized to a probabilistic variant of MMT automata, and have proved some results using this model. Pogosyants is also working on automating these proofs using the Larch tools. However, more work is still necessary to understand the general structure behind such proofs, and to develop guidelines to approach and automate such proofs.

We believe that this is a fruitful area of research, and that many interesting real-time systems are now within the reach of formal methods. Furthermore, as automated tools become more sophisticated, we expect practical machine verification of proofs of real-time systems to be a reasonable goal.

Appendix A

LP Proof Script Files

This appendix contains the proof scripts used to verify the Fischer algorithm described in Chapter 5. The proof is divided into six scripts, corresponding to the *implies* clauses of the automaton and simulation traits specific to the proof of Fischer's algorithm.[1]

At the beginning of several files are some commands that indicate how certain operators are to be ordered in the registry. This significantly reduces the time that LP spends attempting find an order for the operators that does not cause it to loop infinitely during normalization.

These proofs use LP's *box-checking* option, with marks generated by to indicate points at which new proof obligations are introduced (the `<>` marks) and satisfied (matching `[]` marks).[2] These marks are often enough to indicate the structure of the proof, and serve as documentation, as well as checks that LP is proceeding as expected. Comments, preceded by `%`, provide further documentation where necessary. Long commands may be split into several lines, terminated by two periods (`..`). These do not indicate any elision of the script, which is provided here in full, exactly as it is processed by LP.

A.1 The Untimed Aspects of Fischer's Algorithm

```
set script untimedfischer
execute AutomatonFischer_Axioms
```

[1] There is some rearrangement of where the implications are proved. For example, there are actually two scripts for establishing the *implies* clause of the `TimedFischer` trait, one for the sufficient confirmation delay invariant, and the other establishing strong mutual exclusion.

[2] Proof obligations introduced explicitly with a `prove` command do not have redundant `<>` marks. They are matched by `[] conjecture`.

```
set box-checking on
set name UntimedFischer
set proof-methods ⇒,normalization
set immunity ancestor

declare vars p, p1, p2: PC
prove p = (if b then p1 else p2) ⇔ (if b then p:PC = p1 else p = p2) by case b
    <> case bc
    [] case bc
    <> case ¬bc
    [] case ¬bc
  [] conjecture
qed

declare var at:ActionTypes[F]
prove effect(s, at[i], s') ⇒ unchanged(s, s', i) by induction on at:ActionTypes[F]
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
  [] conjecture
qed

prove s'.pc[j] = s.pc[j] ⇒ (enabled(s', at[j]) ⇔ enabled(s, at[j]))
    <> ⇒ subgoal
    resume by induction on at
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
```

```
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove
  isStep(s, at[i], s') ∧  ¬s'.x.free ⇒ at[i] = set[s'.x.owner] ∨ s.x = s'.x
  by induction on at:ActionTypes[F]
  ..
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
  [] conjecture
qed

set proof-methods normalization

prove isStep(s, at[i], s') ∧  ¬s'.x.free ∧ s'.pc[j] = set ⇒ s.pc[j] = set
  resume by case i:UID = j
    <> case ic = jc
```

```
    set proof-methods normalization, ⇒
    resume by induction on at:ActionTypes[F]
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
    [] case ic = jc
    <> case ¬(ic = jc)
    resume by ⇒
      <> ⇒ subgoal
      instantiate s by sc, s' by s'c, at by atc, i by ic, j by jc in UntimedFischer
      [] ⇒ subgoal
    [] case ¬(ic = jc)
  [] conjecture
qed

prove isStep(s, at[i], s') ∧ s'.pc[j] = check ⇒ (s.pc[j] = check ∨ at[i] = set[j])
  resume by case i:UID = j
    <> case ic = jc
    set proof-methods normalization, ⇒
    resume by induction on at:ActionTypes[F]
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
```

```
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
    [] case ic = jc
    <> case ¬(ic = jc)
    resume by ⇒
      <> ⇒ subgoal
      instantiate s by sc, s' by s'c, at by atc, i by ic, j by jc in UntimedFischer
      [] ⇒ subgoal
    [] case ¬(ic = jc)
  [] conjecture
qed

% The invariant

prove start(s) ⇒ inv(s) by ⇒
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove inv(s) ∧ isStep(s, a, s') ⇒ inv(s') by induction on a:Actions[F]
    <> basis subgoal
    resume by case s'.x = s.x
      <> case s'c.x = sc.x
      resume by case u = sc.x.owner
        <> case uc = sc.x.owner
        set proof-methods ⇒, normalization
        resume by induction on a1
          <> basis subgoal
            <> ⇒ subgoal
              <> ⇒ subgoal
              [] ⇒ subgoal
            [] ⇒ subgoal
          [] basis subgoal
          <> basis subgoal
            <> ⇒ subgoal
              <> ⇒ subgoal
              [] ⇒ subgoal
            [] ⇒ subgoal
          [] basis subgoal
          <> basis subgoal
            <> ⇒ subgoal
              <> ⇒ subgoal
```

```
            [] ⇒ subgoal
          [] ⇒ subgoal
        [] basis subgoal
        <> basis subgoal
          <> ⇒ subgoal
            <> ⇒ subgoal
            [] ⇒ subgoal
          [] ⇒ subgoal
        [] basis subgoal
        <> basis subgoal
          <> ⇒ subgoal
            <> ⇒ subgoal
            [] ⇒ subgoal
          [] ⇒ subgoal
        [] basis subgoal
        <> basis subgoal
          <> ⇒ subgoal
            <> ⇒ subgoal
            [] ⇒ subgoal
          [] ⇒ subgoal
        [] basis subgoal
        <> basis subgoal
          <> ⇒ subgoal
            <> ⇒ subgoal
            [] ⇒ subgoal
          [] ⇒ subgoal
        [] basis subgoal
        <> basis subgoal
          <> ⇒ subgoal
            <> ⇒ subgoal
            [] ⇒ subgoal
          [] ⇒ subgoal
        [] basis subgoal
      [] case uc = sc.x.owner
      <> case ¬(uc = sc.x.owner)
      set proof-methods ⇒, normalization
      resume
        <> ⇒ subgoal
          <> ⇒ subgoal
          instantiate s by sc, s' by s'c, at by a1c, i by uc in untimedfischer
          instantiate j by sc.x.owner in untimedfischer
          [] ⇒ subgoal
        [] ⇒ subgoal
      [] case ¬(uc = sc.x.owner)
    [] case s'c.x = sc.x
    <> case ¬(s'c.x = sc.x)
    set proof-methods ⇒, normalization
    resume
      <> ⇒ subgoal
        <> ⇒ subgoal
        instantiate s by sc, s' by s'c, at by a1c, i by uc in untimedfischer
        [] ⇒ subgoal
      [] ⇒ subgoal
    [] case ¬(s'c.x = sc.x)
  [] basis subgoal
 [] conjecture
qed

set log untimedfischer
```

```
statistics
quit
```

A.2 The Milestone Automaton

```
set script intermediate
execute AutomatonIntermediate_Axioms
set name Intermediate
set immunity ancestor
set box-checking on

declare variable stat:Status
prove stat = start ∨ stat = seized ∨ stat = stable by induction on stat
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
    [] basis subgoal
  [] conjecture
qed

prove enabled(s, task(crit[i])) ⇔ ∃ i enabled(s, crit[i])
  resume by case ∃ i enabled(s, crit[i])
    <> case ∃ i:UID enabled(sc, crit[i])
    declare operator ic: → UID
    fix i as ic in *caseHyp
    resume by specializing a:Actions[I] to crit[ic]
      <> specialization subgoal
      [] specialization subgoal
    [] case ∃ i:UID enabled(sc, crit[i])
    <> case ¬∃ i:UID enabled(sc, crit[i])
    resume by case a = crit[i]
      <> case ac = crit[ic]
      [] case ac = crit[ic]
      <> case ¬(ac = crit[ic])
      resume by contradiction
        <> contradiction subgoal
        declare operator i'c: → UID
        fix i as i'c in *contraHyp
        [] contradiction subgoal
      [] case ¬(ac = crit[ic])
    [] case ¬∃ i:UID enabled(sc, crit[i])
  [] conjecture
qed

prove start(s) ⇒ inv(s) by ⇒
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove inv(s) ∧ isStep(s, a, s') ⇒ inv(s') by case s.status = start
    <> case sc.status = start
    resume by induction on a:Actions[I]
      <> basis subgoal
      set proof-methods normalization, ⇒
      resume by induction on a1
```

```
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
    <> basis subgoal
      <> ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
  [] basis subgoal
  <> basis subgoal
  resume by ⇒
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] basis subgoal
  <> basis subgoal
  [] basis subgoal
[] case sc.status = start
<> case ¬(sc.status = start)
resume by case ∃ i(sc.region[i] = try)
  <> case ∃ i (sc.region[i] = try)
  declare operator ic: → UID
  fix i:UID as ic in *hyp
  resume by induction on a:Actions[I]
    <> basis subgoal
    set proof-methods normalization, ⇒
    resume by induction on a1
      <> basis subgoal
        <> ⇒ subgoal
        resume by specializing i:UID to ic
          <> specialization subgoal
          prove ic ≠ uc by contradiction
              <> contradiction subgoal
              [] contradiction subgoal
            [] conjecture
          instantiate j:UID by ic in *hyp
          resume by case j = uc
            <> case jc = uc
            [] case jc = uc
            <> case ¬(jc = uc)
            instantiate j:UID by jc in *hyp
            [] case ¬(jc = uc)
          [] specialization subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
      <> basis subgoal
```

```
          <> ⇒ subgoal
          prove uc ≠ ic by contradiction
              <> contradiction subgoal
              [] contradiction subgoal
            [] conjecture
          resume by specializing i:UID to ic
            <> specialization subgoal
            instantiate j:UID by ic in *hyp
            resume by case j = uc
              <> case jc = uc
              [] case jc = uc
              <> case ¬(jc = uc)
              instantiate j:UID by jc in *hyp
              [] case ¬(jc = uc)
            [] specialization subgoal
          [] ⇒ subgoal
        [] basis subgoal
      [] basis subgoal
      <> basis subgoal
      [] basis subgoal
      <> basis subgoal
      resume by ⇒
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] basis subgoal
    [] case ∃ i (sc.region[i] = try)
    <> case ¬∃ i (sc.region[i] = try)
    [] case ¬∃ i (sc.region[i] = try)
  [] case ¬(sc.status = start)
 [] conjecture
qed

set log intermediate
statistics
quit
```

A.3 Sufficient Confirmation Delay

```
set script SCD
thaw TimedFischer
set box-checking on
set immunity ancestor
set name theorem
set proof-methods ⇒, normalization

%%%%%% Preliminaries

% Put information in registry to speed up ordering

register height
  __[__]:Bounds[F],Tasks[F]→Bounds
    > (.first, .last, bdmap, ∃:Actions[F],Bool→Bool, .basic, +:Time,Time→Time)
  ..
register height .bounds > (.basic, enabled:States[F],Actions[F]→Bool)

%%%%%%% The proof

prove start(s:States[TF]) ⇒ SCD(s)
```

```
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove SCD(s) ∧ inv(s':States[TF]) ∧ isStep(s, addTime(at[i]), s') ⇒ SCD(s')
    <> ⇒ subgoal
      <> ⇒ subgoal
        <> ⇒ subgoal
        resume by case atc[ic] = set[s'c.basic.x.owner]
          <> case atc[ic] = set[s'c.basic.x.owner]
          prove (s'c.now + a) < (s'c.now + b)
            instantiate t by s'c.now, t1 by a, t2 by b in Time
            [] conjecture
          instantiate c by task(set[jc]) in *impliesHyp
          [] case atc[ic] = set[s'c.basic.x.owner]
          <> case ¬(atc[ic] = set[s'c.basic.x.owner])
          instantiate s by sc.basic, s' by s'c.basic, at by atc, i by ic in AutomatonFischer
          % Uses Lemma 5.1
          instantiate j by jc in AutomatonFischer
          instantiate j by sc.basic.x. owner in AutomatonFischer
          % We now know set[jc] and check[sc.basic.x.owner] are enabled in sc and s'c.
          prove atc[ic] ≠ check[sc.basic.x.owner] by contradiction
              <> contradiction subgoal
              [] contradiction subgoal
            [] conjecture
          prove atc[ic] ≠ set[jc] by contradiction
              <> contradiction subgoal
              [] contradiction subgoal
            [] conjecture
          instantiate j by jc in *hyp
          [] case ¬(atc[ic] = set[s'c.basic.x.owner])
        [] ⇒ subgoal
      [] ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove SCD(s) ∧ isStep(s, nu(t), s') ⇒ SCD(s')
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

set log SCD
statistics
quit
```

A.4 Strong Mutual Exclusion

```
set script mutex
thaw TimedFischer
set proof-methods ⇒, normalization
set immunity ancestor
set box-checking on
set name theorem

%%%%%% Preliminaries
```

```
% Put information in registry to speed up ordering

register height
  __[__]:Bounds[F],Tasks[F]→Bounds
    > (.first, .last, bdmap, ∃:Actions[F],Bool→Bool, .basic, +:Time,Time→Time)
  ..
register height .bounds > (.basic, enabled:States[F],Actions[F]→Bool)

%%%%%% The proof

prove start(s:States[TF]) ⇒ StrongMutex(s)
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove
  StrongMutex(s) ∧ SCD(s) ∧ inv(s:States[TF]) ∧ isStep(s, addTime(at[i]), s')
     ⇒ StrongMutex(s')
    by induction on at:ActionTypes[F]
  ..
    % CASE 1: at = try
    <> basis subgoal
      <> ⇒ subgoal
        <> ⇒ subgoal
        resume by case ic1 = ic
          <> case ic1 = ic
          [] case ic1 = ic
          <> case ¬(ic1 = ic)
          critical-pairs *caseHyp with *impliesHyp
          instantiate i by ic1 in *hyp
          resume by case j = ic
            <> case jc = ic
            [] case jc = ic
            <> case ¬(jc = ic)
            critical-pairs *caseHyp with *impliesHyp
            [] case ¬(jc = ic)
          [] case ¬(ic1 = ic)
        [] ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal

    % CASE 2: at = test
    <> basis subgoal
      <> ⇒ subgoal
        <> ⇒ subgoal
        resume by case ic = ic1
          <> case ic = ic1
          resume by case sc.basic.x.free
            <> case sc.basic.x.free
            [] case sc.basic.x.free
            <> case ¬sc.basic.x.free
            [] case ¬sc.basic.x.free
          [] case ic = ic1
          <> case ¬(ic = ic1)
          instantiate j by ic1 in *impliesHyp
          instantiate i by ic1 in *impliesHyp
          resume by case j = ic
```

```
        <> case jc = ic
        [] case jc = ic
        <> case ¬(jc = ic)
        instantiate j by jc in *impliesHyp
        [] case ¬(jc = ic)
      [] case ¬(ic = ic1)
    [] ⇒ subgoal
  [] ⇒ subgoal
[] basis subgoal

% CASE 3: a = set
<> basis subgoal
  <> ⇒ subgoal
    <> ⇒ subgoal
    prove ic1 ≠ ic by contradiction
        <> contradiction subgoal
        [] contradiction subgoal
      [] conjecture
    instantiate j by ic1 in *hyp
    instantiate i by ic1 in *hyp
    [] ⇒ subgoal
  [] ⇒ subgoal
[] basis subgoal

% CASE 4: a = check
<> basis subgoal
  <> ⇒ subgoal
    <> ⇒ subgoal
    instantiate j by ic in *hyp
    resume by case sc.basic.x.owner = ic ∧ ¬sc.basic.x.free
      <> case sc.basic.x.owner = ic ∧ ¬sc.basic.x.free
      prove ic = ic1 by contradiction
          <> contradiction subgoal
          instantiate j by ic1 in *hyp
          instantiate i by ic1 in *hyp
          [] contradiction subgoal
        [] conjecture
      prove ∀ j (s'c.basic.pc[j] ≠ set)
        resume by case j = ic
          <> case jc = ic
          [] case jc = ic
          <> case ¬(jc = ic)
          instantiate j by jc in *impliesHyp
          instantiate c by task(set[jc]) in *hyp
          resume by contradiction
            <> contradiction subgoal
            [] contradiction subgoal
          [] case ¬(jc = ic)
        [] conjecture
      [] case sc.basic.x.owner = ic ∧ ¬sc.basic.x.free

      <> case ¬(sc.basic.x.owner = ic ∧ ¬sc.basic.x.free)
      prove ic1 ≠ ic by contradiction
          <> contradiction subgoal
          [] contradiction subgoal
        [] conjecture
      instantiate j by ic1 in *hyp
      instantiate i by ic1 in *hyp
      resume by case j = ic
```

```
        <> case jc = ic
        [] case jc = ic
        <> case ¬(jc = ic)
        instantiate j by jc in *hyp
        [] case ¬(jc = ic)
      [] case ¬(sc.basic.x.owner = ic ∧ ¬sc.basic.x.free)
    [] ⇒ subgoal
  [] ⇒ subgoal
[] basis subgoal

% CASE 5: a = crit
<> basis subgoal
  <> ⇒ subgoal
    <> ⇒ subgoal
    resume by case ic1 = ic
      <> case ic1 = ic
      instantiate i by ic in *hyp
      resume by case j = ic
        <> case jc = ic
        [] case jc = ic
        <> case ¬(jc = ic)
        instantiate j by jc in *hyp
        [] case ¬(jc = ic)
      [] case ic1 = ic
      <> case ¬(ic1 = ic)
      instantiate j by ic1 in *hyp
      instantiate i by ic1 in *hyp
      resume by case j = ic
        <> case jc = ic
        [] case jc = ic
        <> case ¬(jc = ic)
        instantiate j by jc in *hyp
        [] case ¬(jc = ic)
      [] case ¬(ic1 = ic)
    [] ⇒ subgoal
  [] ⇒ subgoal
[] basis subgoal

% CASE 6: a = exit
<> basis subgoal
  <> ⇒ subgoal
    <> ⇒ subgoal
    resume by case ic1 = ic
      <> case ic1 = ic
      instantiate i by ic in *hyp
      resume by case j = ic
        <> case jc = ic
        [] case jc = ic
        <> case ¬(jc = ic)
        instantiate j by jc in *hyp
        [] case ¬(jc = ic)
      [] case ic1 = ic
      <> case ¬(ic1 = ic)
      instantiate j by ic1 in *hyp
      instantiate i by ic1 in *hyp
      resume by case j = ic
        <> case jc = ic
        [] case jc = ic
        <> case ¬(jc = ic)
```

```
            instantiate j by jc in *hyp
            [] case ¬(jc = ic)
          [] case ¬(ic1 = ic)
        [] ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal

    % CASE 7: a = reset
    <> basis subgoal
      <> ⇒ subgoal
      resume by case i = ic
        <> case ic1 = ic
        [] case ic1 = ic
        <> case ¬(ic1 = ic)
        instantiate i by ic in *hyp
        instantiate j by ic1 in *hyp
        instantiate i by ic1 in *hyp
        [] case ¬(ic1 = ic)
      [] ⇒ subgoal
    [] basis subgoal

    % CASE 8: a = rem
    <> basis subgoal
      <> ⇒ subgoal
        <> ⇒ subgoal
        resume by case ic1 = ic
          <> case ic1 = ic
          [] case ic1 = ic
          <> case ¬(ic1 = ic)
          instantiate j by ic1 in *hyp
          instantiate i by ic1 in *hyp
          resume by case j = ic
            <> case jc = ic
            [] case jc = ic
            <> case ¬(jc = ic)
            instantiate j by jc in *hyp
            [] case ¬(jc = ic)
          [] case ¬(ic1 = ic)
        [] ⇒ subgoal
      [] ⇒ subgoal
    [] basis subgoal
  [] conjecture
qed

prove StrongMutex(s) ∧ isStep(s, nu(t), s') ⇒ StrongMutex(s')
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove StrongMutex(s) ⇒ Mutex(s)
    <> ⇒ subgoal
      <> ⇒ subgoal
        <> ⇒ subgoal
        instantiate i by ic in *hyp
        resume by contradiction
          <> contradiction subgoal
          instantiate i by jc in *hyp
          [] contradiction subgoal
```

```
        [] ⇒ subgoal
      [] ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

set log mutex
statistics
quit
```

A.5 The Simulation from the Milestones to the Specification

```
set script i2m
thaw I2M
set name theorem
set immunity ancestor
set box-checking on

%%%%%% Preliminaries

% Put information in registry to speed up ordering

register height
  __[__]:Bounds[I],Tasks[I]→Bounds
    > (.first, .last, bdmap:Tasks[I]→Bounds, ∃:Actions[I],Bool→Bool,
        .basic:States[TI]→States[I], +:Time,Time→Time)
  ..
register height
  __[__]:Bounds[M],Tasks[M]→Bounds
    > (.first, .last, bdmap:Tasks[M]→Bounds, ∃:Actions[M],Bool→Bool,
        .basic:States[TM]→States[M], +:Time,Time→Time)
  ..
register height
  .bounds:States[TI]→Bounds[I]
     > (.basic:States[TI]→States[I], enabled:States[I],Actions[I]→Bool)
  ..
register height
  .bounds:States[TM]→Bounds[M]
     > (.basic:States[TM]→States[M], enabled:States[M],Actions[M]→Bool)
  ..
register height __[__]:Regions,UID→Region >  ∨

% Introduce constants that will be used to replace variables

declare operators
  uc:           → States[TM]    % Used by LP for u in hypotheses
  sc, s'c:      → States[TI]    % Ditto for s and s'
  u'c:          → States[TM]    % To abbreviate STEP(uc, ...)
  ac, a'c:      → Actions[M]    % To abbreviate actions
  ..

% Put information in registry to ensure intended orientation of equations

register top uc, u'c
register height u'c > uc
register height s'c > sc

% Some preliminary lemmas.
```

```
prove effect(s, a, STEP(s, a))
  rewrite conjecture with reversed I2M
  [] conjecture
qed

prove enabled(s:States[M], task(crit[i])) ⇔ ∃ i:UID enabled(s:States[M], crit[i])
  resume by case ∃ i:UID enabled(s:States[M], crit[i])
    <> case ∃ i:UID  enabled(sc, crit[i])
    declare operator ic: → UID
    fix i:UID as ic in *caseHyp
    resume by specializing a:Actions[M] to crit[ic]
      <> specialization subgoal
      [] specialization subgoal
    [] case ∃ i:UID  enabled(sc, crit[i])
    <> case ¬∃ i:UID  enabled(sc, crit[i])
    resume by case a:Actions[M] = crit[i]
      <> case ac = crit[ic]
      [] case ac = crit[ic]
      <> case ¬(ac = crit[ic])
      resume by contradiction
        <> contradiction subgoal
        declare operator i'c: → UID
        fix i:UID as i'c in *contraHyp
        [] contradiction subgoal
      [] case ¬(ac = crit[ic])
    [] case ¬∃ i:UID  enabled(sc, crit[i])
  [] conjecture
qed

%%%% The proof of the simulation

prove start(s:States[TI]) ⇒ ∃ u:States[TM] (g(s,u) ∧ start(u:States[TM])) by ⇒
    <> ⇒ subgoal
    declare operators nullbounds: → Bounds[M], startregs: → Regions
    assert ∀ i:UID (startregs[i] = rem)
    assert ∀ c:Tasks[M] (nullbounds[c] = unbounded)
    resume by specializing u to [[startregs], 0, nullbounds]
      <> specialization subgoal
      resume by induction on c:Tasks[M]
        <> basis subgoal
        resume by induction on a7
          <> basis subgoal
          resume by induction on a5
            <> basis subgoal
            [] basis subgoal
            <> basis subgoal
            [] basis subgoal
            <> basis subgoal
            [] basis subgoal
            <> basis subgoal
            [] basis subgoal
          [] basis subgoal
        [] basis subgoal
      [] specialization subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove g(s,u) ⇒ s:States[TI].now = u.now by ⇒
```

```
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

declare variable alpha:StepSeq[TM]
prove
  g(s,u) ∧ isStep(s:States[TI], a, s') ∧ inv(s:States[TI]) ∧ inv(u)
   ⇒ ∃ alpha (execFrag(alpha) ∧ first(alpha) = u ∧ g(s',last(alpha))
                 ∧ trace(alpha) = trace(a:Actions[TI]) )
  by induction a:Actions[TI]
  ..
    <> basis subgoal

    % CASE: a = nu(s'c.now)
    resume by ⇒
      <> ⇒ subgoal
      assert u'c = STEP(uc, nu(s'c.now))
      resume by specializing alpha to ({uc}) { nu(s'c.now), u'c }
        <> specialization subgoal
        resume by induction on c:Tasks[M]
          <> basis subgoal
          resume by induction on a7
            <> basis subgoal
            resume by induction on a5
              <> basis subgoal
              % CASE 1: c = task(try[u1])
              instantiate c:Tasks[M] by task(try[u1]) in *impliesHyp
              [] basis subgoal
              <> basis subgoal
              % CASE 2: c = task(crit[u1])
              resume by cases
                enabled(sc.basic, seize),
                enabled(sc.basic, stabilize),
                enabled(sc.basic, task(crit[u1])),
                ¬enabled(uc.basic, task(crit[u1]))
                ..
                <> case justification
                resume by contradiction
                  <> contradiction subgoal
                  instantiate stat by sc.basic.status in AutomatonIntermediate
                  declare operator ic: → UID
                  fix i:UID as ic in *contraHyp
                  [] contradiction subgoal
                [] case justification
                <> case enabled(sc.basic, seize)
                prove s'c.now ≤ (sc.bounds[task(seize)].last + a + a + c)
                  instantiate z by sc.bounds[c].last+a+a+c in Transitivity
                  [] conjecture
                [] case enabled(sc.basic, seize)
                <> case enabled(sc.basic, stabilize)
                prove s'c.now ≤ (sc.bounds[task(stabilize)].last + a + c)
                  instantiate z by sc.bounds[c].last+a+c in Transitivity
                  [] conjecture
                [] case enabled(sc.basic, stabilize)
                <> case enabled(sc.basic, task(crit[u1c]))
                [] case enabled(sc.basic, task(crit[u1c]))
                <> case ¬enabled(uc.basic, task(crit[u1c]))
                instantiate c:Tasks[M] by task(crit[u1c]) in *impliesHyp
```

```
            [] case ¬enabled(uc.basic, task(crit[u1c]))
          [] basis subgoal
          <> basis subgoal
          % CASE 3: c = task(exit[u1])
          instantiate c:Tasks[M] by task(exit[u1]) in *impliesHyp
          [] basis subgoal
          <> basis subgoal
          % CASE 4: c = task(rem[u1])
          resume by case enabled(uc.basic, task(rem[u1]))
            <> case enabled(uc.basic, task(rem[u1c]))
            instantiate a:Actions[I] by rem[u1c], i by u1c in *impliesHyp
            [] case enabled(uc.basic, task(rem[u1c]))
            <> case ¬enabled(uc.basic, task(rem[u1c]))
            instantiate c:Tasks[M] by task(rem[u1c]) in *impliesHyp
            [] case ¬enabled(uc.basic, task(rem[u1c]))
          [] basis subgoal
        [] basis subgoal
      [] basis subgoal
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal

resume by induction on a3
  <> basis subgoal
  resume by induction on a1
    <> basis subgoal
    % CASE 1: a = addTime(try[u1])
    resume by ⇒
      <> ⇒ subgoal
      assert (ac = try[u1c]); u'c = STEP(uc, addTime(ac))
      resume by specializing alpha to ({uc}) { addTime(ac), u'c }
        <> specialization subgoal
        instantiate c by task(ac) in *impliesHyp / c-op(.first)

        prove u'c.basic.region[i] = s'c.basic.region[i]
          resume by case i = u1c
            <> case ic = u1c
            [] case ic = u1c
            <> case ¬(ic = u1c)
            instantiate j by ic in *impliesHyp, theorem
            [] case ¬(ic = u1c)
          [] conjecture

        % The rest are checking the bounds in the post-states.
        %  We do this in two parts, starting with the bound for the rem action
        prove
          (s'c.basic.region[i] = exit
             ⇒ s'c.bounds[task(rem[i])].last ≤ u'c.bounds[task(rem[i])].last)
          ..
          resume by case i = u1c
            <> case ic = u1c
            [] case ic = u1c
            <> case ¬(ic = u1c)
            resume by ⇒
              <> ⇒ subgoal
              instantiate c:Tasks[I] by task(rem[ic]) in *impliesHyp
              instantiate c:Tasks[M] by task(rem[ic]) in theorem
              instantiate j by ic in *impliesHyp
```

```
      instantiate i by ic in *impliesHyp / c-op(rem:→ActionTypes[M])
      [] ⇒ subgoal
    [] case ¬(ic = u1c)
  [] conjecture

resume by case ¬enabled(u'c.basic, task(crit[i]))
  <> case ¬enabled(u'c.basic, task(crit[ic]))
  instantiate c:Tasks[M] by task(crit[ic]) in theorem
  resume by case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    <> case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    [] case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    <> case ¬∀ j:UID ¬(s'c.basic.region[j] = crit)
    [] case ¬∀ j:UID ¬(s'c.basic.region[j] = crit)
  [] case ¬enabled(u'c.basic, task(crit[ic]))
  <> case ¬(¬enabled(u'c.basic, task(crit[ic])))
  resume by case enabled(uc.basic, task(crit[ic]))
    <> case enabled(uc.basic, task(crit[ic]))
    resume by ∧
      <> ∧ subgoal
      resume by ⇒
        <> ⇒ subgoal
        resume by case enabled(sc.basic, seize)
          <> case enabled(sc.basic, seize)
          [] case enabled(sc.basic, seize)
          <> case ¬enabled(sc.basic, seize)
          instantiate c by task(seize) in *impliesHyp
          instantiate c:Tasks[M] by task(crit[i]) in theorem
          declare operator i'c: → UID
          fix i:UID as i'c in *caseHyp
          [] case ¬enabled(sc.basic, seize)
        [] ⇒ subgoal
      [] ∧ subgoal
      <> ∧ subgoal
      resume by ⇒
        <> ⇒ subgoal
        [] ⇒ subgoal
      [] ∧ subgoal
      <> ∧ subgoal
      resume by ⇒
        <> ⇒ subgoal
        prove enabled(sc.basic, task(crit[ic]))
          resume by case ∃ i (sc.basic.region[i] = try)
            <> case ∃ i:UID (sc.basic.region[i] = try)
            declare operator i'c: → UID
            fix i:UID as i'c in *impliesHyp
            [] case ∃ i:UID (sc.basic.region[i] = try)
            <> case ¬∃ i:UID (sc.basic.region[i] = try)
            [] case ¬∃ i:UID (sc.basic.region[i] = try)
          [] conjecture
        [] ⇒ subgoal
      [] ∧ subgoal
    [] case enabled(uc.basic, task(crit[ic]))
    <> case ¬enabled(uc.basic, task(crit[ic]))
    % seize, stabilize, crit[i] are not enabled in sc
    %  (because crit[i] is not enabled in uc, and g(sc,uc) )
    resume by case ¬∃ i:UID (sc.basic.region[i] = try)
      <> case ¬∃ i:UID (sc.basic.region[i] = try)
      resume by ⇒
        <> ⇒ subgoal
```

```
            [] ⇒ subgoal
          [] case ¬∃ i:UID (sc.basic.region[i] = try)
          <> case ¬(¬∃ i:UID (sc.basic.region[i] = try))
          resume by ∧
            <> ∧ subgoal
            resume by ⇒
              <> ⇒ subgoal
              prove ¬enabled(sc.basic, seize) by contradiction
                  <> contradiction subgoal
                  [] contradiction subgoal
                [] conjecture
              [] ⇒ subgoal
            [] ∧ subgoal
            <> ∧ subgoal
            resume by ⇒
              <> ⇒ subgoal
              [] ⇒ subgoal
            [] ∧ subgoal
            <> ∧ subgoal
            resume by ⇒
              <> ⇒ subgoal
              declare operator i'c: → UID
              fix i:UID as i'c in *caseHyp / c-op(sc)
              instantiate i by i'c in *caseHyp
              [] ⇒ subgoal
            [] ∧ subgoal
          [] case ¬(¬∃ i:UID (sc.basic.region[i] = try))
        [] case ¬enabled(uc.basic, task(crit[ic]))
      [] case ¬(¬enabled(u'c.basic, task(crit[ic])))
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal
% CASE 2: a = addTime(crit[u1])
resume by ⇒
  <> ⇒ subgoal
  assert (ac = crit[u1c]); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    <> specialization subgoal
    instantiate c:Tasks[M] by task(crit[u1c]) in *impliesHyp / c-op(.first)
    prove u'c.basic.region[i] = s'c.basic.region[i]
      resume by case i = u1c
        <> case ic = u1c
        [] case ic = u1c
        <> case ¬(ic = u1c)
        instantiate j by ic in *impliesHyp, theorem
        [] case ¬(ic = u1c)
      [] conjecture
    prove
      (s'c.basic.region[i] = exit
        ⇒ s'c.bounds[task(rem[i])].last ≤ u'c.bounds[task(rem[i])].last)
      ..
      resume by case i = u1c
        <> case ic = u1c
        [] case ic = u1c
        <> case ¬(ic = u1c)
        resume by ⇒
          <> ⇒ subgoal
          instantiate c:Tasks[I] by task(rem[ic]) in *impliesHyp
```

```
     instantiate c:Tasks[M] by task(rem[ic]) in theorem
     instantiate j by ic in *impliesHyp
     instantiate i by ic in *impliesHyp / c-op(rem:→ActionTypes[M])
     [] ⇒ subgoal
    [] case ¬(ic = u1c)
   [] conjecture
  resume by ⇒
   <> ⇒ subgoal
   [] ⇒ subgoal
  [] specialization subgoal
 [] ⇒ subgoal
[] basis subgoal
<> basis subgoal
% CASE 3: a = addTime(exit[u1])
resume by ⇒
 <> ⇒ subgoal
 assert (ac = exit[u1c]); u'c = STEP(uc, addTime(ac))
 resume by specializing alpha to ({uc}) { addTime(ac), u'c }
  <> specialization subgoal
  instantiate c by task(ac) in *impliesHyp / c-op(.first)
  prove u'c.basic.region[i] = s'c.basic.region[i]
   resume by case i = u1c
    <> case ic = u1c
    [] case ic = u1c
    <> case ¬(ic = u1c)
    instantiate j by ic in *impliesHyp, theorem
    [] case ¬(ic = u1c)
   [] conjecture
  prove
   (s'c.basic.region[i] = exit
    ⇒ s'c.bounds[task(rem[i])].last ≤ u'c.bounds[task(rem[i])].last)
   ..
   resume by case i = u1c
    <> case ic = u1c
    [] case ic = u1c
    <> case ¬(ic = u1c)
    resume by ⇒
     <> ⇒ subgoal
     instantiate c:Tasks[I] by task(rem[ic]) in *impliesHyp
     instantiate c:Tasks[M] by task(rem[ic]) in theorem
     instantiate j by ic in *impliesHyp
     instantiate i by ic in *impliesHyp / c-op(rem:→ActionTypes[M])
     [] ⇒ subgoal
    [] case ¬(ic = u1c)
   [] conjecture
  resume by case ¬enabled(u'c.basic, task(crit[i]))
   <> case ¬enabled(u'c.basic, task(crit[ic]))
   instantiate c:Tasks[M] by task(crit[ic]) in theorem
   resume by case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    <> case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    [] case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    <> case ¬∀ j:UID ¬(s'c.basic.region[j] = crit)
    [] case ¬∀ j:UID ¬(s'c.basic.region[j] = crit)
   [] case ¬enabled(u'c.basic, task(crit[ic]))
   <> case ¬(¬enabled(u'c.basic, task(crit[ic])))
   prove ¬enabled(uc.basic, task(crit[i])) by contradiction
     <> contradiction subgoal
     instantiate j by u1c in *contraHyp
     [] contradiction subgoal
```

```
        [] conjecture

      instantiate c:Tasks[M] by task(crit[ic]) in theorem
      % We now know u'c.bounds[task(crit[ic])] = [s'c.now, (4*a) + (2*c) + s'c.now]
      prove inv(s'c)
        instantiate
          s by sc, s' by s'c, a:Actions[TI] by addTime(exit[u1c]) in Invariants
          ..
        [] conjecture
      resume by ∧
        <> ∧ subgoal
        prove ¬∀ j:UID ¬(sc.basic.region[j] = crit) by contradiction
            <> contradiction subgoal
            [] contradiction subgoal
          [] conjecture
        resume by ⇒
          <> ⇒ subgoal
          [] ⇒ subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        resume by ⇒
          <> ⇒ subgoal
          [] ⇒ subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        resume by ⇒
          <> ⇒ subgoal
          instantiate c:Tasks[I] by task(crit[ic]) in *impliesHyp
          instantiate c:Tasks[I] by task(crit[ic]) in theorem / c-op(.last)
          instantiate z by a+a+a+a+c+c+s'c.now in Transitivity
          [] ⇒ subgoal
        [] ∧ subgoal
      [] case ¬(¬enabled(u'c.basic, task(crit[ic])))
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
% CASE 4: a = addTime(rem[u1])
<> basis subgoal
resume by ⇒
  <> ⇒ subgoal
  assert (ac = rem[u1c]); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    <> specialization subgoal
    instantiate c by task(ac) in *impliesHyp / c-op(.first)
    prove u'c.basic.region[i] = s'c.basic.region[i]
      resume by case i = u1c
        <> case ic = u1c
        [] case ic = u1c
        <> case ¬(ic = u1c)
        instantiate j by ic in *impliesHyp, theorem
        [] case ¬(ic = u1c)
      [] conjecture
    prove
      (s'c.basic.region[i] = exit
        ⇒ s'c.bounds[task(rem[i])].last ≤ u'c.bounds[task(rem[i])].last)
      ..
      resume by case i = u1c
        <> case ic = u1c
        [] case ic = u1c
```

```
    <> case ¬(ic = u1c)
    resume by ⇒
      <> ⇒ subgoal
      instantiate c:Tasks[I] by task(rem[ic]) in *impliesHyp
      instantiate c:Tasks[M] by task(rem[ic]) in theorem
      instantiate j by ic in *impliesHyp
      instantiate i by ic in *impliesHyp / c-op(rem:→ActionTypes[M])
      [] ⇒ subgoal
    [] case ¬(ic = u1c)
  [] conjecture
resume by case ¬enabled(u'c.basic, task(crit[i]))
  <> case ¬enabled(u'c.basic, task(crit[ic]))
  instantiate c:Tasks[M] by task(crit[ic]) in theorem
  resume by case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    <> case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    [] case ∀ j:UID ¬(s'c.basic.region[j] = crit)
    <> case ¬∀ j:UID ¬(s'c.basic.region[j] = crit)
    [] case ¬∀ j:UID ¬(s'c.basic.region[j] = crit)
  [] case ¬enabled(u'c.basic, task(crit[ic]))
  <> case ¬(¬enabled(u'c.basic, task(crit[ic])))
  prove enabled(uc.basic, task(crit[ic]))
    declare operator i'c: → UID
    fix i:UID as i'c in *caseHyp
    resume by specializing i:UID to i'c
      <> specialization subgoal
      prove i'c ≠ u1c by contradiction
          <> contradiction subgoal
          [] contradiction subgoal
        [] conjecture
      instantiate j by i'c in *impliesHyp
      resume by contradiction
      <> contradiction subgoal
        prove jc ≠ u1c by contradiction
            <> contradiction subgoal
            [] contradiction subgoal
          [] conjecture
        instantiate j by jc in *impliesHyp
        [] contradiction subgoal
      [] specialization subgoal
    [] conjecture
  instantiate c:Tasks[M] by task(crit[ic]) in theorem
  resume by ∧
    <> ∧ subgoal
    resume by ⇒
      <> ⇒ subgoal
      instantiate c by task(seize) in *impliesHyp
      prove enabled(sc.basic, seize)
        declare operator i'c: → UID
        fix i:UID as i'c in (*impliesHyp / c-op(s'c)) ¬ c-op(sc)
        resume by specializing i:UID to i'c
          <> specialization subgoal
          prove i'c ≠ u1c by contradiction
              <> contradiction subgoal
              [] contradiction subgoal
            [] conjecture
          instantiate j by i'c in *impliesHyp
          resume by contradiction
            <> contradiction subgoal
            prove jc ≠ u1c by contradiction
```

```
                      <> contradiction subgoal
                      [] contradiction subgoal
                    [] conjecture
                  instantiate j by jc in*impliesHyp
                  [] contradiction subgoal
                [] specialization subgoal
              [] conjecture
            [] ⇒ subgoal
          [] ∧ subgoal
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            [] ⇒ subgoal
          [] ∧ subgoal
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            declare operator i'c: → UID
            fix i:UID as i'c in *impliesHyp / c-op(stable:→Status)
            [] ⇒ subgoal
          [] ∧ subgoal
        [] case ¬(¬enabled(u'c.basic, task(crit[ic])))
      [] specialization subgoal
    [] ⇒ subgoal
  [] basis subgoal
[] basis subgoal
<> basis subgoal
% CASE 5: a = addTime(seize)
resume by ⇒
  <> ⇒ subgoal
  resume by specializing alpha to {uc}
    <> specialization subgoal
    prove
      (sc.basic.region[i] = exit
        ⇒ s'c.bounds[task(rem[i])].last ≤ uc.bounds[task(rem[i])].last)
      ..
      resume by ⇒
        <> ⇒ subgoal
        instantiate c:Tasks[I] by task(rem[ic]) in *impliesHyp
        instantiate c:Tasks[M] by task(rem[ic]) in theorem
        instantiate i by ic in *impliesHyp / c-op(rem:→ActionTypes[M])
        [] ⇒ subgoal
      [] conjecture
    instantiate c by task(stabilize) in *impliesHyp
    prove  (a + a + c + s'c.now) ≤ (sc.bounds[task(seize)].last + a + a + c)
      resume by case a+a+c = infinity
        <> case a + a + c = infinity
        [] case a + a + c = infinity
        <> case ¬(a + a + c = infinity)
        instantiate t by a+a+c in Time
        [] case ¬(a + a + c = infinity)
      [] conjecture
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal
% CASE 6: a = addTime(stabilize)
resume by ⇒
  <> ⇒ subgoal
```

```
        resume by specializing alpha to {uc}
          <> specialization subgoal
          prove
            (sc.basic.region[i] = exit
               ⇒ s'c.bounds[task(rem[i])].last ≤ uc.bounds[task(rem[i])].last)
            ..
            resume by ⇒
              <> ⇒ subgoal
              instantiate c:Tasks[I] by task(rem[ic]) in *impliesHyp
              instantiate c:Tasks[M] by task(rem[ic]) in theorem
              instantiate i by ic in *impliesHyp / c-op(rem:→ActionTypes[M])
              [] ⇒ subgoal
            [] conjecture
          instantiate c:Tasks[I] by task(crit[i]) in *impliesHyp
          prove (a+c+s'c.now) ≤ (sc.bounds[task(stabilize)].last + a + c)
            resume by case a+c = infinity
              <> case a + c = infinity
              [] case a + c = infinity
              <> case ¬(a + c = infinity)
              instantiate t by a+c in Time
              [] case ¬(a + c = infinity)
            [] conjecture
          [] specialization subgoal
        [] ⇒ subgoal
      [] basis subgoal
    [] basis subgoal
  [] conjecture
qed

set log i2m
statistics
quit
```

A.6 The Simulation from Fischer's Algorithm to the Milestones

```
set script f2i
thaw F2I
set name theorem
set immunity ancestor
set box-checking on

%%%%%% Preliminaries

% Put information in registry to speed up ordering

register height
  __[__]:Bounds[F],Tasks[F]→Bounds
    > (.first, .last, bdmap:Tasks[F]→Bounds, ∃:Actions[F],Bool→Bool,
       .basic:States[TF]→States[F], +:Time,Time→Time)
  ..
register height
  __[__]:Bounds[I],Tasks[I]→Bounds
    > (.first, .last, bdmap:Tasks[I]→Bounds, ∃:Actions[I],Bool→Bool,
       .basic:States[TI]→States[I], +:Time,Time→Time)
  ..
register height
```

```
  .bounds:States[TF]→Bounds[F]
     > (.basic:States[TF]→States[F], enabled:States[F],Actions[F]→Bool)
  ..
register height
  .bounds:States[TI]→Bounds[I]
     > (.basic:States[TI]→States[I], enabled:States[I],Actions[I]→Bool)
  ..
register height __[__]:Regions,UID→Region > ∨


% Introduce constants that will be used to replace variables

declare operators
  uc:          → States[TI]    % Used by LP for u in hypotheses
  sc, s'c:     → States[TF]    % Ditto for s and s'
  u'c, u''c:   → States[TI]    % To abbreviate STEP(uc, ...), STEP(u'c, ...)
  ac, a'c:     → Actions[I]    % To abbreviate actions
  ..

% Put information in registry to ensure intended orientation of equations

register top uc, u'c, u''c
register height u''c > u'c > uc
register height s'c > sc

% Define some abbreviations for useful sets of facts

define-class $firstHyp     *impliesHyp / contains-operator(.first)
define-class $wdef         (F2I / contains-operator(w)) ¬ contains-operator(f)

% Some preliminary lemmas
prove effect(s, a, STEP(s, a))
  rewrite conjecture with reversed F2I
  [] conjecture
qed

prove
  inv(s:States[TF]) ∧ isStep(s:States[TF], addTime(at[i]), s')
   ⇒ ∀ j (j:UID ≠ i ⇒ w(s',j) = w(s,j))
  ..
  resume by ⇒
    <> ⇒ subgoal
    resume by ⇒
      <> ⇒ subgoal
      instantiate
        s by sc, s' by s'c, at by atc, i by ic, j by jc
        in TimedFischer / c-v(at':ActionTypes[F])
        ..
      instantiate
        s by sc.basic, s' by s'c.basic, at by atc, i by ic, j by jc
        in AutomatonFischer / c-v(j:UID)
        ..
      instantiate s by sc, i by jc in $wdef
      instantiate s by s'c, i by jc in $wdef
      resume by cases
        sc.basic.pc[jc] = test,
        sc.basic.pc[jc] = set,
        sc.basic.pc[jc] = check,
        ¬(sc.basic.pc[jc] ∈ { test, set, check })
```

```
        ..
        <> case justification
        [] case justification
        <> case sc.basic.pc[jc] = test
        [] case sc.basic.pc[jc] = test
        <> case sc.basic.pc[jc] = set
        [] case sc.basic.pc[jc] = set
        <> case sc.basic.pc[jc] = check
        [] case sc.basic.pc[jc] = check
        <> case ¬(sc.basic.pc[jc] ∈ {test, set, check})
        [] case ¬(sc.basic.pc[jc] ∈ {test, set, check})
      [] ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

% Now the proof obligations to check the forward simulation

prove start(s:States[TF]) ⇒ ∃ u (f(s,u) ∧ start(u)) by ⇒
    <> ⇒ subgoal
    declare operators nullbounds: → Bounds[I], startregs: → Regions
    assert ∀ i:UID (startregs[i] = rem)
    assert ∀ c:Tasks[I] (nullbounds[c] = unbounded)
    resume by specializing u to [[startregs, start], 0, nullbounds]
      <> specialization subgoal
      resume by induction on c:Tasks[I]
        <> basis subgoal
        resume by induction on a3
          <> basis subgoal
          resume by induction on a1
            <> basis subgoal
            [] basis subgoal
            <> basis subgoal
            [] basis subgoal
            <> basis subgoal
            [] basis subgoal
            <> basis subgoal
            [] basis subgoal
          [] basis subgoal
          <> basis subgoal
          [] basis subgoal
          <> basis subgoal
          [] basis subgoal
        [] basis subgoal
      [] specialization subgoal
    [] ⇒ subgoal
  [] conjecture
qed

prove f(s,u) ⇒ s:States[TF].now = u.now by ⇒
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] conjecture
qed

set proof-method explicit-commands
declare variable alpha:StepSeq[TI]
prove
  f(s,u) ∧ isStep(s:States[TF], a, s') ∧ inv(s:States[TF]) ∧ inv(u) ∧ StrongMutex(s)
```

```
  ⇒ ∃ alpha (execFrag(alpha)
             ∧ first(alpha) = u ∧ f(s',last(alpha))
             ∧ trace(alpha) = trace(a:Actions[TF]) )
by induction on a:Actions[TF]
..
  <> basis subgoal

  % CASE: nu(s'c.now)
  resume by ⇒
    <> ⇒ subgoal
    set proof-methods normalization
    assert u'c = STEP(uc, nu(s'c.now))
    resume by specializing alpha to ({uc}) { nu(s'c.now), u'c }
      <> specialization subgoal

      resume by ∧
        <> ∧ subgoal
        % Trying to prove ∀ c:Tasks[I] (s'c.now ≤ uc.bounds[c].last)
        resume by induction on c:Tasks[I]
          <> basis subgoal

          prove s'c.now ≤ (sc.bounds[c].last + t)
            instantiate t by sc.bounds[c].last, t1:Time by t in Time
            [] conjecture

          resume by induction on a3
            <> basis subgoal
            resume by induction on a1
              <> basis subgoal

              % CASE 1: c = task(try[u1])
              instantiate c:Tasks[I] by task(try[u1]) in *impliesHyp
              [] basis subgoal
              <> basis subgoal

              % CASE 2: c = task(crit[u1])
              resume by case ¬enabled(uc.basic, task(crit[u1]))
                <> case ¬enabled(uc.basic, task(crit[u1c]))
                instantiate c:Tasks[I] by task(crit[u1c]) in *impliesHyp
                [] case ¬enabled(uc.basic, task(crit[u1c]))
                <> case ¬(¬enabled(uc.basic, task(crit[u1c])))
                declare operator dummyi:→UID
                fix i as dummyi in *caseHyp
                declare operator critTask:→Tasks[I]
                assert task(crit[u1c]) = critTask
                prove critTask = task(crit[sc.basic.x.owner])
                  instantiate
                    a:Actions[I] by crit[u1c], a':Actions[I] by crit[sc.basic.x.owner]
                    in AutomatonIntermediate
                    ..
                  [] conjecture

                resume by case sc.basic.pc[sc.basic.x.owner] = lvtry
                  <> case sc.basic.pc[sc.basic.x.owner] = lvtry
                  instantiate i by sc.basic.x.owner in *impliesHyp / c-op(lvtry)
                  [] case sc.basic.pc[sc.basic.x.owner] = lvtry
                  <> case ¬(sc.basic.pc[sc.basic.x.owner] = lvtry)
                  [] case ¬(sc.basic.pc[sc.basic.x.owner] = lvtry)
                [] case ¬(¬enabled(uc.basic, task(crit[u1c])))
```

```
  [] basis subgoal
  <> basis subgoal

  % CASE 3: c = task(exit[u1])
  instantiate c:Tasks[I] by task(exit[u1]) in *impliesHyp
  [] basis subgoal
  <> basis subgoal

  % CASE 4: c = task(rem[u1])
  resume by case ¬enabled(uc.basic, task(rem[u1]))
    <> case ¬enabled(uc.basic, task(rem[u1c]))
    instantiate c:Tasks[I] by task(rem[u1c]) in *impliesHyp
    [] case ¬enabled(uc.basic, task(rem[u1c]))
    <> case ¬(¬enabled(uc.basic, task(rem[u1c])))
    instantiate i by u1c in *impliesHyp
    resume by case sc.basic.pc[u1c] = reset
      <> case sc.basic.pc[u1c] = reset
      [] case sc.basic.pc[u1c] = reset
      <> case ¬(sc.basic.pc[u1c] = reset)
      [] case ¬(sc.basic.pc[u1c] = reset)
    [] case ¬(¬enabled(uc.basic, task(rem[u1c])))
  [] basis subgoal
[] basis subgoal
<> basis subgoal

% CASE 5: c = task(seize)
resume by case ¬enabled(uc.basic, task(seize))
  <> case ¬enabled(uc.basic, task(seize))
  instantiate c by task(seize) in *impliesHyp
  [] case ¬enabled(uc.basic, task(seize))
  <> case ¬(¬enabled(uc.basic, task(seize)))
  resume by case sc.basic.x.free
    <> case sc.basic.x.free
    declare operator ic:→UID
    fix i as ic in *hyp / c-op(seize)
    prove w(sc, ic) ≥ s'c.now
      instantiate i by ic, s by sc in $wdef
      resume by cases
        sc.basic.pc[ic] = test,
        sc.basic.pc[ic] = set,
        sc.basic.pc[ic] = check,
        ¬(sc.basic.pc[ic] ∈ { test, set, check })
        ..
        <> case justification
        [] case justification
        <> case sc.basic.pc[ic] = test
        [] case sc.basic.pc[ic] = test
        <> case sc.basic.pc[ic] = set
        [] case sc.basic.pc[ic] = set
        <> case sc.basic.pc[ic] = check
        [] case sc.basic.pc[ic] = check
        <> case ¬(sc.basic.pc[ic] ∈ {test, set, check})
        [] case ¬(sc.basic.pc[ic] ∈ {test, set, check})
      [] conjecture
    [] case sc.basic.x.free
    <> case ¬sc.basic.x.free

    % Now we know ∃ i:UID (sc.basic.pc[i] = reset)
    declare operator ic:→UID
```

```
            fix i as ic in *caseHyp / c-op(reset:→PC)
            instantiate c by task(reset[ic]) in *impliesHyp / c-op(.last)
            instantiate i by ic in *impliesHyp / c-op(reset:→PC)
            [] case ¬sc.basic.x.free
          [] case ¬(¬enabled(uc.basic, task(seize)))
        [] basis subgoal
        <> basis subgoal

        % CASE 6: c = task(stabilize)
        instantiate c by task(stabilize) in *impliesHyp
        resume by case ¬enabled(uc.basic, task(stabilize))
          <> case ¬enabled(uc.basic, task(stabilize))
          [] case ¬enabled(uc.basic, task(stabilize))
          <> case ¬(¬enabled(uc.basic, task(stabilize)))
          declare operator ic:→UID
          fix i as ic in *caseHyp
          instantiate i by ic in *impliesHyp / c-op(set:→PC)
          [] case ¬(¬enabled(uc.basic, task(stabilize)))
        [] basis subgoal
      [] basis subgoal
    [] ∧ subgoal
    <> ∧ subgoal
    resume by ∧
      <> ∧ subgoal
      [] ∧ subgoal
      <> ∧ subgoal
      prove w(s'c, i) = w(sc, i)
        declare operator ic:→UID
        resume by generalizing i:UID from ic
          <> generalization subgoal
          instantiate s by sc, i by ic in $wdef
          instantiate s by s'c, i by ic in $wdef
          resume by cases
            sc.basic.pc[ic] = test,
            sc.basic.pc[ic] = set,
            sc.basic.pc[ic] = check,
            ¬(sc.basic.pc[ic] ∈ { test, set, check })
            ..
            <> case justification
            [] case justification
            <> case sc.basic.pc[ic] = test
            [] case sc.basic.pc[ic] = test
            <> case sc.basic.pc[ic] = set
            [] case sc.basic.pc[ic] = set
            <> case sc.basic.pc[ic] = check
            [] case sc.basic.pc[ic] = check
            <> case ¬(sc.basic.pc[ic] ∈ {test, set, check})
            [] case ¬(sc.basic.pc[ic] ∈ {test, set, check})
          [] generalization subgoal
        [] conjecture
      [] ∧ subgoal
    [] ∧ subgoal
  [] specialization subgoal
 [] ⇒ subgoal
[] basis subgoal
<> basis subgoal
resume by induction on a7
  <> basis subgoal
  resume by induction on a5
```

```
<> basis subgoal

% CASE 1: try[u1c]
resume by ⇒
  <> ⇒ subgoal
  set proof-methods normalization
  assert (ac = try[u1c]); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    <> specialization subgoal
    instantiate c by task(ac) in $firstHyp

    prove
       (s'c.basic.pc[i] = crit ⇔ sc.basic.pc[i] = crit)
        ∧ (s'c.basic.pc[i] = reset ⇔ sc.basic.pc[i] = reset)
        ∧ (s'c.basic.pc[i] = set ⇔ sc.basic.pc[i] = set)
      ..
      resume by case i = u1c
        <> case ic = u1c
        [] case ic = u1c
        <> case ¬(ic = u1c)
        instantiate j by ic in *impliesHyp / c-op(u1c)
        [] case ¬(ic = u1c)
      [] conjecture

    prove ∃ i:UID (w(s'c, i) ≤ u'c.bounds[task(seize)].last)
      instantiate c by task(seize) in theorem
      resume by case ¬enabled(u'c.basic, seize)
        <> case ¬enabled(u'c.basic, seize)
        [] case ¬enabled(u'c.basic, seize)
        <> case ¬(¬enabled(u'c.basic, seize))
        resume by case enabled(uc.basic, seize)
          <> case enabled(uc.basic, seize)
          declare operator ic:→UID
          fix i as ic in *hyp / c-op(w)
          resume by specializing i:UID to ic
            <> specialization subgoal
            instantiate
              s by sc, s' by s'c, at: ActionTypes[F] by try, i by u1c, j by ic
              in theorem / c-op(w)
              ..
            resume by case ic = u1c
              <> case ic = u1c
              instantiate s by sc, i by u1c in $wdef
              [] case ic = u1c
              <> case ¬(ic = u1c)
              [] case ¬(ic = u1c)
            [] specialization subgoal
          [] case enabled(uc.basic, seize)
          <> case ¬enabled(uc.basic, seize)
          resume by specializing i:UID to u1c
            <> specialization subgoal
            instantiate s by s'c, i by u1c in $wdef
            [] specialization subgoal
          [] case ¬enabled(uc.basic, seize)
        [] case ¬(¬enabled(u'c.basic, seize))
      [] conjecture

    resume by case i = u1c
      <> case ic = u1c
```

```
[] case ic = u1c
<> case ¬(ic = u1c)
instantiate j by ic in (*impliesHyp, theorem) / c-op(u1c)
resume by ∧
  <> ∧ subgoal
  instantiate c:Tasks[I] by task(crit[ic]) in theorem / c-op(u'c)
  resume by case ¬enabled(u'c.basic, task(crit[ic]))
    <> case ¬enabled(u'c.basic, task(crit[ic]))
    [] case ¬enabled(u'c.basic, task(crit[ic]))
    <> case ¬(¬enabled(u'c.basic, task(crit[ic])))
    resume by ⇒
      <> ⇒ subgoal
      declare operator dummyi:→UID
      fix i as dummyi in *caseHyp
      [] ⇒ subgoal
    [] case ¬(¬enabled(u'c.basic, task(crit[ic])))
  [] ∧ subgoal
  <> ∧ subgoal
  instantiate c:Tasks[I] by task(rem[ic]) in theorem / c-op(u'c)
  resume by ⇒
    <> ⇒ subgoal
    instantiate i by ic in *impliesHyp
    [] ⇒ subgoal
  [] ∧ subgoal
  <> ∧ subgoal
  resume by case ¬enabled(u'c.basic, task(crit[ic]))
    <> case ¬enabled(u'c.basic, task(crit[ic]))
    [] case ¬enabled(u'c.basic, task(crit[ic]))
    <> case ¬(¬enabled(u'c.basic, task(crit[ic])))
    resume by ⇒
      <> ⇒ subgoal
      declare operator dummyi: → UID
      fix i as dummyi in *caseHyp
      instantiate i by ic in *impliesHyp
      [] ⇒ subgoal
    [] case ¬(¬enabled(u'c.basic, task(crit[ic])))
  [] ∧ subgoal
  <> ∧ subgoal
  resume by ⇒
    <> ⇒ subgoal
    instantiate i by ic in *impliesHyp
    [] ⇒ subgoal
  [] ∧ subgoal
  <> ∧ subgoal
  instantiate c by task(seize) in theorem / c-op(u'c)
  resume by case ¬enabled(u'c.basic, task(seize))
    <> case ¬enabled(u'c.basic, task(seize))
    [] case ¬enabled(u'c.basic, task(seize))
    <> case ¬(¬enabled(u'c.basic, task(seize)))
    resume by ⇒
      <> ⇒ subgoal
      instantiate c by task(reset[ic]) in *impliesHyp
      instantiate i by ic in *impliesHyp
      resume by case enabled(uc.basic, task(seize))
        <> case enabled(uc.basic, task(seize))
        [] case enabled(uc.basic, task(seize))
        <> case ¬enabled(uc.basic, task(seize))
        instantiate t by a+a+c in Time
        resume by case a+a+c = infinity
```

```
                <> case a + a + c = infinity
                [] case a + a + c = infinity
                <> case ¬(a + a + c = infinity)
                prove sc.bounds[task(reset[ic])].last ≤ (sc.now + c)
                  instantiate c by task(reset[ic]) in *hyp
                  prove (sc.now +a) ≤ (sc.now + c)
                    instantiate t by sc.now in Time
                    [] conjecture
                  [] conjecture
                instantiate t by a in Time
                [] case ¬(a + a + c = infinity)
              [] case ¬enabled(uc.basic, task(seize))
            [] ⇒ subgoal
          [] case ¬(¬enabled(u'c.basic, task(seize)))
        [] ∧ subgoal
        <> ∧ subgoal
        instantiate c by task(stabilize) in theorem / c-op(u'c)
        resume by cases ¬enabled(u'c.basic, task(stabilize))
          <> case ¬enabled(u'c.basic, task(stabilize))
          [] case ¬enabled(u'c.basic, task(stabilize))
          <> case ¬(¬enabled(u'c.basic, task(stabilize)))
          resume by ⇒
            <> ⇒ subgoal
            instantiate c by task(set[ic]) in *impliesHyp / c-op(s'c)
            instantiate i by ic in *impliesHyp
            [] ⇒ subgoal
          [] case ¬(¬enabled(u'c.basic, task(stabilize)))
        [] ∧ subgoal
      [] case ¬(ic = u1c)
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal

% CASE 2: test[u1c]
resume by ⇒
  <> ⇒ subgoal
  set proof-methods normalization
  resume by specializing alpha to {uc}
    <> specialization subgoal
    resume by case sc.basic.x.free
      <> case sc.basic.x.free
      prove ∃ i:UID (w(s'c, i) ≤ uc.bounds[task(seize)].last)
        declare operator ic:→UID
        fix i as ic in *hyp / c-op(w)
        resume by specializing i:UID to ic
          <> specialization subgoal
          resume by case ic = u1c
            <> case ic = u1c
            instantiate s by s'c, i by u1c in $wdef
            instantiate s by sc, i by u1c in $wdef
            instantiate
              t by a, t1 by sc.now, t2 by sc.bounds[task(test[u1c])].last in Time
              ..
            [] case ic = u1c
            <> case ¬(ic = u1c)
            instantiate
              s by sc, s' by s'c, at by test, i by u1c, j by ic in theorem / c-op(w)
              ..
```

```
      [] case ¬(ic = u1c)
    [] specialization subgoal
  [] conjecture
resume by case i = u1c
  <> case ic = u1c
  instantiate c by task(stabilize) in *impliesHyp
  [] case ic = u1c
  <> case ¬(ic = u1c)
  instantiate j by ic in *impliesHyp / c-op(u1c)
  resume by ∧
    <> ∧ subgoal
    resume by ⇒
      <> ⇒ subgoal
      instantiate c:Tasks[F] by task(rem[ic]) in *impliesHyp / c-op(s'c)
      instantiate i by ic in *impliesHyp
      [] ⇒ subgoal
    [] ∧ subgoal
    <> ∧ subgoal
    resume by ⇒
      <> ⇒ subgoal
      instantiate c by task(set[ic]) in *impliesHyp / c-op(s'c)
      instantiate i by ic in *impliesHyp
      [] ⇒ subgoal
    [] ∧ subgoal
  [] case ¬(ic = u1c)
[] case sc.basic.x.free
<> case ¬sc.basic.x.free
prove s'c.basic.pc[i] = sc.basic.pc[i]
  resume by case i = u1c
    <> case ic = u1c
    [] case ic = u1c
    <> case ¬(ic = u1c)
    instantiate j by ic in *impliesHyp / c-op(u1c)
    [] case ¬(ic = u1c)
  [] conjecture
resume by ∧
  <> ∧ subgoal
  resume by ⇒
    <> ⇒ subgoal
    [] ⇒ subgoal
  [] ∧ subgoal
  <> ∧ subgoal
  resume by ⇒
    <> ⇒ subgoal
    instantiate i by ic in *impliesHyp
    [] ⇒ subgoal
  [] ∧ subgoal
  <> ∧ subgoal
  resume by ⇒
    <> ⇒ subgoal
    instantiate i by ic in *impliesHyp
    [] ⇒ subgoal
  [] ∧ subgoal
  <> ∧ subgoal
  resume by ⇒
    <> ⇒ subgoal
    instantiate i by ic in *impliesHyp
    [] ⇒ subgoal
  [] ∧ subgoal
```

```
        <> ∧ subgoal
        resume by ⇒
          <> ⇒ subgoal
          instantiate i by ic in *impliesHyp
          [] ⇒ subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        resume by ⇒
          <> ⇒ subgoal
          instantiate i by ic in *impliesHyp
          [] ⇒ subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        resume by case enabled(uc.basic, seize)
          <> case enabled(uc.basic, seize)
          declare operator ic:→ UID
          fix i as ic in *caseHyp / c-op(reset:→PC)
          resume by specializing i:UID to u1c
            <> specialization subgoal
            instantiate i by ic in *impliesHyp / c-op(seize)
            instantiate s by s'c, i by u1c in $wdef
            instantiate c by task(reset[ic]) in *impliesHyp
            prove (sc.now + a+a) ≤ (sc.bounds[task(reset[ic])].last +a+a)
              instantiate t by a in Time
              [] conjecture
            [] specialization subgoal
          [] case enabled(uc.basic, seize)
          <> case ¬enabled(uc.basic, seize)
          instantiate c by task(seize) in *hyp
          [] case ¬enabled(uc.basic, seize)
        [] ∧ subgoal
      [] case ¬sc.basic.x.free
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal

% CASE 3: set[u1]
resume by ⇒
  <> ⇒ subgoal
  instantiate j by s'c.basic.x.owner in *impliesHyp
  % StrongMutex ⇒ sc.basic.pc[i] ≠ crit, lvtry, reset
  set proof-methods normalization

  prove s'c.basic.pc[i] ∉ { crit, reset }
    resume by case i = s'c.basic.x.owner
      <> case ic = s'c.basic.x.owner
      [] case ic = s'c.basic.x.owner
      <> case ¬(ic = s'c.basic.x.owner)
      instantiate j by ic in *impliesHyp
      [] case ¬(ic = s'c.basic.x.owner)
    [] conjecture

  resume by case sc.basic.x.free
    <> case sc.basic.x.free
    prove ∃ i:UID (uc.basic.region[i] = try)
      resume by specializing i:UID to s'c.basic.x.owner
        <> specialization subgoal
        [] specialization subgoal
```

```
  [] conjecture

resume by case ∀ j:UID (s'c.basic.pc[j] ≠ set)
  <> case ∀ j:UID (s'c.basic.pc[j] ≠ set)

  assert (ac = seize); u'c = STEP(uc, addTime(ac))
  assert a'c = stabilize; u''c = STEP(u'c, addTime(a'c))
  resume by specializing
    alpha to (({uc}) { addTime(ac), u'c }) { addTime(a'c), u''c }
    ..
    <> specialization subgoal
    instantiate c by task(ac) in $firstHyp

    resume by case i = s'c.basic.x.owner
      <> case ic = s'c.basic.x.owner
      [] case ic = s'c.basic.x.owner
      <> case ¬(ic = s'c.basic.x.owner)
      instantiate j by ic in *Hyp
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] case ¬(ic = s'c.basic.x.owner)
    [] specialization subgoal
  [] case ∀ j:UID (s'c.basic.pc[j] ≠ set)
  <> case ¬∀ j:UID (s'c.basic.pc[j] ≠ set)
  assert (ac = seize); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    <> specialization subgoal
    instantiate c by task(ac) in $firstHyp

    prove ∃ i:UID (s'c.basic.pc[i] = set)
      declare operator ic:→UID
      fix j as ic in *caseHyp
      resume by specializing i:UID to ic
        <> specialization subgoal
        [] specialization subgoal
      [] conjecture

    resume by case i = s'c.basic.x.owner
      <> case ic = s'c.basic.x.owner
      [] case ic = s'c.basic.x.owner
      <> case ¬(ic = s'c.basic.x.owner)
      instantiate j by ic in *impliesHyp
      resume by ∧
        <> ∧ subgoal
        resume by ⇒
          <> ⇒ subgoal
          instantiate i by ic in *impliesHyp
          [] ⇒ subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        resume by ⇒
          <> ⇒ subgoal
          instantiate c by task(set[ic]) in *impliesHyp
          [] ⇒ subgoal
        [] ∧ subgoal
      [] case ¬(ic = s'c.basic.x.owner)
    [] specialization subgoal
```

```
  [] case ¬∀ j:UID (s'c.basic.pc[j] ≠ set)
[] case sc.basic.x.free
<> case ¬sc.basic.x.free
resume by case ∀ j:UID (s'c.basic.pc[j] ≠ set)
  <> case ∀ j:UID (s'c.basic.pc[j] ≠ set)
  assert (ac = stabilize); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    <> specialization subgoal
    instantiate c by task(ac) in $firstHyp
    prove ∃ i:UID (sc.basic.pc[i] = set)
      resume by specializing i:UID to s'c.basic.x.owner
        <> specialization subgoal
        [] specialization subgoal
      [] conjecture

    resume by case i = s'c.basic.x.owner
      <> case ic = s'c.basic.x.owner
      prove ¬∀ i:UID ¬(sc.basic.pc[i] = set) by contradiction
        <> contradiction subgoal
          [] contradiction subgoal
        [] conjecture
      [] case ic = s'c.basic.x.owner
      <> case ¬(ic = s'c.basic.x.owner)
      instantiate j by ic in *Hyp
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] case ¬(ic = s'c.basic.x.owner)
    [] specialization subgoal
  [] case ∀ j:UID (s'c.basic.pc[j] ≠ set)
  <> case ¬∀ j:UID (s'c.basic.pc[j] ≠ set)
  resume by specializing alpha to {uc}
    <> specialization subgoal

    prove uc.bounds[task(seize)].last = infinity
      instantiate c by task(seize) in *hyp
      [] conjecture

    prove ∃ i:UID (sc.basic.pc[i] = set)
      resume by specializing i:UID to s'c.basic.x.owner
        <> specialization subgoal
        [] specialization subgoal
      [] conjecture

    prove ∃ i:UID (s'c.basic.pc[i] = set)
      declare operator ic:→UID
      fix j as ic in *caseHyp
      resume by specializing i:UID to ic
        <> specialization subgoal
        [] specialization subgoal
      [] conjecture

    prove ¬∀ i:UID ¬(sc.basic.pc[i] = set) by contradiction
        <> contradiction subgoal
        [] contradiction subgoal
      [] conjecture

    resume by case i = s'c.basic.x.owner
```

```
          <> case ic = s'c.basic.x.owner
          instantiate c:Tasks[I] by task(crit[ic]) in *impliesHyp
          [] case ic = s'c.basic.x.owner
          <> case ¬(ic = s'c.basic.x.owner)
          instantiate j by ic in *impliesHyp
          instantiate i by ic in *impliesHyp
          resume by ∧
            <> ∧ subgoal
            resume by ⇒
              <> ⇒ subgoal
              [] ⇒ subgoal
            [] ∧ subgoal
            <> ∧ subgoal
            resume by ⇒
              <> ⇒ subgoal
              [] ⇒ subgoal
            [] ∧ subgoal
          [] case ¬(ic = s'c.basic.x.owner)
        [] specialization subgoal
      [] case ¬∀ j:UID (s'c.basic.pc[j] ≠ set)
    [] case ¬sc.basic.x.free
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal

% CASE 4: check[u1c]
resume by ⇒
  <> ⇒ subgoal
  set proof-methods normalization
  resume by specializing alpha to {uc}
    <> specialization subgoal

    prove ∃ i:UID (w(s'c, i) ≤ uc.bounds[task(seize)].last)
      declare operator ic:→UID
      fix i as ic in *hyp / c-op(w)
      resume by specializing i:UID to ic
        <> specialization subgoal
        resume by case ic = u1c
          <> case ic = u1c
          resume by case ¬sc.basic.x.free ∧ sc.basic.x.owner = u1c
            <> case ¬sc.basic.x.free ∧ sc.basic.x.owner = u1c
            prove ¬enabled(uc.basic, seize) by contradiction
                <> contradiction subgoal
                declare operator ic1:→UID
                fix i as ic1 in *contraHyp / c-op(reset:→PC)
                instantiate i by ic1 in *impliesHyp
                [] contradiction subgoal
              [] conjecture
            instantiate c by task(seize) in *hyp
            [] case ¬sc.basic.x.free ∧ sc.basic.x.owner = u1c
            <> case ¬(¬sc.basic.x.free ∧ sc.basic.x.owner = u1c)
            instantiate s by s'c, i by u1c in $wdef
            instantiate s by sc, i by u1c in $wdef
            prove (sc.now + a+a) ≤ (sc.bounds[task(check[u1c])].last + a+a)
              instantiate t by a in Time
              [] conjecture
            [] case ¬(¬sc.basic.x.free ∧ sc.basic.x.owner = u1c)
          [] case ic = u1c
          <> case ¬(ic = u1c)
```

```
      instantiate
        s by sc, s' by s'c, at by check, i by u1c, j by ic in theorem / c-op(w)
        ..
      [] case ¬(ic = u1c)
    [] specialization subgoal
  [] conjecture

resume by case sc.basic.x.free
  <> case sc.basic.x.free
  resume by case i = u1c
    <> case ic = u1c
    [] case ic = u1c
    <> case ¬(ic = u1c)
    instantiate j by ic in *impliesHyp / c-op(u1c)
    resume by ∧
      <> ∧ subgoal
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] ∧ subgoal
      <> ∧ subgoal
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] ∧ subgoal
    [] case ¬(ic = u1c)
  [] case sc.basic.x.free
  <> case ¬sc.basic.x.free

  prove
     (s'c.basic.pc[i] = crit ⇔ sc.basic.pc[i] = crit)
      ∧ (s'c.basic.pc[i] = reset ⇔ sc.basic.pc[i] = reset)
      ∧ (s'c.basic.pc[i] = set ⇔ sc.basic.pc[i] = set)
    ..
    resume by case i = u1c
      <> case ic = u1c
      resume by case sc.basic.x.owner = u1c
        <> case sc.basic.x.owner = u1c
        [] case sc.basic.x.owner = u1c
        <> case ¬(sc.basic.x.owner = u1c)
        [] case ¬(sc.basic.x.owner = u1c)
      [] case ic = u1c
      <> case ¬(ic = u1c)
      instantiate j by ic in *impliesHyp
      [] case ¬(ic = u1c)
    [] conjecture

  resume by case i = u1c
    <> case ic = u1c
    resume by case sc.basic.x.owner = u1c
      <> case sc.basic.x.owner = u1c
      prove (a + sc.now) ≤ (sc.bounds[task(check[u1c])].last + a)
        instantiate t by a in Time
        [] conjecture
      [] case sc.basic.x.owner = u1c
      <> case ¬(sc.basic.x.owner = u1c)
      [] case ¬(sc.basic.x.owner = u1c)
```

```
        [] case ic = u1c
        <> case ¬(ic = u1c)
        instantiate j by ic in *impliesHyp / c-op(u1c)
        resume by ∧
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            [] ⇒ subgoal
          [] ∧ subgoal
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            instantiate i by ic in *impliesHyp
            [] ⇒ subgoal
          [] ∧ subgoal
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            instantiate i by ic in *impliesHyp
            [] ⇒ subgoal
          [] ∧ subgoal
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            instantiate i by ic in *impliesHyp
            [] ⇒ subgoal
          [] ∧ subgoal
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            instantiate i by ic in *impliesHyp
            [] ⇒ subgoal
          [] ∧ subgoal
          <> ∧ subgoal
          resume by ⇒
            <> ⇒ subgoal
            instantiate i by ic in *impliesHyp
            [] ⇒ subgoal
          [] ∧ subgoal
        [] case ¬(ic = u1c)
      [] case ¬sc.basic.x.free
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal

% CASE 5: crit[u1c]
resume by ⇒
  <> ⇒ subgoal
  instantiate i by u1c in *impliesHyp  % StrongMutex!
  set proof-methods normalization
  assert (ac = crit[u1c]); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    <> specialization subgoal
    instantiate c by task(ac) in $firstHyp
    resume by ∧
      <> ∧ subgoal

      prove
```

```
  ¬(  ∃ i:UID (s'c.basic.pc[i] = set)
       ∧ ∀ i:UID (s'c.basic.pc[i] ∉ { crit, reset }))
     ∧  ¬∀ i:UID (s'c.basic.pc[i] ∉ { crit, reset, set })
     ∧ ∃ i:UID (s'c.basic.pc[i] ∈ { crit, reset })
    ..
    resume by ∧
      <> ∧ subgoal
      resume by contradiction
        <> contradiction subgoal
        [] contradiction subgoal
      [] ∧ subgoal
      <> ∧ subgoal
      resume by specializing i:UID to u1c
        <> specialization subgoal
        [] specialization subgoal
      [] ∧ subgoal
      <> ∧ subgoal
      resume by specializing i:UID to u1c
        <> specialization subgoal
        [] specialization subgoal
      [] ∧ subgoal
    [] conjecture

  prove u'c.bounds[task(seize)].last = infinity
    prove ¬enabled(u'c.basic, seize) by contradiction
        <> contradiction subgoal
        [] contradiction subgoal
      [] conjecture
    [] conjecture

  resume by case i = u1c
    <> case ic = u1c
    [] case ic = u1c
    <> case ¬(ic = u1c)
    instantiate j by ic in (theorem, *impliesHyp) / c-op(u1c)
    resume by ∧
      <> ∧ subgoal
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] ∧ subgoal
      <> ∧ subgoal
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] ∧ subgoal
    [] case ¬(ic = u1c)
  [] ∧ subgoal
  <> ∧ subgoal
  resume by case i = u1c
    <> case ic = u1c
    [] case ic = u1c
    <> case ¬(ic = u1c)
    instantiate i by ic in *impliesHyp
    [] case ¬(ic = u1c)
  [] ∧ subgoal
[] specialization subgoal
```

```
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal

% CASE 6: exit[u1c]
resume by ⇒
  <> ⇒ subgoal
  instantiate i by u1c in *impliesHyp
  set proof-methods normalization
  assert (ac = exit[u1c]); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c  }
    <> specialization subgoal

    instantiate c by task(ac) in $firstHyp

    prove
          (s'c.basic.pc[i] ∉ { crit, reset } ⇔ sc.basic.pc[i] ∉ { crit, reset })
       ∧  (s'c.basic.pc[i] ∈ { crit, reset } ⇔ sc.basic.pc[i] ∈ { crit, reset })
       ∧  (s'c.basic.pc[i] = set ⇔ sc.basic.pc[i] = set)
      ..
      resume by case i = u1c
        <> case ic = u1c
        [] case ic = u1c
        <> case ¬(ic = u1c)
        instantiate j by ic in *impliesHyp
        [] case ¬(ic = u1c)
      [] conjecture

    prove ∃ i:UID (w(s'c, i) ≤ u'c.bounds[task(seize)].last)
      resume by case ¬enabled(u'c.basic, task(seize))
        <> case ¬enabled(u'c.basic, task(seize))
        [] case ¬enabled(u'c.basic, task(seize))
        <> case ¬(¬enabled(u'c.basic, task(seize)))
        prove ¬enabled(uc.basic, seize) by contradiction
            <> contradiction subgoal
            [] contradiction subgoal
          [] conjecture
        declare operator ic:→UID
        fix i as ic in *caseHyp / c-op(try:→Region)
        instantiate i by u1c in *hyp
        prove ic ≠ u1c by contradiction
            <> contradiction subgoal
            [] contradiction subgoal
          [] conjecture
        resume by specializing i:UID to ic
          <> specialization subgoal
          prove inv(s'c)
            instantiate
              s by sc, s' by s'c, a: Actions[TF] by addTime(exit[u1c]) in Invariants
              ..
            [] conjecture
          resume by case s'c.basic.pc[ic] = check, s'c.basic.pc[ic] = test
            <> case justification
            instantiate j by ic in *impliesHyp, theorem
            instantiate i by ic in *impliesHyp          % StrongMutex
            [] case justification
            <> case s'c.basic.pc[ic] = check
            instantiate s by s'c, i:UID by ic in $wdef
            instantiate c by task(check[ic]) in theorem
```

```
            instantiate t:Time by a in Time
            [] case s'c.basic.pc[ic] = check
            <> case s'c.basic.pc[ic] = test
            instantiate s by s'c, i by ic in $wdef
            instantiate c by task(test[ic]) in theorem
            prove (s'c.bounds[task(test[ic])].last + a) ≤ (sc.now +a+a)
              instantiate t by a in Time
              [] conjecture
            prove (s'c.bounds[task(test[ic])].last + a) ≤ (sc.now +a+a+c)
              instantiate t by sc.now + a + a, t1 by c in Time
              [] conjecture
            [] case s'c.basic.pc[ic] = test
          [] specialization subgoal
        [] case ¬(¬enabled(u'c.basic, task(seize)))
      [] conjecture

    resume by case i = u1c
      <> case ic = u1c
      resume by case ¬enabled(u'c.basic, task(seize))
        <> case ¬enabled(u'c.basic, task(seize))
        [] case ¬enabled(u'c.basic, task(seize))
        <> case ¬(¬enabled(u'c.basic, task(seize)))
        instantiate c by task(reset[u1c]) in *impliesHyp
        prove ¬enabled(uc.basic, seize) by contradiction
            <> contradiction subgoal
            [] contradiction subgoal
          [] conjecture
        instantiate t by sc.now + a+a in Time
        [] case ¬(¬enabled(u'c.basic, task(seize)))
      [] case ic = u1c
      <> case ¬(ic = u1c)
      instantiate j by ic in (*impliesHyp, theorem) / c-op(u1c)
      instantiate i by ic in *impliesHyp
      resume by case ¬enabled(u'c.basic, task(rem[ic]))
        <> case ¬enabled(u'c.basic, task(rem[ic]))
        [] case ¬enabled(u'c.basic, task(rem[ic]))
        <> case ¬(¬enabled(u'c.basic, task(rem[ic])))
        [] case ¬(¬enabled(u'c.basic, task(rem[ic])))
      [] case ¬(ic = u1c)
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal

% CASE 7: reset[u1]
resume by ⇒
  <> ⇒ subgoal
  set proof-methods normalization
  resume by specializing alpha to {uc}
    <> specialization subgoal
    instantiate i by u1c in *impliesHyp / c-op(reset:→PC) / c-op(.owner)
    prove w(s'c, i) = w(sc,i)
      resume by case i = u1c
        <> case ic = u1c
        instantiate s by s'c, i by u1c in $wdef
        instantiate s by sc, i by u1c in $wdef
        [] case ic = u1c
        <> case ¬(ic = u1c)
        instantiate
```

```
          s by sc, s' by s'c, at by reset, i by u1c, j by ic in theorem / c-op(w)
          ..
        [] case ¬(ic = u1c)
      [] conjecture

    resume by case i = u1c
      <> case ic = u1c
      resume by ∧
        <> ∧ subgoal
        instantiate i by u1c in *impliesHyp
        instantiate
          t by a, t1 by s'c.now, t2 by sc.bounds[task(reset[u1c])].last in Time
          ..
        [] ∧ subgoal
        <> ∧ subgoal
        resume by specializing i:UID to u1c
          <> specialization subgoal
          [] specialization subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        resume by specializing i:UID to u1c
          <> specialization subgoal
          [] specialization subgoal
        [] ∧ subgoal
      [] case ic = u1c
      <> case ¬(ic = u1c)
      instantiate j by ic in *impliesHyp / c-op(u1c)
      resume by ∧
        <> ∧ subgoal
        resume by ⇒
          <> ⇒ subgoal
          instantiate i by ic in *impliesHyp
          [] ⇒ subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        instantiate i by ic in *impliesHyp
        [] ∧ subgoal
        <> ∧ subgoal
        instantiate i by ic in *impliesHyp
        [] ∧ subgoal
        <> ∧ subgoal
        instantiate i by ic in *impliesHyp
        [] ∧ subgoal
        <> ∧ subgoal
        resume by specializing i:UID to u1c
          <> specialization subgoal
          [] specialization subgoal
        [] ∧ subgoal
        <> ∧ subgoal
        resume by specializing i:UID to u1c
          <> specialization subgoal
          [] specialization subgoal
        [] ∧ subgoal
      [] case ¬(ic = u1c)
    [] specialization subgoal
  [] ⇒ subgoal
[] basis subgoal
<> basis subgoal
```

```
% CASE 8: rem[u1]
resume by ⇒
  <> ⇒ subgoal
  set proof-methods normalization
  assert (ac = rem[u1c]); u'c = STEP(uc, addTime(ac))
  resume by specializing alpha to ({uc}) { addTime(ac), u'c }
    <> specialization subgoal
    instantiate c by task(ac) in $firstHyp

    prove
       (s'c.basic.pc[i] = crit ⇔ sc.basic.pc[i] = crit)
        ∧ (s'c.basic.pc[i] = reset ⇔ sc.basic.pc[i] = reset)
        ∧ (s'c.basic.pc[i] = set ⇔ sc.basic.pc[i] = set)
      ..
      resume by case i = u1c
        <> case ic = u1c
        [] case ic = u1c
        <> case ¬(ic = u1c)
        instantiate j by ic in *impliesHyp / c-op(u1c)
        [] case ¬(ic = u1c)
      [] conjecture

    prove ∃ i:UID (w(s'c, i) ≤ u'c.bounds[task(seize)].last)
      declare operator ic:→UID
      fix i as ic in *hyp / c-op(w)
      resume by specializing i:UID to ic
        <> specialization subgoal
        resume by case ¬enabled(u'c.basic, seize)
          <> case ¬enabled(u'c.basic, seize)
          [] case ¬enabled(u'c.basic, seize)
          <> case ¬(¬enabled(u'c.basic, seize))
          prove enabled(uc.basic, seize)
            declare operator ic1:→UID
            fix i as ic1 in *caseHyp / c-op(try:→Region)
            prove ic1 ≠ u1c by contradiction
                <> contradiction subgoal
                [] contradiction subgoal
              [] conjecture
            resume by specializing i:UID to ic1
              <> specialization subgoal
              instantiate j by ic1 in theorem
              resume by case j = u1c
                <> case jc = u1c
                [] case jc = u1c
                <> case ¬(jc = u1c)
                instantiate j by jc in theorem, *hyp
                [] case ¬(jc = u1c)
              [] specialization subgoal
            [] conjecture
          resume by case ic = u1c
            <> case ic = u1c
            instantiate s by sc, i:UID by u1c in $wdef
            [] case ic = u1c
            <> case ¬(ic = u1c)
            instantiate
              s by sc, s' by s'c, at: ActionTypes[F] by rem, i by u1c, j by ic
              in theorem / c-op(w)
              ..
            [] case ¬(ic = u1c)
```

```
      [] case ¬(¬enabled(u'c.basic, seize))
    [] specialization subgoal
  [] conjecture

resume by case i = u1c
  <> case ic = u1c
  [] case ic = u1c
  <> case ¬(ic = u1c)
  instantiate j by ic in (theorem, *impliesHyp) / c-op(u1c)
  resume by ∧
    <> ∧ subgoal
    resume by case ¬enabled(u'c.basic, task(crit[ic]))
      <> case ¬enabled(u'c.basic, task(crit[ic]))
      [] case ¬enabled(u'c.basic, task(crit[ic]))
      <> case ¬(¬enabled(u'c.basic, task(crit[ic])))
      resume by ⇒
        <> ⇒ subgoal
        declare operator dummyi: → UID
        fix i as dummyi in *caseHyp
        [] ⇒ subgoal
      [] case ¬(¬enabled(u'c.basic, task(crit[ic])))
    [] ∧ subgoal
    <> ∧ subgoal
    resume by case ¬enabled(u'c.basic, task(rem[ic]))
      <> case ¬enabled(u'c.basic, task(rem[ic]))
      [] case ¬enabled(u'c.basic, task(rem[ic]))
      <> case ¬(¬enabled(u'c.basic, task(rem[ic])))
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] case ¬(¬enabled(u'c.basic, task(rem[ic])))
    [] ∧ subgoal
    <> ∧ subgoal
    resume by case ¬enabled(u'c.basic, task(crit[ic]))
      <> case ¬enabled(u'c.basic, task(crit[ic]))
      [] case ¬enabled(u'c.basic, task(crit[ic]))
      <> case ¬(¬enabled(u'c.basic, task(crit[ic])))
      resume by ⇒
        <> ⇒ subgoal
        declare operator dummyi: → UID
        fix i as dummyi in *caseHyp
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] case ¬(¬enabled(u'c.basic, task(crit[ic])))
    [] ∧ subgoal
    <> ∧ subgoal
    resume by case ¬enabled(u'c.basic, task(rem[ic]))
      <> case ¬enabled(u'c.basic, task(rem[ic]))
      [] case ¬enabled(u'c.basic, task(rem[ic]))
      <> case ¬(¬enabled(u'c.basic, task(rem[ic])))
      resume by ⇒
        <> ⇒ subgoal
        instantiate i by ic in *impliesHyp
        [] ⇒ subgoal
      [] case ¬(¬enabled(u'c.basic, task(rem[ic])))
    [] ∧ subgoal
    <> ∧ subgoal
    resume by case ¬enabled(u'c.basic, task(seize))
```

```
                  <> case ¬enabled(u'c.basic, task(seize))
                  [] case ¬enabled(u'c.basic, task(seize))
                  <> case ¬(¬enabled(u'c.basic, task(seize)))
                  resume by ⇒
                    <> ⇒ subgoal
                    instantiate c by task(reset[ic]) in *impliesHyp
                    instantiate i by ic in *impliesHyp
                    resume by case enabled(uc.basic, task(seize))
                      <> case enabled(uc.basic, task(seize))
                      [] case enabled(uc.basic, task(seize))
                      <> case ¬enabled(uc.basic, task(seize))
                      prove (sc.now + a + a + a) ≤ (sc.now + a + a + c)
                        instantiate t by sc.now + a+a in Time
                        [] conjecture
                      instantiate t by a+a in Time
                      [] case ¬enabled(uc.basic, task(seize))
                    [] ⇒ subgoal
                  [] case ¬(¬enabled(u'c.basic, task(seize)))
                [] ∧ subgoal
                <> ∧ subgoal
                resume by case ¬enabled(u'c.basic, task(stabilize))
                  <> case ¬enabled(u'c.basic, task(stabilize))
                  [] case ¬enabled(u'c.basic, task(stabilize))
                  <> case ¬(¬enabled(u'c.basic, task(stabilize)))
                  resume by ⇒
                    <> ⇒ subgoal
                    instantiate i by ic in *impliesHyp
                    [] ⇒ subgoal
                  [] case ¬(¬enabled(u'c.basic, task(stabilize)))
                [] ∧ subgoal
              [] case ¬(ic = u1c)
            [] specialization subgoal
          [] ⇒ subgoal
        [] basis subgoal
      [] basis subgoal
    [] basis subgoal
  [] conjecture
qed

set log f2i
statistics
quit
```

Bibliography

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, PA, June 1990. IEEE Computer Science Press.

[AD90] Rajeev Alur and David Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *Automata, Languages and Programming: Proceedings of the ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, Warwick University, England, July 1990. To appear in *Theoretical Computer Science*.

[AH89] Rajeev Alur and Thomas Henzinger. A really temporal logic. In *Proceedings of Thirtieth Annual ACM Symposium on Foundations of Computer Science*, pages 164–169, Research Triangle Park, NC, October/November 1989. IEEE Computer Science Press.

[AH90] Rajeev Alur and Thomas Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 390–401, Philadelphia, PA, June 1990. IEEE Computer Science Press.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AL92] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. Research report, DEC Systems Research Center, December 1992.

[AS85] Bowen Alpern and Fred Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[BH81] Arthur Bernstein and Paul Harter. Proving real-time properties of programs with temporal logic. In *Proceedings of the Eighth Annual ACM Symposium for Operating System Principles*, pages 1–11, Pacific Grove, CA, December 1981. ACM Press.

[BL93] James Burns and Nancy Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[CAZ92] Baio Chen, Gopal Agrawal, and Wei Zhao. Optimal synchronous capacity allocation for hard real-time communications with the timed token protocol. In *Proceedings of the Thirteenth Real-Time Systems Symposium*, pages 198–207, Phoenix, AZ, December 1992. IEEE Computer Society Press.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[CG87] E.M. Clarke and O. Grümberg. Research on automatic verification of finite-state concurrent systems. In Joseph Traub, Barbara Grosz, Butler Lampson, and Nils Nilsson, editors, *Annual Reviews of Computer Science*, volume 2, pages 269–290. Annual Reviews Inc., 1987.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, May 1979.

[CR83] J. Coolahan and N. Roussopoulus. Timing requirements for time-driven systems using augmented Petri nets. *IEEE Transactions on Software Engineering*, SE-9(5):603–616, September 1983.

[Dij65] Edsger Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 11(5):341–346, September 1965.

[DS89] Jim Davies and Steve Schneider. An introduction to timed CSP. Technical Report PRG-75, Oxford University Computing Laboratory, 1989.

[EGL92] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G.v.Bochmann and D.K.Probst, editors, *Computer Aided Verification: Fourth International Workshop, CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 44–55, Montreal, Canada, June/July 1992. Springer-Verlag.

[Fis85] Michael Fischer. Re: Where are you? E-mail to Leslie Lamport. ARPAnet message number 8506252257.AA07636@YALE.BULLDOG.YALE.ARPA, June 25, 1985. 18:56:29EDT.

[FKL95] Alan Fekete, Frans Kaashoek, and Nancy Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. In *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*, 1995. To appear.

[Flo67] R. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[FvR95] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in Horus. Technical Report CORNELLCS:TR95-1491, Cornell University Department of Computer Science, Ithaca, NY 14853, March 1995.

[Gaw92] Rainer Gawlick. Bounded concurrent time-stamping made simple. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, Cambridge, MA 02139, June 1992.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, DEC Systems Research Center, December 1991.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GHS83] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[Haa81] Volkmar Haase. Real-time behavior of programs. *IEEE Transactions on Software Engineering*, SE-7(5):494–501, September 1981.

[HLP90] Eyal Harel, Orna Lichtenstein, and Amir Pnueli. Explicit clock temporal logic. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 402–413, Philadelphia, PA, June 1990. IEEE Computer Science Press.

[HMP94] Thomas Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112(2):273–337, August 1994.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993.

[Jef92] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, AZ, December 1992. IEEE Computer Society Press.

[KMP93] Yonit Kesten, Zohar Manna, and Amir Pnueli. Temporal verification and simulation and refinement. In *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 273–346, REX School/Symposium, Noordwijkerhout, the Netherlands, June 1993. Springer-Verlag.

[KR93] Hermann Kopetz and Johannes Reisinge. The non-blocking write protocol nbw: A solution to a real-time synchronization problem. In *Proceedings of the Fourteenth Real-Time Systems Symposium*, pages 131–137, Raleigh-Durham, NC, December 1993. IEEE Computer Society Press.

[KVdR83] Ron Koymans, J. Vytopil, and Willem-Paul de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the Second Annual ACM*

Symposium on the Principles of Distributed Computing, pages 187–197. ACM Press, August 1983.

[LA92] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6(2):121–139, 1992.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[Lam91] Leslie Lamport. The temporal logic of actions. Research Report 79, DEC Systems Research Center, December 1991.

[Lam93a] Leslie Lamport. Verification and specification of concurrent programs. In *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 347–374, REX School/Symposium, Noordwijkerhout, the Netherlands, June 1993. Springer-Verlag.

[Lam93b] Butler Lampson. Principles for computer system design, 1993. Turing Award Lecture.

[LeL77] Gérard LeLann. Distributed systems, towards a formal approach. In *Information Processing 77: Proceedings of the Seventh IFIP World Congress*, pages 155–160, Toronto, Ontario, 1977.

[LL90] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In Jan van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 18, pages 1157–1199. Elsevier and MIT Press, 1990.

[LS84] Simon Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, April 1984.

[LS92] Nancy Lynch and Nir Shavit. Timing-based mutual exclusion. In *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, pages 2–11, Phoenix, Arizona, December 1992. IEEE Computer Society Press.

[LS93] Nancy Lynch and Boaz Patt-Shamir. Distributed algorithms. Lecture Notes for 6.852. Technical Report MIT/LCS/RSS-20, Massachusetts Institute of Technology, Cambridge, MA 02139, January 1993.

[LSGL94] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In *Formal Description Techniques VII: Proceedings of FORTE'94*, pages 259–273, Berne, Switzerland, October 1994. IFIP WG6.1, Chapman and Hall.

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, Cambridge, MA 02139, April 1987. Also, MIT/LCS/TR-387.

[LT89] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[LV91] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446, REX Workshop, Mook, the Netherlands, June 1991. Springer-Verlag. Also, MIT/LCS/TM-458.

[LVara] Nancy Lynch and Frits Vaandrager. Forward and backward simulations—Part I: Untimed systems. *Information and Computation*, to appear. Also, MIT/LCS/TM-486.b.

[LVarb] Nancy Lynch and Frits Vaandrager. Forward and backward simulations—Part II: Timing-based systems, to appear. Submitted for publication. Also, MIT/LCS/TM-487.b.

[Lyn89a] Nancy Lynch. A hundred impossiblility proofs for distributed computing. In *Proceedings of the Eighth Annual ACM Symposium on the Principles of Distributed Computing*, pages 1–27, Edmonton, Alberta, August 1989. ACM Press. Also, MIT/LCS/TM-394.

[Lyn89b] Nancy Lynch. Multivalued possibilities mappings. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Sys-*

tems: Models, Formalism, Correctness, volume 430 of *Lecture Notes in Computer Science*, pages 519–543, REX Workshop, Mook, The Netherlands, June 1989. Springer-Verlag. Also, MIT/LCS/TM-422.

[Lyn93] Nancy Lynch. Simulation techniques for proving properties of real-time systems. In *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 375–424, REX School/Symposium, Noordwijkerhout, the Netherlands, June 1993. Springer-Verlag. Also, MIT/LCS/TM-494.

[MBRS94] Dalia Malki, Ken Birman, Aleta Ricciardi, and André Schiper. Uniform actions in asynchronous distributed systems. In *Proceedings of Thirteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1994. Also, CORNELLCS:TR94-1447.

[Mil89] Robin Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall International, 1989.

[MMT91] Michael Merritt, Francemary Modugno, and Mark Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, editors, *CONCUR'91: Second International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423, Amsterdam, The Netherlands, August 1991. Springer-Verlag.

[MSST93] Sarit Mukherjee, Debanjan Saha, Manas C. Saksena, and Satish K. Tripathi. A bandwidth allocation scheme for time constrained message transmission on a slotted ring lan. In *Proceedings of the Fourteenth Real-Time Systems Symposium*, pages 44–54, Raleigh-Durham, NC, December 1993. IEEE Computer Society Press.

[Nip89] Tobias Nipkow. Formal verification of data type refinement. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 561–589, REX Workshop, Mook, The Netherlands, June 1989. Springer-Verlag.

[NS91] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In J. W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 526–548, REX Workshop, Mook, the Netherlands, June 1991. Springer-Verlag.

[Ost89] J. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press (John Wiley & Sons), 1989.

[PF77] Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 91–97, Boulder, CO, May 1977. ACM Press.

[Pog95] Anna Pogosyants. Incorporating specialized theories into a general purpose theorem prover. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, Cambridge, MA 02139, February 1995.

[PS95] Anna Pogosyants and Roberto Segala. Formal verification of timed properties for randomized distributed algorithms. In *Proceedings of Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, 1995. To appear.

[Sch93] Fred Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems*, chapter 2. Addison-Wesley, second edition, 1993.

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, MIT Dept. of Electrical Engineering and Computer Science, Cambridge, MA 02139, 1995.

[SGG+93] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer Aided Verification: Fifth International Conference, CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 305–319, Elounda, Greece, June/July 1993. Springer-Verlag.

[Sha89] A. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

[Sha92] A. Udaya Shankar. A simple assertional proof system for real-time systems. In *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium.*, pages 167–176, Phoenix, AZ, December 1992. IEEE Computer Society Press.

[Sha93] N. Shankar. Verification of real-time systems using PVS. In *Computer Aided Verification: Fifth International Conference, CAV'93*, number 697 in Lecture Notes in Computer Science, pages 280–291, Elounda, Greece, June/July 1993. Springer-Verlag.

[SL87] A. Udaya Shankar and Simon Lam. Time-dependent distributed systems: Proving safety, liveness and real-time properties. *Distributed Computing*, 2(2):61–79, April 1987.

[SLL93a] Jørgen Søgaard-Andersen, Nancy Lynch, and Butler Lampson. Correctness of at-most-once message delivery protocols. In *Formal Description Techniques VI: Proceedings of FORTE'93*, pages 387–402, Boston, MA, October 1993.

[SLL93b] Jørgen Søgaard-Andersen, Nancy Lynch, and Butler Lampson. Correctness of communication protocols: A case study. Technical Report MIT/LCS/TR-589, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA 02139, November 1993.

[Söy94] Ekrem Söylemez. Automatic verification of the timing properties of MMT automata. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, Cambridge, MA 02139, February 1994.

[Tel88] Gerard Tel. Assertional verification of a timer based protocol. In Timo Lepistö and Arta Salomaa, editors, *Automata, Languages and Programming: Proceedings of ICALP'88*, volume 317 of *Lecture Notes in Computer Science*, pages 600–614, Tampere, Finland, July 1988. Springer-Verlag.

[vRHB94] Robbert van Renesse, Takako Hickey, and Ken Birman. Design and performance of Horus: A lightweight group communication system. Technical Report CORNELLCS:TR94-1442, Cornell University Department of Computer Science, Ithaca, NY 14853, August 1994.

[Wan91] Yi Wang. CCS + time = an interleaving model for real time systems. In J. Leach Albert, B. Monien, and M. Rodriguez Artalejo, editors, *Automata, Languages and Programming: Proceedings of ICALP'91*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228, Madrid, Spain, July 1991. Springer-Verlag.

[WLL88] Jennifer Lundelius Welch, Leslie Lamport, and Nancy Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proceedings of the Seventh Annual ACM Symposium on the Principles of Distributed Computing*, pages 28–43, Toronto, Ontario, August 1988. ACM Press. Also, see MIT/LCS/TM-361.

[WPD94] Yi Wang, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII: Proceedings of FORTE'94*, pages 243–258, Berne, Switzerland, October 1994. IFIP WG6.1, Chapman and Hall.

[Zho92] Hongyi Zhou. Performance effects of information sharing in a distributed multiprocessor real-time scheduler. In *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, pages 46–55, Phoenix, AZ, December 1992. IEEE Computer Society Press.