Proving Temporal Properties of Concurrent Programs: A Non-Temporal Approach

Bowen L. Alpern

Ph.D. Thesis
TR 86-732
February 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

PROVING
TEMPORAL PROPERTIES OF CONCURRENT PROGRAMS:
A NON-TEMPORAL APPROACH

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Bowen Lewis Alpern

June 1986

PROVING
TEMPORAL PROPERTIES OF CONCURRENT PROGRAMS:
A NON-TEMPORAL APPROACH

Bowen Lewis Alpern, Ph.D.
Cornell University 1986

This thesis develops a new method for proving properties of concurrent programs and gives formal definitions for safety and liveness. A property is specified by a property recognizer—a finite-state machine that accepts the sequences of program states in the property it specifies. A property recognizer can be constructed for any temporal logic formula.

To prove that a program satisfies a property specified by a deterministic property recognizer, one must show that any history of the program will be accepted by the recognizer. This is done by demonstrating that proof obligations derived from the recognizer are met. These obligations require the program prover to exhibit certain invariant assertions and variant functions and to prove the validity of certain predicates and Hoare triples. Thus, the same techniques used to prove

total correctness of a while loop can be used to prove temporal properties of concurrent programs. No temporal inference is required.

The invariant assertions required by the proof obligations establish a correspondence between the states of the program and those of the recognizer. Such correspondences can be denoted by property outlines, a generalization of proof outlines.

Some non-deterministic property recognizers have no deterministic equivalents. To prove that a program satisfies a non-deterministic property, a deterministic sub-property that the program satisfies must be found. This is shown possible, provided the program state space is finite.

Finally, safety properties are formalized as the closed sets of a topological space and liveness properties as its dense sets. Every property is shown to be the intersection of a safety property and a liveness property. A technique for separating a property specified by a deterministic property recognizer into its safety and liveness aspects is also presented.

Biographical Sketch

Born November 17th 1952, Bowen Lewis Alpern enjoyed a classically unhappy childhood. He grew up in Ann Arbor, entering the University of Michigan in the waning days of the International Cultural Revolution. He fell in love and studied mathematics and computer science. They spent two years in graduate school at Santa Cruz. Upon the dissolution of the relationship, he returned to Ann Arbor. Commuting to a programming job in Wayne, he came to marxism discussing Isaac Deutscher's biography of Trotsky in the kitchen of Walden coop. He retired in the spring of 1979. He performed the rites of matrimony on Debbie Gale and Steve Gurevitz in another kitchen that summer. In the fall, he matriculated at Cornell where he received a master's degree and developed a pronounced distaste for referring to himself in the third person. After completing his dissertation, he will join IBM's Exploratory Computer Science group. He is a member of the ACM and the DSA and has participated in a number of programs offered by Werner Erhart and Associates. He met Robin (a.k.a. Elise, a.k.a. Kelly) Mallison while sunbathing at "the rock" of the lower reservoir on Six-Mile Creek in the summer of 1984. They married the following August. They have two cats.

To those who live out of their commitments rather than their circumstances.

Acknowledgements

Let me begin by acknowledging my advisor, colleague, and friend, Fred B. Schneider. His guidance has been ruthless and compassionate. His commitment to me and to my becoming the best possible scientist has been absolute. His dedication remains an inspiration. He has been the perfect advisor.

Other Cornell professors whose challenge and encouragement has made a difference in my life include Robert Constable, Juris Hartmanis, John Hopcroft, Steven Kaplan, David Gries, and Ken Birmin. The support staff of the Computer Science Department deserves thanks for creating ongoingly an environment in which research is possible. Special thanks to Diane Duke for making sure that I always had food on the table and a roof over my head. The graduate student community has been an unending source of stimulation and friendship. Many people come to mind. I particularly wish to acknowledge Susan Horwitz, Tom Reps, Dean Jacobs, Jan Prins, Mark Krentel, and Mimi Bussan.

And then there is my wife. Robin neither put me through graduate school nor typed my thesis. She has allowed me the privilege of participating in her life. There is no greater honor.

I happily acknowledge Werner Erhart and Associates. I cannot say that I would not have finished my thesis had I not participated in "the work." (Indeed, I might have finished earlier.) I can and do say that it has had a powerful impact on my life. I recommend it without qualification to everyone. Particular thanks go to Terry Miller and the Syracuse Seminar for keeping my honest and in touch with my commitments.

Who have I left out? Family, friends, teachers, authors, heros, casual acquaintances. . . The list goes on without end; the web of indebtedness is seamless. This has been a hard section to write. It is clear that I haven't done justice to the contribution people are in my life. In a trite old phrase "words fail me."

Table of Contents

List of Figures

CHAPTER 1

Introduction

Experience has shown that while it may be possible to understand a sequential program by considering some subset of its executions, this is impossible for concurrent programs. Consequently, over the past 15 years, there has been increasing interest in ways to deduce properties of program behavior from the program text itself. The program text obviously contains all the information needed to decide what executions are possible. Moreover, while the number of possible executions is likely to be intractably large, only a single program text need be analyzed.

An execution of a program can be viewed as a potentially infinite sequence of states called a *history*. In a history, the first state is an initial state of the program and each following state results from executing a single atomic action in the preceding state. In a concurrent or distributed program, a history is the sequence of states that results from interleaving the atomic actions of the processes as they execute.

A *property* defines a set of sequences of states; we write

$$\sigma \models P$$

to denote that σ is a sequence in the set defined by P. A program *satisfies* a property if each of its histories is in the set defined by the property. A property can be specified as a predicate on sequences. This allows the essence of the property to be made explicit.

Some examples of properties frequently arising in practice follow.

- *Partial Correctness* includes all sequences of program states such that, if the first state in the sequence satisfies some given precondition *Pre* and the sequence is finite, then in the final state the program counter denotes the end of the program and some given postcondition *Post* is satisfied.

- *Abortion Freedom* includes all sequences of program states such that, if the first state in the sequence satisfies some given precondition *Pre* and the sequence is finite, then the final state is not one in which the program aborts.

- *Termination* includes all sequences of program states such that, if the first state in the sequence satisfies some given precondition *Pre*, then the sequence is finite.

- *Normal Termination*, which is the conjunction of an Abortion Freedom property and a Termination property—both with the same precondition *Pre*—includes all sequences of program states such that, if the first state satisfies *Pre*, then the sequence is finite and the final state is not one in which the program aborts.

- *Total Correctness*, which is the conjunction of a Partial Correctness property and a Normal Termination property—both with the same precondition *Pre*—includes all sequences such that, if the first state satisfies *Pre*, then the sequence is finite, the final state satisfies some given postcondition *Post*, and the final value of the program counter denotes the end of the program.

- *Mutual Exclusion* includes all sequences in which there is no state where the program counters for two or more processes denote control points inside critical sections.

- *Deadlock Freedom* includes all sequences in which there is no state where both (i) some process has no enabled atomic actions and (ii) no subsequent execution by any other process can alter that.

- *Guaranteed Service* includes all sequences in which a process that requests service eventually gets serviced.

- *First-come First-serve* includes all sequences in which processes that request service in one order are not serviced in another order.

- *Starvation Freedom* includes all sequences in which a process with an atomic action that is enabled frequently enough will make progress eventually.

Formulas of temporal logic can be interpreted as predicates on sequences of states, and various formulations of temporal logic have been used for specifying properties of interest to designers of concurrent programs [Lamport 83a] [Lamport 83b] [Manna & Pnueli 81a] [Wolper 83]. While there is not general agreement on the details of such a specification language, there is agreement that temporal logic provides a good basis for such a language and it, or something close to it, is sufficiently expressive.

Temporal logic has also been used in proving properties of concurrent programs [Pnueli 77] [Manna & Pnueli 81b] [Manna & Pnueli 84] [Owicki & Lamport 82]. Here, a program is regarded as defining a collection of temporal logic axioms. The programmer proves a property of interest by using these axioms along with program-independent axioms and inference rules of temporal logic [Manna & Pnueli 83]. Various packagings of the approach avoid the necessity of making temporal inferences by restricting the class of properties that can be proved. Examples include Hoare's logic for Partial Correctness of sequential programs [Hoare 69] and its extension to concurrent programs [Owicki & Gries 76], GHL (Generalized Hoare Logic) for proving safety properties of concurrent programs [Lamport 80] [Lamport & Schneider 84], and proof lattices for proving liveness properties [Owicki & Lamport 82].

This thesis introduces a new approach for proving properties of (concurrent) programs. The approach can handle a broad class of properties, including any property that can be expressed in temporal logic. Using our approach, to prove that a program satisfies some given property, *invariance obligations* and *variance obligations* are constructed. Invariance obligations are discharged by finding certain *invariant assertions* and showing that they are preserved by execution; variance obligations are discharged by finding *variant functions* and showing that they decrease following certain events. Hoare's partial correctness logic is used to show that the invariant assertions are preserved by execution and that the variant functions are decreased by execution.

We proceed as follows. Chapter 2 reviews the concepts we will need to treat concurrent programs. Chapter 3 defines *property recognizers*, an automata-theoretic method of specifying properties. A procedure for translating a temporal logic specification into an equivalent property recognizer specification is also given. Chapter 4 shows how to extract proof obligations from a deterministic property recognizer. To prove that a program satisfies a deterministic property, one must first find the assertions and the functions entailed by the proof obligations then verify that the assertions are invariant and that the variant functions decrease. Chapter 5 presents *property outlines*, a generalization of proof outlines used to establish invariance assertions. Chapter 6 uses our method to prove three properties of a Mutual Exclusion protocol. Chapter 7 extends our approach to handle properties specified by non-deterministic property recognizers. Chapter 8 concerns the meta-properties, *safety* and *liveness*. Chapter 9 discusses related work. And, Chapter 10 is a conclusion.

CHAPTER 2

Concurrent Programs

A concurrent program π consists of a collection of atomic actions $\mathcal{A}_\pi$ and a set of initial states $Init_\pi$. Knowing the atomic actions of a concurrent program is necessary in order to understand its execution, since they define the grain of interleaving of processes. The atomic actions in a process ϕ define its *control points* $\mathcal{L}_\phi$—the set of values that can be stored in its program counter, pc_ϕ. We can denote the control points of a program by naming them within braces in the program text; this results in a *control-point annotation*. For example, program θ of Figure 2.1 consists of two sequential processes, μ and η, each with a single atomic action and two control points. The atomic action in process μ is called α_1 and the control points in μ are labeled 1 and 2.

The program counter pc_ϕ for process ϕ is a variable that can be used in defining sets of program states. For example,

$$Init_\theta \;=\; pc_\mu=1 \wedge pc_\eta=3$$

is the set of initial states off θ. The program counter of a sequential process differs from other program variables in that usually only a single process may update it and direct assignments to it are not permitted. Each atomic action, however, changes the value of the program counter. For example, atomic action α_1 in θ changes pc_μ (from 1 to 2) as well as incrementing x. The assignment to

```
θ: cobegin
     μ: {1:}
         α1: x := x+1
         {2:}
   //
     η: {3:}
         α2: x := x+1
         {4:}
   coend
```

Figure 2.1. Simple Program

pc_μ by α_1, though not explicit, can be deduced from the position of α_1 in the program text.

We define atomic actions to be executed indivisibly and to completion, so an atomic action cannot be started unless it will terminate. We therefore assume an atomic action is delayed until the program state is one that will permit its termination. Using angle brackets to denote an atomic action, α_1 of θ is

$$\langle \textbf{if } pc_\mu=1 \rightarrow pc_\mu, x := 2, x+1 \textbf{ fi} \rangle . \tag{2.1}$$

In the Guarded Command notation [Dijkstra 76] used here, the semantics of **if** require that

$$\textbf{if } B0 \rightarrow S0 \ [] \ \ldots \ [] \ Bn \rightarrow Sn \textbf{ fi}$$

abort if executed in a state where none of the *guards* $B0, \ldots, Bn$ holds. Thus, (2.1) is delayed until the program counter for process μ is 1, and then (without interruption) atomically updates the program counter and increments x. An atomic action might be delayed for reasons other than the program counter value. A **P** operation in process ψ on a general semaphore *sem*,

$$\ldots \{a:\} \ \mathbf{P}(sem) \ \{b:\} \ldots$$

defines an atomic action β:

$$\langle \mathbf{if}\ pc_\psi = a \wedge sem > 0 \ \rightarrow \ pc_\psi,\ sem := b,\ sem-1\ \mathbf{fi} \rangle \qquad (2.2)$$

An atomic action is ***enabled*** in any state where its execution would not be delayed. Let $Enabled(\alpha)$ be the set of states in which α is enabled. For (2.1),

$$Enabled(\alpha_1) \ \equiv \ pc_\mu = 1$$

and for (2.2),

$$Enabled(\beta) \ \equiv \ pc_\psi = a \ \wedge \ sem > 0.$$

We can use *Enabled* to characterize states in which a program π is ***blocked*** and can make no further progress because there are and will be no enabled atomic actions:

$$Blocked_\pi \ \equiv \ \bigwedge_{\alpha : \alpha \in \mathcal{A}_\pi} \neg Enabled(\alpha)$$

The effects of an atomic action α can be defined in terms of the relation between the program state before and after it is executed. This relation can be described by a ***triple*** $\{P\}\ \alpha\ \{Q\}$, which is valid if, whenever α is enabled in a state satisfying P, its execution terminates in a state satisfying Q. Since an atomic action does not terminate if it is executed in a state in which it is not enabled, $\{P\}\ \alpha\ \{Q\}$ expresses the Partial Correctness of α with respect to P and Q. In such a triple, P is called the *precondition* and Q the *postcondition*.

Programming logics to prove validity of a triple involving a sequential program π are well known [Hoare 69]. One is summarized in Figure 2.2. Thus, if atomic actions are described by sequential programs, then such a logic and the following inference rule can be used to infer their semantics.

Skip Axiom: $\{P\}$ **skip** $\{P\}$

Assignment Axiom: $\{P^{\bar{x}}_{\bar{e}}\}\ \bar{x} := \bar{e}\ \{P\}$

if Rule: $\dfrac{\{P \wedge B0\}\ S1\ \{Q\}, \quad \cdots \quad \{P \wedge Bn\}\ Sn\ \{Q\}}{\{P\}\ \textbf{if}\ B0 \rightarrow S0 [] \ \ldots\ [] \ Bn \rightarrow Sn\ \textbf{fi}\ \{Q\}}$

do Rule: $\dfrac{\{P \wedge B0\}\ S1\ \{P\}, \quad \cdots \quad \{P \wedge Bn\}\ Sn\ \{P\}}{\{P\}\ \textbf{do}\ B0 \rightarrow S0\ []\ \ldots\ []\ Bn \rightarrow Sn\ \textbf{od}\ \{P \wedge \neg B0 \wedge \ldots \wedge \neg Bn\}}$

Rule of Consequence: $\dfrac{P \Rightarrow P', \quad \{P'\}\ S\ \{Q'\}, \quad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$

Conjunction Rule: $\dfrac{\{P\}\ S\ \{Q\}, \quad \{P'\}\ S\ \{Q'\}}{\{P \wedge P'\}\ S\ \{Q \wedge Q'\}}$

Figure 2.2. Partial Correctness Logic

Atomic Action Rule: $\dfrac{\{P\}\ S\ \{Q\}}{\{P\}\ \langle S \rangle\ \{Q\}}$

For example, returning to θ of Figure 2.1, we can establish the validity of $\{x=0\}\ \alpha_1\ \{x=1\}$ as follows:

$\{x=0\}\ \ pc_\mu, x := 2, x+1\ \ \{x=1\}$ (Assignment Axiom)
$\{x=0 \wedge pc_\mu=1\}\ \ pc_\mu, x := 2, x+1\ \ \{x=1\}$ (Rule of Consequence)
$\{x=0\}$ **if** $pc_\mu=1 \rightarrow pc_\mu, x := 2, x+1$ **fi** $\{x=1\}$ (**if** Rule)
$\{x=0\}$ $\langle$**if** $pc_\mu=1 \rightarrow pc_\mu, x := 2, x+1$ **fi**$\rangle$ $\{x=1\}$ (Atomic Action Rule)
$\{x=0\}\ \ \alpha_1\ \ \{x=1\}$ (definition of α_1)

This type of reasoning, which we employ frequently in the sequel, is facilitated by the following derived rule of inference.

Await Rule: $\dfrac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \langle \textbf{if}\ B \rightarrow S\ \textbf{fi}\ \rangle\ \{Q\}}$

The Await Rule follows from the **if** Rule and the Atomic Action Rule.

CHAPTER 3

Specifying Properties

This chapter presents an automata-theoretic method of specifying properties in which properties are specified by *property recognizers*, automata similar to Buchi automata [Eilenberg 74]. Property recognizers are not being advocated as a specification language; they are merely a convenient starting point for our verification method. A mechanical procedure for converting a temporal logic formula into a property recognizer is also given. Thus, using property recognizers is not a restriction.

3.1. Property Recognizers

A property recognizer accepts those sequences of program states that are in the property it specifies. Properties can contain infinite sequences as well as finite ones, so a property recognizer must be able to accept both kinds of sequences. Recall that an ordinary finite-state automaton accepts a finite sequence if and only if it enters an accepting state after reading the final symbol [Hopcroft & Ullman 79]. A Buchi automaton is a finite-state automaton with an acceptance criterion that allows it to accept infinite sequences—it accepts an infinite sequence if and only if it enters an accepting state infinitely often while reading that sequence [Eilenberg 74]. A property recognizer is an automaton that behaves like an ordinary finite-state automaton on finite input sequences and like a Buchi automaton on infinite input sequences.

An example of a property recognizer, m_{exmpl}, is illustrated in Figure 3.1.

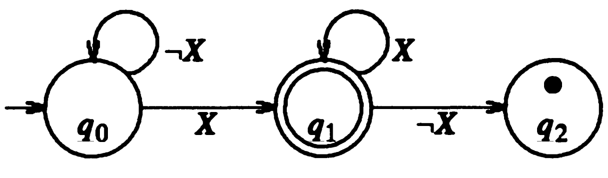


Figure 3.1. m_{exmpl}

It defines the set of sequences where each has a (possibly empty) prefix of states in which each satisfies predicate $\neg X$, immediately followed by either (1) an infinite sequence of states in which X holds throughout, or (2) a finite sequence of states in which X holds on all but the last state.

Property recognizer m_{exmpl} contains three *automaton states* labelled q_0, q_1, and q_2. *Start states* are denoted by arcs with no origin; *infinite-accepting states* by concentric circles; and *finite-accepting states* by bullets (●). Thus, automaton state q_0 is a start state, q_1 is an infinite-accepting state, and q_2 is a finite-accepting state. An infinite sequence is accepted by the property recognizer if and only if it causes the recognizer to be infinitely-often in some infinite-accepting state. A finite sequence is accepted by the property recognizer if and only if it causes the recognizer to halt (after reading the final symbol of its input) in some finite-accepting state. A state is *terminal* if and only if it is both infinite-accepting and finite-accepting and there is an arc from it to itself labelled *True*. There are no terminal states in m_{exmpl}.

Arcs between automaton states are labelled by program state predicates called *transition predicates*. These define transitions between automaton states based on the next symbol read from the input. For example, the arc labelled X from q_0 to q_1 in m_{exmpl} means that whenever the property recognizer is in automaton state q_0 and the next symbol is a program state satisfying X, then m_{exmpl} makes a transition to automaton state q_1. If an input symbol does not satisfy any transition predicate on an arc out of the current automaton state, then the input is rejected by the property recognizer; the transition is said to be *undefined* for that symbol. This is used in m_{exmpl} to ensure that every finite sequence ends with a single program state satisfying $\neg X$—no further transitions are possible from q_2 because there are no arcs emanating from it.

When more than one transition is possible from some automaton state for some input symbol or there is more than one start state, the property recognizer is *non-deterministic*; otherwise it is *deterministic*. Property recognizer m_{exmpl} is deterministic because disjoint transition predicates emanate from each automaton state.

Before formally defining property recognizers, some notation for sequences will be useful. For any sequence $\sigma = s_0 s_1 \ldots$,

$$\begin{aligned}
\sigma[i] &\equiv s_i \\
\sigma[..i] &\equiv s_0 s_1 \ldots s_i \\
\sigma[i..] &\equiv s_i s_{i+1} \ldots \\
|\sigma| &\equiv \text{the length of } \sigma \text{ } (\omega \text{ if } \sigma \text{ is infinite}).
\end{aligned}$$

Formally, a property recognizer m for a property of program π is a sextuple $\langle S, Q, Q_0, Q_\infty, Q_F, \delta \rangle$, where

S is the set of program states of π,
Q is the set of automaton states of m,
$Q_0 \subseteq Q$ is the set of start states of m,
$Q_\infty \subseteq Q$ is the set of infinite-accepting states of m,
$Q_F \subseteq Q$ is the set of finite-accepting states of m, and
$\delta \in (Q \times S) \rightarrow 2^Q$ is the *transition function* of m.

The transition predicates can be derived from the transition function δ. The transition predicate T_{ij} associated with the arc from state q_i to state q_j is the predicate that holds for all program states s such that $q_j \in \delta(q_i, s)$. Transition function δ can be extended to handle finite sequences of program states:

$$\delta^*(q, \sigma) = \begin{cases} \{q\} & \text{if } |\sigma|=0 \\ \{q' \mid q'' \in \delta(q, \sigma[0]) \wedge q' \in \delta^*(q'', \sigma[1..])\} & \text{if } 0<|\sigma|<\omega \end{cases}$$

A finite sequence σ is accepted by m if and only if $\delta^*(q_0, \sigma) \cap Q_F \neq \emptyset$. To characterize acceptance of infinite sequences, define a *run* of m on a sequence σ to be a sequence of automaton states ρ that m could be in while reading input σ. Thus, for ρ to be a run, $\rho[0] \in Q_0$, and $(\forall i: 0<i<|\sigma|: \rho[i] \in \delta(\rho[i-1], \sigma[i-1]))$. Let $\Gamma_m(\sigma)$ be the set of runs of m on σ. (If m is deterministic, this set has exactly one element.) Let $INF_m(\sigma)$ be the set of automaton states that appear infinitely often in any element of $\Gamma_m(\sigma)$. An infinite sequence σ is accepted by m if and only if $INF_m(\sigma) \cap Q_\infty \neq \emptyset$.

As is well known, any set of finite sequences that can be recognized by a non-deterministic finite state machine can be recognized by some deterministic finite state machine. Unfortunately, Buchi automata, hence property recognizers, do not enjoy this equivalence—there are sets of infinite sequences that can be recognized by non-deterministic property recognizers but by no deterministic one [Eilenberg 74]. This will necessitate developing two different techniques for proving properties: one for properties that can be specified by deterministic property recognizers and one for properties that require non-deterministic property recognizers.

Examples of Property Recognizers

A property recognizer m_{mutex} for the Mutual Exclusion property for two processes, ϕ and ψ, is given in Figure 3.2. The transition predicate Cs_ϕ (Cs_ψ) holds on any state in which process ϕ (ψ) is executing in its critical section.

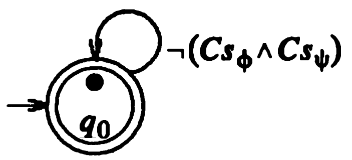


Figure 3.2. m_{mutex}

This recognizer accepts all sequences that do not contain a state satisfying both Cs_ϕ and Cs_ψ.

The property recognizer for Guaranteed Service, m_{gs} of Figure 3.3, specifies that if process ϕ ever makes a request then that request will get serviced.

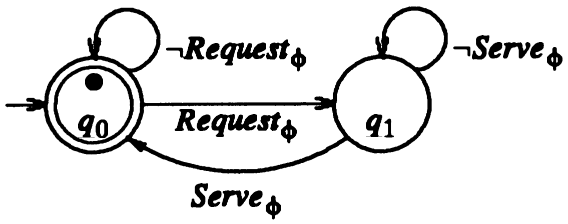


Figure 3.3. m_{gs}

As long as ϕ makes no request, the automaton remains in the accepting state q_0. When ϕ makes a request, m_{gs} enters the non-accepting state q_0 and remains there until the request is serviced.

The property recognizer for the First-come First-serve property, m_{fcfs} of Figure 3.4, guarantees that if process ϕ makes a request before process ψ, then the prior request will be serviced first.

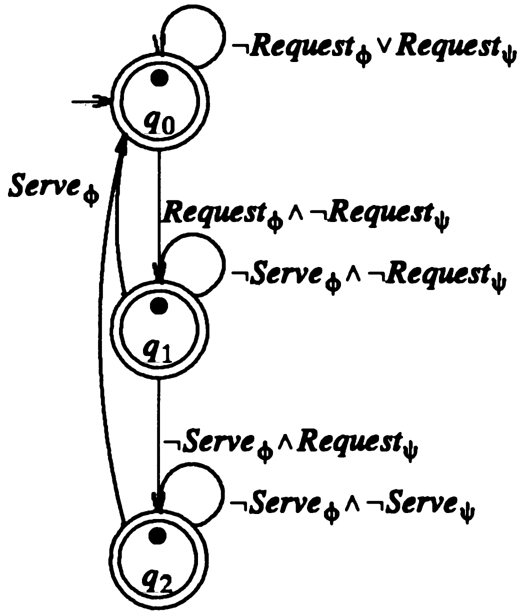


Figure 3.4. m_{fcfs}

Until ϕ makes a request when there is no prior pending request from ψ, m_{fcfs}

remains in q_0. Then it enters q_1 and remains there until ϕ's request is served or ψ makes a request. Thereupon, m_{fcfs} enters q_2, where it remains until one of the requests is serviced. If the wrong request is serviced then it attempts an undefined transition and the sequence is rejected. (A symmetric automaton would guarantee that process ψ is not discriminated against.)

A property recognizer m_{pc} for Partial Correctness is shown in Figure 3.5. In it, *Pre* is a transition predicate that holds for states satisfying the precondition. Transition predicate *Done* holds for states in which the program counter denotes the end of the program. And, transition predicate *Post* holds for states satisfying the postcondition.

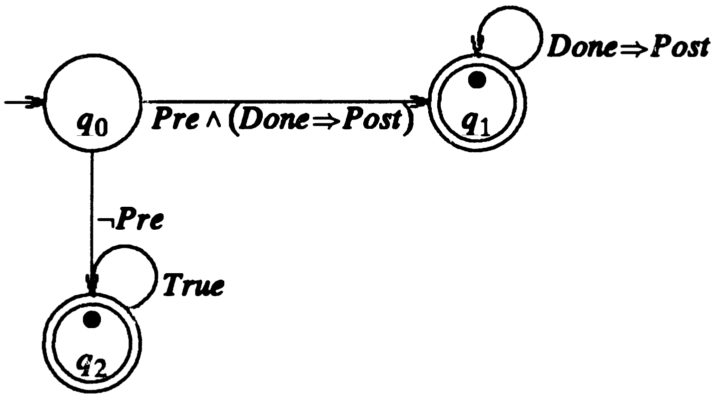


Figure 3.5. m_{pc}

If the first symbol of a sequence read by this automaton is a program state that does not satisfy *Pre*, then the automaton enters the terminal state q_2 and (eventually) accepts the sequence. It accepts other sequences unless they contain a state that satisfies *Done* but not *Post*. (If the sequence is a history then only its final state will satisfy *Done*.)

The property recognizer for Total Correctness is shown in Figure 3.6.

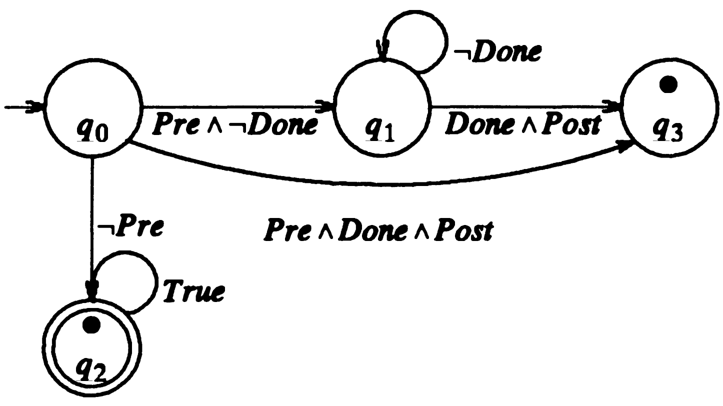


Figure 3.6. m_{tc}

Of the histories that begin by satisfying *Pre*, this automaton accepts only those that are finite and end in a state satisfying both *Done* and *Post*.

3.2. Translating Temporal Logic into Property Recognizers

To use our approach to prove that a program satisfies a property expressed in temporal logic, it will be necessary to build a property recognizer for the property. One method for translating from temporal logic to Buchi automata (from which property recognizers can easily be obtained) is based on the tableau for a temporal formula [Wolper 84]. Another method is presented below.

3.2.1. Temporal Logic

The temporal logic that follows is based on [Manna & Pnueli 81a]. An *atomic formula* of the logic is a predicate on program states. A *temporal formula* is built from these atomic formulas using the propositional connectives, $\wedge$, $\vee$, and

$\neg$, the unary temporal connectives, $\Diamond$ (read *eventually*) and $\bigcirc$ (*next*[1]), and the binary temporal connective $\mathcal{U}$ (*until*).

A sequence σ satisfies an atomic formula X if and only if X holds on $\sigma[0]$. The propositional connectives have their usual meanings. A sequence σ satisfies $\bigcirc A$ if and only if A holds on $\sigma[1..]$. A sequence σ satisfies $\Diamond A$ if and only if A holds on $\sigma[i..]$ for some $0 \le i < |\sigma|$. Finally, σ satisfies $A \mathcal{U} B$ if and only if B holds on $\sigma[i..]$ for some $0 \le i < |\sigma|$ and A holds on $\sigma[j..]$ for all $0 \le j < i$. The connectives $\Rightarrow$, $\Box$ (*henceforth*) and $\mathcal{N}$ (*unless*) are defined as follows:

$$
\begin{array}{lcl}
A \Rightarrow B & \equiv & \neg A \vee B \\
\Box A & \equiv & \neg \Diamond \neg A \\
A \mathcal{N} B & \equiv & \neg(\neg A \mathcal{U} B).
\end{array}
$$

It will be convenient to restrict our attention to temporal formulas in which only atomic formulas appear negated. Any temporal formula can be put in this form using DeMorgan's Laws and the following identities:

$$
\begin{array}{lcl}
A \Rightarrow B & = & \neg A \vee B, \\
\neg \Box A & = & \Diamond \neg A, \\
\neg \Diamond A & = & \Box \neg A, \\
\neg(A \mathcal{U} B) & = & \neg A \mathcal{N} B, \text{ and} \\
\neg(A \mathcal{N} B) & = & \neg A \mathcal{U} B.
\end{array}
$$

For the remainder of this chapter, we shall assume that the only negated subformulas of a temporal formula are atomic.

---

[1] We do not endorse use of the next operator in specifications of concurrent programs. (Arguments against such use are given in [Lamport & Owicki 81].) We include it only for expressive completeness.

3.2.2. Alternating Property Recognizers

Alternating property recognizers bear the same relationship to ordinary property recognizers as alternating finite automata (and alternating Turing machines) [Chandra et al. 80] bear to ordinary finite automata (and Turing machines). An *alternating property recognizer M* for a property of program π is a sextuple $\langle S, Q, \mathcal{Q}_0, Q_\infty, Q_F, \Delta \rangle$, where

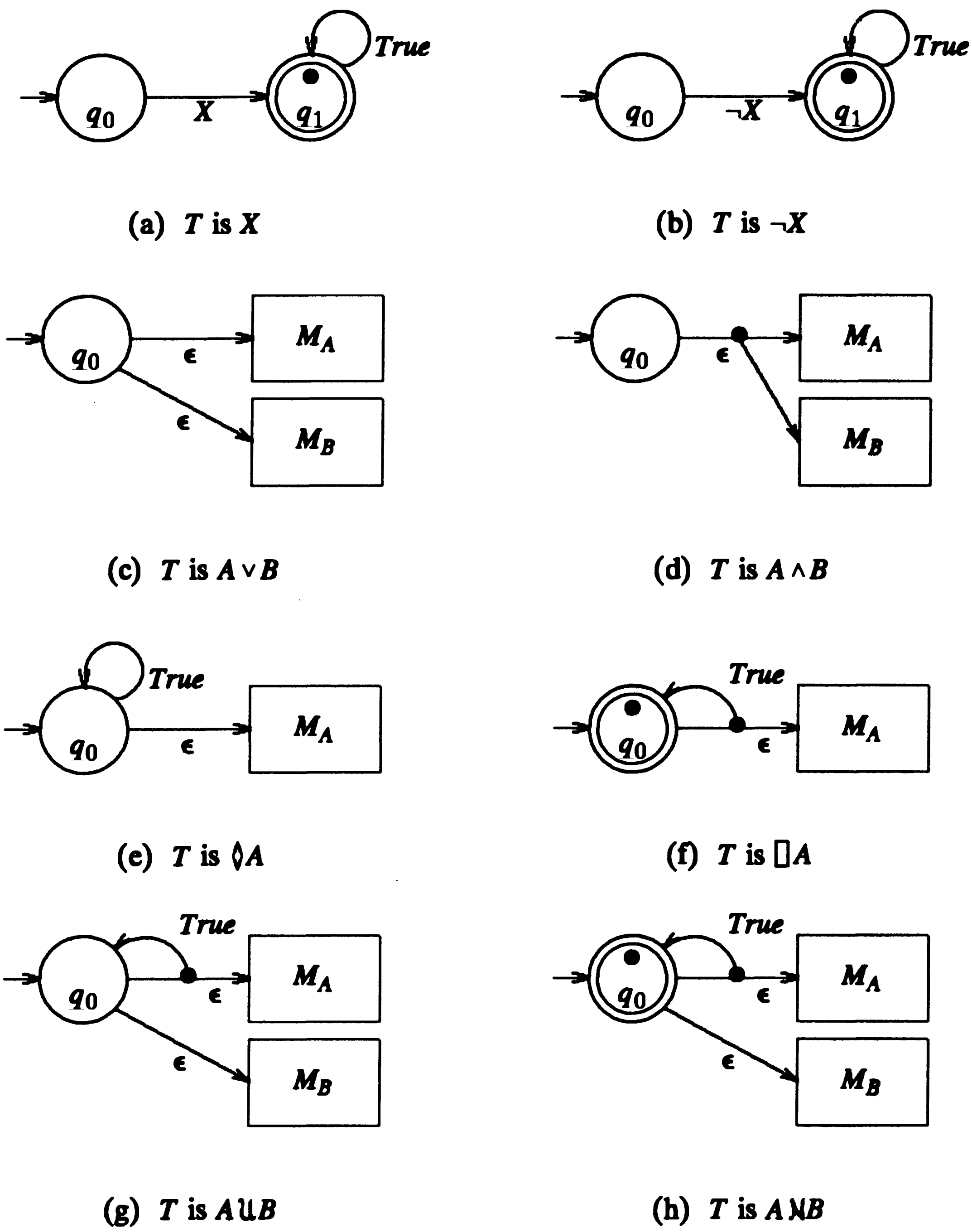
S is the set of program states of π
Q is the set of automaton states of m,
$\mathcal{Q}_0 \subseteq 2^Q$ is a collection of sets of start states of m,
$Q_\infty \subseteq Q$ is the set of infinite-accepting states of m,
$Q_F \subseteq Q$ is the set of finite-accepting states of m, and
$\Delta \in (Q \times S) \to 2^{2^Q}$ is the *transition function* of m.

Notice that the elements of this sextuple, except for $\mathcal{Q}_0$ and Δ, have the same form as the corresponding elements of the sextuple for an ordinary property recognizer. As it reads its input, an alternating property recognizer is simultaneously in a set of states called a *cluster*. $\mathcal{Q}_0$ is the set of possible starting clusters; M starts simultaneously in each state of one of these clusters. The range of the transition function Δ is the set of clusters (i.e. 2^{2^Q}) rather than just the sets of states (i.e. 2^Q). Reading a symbol s in a state q the alternating property recognizer non-deterministically chooses from a set of *hyper-transition*, each of which identifies a cluster to be entered. There is a *transition* from a cluster C to cluster C' under s if and only if C' is the union of destinations of hyper-transitions under s from the states of C.

A run of an alternating property recognizer M on input σ is a tree. The nodes of the tree are states M. The children of a node q at level k is the cluster identified by a hyper-transition from q under $\sigma[k]$. The alternating property recognizer accepts a finite input if it has a run in which every path ends in a finite-accepting state. It accepts an infinite input if it has a run in which every infinite path has infinitely many occurrences of an infinite-accepting state.

3.2.3. From Temporal Logic to Alternating Property Recognizer

Figure 3.7 shows how to construct an alternating property recognizer that recognizes the property specified by a given temporal formula. Certain notational conventions are used in this figure that need to be explained. A simple arc labelled ϵ form q to q' (Figure 3.7 (c) (e) (g) and (h)) indicates that any hyper-transition from q' is to be replicated for q. (If there are no non-ϵ-arcs into q' then it is superfluous.) A *hyper-arc*—an arc with more than one destination or *prong*—labelled ϵ form q to q' and q'' (Figure 3.7 (d)) indicates that, where there are hyper-transitions from q' to C' and from q'' to C'', there is a hyper-transition from q to $C' \cup C''$. (Again, q' or q'' may be superfluous.) A hyper-arc form q with one prong labelled ϵ to q' and another prong labelled P to q'' (Figure 3.7 (f) (g) and (h)) indicates that, where there is a hyper-transition under a program state that satisfies P from q' to a cluster C', there is a hyper-transition from q to $C' \cup \{q''\}$. The construction of an alternating property recognizer from a temporal formula can now be explained.



(a) T is X

(b) T is $\neg X$

(c) T is $A \vee B$

(d) T is $A \wedge B$

(e) T is $\Diamond A$

(f) T is $\Box A$

(g) T is $A \,\mathcal{U}\, B$

(h) T is $A \,\mathcal{N}\, B$

Figure 3.7. Constructing Alternating Property Recognizer M_T

The alternating property recognizer for an atomic formula X (Figure 3.7(a)) is an ordinary property recognizer with two states. The transition predicate between the two states requires that the first input symbol (i.e. program state) satisfy X. The alternating property recognizer for the negation of an atomic proposition $\neg X$ (Figure 3.7(b)) is similar. The alternating property recognizer for the disjunction of two properties (Figure 3.7(c)) is formed from their alternating property recognizers—this automaton non-deterministically chooses which sub-automaton to enter at the beginning of an input sequence. The alternating property recognizer for the conjunction of two properties (Figure 3.7(d)) is also formed from their alternating property recognizers. This automaton simultaneously enters both sub-automata at the beginning of an input sequence. The alternating property recognizer for $\Diamond A$ (Figure 3.7(e)) non-deterministically ignores some prefix of its input and requires that the alternating property recognizer for A accept the corresponding suffix. The alternating property recognizer for $\Box A$ (Figure 3.7(f)) simultaneously requires that the alternating property recognizer for A accept every suffix of its input. The alternating property recognizer for $A \mathcal{U} B$ (Figure 3.7(g)) requires that the recognizer for B accept some suffix of the input and that the recognizer for A accept every previous suffix. The alternating property recognizer for $A \mathcal{N} B$ (Figure 3.7(h)) requires that every suffix be accepted by the recognizer for A until and unless some suffix is recognized by the recognizer for B.

3.2.4. From Alternating to Ordinary Property Recognizer

We will now explain how to construct an ordinary property recognizer m that accepts the same set of sequences as a given alternating property recognizer M. Each state of m will simulate a cluster of M. The k^{th} state in a run of m will represent the cluster of states at the k^{th} level in a run of M. A state of m will be finite-accepting if and only if each of the states in the cluster it represents is finite-accepting.

However, it will not work to have the infinite-accepting states of m be those that represent clusters of infinite-accepting states of M. Even if every path in a run of M contains infinitely many infinite-accepting states, it is still possible that no level of the run would contain only such states. To handle this problem, imagine that a run is divided into phases. We begin at the root of the run. In each phase, infinite-accepting states and their descendants are marked. If, at some level, all of the states are marked, the current phase ends and a new one begins. Clearly, a run is accepting if and only if it has infinitely many phases. The states of m will keep track of which of the states they represent is marked. The infinite-accepting states of m are those that represent only marked states.

3.2.5. Example: Guaranteed Service

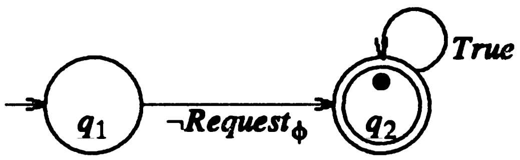
The Guaranteed Service property of Figure 3.3 can be expressed as the temporal logic formula

$$\Box(Request_{\phi} \Rightarrow \Diamond Serve_{\phi}),$$

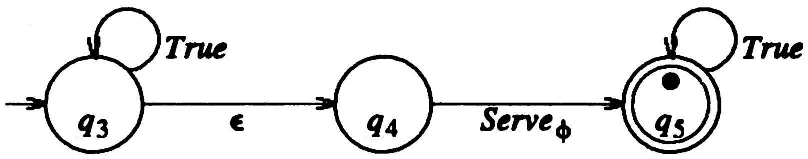
which requires that any request by process ϕ be serviced eventually. In normal form, this formula is

$$\Box(\neg Request_{\phi} \vee \Diamond Serve_{\phi}). \tag{3.1}$$

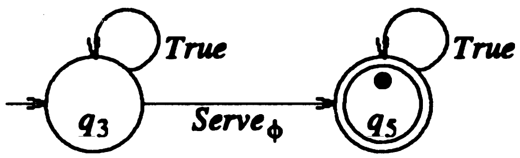
The alternating property recognizer for $\neg Request_{\phi}$ is:



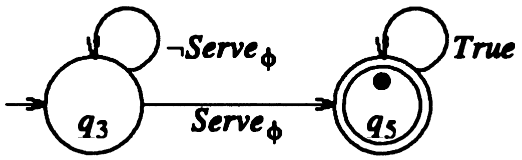
The alternating property recognizer for $\Diamond Serve_{\phi}$ is:



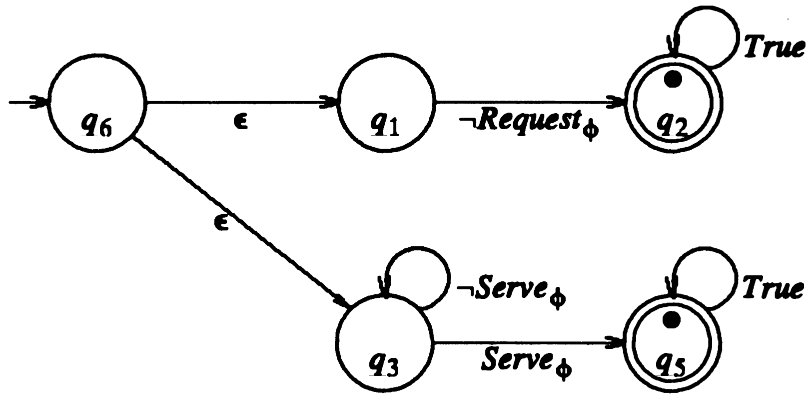
Removing the ϵ-arc, we have:



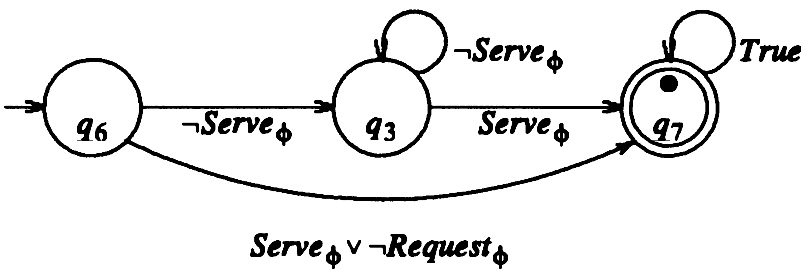
Since there is never any reason to take the transition from q_3 back to itself when transition to the terminal state q_5 is possible, this automaton is equivalent to:



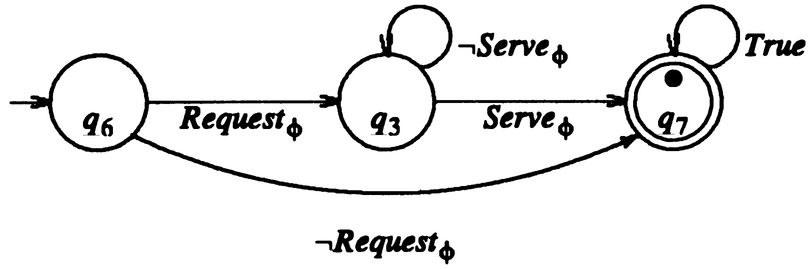
The alternating property recognizer for $\neg Request_\phi \vee \lozenge Serve_\phi$ is:



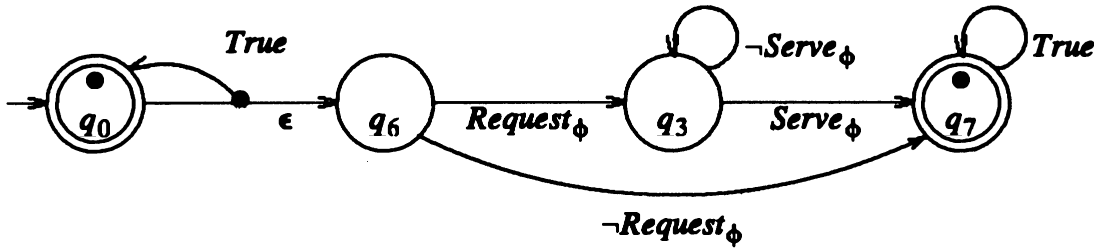
Notice that both q_2 and q_5 are terminal states. They can be combined since the occurrence of either state in a path through any run of the automaton guarantees that that path is accepting. Removing ϵ-arcs and identifying q_2 and q_5, we have:



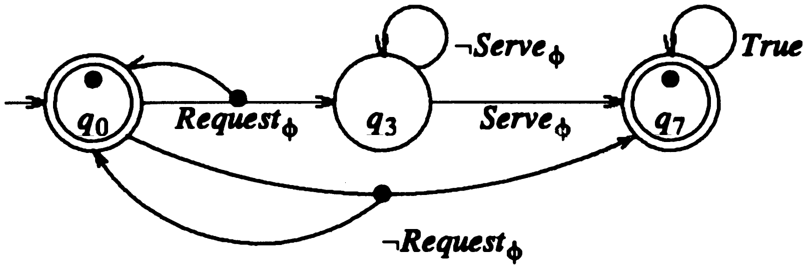
Notice that their is never any reason to enter q_3 from q_6 when it is possible to enter q_7, so the predicate on the arc from q_6 to q_3 can be strengthened to $Request_\phi \wedge \neg Serve_\phi$. Assuming that requests cannot be serviced instantaneously (that is, $Request_\phi \wedge Serve_\phi = False$), the automaton can be simplified to:



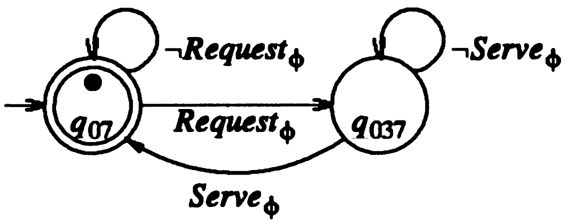
Finally, the alternating property recognizer for Guaranteed Service (3.1) is:



Eliminating the ϵ's, this automaton becomes:



This alternating property recognizer has four clusters: $\{q_0\}$, $\{q_0, q_7\}$, $\{q_0, q_3\}$, and $\{q_0, q_3, q_7\}$. The first cluster can be subsumed by the second, and the third by the fourth, because a terminal state like q_7 can be added to a cluster without affecting the set of sequences accepted by an alternating property recognizer. It is not necessary to mark the states in these clusters since a phase ends whenever the automaton enters a cluster without q_3. The ordinary property recognizer that simulates this alternating property recognizer is:



This is the property recognizer for Guaranteed Service m_{gs} of Figure 3.3.

CHAPTER 4

Verifying Deterministic Properties, Part I: Proof Obligations

4.1. Proof Obligations

The basis for our approach to verifying that a program π satisfies a property P is the observation that if a property recognizer for P accepts every history of π, then π satisfies P. Assume m is a deterministic property recognizer for P. (Non-deterministic property recognizers are considered in Chapter 7). One can think of m as simulating—in an abstract way—any program π that satisfies P. Thus, to show that π satisfies P, we demonstrate a correspondence between m and π. This is done by defining a *correspondence invariant* for each automaton state. Correspondence invariant C_i for automaton state q_i is a predicate that holds on a program state s if in some history of π m enters q_i upon reading s. Thus, if m is ever in automaton state q_i then the program state just read must satisfy C_i. Constraints satisfied by correspondence invariants are defined inductively, as follows.

The initial automaton state is q_0. If, upon reading an initial program state s_0, m enters automaton state q_j then s_0 must satisfy $Init_\pi$ and T_{0j}, the transition predicate labelling the arc from q_0 to q_j. Therefore, the correspondence invariant holds for q_j provided $(Init_\pi \wedge T_{0j}) \Rightarrow C_j$. Since this must hold for any automaton state q_j entered upon reading the first symbol of any history of π, we require

$$(\forall j\colon\ q_j \in Q\colon\ (Init_\pi \wedge T_{0j}) \Rightarrow C_j) \tag{4.1}$$

to ensure that the appropriate correspondence invariant holds after the first state in any history of π is read by m.

Now consider the k^{th} state in a history of π, and assume the appropriate correspondence invariant was satisfied for every previous program state in that history. Suppose m is in state q_i and that upon reading the next input symbol, program state s_k, a transition is made to automaton state q_j. Thus, s_{k-1} satisfies C_i and s_k satisfies transition predicate T_{ij}. The appropriate correspondence invariant C_j will hold provided $\{C_i\}\ \alpha\ \{T_{ij} \Rightarrow C_j\}$ is valid for any atomic action α of π. (If α is not enabled in s_{k-1} then the triple is trivially valid.) Generalizing to handle any atomic action and any automaton state that m might be in when s_k is read, we get:

$$\begin{array}{l} \text{For all } \alpha\colon\ \alpha \in A_\pi\colon \\ \quad \text{For all } i\colon\ q_i \in Q\colon \\ \quad\quad \{C_i\}\ \alpha\ \{\bigwedge_{j:q_j \in Q} (T_{ij} \Rightarrow C_j)\} \end{array} \tag{4.2}$$

We therefore conclude that any collection of predicates satisfying (4.1) and (4.2) are correspondence invariants for m and π.

In order to establish that π satisfies P, we must show that every history of π is accepted by m. There are exactly three ways that m might fail to accept σ, a history of π:

(1) m could attempt an undefined transition when reading σ.

(2) If σ is finite, m could halt in a non-finite-accepting state.

(3) If σ is infinite, m could refrain from entering infinite-accepting states after some finite prefix of σ.

Thus, in order to prove that every history of π satisfies P, it suffices to show that (1)−(3) are impossible.

Two obligations ensure that (1) is impossible. First, we must show that m can make some transition from its start state upon reading any initial program state:

$$Init_\pi \Rightarrow \bigvee_{j:q_j \in Q} T_{0j} \tag{4.3}$$

Second, we must show that m can always make a transition upon reading subsequent states in a history. If m is in state q_i then the program state just read by m satisfies correspondence invariant C_i. To avoid an undefined transition, any atomic action α that is then executed must transform the program state so that one of the transition predicates T_{ij} emanating from q_i holds. This is guaranteed by

$$\begin{array}{l} \text{For all } \alpha\text{: } \alpha \in \mathcal{A}_\pi\text{:} \\ \quad \text{For all } i\text{: } q_i \in Q\text{:} \\ \quad\quad \{C_i\}\ \alpha\ \{\bigvee_{j:q_j \in Q} T_{ij}\} \end{array} \tag{4.4}$$

We can exploit the fact that m is deterministic to combine and simplify the obligations derived so far. In a deterministic property recognizer, the transition predicates on arcs emanating from any automaton state are disjoint. Thus,

$$(\forall i,j,k:\ q_i,\ q_j,\ q_k \in Q\ \wedge\ j \neq k:\ (T_{ij} \wedge T_{ik}) = False). \tag{4.5}$$

Using (4.5), we combine (4.1) and (4.3), obtaining

Simulation Basis: $$Init_\pi \Rightarrow (\bigvee_{j:q_j \in Q} (T_{0j} \wedge C_j)), \tag{4.6}$$

and combine (4.2) and (4.4), obtaining

Simulation Induction: For all α: $\alpha \in A_\pi$:
For all i: $q_i \in Q$:
$$\{C_i\}\ \alpha\ \{\bigvee_{j:q_j \in Q} (T_{ij} \wedge C_j)\}\ . \tag{4.7}$$

To ensure that it is impossible for m to halt in a non finite-accepting state—(2) above—the correspondence invariant for any non-finite-accepting state must hold only for program states in which subsequent execution by π is guaranteed. Since C_i holds of the program state just read by m, and $Blocked_\pi$ holds for all program states of π in which subsequent execution is not inevitable, we require

Finite Acceptance: $$(\forall i:\ q_i \in Q - Q_F:\ C_i \Rightarrow \neg Blocked_\pi). \tag{4.8}$$

Finally, we must ensure that every infinite history of π is accepted by m, hence (3) is impossible. A set Q' of automaton states is ***strongly connected*** if and only if there is a sequence of transitions from any element of Q' to any other without involving an automaton state outside of Q'. A ***reject knot*** κ is a maximal strongly connected subset of Q containing no infinite-accepting states. It may, however, contain finite-accepting states. In order to show that every infinite history of π is accepted by m, we must prove that the run for no infinite history of π

is restricted to automaton states in $Q-Q_\infty$. We do this by constructing a *variant function* v_κ for each reject knot κ.

A variant function is a function from automaton and program states[1] to the Natural Numbers. Whenever $v_\kappa(q,s)=0$ for $q \in \kappa$ and $s \in S_\pi$, we require that q be a finite-accepting state and that s be the final state in any history in which it occurs:

$$\textbf{Knot Exit:}\ (\forall i\colon\ q_i \in \kappa\colon\ (v_\kappa(q_i)=0) \Rightarrow \mathit{Blocked}_\pi \vee \neg C_i) \tag{4.9}$$

This means that if $v_\kappa(q) = 0$, either the history is finite and will be accepted by m or some progress towards accepting the history will be made. Finally, to ensure that the variant function does reach 0, we require that it is decreased by every atomic action in π that might be executed:

$$\begin{array}{l} \textbf{Knot Variance:}\ \text{For all } \alpha\colon\ \alpha \in A_\pi\colon \\ \qquad \text{For all } q_i \in \kappa\colon \\ \qquad\quad \{C_i \wedge 0<v_\kappa(q_i)=V\}\ \alpha\ \{\bigwedge_{q_j \in \kappa}((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j)<V)\} \end{array} \tag{4.10}$$

Note, requiring that $v_\kappa(q)$ be decreased by any enabled atomic action does not preclude proving properties under various fairness assumptions. To prove that a property P holds assuming some fairness property F, proof obligations are extracted from the property recognizer for $F \Rightarrow P$.

The five proof obligations—Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10)—are of three basic forms. Simulation Basis (4.6), Finite Acceptance (4.8), and Knot

[1]The program state argument is often left implicit.

Exit (4.9) involve proving that predicate logic formulas are valid. Simulation Induction (4.7) involves proving invariance of some assertions. Knot Variance (4.10) involves proving that certain events cause variant functions to be decreased. Of course, the intellectual challenge in proving that a program satisfies a property lies not in checking the proof obligations, but in devising the correspondence invariants and variant functions. The proof obligations, however, do give insight into forms the correspondence invariants and variant function must take.

4.2. A Detailed Example

To illustrate our verification method, we prove that if program θ of Figure 2.1 is started in a state where $x=0$ then it will terminate with $x=2$. This is an instance of Total Correctness.

For θ, we have

$$\begin{aligned} Init_\theta \ &\equiv \ pc_\mu=1 \wedge pc_\eta=3 \\ Blocked_\theta \ &\equiv \ pc_\mu=2 \wedge pc_\eta=4 \end{aligned}$$

and $\mathcal{A}_\theta=\{\alpha_1, \alpha_2\}$, where

$$\begin{aligned} \alpha_1 \ &\equiv \ \langle \textbf{if } pc_\mu=1 \rightarrow pc_\mu, x := 2, x+1 \textbf{ fi} \rangle \\ \alpha_2 \ &\equiv \ \langle \textbf{if } pc_\eta=3 \rightarrow pc_\eta, x := 4, x+1 \textbf{ fi} \rangle . \end{aligned}$$

A property recognizer m_{tc} for Total Correctness appears in Figure 3.6. For predicates *Pre*, *Post*, and *Done* we choose:

$$Pre \equiv x=0$$
$$Post \equiv x=2$$
$$Done \equiv pc_\mu=2 \wedge pc_\eta=4$$

Thus, m_{tc} accepts every sequence of states such that if $x=0$ holds for the first state, then the sequence is finite and the final state is one in which $x=2$ and both μ and η have terminated.

We first define correspondence invariants for each of the four automaton states of m_{tc}.

$$C_0 \equiv False$$
$$\begin{aligned} C_1 \equiv\ & pc_\mu=1 \Rightarrow ((pc_\eta=3 \Rightarrow x=0) \wedge (pc_\eta=4 \Rightarrow x=1)) \wedge \\ & pc_\mu=2 \Rightarrow ((pc_\eta=3 \Rightarrow x=1) \wedge pc_\eta \neq 4)) \wedge \\ & pc_\eta=3 \Rightarrow ((pc_\mu=1 \Rightarrow x=0) \wedge (pc_\mu=2 \Rightarrow x=1)) \wedge \\ & pc_\eta=4 \Rightarrow ((pc_\mu=1 \Rightarrow x=1) \wedge pc_\mu \neq 2)) \end{aligned}$$
$$C_2 \equiv True$$
$$C_3 \equiv pc_\mu=2 \wedge pc_\eta=4 \wedge x=2$$

To satisfy Simulation Basis (4.6), we must show that

$$Init_\theta \Rightarrow ((False \wedge C_0) \vee (Pre \wedge \neg Done \wedge C_1) \vee (\neg Pre \wedge C_2) \vee (Pre \wedge Done \wedge Post \wedge C_3)$$

is valid. Substituting, we get

$$\begin{aligned} (pc_\mu=1 \wedge pc_\eta=3) \Rightarrow (\ & False \\ \vee\ & (x=0 \wedge \neg (pc_\mu=2 \wedge pc_\eta=4) \wedge C_1) \\ \vee\ & (x \neq 0) \\ \vee\ & (x=0 \wedge pc_\mu=2 \wedge pc_\eta=4 \wedge x=2)\ ), \end{aligned}$$

which is valid.

To satisfy Simulation Induction (4.7), we must show for each $\alpha \in A_\theta$ that the following partial correctness logic triples are valid:

$$\{C_0\}\ \alpha\ \{(T_{00}\wedge C_0)\vee(T_{01}\wedge C_1)\vee(T_{02}\wedge C_2)\vee(T_{03}\wedge C_3)\} \tag{4.11}$$

$$\{C_1\}\ \alpha\ \{(T_{10}\wedge C_0)\vee(T_{11}\wedge C_1)\vee(T_{12}\wedge C_2)\vee(T_{13}\wedge C_3)\} \tag{4.12}$$

$$\{C_2\}\ \alpha\ \{(T_{20}\wedge C_0)\vee(T_{21}\wedge C_1)\vee(T_{22}\wedge C_2)\vee(T_{23}\wedge C_3)\} \tag{4.13}$$

$$\{C_3\}\ \alpha\ \{(T_{30}\wedge C_0)\vee(T_{31}\wedge C_1)\vee(T_{32}\wedge C_2)\vee(T_{33}\wedge C_3)\} \tag{4.14}$$

Since the triples for α_2 are symmetric with those for α_1, we prove only the former.

Because $T_{00} = \mathit{False}$, $C_2 = \mathit{True}$, and $\mathit{Pre} \wedge \mathit{Post} = \mathit{False}$, we need only show

$$\{\mathit{False}\}\ \alpha_1\ \{(\mathit{Pre}\wedge\neg\mathit{Done}\wedge C_1)\vee(\neg\mathit{Pre})\}$$

to prove (4.11) valid. This follows from the Assignment Axiom and Rule of Consequence.

After simplifying (4.12), ($T_{10} = T_{12} = \mathit{False}$), we get

$$\{C_1\}\ \alpha_1\ \{(\neg\mathit{Done}\wedge C_1)\vee(\mathit{Done}\wedge\mathit{Post}\wedge C_3)\}. \tag{4.15}$$

From the definition of α_1 (2.1) and the Await Rule, it suffices to demonstrate

$$\{C_1\wedge pc_\mu=1\}\ pc_\mu,\, x := 2,\, x+1\ \{(\neg\mathit{Done}\wedge C_1)\vee(\mathit{Done}\wedge\mathit{Post}\wedge C_3)\}$$

to prove (4.15). Expanding and substituting, we get

$$\begin{array}{l}\{(pc_\eta=3\Rightarrow x=0)\ \wedge\ (pc_\eta=4\Rightarrow x=1)\ \wedge\ pc_\mu=1\}\\ pc_\mu,\, x := 2,\, x+1\\ \{(\neg(pc_\mu=2\ \wedge\ pc_\eta=4)\wedge C_1)\vee(pc_\mu=2\ \wedge\ pc_\eta=4\ \wedge\ x=2)\},\end{array}$$

which follows from the Assignment Axiom and Rule of Consequence.

Triple (4.13) simplifies to $\{\mathit{True}\}\ \alpha_1\ \{\mathit{True}\}$, which is valid.

Because T_{30}, T_{31}, T_{32}, and T_{33} are all False, triple (4.14) simplifies to $\{C_3\}\ \alpha_1\ \{\mathit{False}\}$. From definition of α_1 (2.1) and the Await Rule, it suffices to show

$$\{C_3 \wedge pc_\mu = 1\}\ \ pc_\mu,\ x := 2,\ x+1\ \ \{False\}.$$

This follows from the Assignment Axiom and Rule of Consequence, because $(C_3 \wedge pc_\mu = 1) = False$.

To satisfy Finite Acceptance (4.8), we must prove that

$$(C_0 \Rightarrow \neg Blocked_\theta) \wedge (C_1 \Rightarrow \neg Blocked_\theta)$$

since $Q_F = \{q_2, q_3\}$. Substituting and simplifying, we get

$$(False \Rightarrow (pc_\mu \neq 2 \vee pc_\eta \neq 4)) \wedge (C_1 \Rightarrow (pc_\mu \neq 2 \vee pc_\eta \neq 4)),$$

which is valid.

The final two obligations concern reject knots. There is a single reject knot $\kappa = \{q_1\}$ in $m_{\kappa c}$. Define

$$v_\kappa(q_1) \ = \ (2 - pc_\mu) + (4 - pc_\eta).$$

Knot Exit (4.9) requires that

$$(v_\kappa(q_1) = 0) \Rightarrow Blocked_\theta \vee \neg C_1.$$

This is valid because

$$\begin{aligned}(v_\kappa(q_1) = 0) &\Rightarrow (pc_\mu = 2 \wedge pc_\eta = 4)\\ &= Blocked_\theta\ .\end{aligned}$$

To satisfy Knot Variance (4.10), we must establish the validity of 2 triples:

$$\{C_1 \wedge 0 < v_\kappa(q_1) = V\}\ \ \alpha_1\ \ \{(\neg Done \wedge C_1) \Rightarrow v_k(q_1) < V\} \tag{4.16}$$

$$\{C_1 \wedge 0 < v_\kappa(q_1) = V\}\ \ \alpha_2\ \ \{(\neg Done \wedge C_1) \Rightarrow v_k(q_1) < V\} \tag{4.17}$$

We give details only for the first; the second is similar. Using definition of α_1 (2.1), the Await Rule, and the Rule of Consequence, to prove (4.16) it suffices to prove

$$\{C_1 \wedge 0<\nu_\kappa(q_1)=V \wedge pc_\mu=1\}\ pc_\mu,\, x := 2,\, x+1\ \{\nu_\kappa(q_1)<V\}.$$

This is valid because changing pc_μ from 1 to 2 decreases ν_κ.

4.3. Soundness and Relative Completeness

The soundness and relative completeness of our approach is shown below. We first show that the proof obligations are sound. We then show that they are complete relative to predicate logic and partial correctness logic (Figure 2.2). Since partial correctness logic is known to be complete relative to predicate logic, our proof obligations are also complete relative to predicate logic.

Soundness Theorem 4.1: If for a program π and deterministic property recognizer m_P for property P there are correspondence invariants and variant functions such that Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10) are valid, then π satisfies P.

Proof: Assume that the proof obligations are valid for some correspondence invariants and variant functions and that σ is a history of π. We must show that σ satisfies P.

By induction on n,

$$\delta^*(q_0, \sigma[..n])=q_i \Rightarrow C_i(\sigma[n])$$

due to Simulation Basis (4.6) and Simulation Induction (4.7). A similar inductive argument shows that m_P cannot attempt an undefined transition when reading $\sigma[n]$.

We now show that if σ is finite then it is accepted by m_P. Without loss of generality, let $\sigma[n]$ be the final state of σ. We must show $\delta^*(q_0, \sigma[..n]) \in Q_F$. Due to Finite Acceptance (4.8), if $\delta^*(q_0, \sigma[..n])$ is a non-finite-accepting state, then π cannot be blocked in $\sigma[n]$ and this contradicts the assumption that $\sigma[n]$ is the final state of σ. Thus, we conclude that $\delta^*(q_0, \sigma[..n])$ is a finite-accepting state, and, by definition, m_P accepts σ, hence σ satisfies P.

Finally, we show that if σ is infinite then it is accepted by m_P. By Knot Exit (4.9) and Knot Variance (4.10), if m_P enters a reject knot κ upon reading $\sigma[n]$, then it must exit κ before reading the $v_\kappa(\delta^*(q_0, \sigma[..n]), \sigma[n])+n^{th}$ symbol of σ. By the definition of a reject knot, m_P cannot reenter κ after exiting it, without first entering an infinite-accepting state. Since there are finitely many reject knots and σ is infinite, m_P must enter an infinite-accepting state infinitely often. Thus, by definition, m_P will accept σ, hence σ satisfies P. □

Relative Completeness Theorem 4.2: If a program π satisfies a property P that is accepted by a deterministic property recognizer m_P, then there exist correspondence invariants and variant functions, for which Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10) are valid.

Proof: Assume m_P accepts every history of π. We must show that (4.6)−(4.10) for π and m_P are valid.

Choose correspondence invariants and variant functions as follows. First, for each automaton state q_i, define

$$C_i(s) \equiv (\exists \sigma, n\colon \sigma \in H_\pi, 0 \le n\colon s=\sigma[n] \wedge \delta^*(q_0, \sigma[..n])=q_i),$$

where H_π is the set of histories of π. Thus, $C_i(s)$ holds for a program state s if and only if there is some history of π in which s caused m_P to make a transition to q_i. Next, for each reject knot κ and each $q_i \in \kappa$, define

$$v_\kappa(q_i, s) \equiv \begin{cases} 0, & \text{if } \neg C_i(s) \vee \mathit{Blocked}_\pi(s) \\ 1+\max\limits_{v}(\exists \sigma, n\colon \sigma \in H_\pi, 0 \le n\colon s=\sigma[n] & \\ \qquad \wedge\ \delta^*(q_0, \sigma[..n])=q_i & \\ \qquad \wedge\ \neg \mathit{Blocked}_\pi(\sigma[n+v]) & \\ \qquad \wedge\ \forall j\colon 0 \le j \le v\colon \delta^*(q_0, \sigma[..n+j]) \in \kappa & \\ \qquad), & \text{if } C_i(s) \wedge \neg \mathit{Blocked}_\pi(s) \end{cases}$$

Thus, $v_\kappa(q_i, s)$ is the maximum number of atomic actions π can execute when in state s and m_P is in q_i before m_P will halt or leave κ.

It remains to prove that (4.6)−(4.10) are valid with these correspondence invariants and variant functions. We consider each proof obligation in turn.

Simulation Basis (4.6). Since π satisfies P, every initial state of π must satisfy some transition predicate T_{0j}. By construction, this initial state will also satisfy C_j. Thus, (4.6) is valid.

Simulation Induction (4.7). Consider any program history σ and suppose $\delta^*(q_0, \sigma[..n]) = q_i$ for some n. By construction, $C_i(\sigma[n])$. Consider an atomic action α from A_π that terminates in a state s when started in state $\sigma[n]$. Clearly, σs is the prefix of some history σ' of π. Since m_P accepts every history of π, m_P must accept σ', so there must exist an automaton state q_j such that $\sigma'[n+1]$ satisfies T_{ij}. By construction, $C_j(\sigma'[n+1])$. Therefore, $\{C_i\}\ \alpha\ \{\bigvee_{j:q_j \in Q} (T_{ij} \wedge C_j)\}$ is valid for any atomic action that terminates when started in a state satisfying C_i. Since $\{C_i\}\ \alpha\ \{T_{ij} \wedge C_j\}$ is valid for any atomic action α that does not terminate when started in a state satisfying C_i, we have shown that (4.7) is valid.

Finite Acceptance (4.8). Consider any program state $\sigma[n]$ in some history σ of π. Suppose $\delta^*(q_0, \sigma[..n]) = q_j$. Thus, by construction $C_j(\sigma[n])$. If $q_j \in Q - Q_F$, then $\sigma[n]$ also satisfies $\neg Blocked_\pi$. Otherwise, $\sigma[n]$ would have to be the final state of σ, which would cause m_P to reject σ, contradicting the assumption that every history of π is accepted by m_P. Thus, $C_j \Rightarrow \neg Blocked_\pi$ is valid, so (4.8) is valid.

Knot Exit (4.9). The proof that (4.9) is valid is trivial, by construction of v_κ.

Knot Variance (4.10). If α is not enabled in a state satisfying some correspondence invariant C_i for an automaton state $q_i \in \kappa$, then

$$\{C_i \wedge v_\kappa(q_i) = V\}\ \alpha\ \{\bigwedge_{q_j \in \kappa} ((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < V)\} \tag{4.18}$$

is trivially valid.

Suppose α is enabled and terminates in state s' when started in state s. By construction of C_i, there must exist a history σ_1 and an integer n_1 such that $\sigma_1[n_1]=s$ and $\delta^*(q_0, \sigma_1[..n_1])=q_i$. Suppose s' satisfies T_{ij} and C_j for some $q_j \in \kappa$. By construction of v_κ, there must exist a history σ_2 and an integer n_2 such that $\sigma_2[n_2]=s'$, $\delta^*(q_0, \sigma_2[..n_2])=q_j$, and $\neg Blocked_\pi(\sigma_2[v_\kappa(q_j, s')])$ $(\forall j\colon 0 \le j \le v_\kappa(q_j, s')\colon \delta^*(q_0, \sigma_2[..n_2+j] \in \kappa)$. Let $\sigma=\sigma_1[..n_1]\sigma_2[n_2..]$. Since α terminates in s' when started in state s, σ is a history of π. By the construction of v_κ, we conclude $v_\kappa(q_j, s')+1 \le v_\kappa(q_i, s)$. So, (4.18) is valid. This concludes the relative completeness proof. □

4.4. A Note on Variant Functions

Our variant functions have two arguments, the automaton state and the program state. If, as is often the case, a reject knot contains only one automaton state, the first argument is superfluous. Even if a reject knot has more than one state, it is not immediately obvious that this first argument is necessary. We now give an example requiring a variant function with both arguments.

Consider property recognizer m_{vf} in Figure 4.1. There is one reject knot which contains states q_1 and q_2. To reject an infinite sequence (without attempting an undefined transition), m_{vf} must "loop" in q_1 under W or Y, "loop" in q_2 under W or X, and/or "ping-pong" between q_1 and q_2 under Z.

A contrived program that satisfies this property is shown in Figure 4.2. The program is a loop containing three guarded commands. Notice that the second

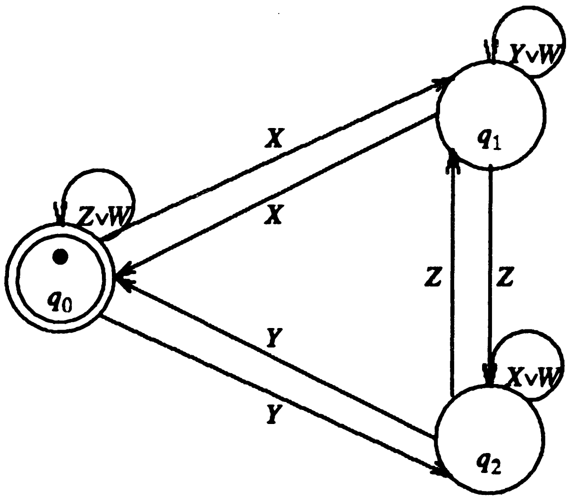


Figure 4.1. Property Recognizer m_{vf}

$\{\ 0 \le a \wedge 0 \le b\ \}$

do $a+b>2 \rightarrow W,\ Z,\ a,\ b := False,\ True,\ a-1,\ b-1$
$\quad W,\ X,\ Y,\ Z := True,\ False,\ False,\ False;$

$[]\ a>0 \rightarrow W,\ X,\ a,\ b := False,\ True,\ a-1,\ b+1$
$\quad W,\ X,\ Y,\ Z := True,\ False,\ False,\ False;$

$[]\ b>0 \rightarrow W,\ Y,\ a,\ b := False,\ True,\ a+1,\ b-1$
$\quad W,\ X,\ Y,\ Z := True,\ False,\ False,\ False;$

od

Figure 4.2. A Contrived Program

assignment is the same for each guarded command. It sets W and resets the other predicates. In effect, this blots out the transition information of the preceding assignment. To see that this program satisfies the property, take

$$C_0 \equiv True \qquad C_1 \equiv a+b>0 \qquad C_2 \equiv a+b>0$$

to be the correspondence predicates, and

$$v(q_1) \equiv (a+b)^2 + 2a + |\neg W| \qquad v(q_2) \equiv (a+b)^2 + 2b + |\neg W|$$

(where $|True|$ is 1 and $|False|$ is 0) to be the variant function. Note that the first guarded command decreases both $v(q_1)$ and $v(q_2)$, while the second (third) decreases (increases) $v(q_1)$ and increases (decreases) $v(q_2)$. We leave it to the reader to check that v and the C_i's meet the proof obligations for m_{vf}.

To see that every variant function for this program requires two arguments, consider two program states

$$s_1 \equiv W \wedge \neg X \wedge \neg Y \wedge \neg Z \wedge a=1 \wedge b=3, \text{ and}$$
$$s_2 \equiv W \wedge \neg X \wedge \neg Y \wedge \neg Z \wedge a=3 \wedge b=1.$$

Assume there is a variant function f with one argument, the program state. Suppose the program is in s_1 and m_{vf} is in q_1. The third guarded command can be executed twice without taking m_{vf} out of the reject knot. The resulting program state is s_2. Since f is to be a variant function, $f(s_1) > f(s_2)$. However, starting from s_2 with m_{vf} in q_2, the second guarded command can be executed twice without taking m_{vf} out of the reject knot. The resulting program state is s_1. So, $f(_2) > f(s_1)$. This contradicts the assumption that f is a variant function. In this case, a variant function must have two arguments.

CHAPTER 5

Verifying Deterministic Properties, Part II: Property Outlines

In this chapter, we describe *property outlines* to give a compact representation for the correspondence invariants and the Simulation Induction (4.7) obligations of a given property recognizer and program. Property outlines play much the same role in our approach to verification as proof outlines do for verifying Partial Correctness using Hoare's partial correctness logic—they make it easy to do verification informally and make it easy to present a proof. In fact, proof outlines are a special case of property outlines (see section 5.4).

5.1. Proof Outlines

A *proof outline* for a concurrent program π is the text of π annotated with an *assertion* P^l at each control point l. Each assertion is a first-order predicate logic formula involving the program variables and program counters of π.[1] A proof outline is *valid* provided:

> **Proof Outline Validity:** Executing any enabled atomic action in a state where the assertions associated with the control points denoted by program counters hold, produces a state in which the assertions associated with the control points denoted by program counters still hold.

[1] The conjunct $pc_\phi = l$ is often left implicit and omitted from P^l in a proof outline for process ϕ.

Proving validity of a proof outline for a concurrent program can be reduced to proving the validity of a collection of partial correctness logic triples [Owicki & Gries 76].[2] This is done as follows, where $pre(\alpha)$ is the assertion immediately preceding α in the proof outline and $post(\alpha)$ is the assertion immediately following it.

Sequential Correctness: For each atomic action α in the proof outline, prove

$$\{pre(\alpha)\}\ \alpha\ \{post(\alpha)\}.$$

Interference Freedom: For each atomic action α in the proof outline and every assertion R in a process different from the one containing α, prove:

$$\{pre(\alpha) \wedge R\}\ \alpha\ \{R\}.$$

5.2. Property Outlines

A property outline for property recognizer m and program π is obtained by adding information about correspondence invariants to a control-point annotation for π. For each control point l and for each automaton state q, the property outline specifies what must be true when the program counter denotes l and the property recognizer is in state q. This is done by placing a property assertion at each control point in a control-point annotation for π.

A *property assertion*

$q_0{\sim}P_0 \mid \ldots \mid q_n{\sim}P_n$,

[2]See the formulation of this approach in [Schneider & Andrews 86].

where $q_0, \ldots, q_n$ are the automaton states of m and $P_0, \ldots, P_n$ are first-order predicate logic formulas involving the program variables of π (possibly including program counters), holds in an automaton state q_i and program state s if s satisfies P_i. A property outline for π and m is *valid* provided:

> **Property Outline Validity:** Executing any enabled atomic action in an automaton state q and a program state s where the property assertions associated with the control points denoted by program counters hold, produces a program state s' that causes the property recognizer to enter q' in which the property assertions associated with the control points denoted by program counters still hold.

Figure 5.1 is a valid property outline for the program of Figure 2.1 and the property recognizer m_{tc} for Total Correctness.

We can exploit the similarity in the definition of validity for proof outlines and for property outlines in developing a procedure to establish the validity of a property outline. Define a *property triple*

$$\{\tilde{P}\} \ \alpha \ \{\tilde{Q}\}, \tag{5.1}$$

where $\tilde{P}$ and $\tilde{Q}$ are property assertions[3], to be valid, if in every state satisfying $\tilde{P}$ any enabled atomic action terminates in a state satisfying $\tilde{Q}$. Note that (5.1) cannot be a partial correctness logic triple because it contains property assertions in its pre- and postcondition. However, the interpretation of (5.1) is quite similar to

[3]By convention, names of property assertions are uppercase letters with a "~" over them.

$$
\begin{array}{l}
\theta\text{: } \textbf{cobegin} \\
\quad \mu\text{: } \{1\text{: } q_0 \sim \mathit{False} \\
\qquad\qquad \mid q_1 \sim (pc_\eta = 3 \Rightarrow x = 0) \wedge (pc_\eta = 4 \Rightarrow x = 1) \\
\qquad\qquad \mid q_2 \sim \mathit{True} \\
\qquad\qquad \mid q_3 \sim \mathit{False}\} \\
\qquad \alpha_1\text{: } x := x+1 \\
\qquad \{2\text{: } q_0 \sim \mathit{False} \\
\qquad\qquad \mid q_1 \sim (pc_\eta = 3 \Rightarrow x = 1) \wedge pc_\eta \neq 4 \\
\qquad\qquad \mid q_2 \sim \mathit{True} \\
\qquad\qquad \mid q_3 \sim pc_\mu = 2 \wedge pc_\eta = 4 \wedge x = 2\} \\
\quad // \\
\quad \eta\text{: } \{3\text{: } q_0 \sim \mathit{False} \\
\qquad\qquad \mid q_1 \sim (pc_\mu = 1 \Rightarrow x = 0) \wedge (pc_\mu = 2 \Rightarrow x = 1) \\
\qquad\qquad \mid q_2 \sim \mathit{True} \\
\qquad\qquad \mid q_3 \sim \mathit{False}\} \\
\qquad \alpha_2\text{: } x := x+1 \\
\qquad \{4\text{: } q_0 \sim \mathit{False} \\
\qquad\qquad \mid q_1 \sim (pc_\mu = 1 \Rightarrow x = 1) \wedge pc_\mu \neq 2 \\
\qquad\qquad \mid q_2 \sim \mathit{True} \\
\qquad\qquad \mid q_3 \sim pc_\mu = 2 \wedge pc_\eta = 4 \wedge x = 2\} \\
\quad \textbf{coend}
\end{array}
$$

Figure 5.1. Example Property Outline

the interpretation of a partial correctness logic triple. In fact, if we can show how to establish the validity of a property triple like (5.1) and one like

$$\{\vec{P} \wedge \vec{R}\} \;\; \alpha \;\; \{\vec{R}\}, \tag{5.2}$$

then we have solved the problem of establishing the validity of a property outline. This is because we can use Sequential Correctness and Interference Freedom to reduce the problem to showing that a collection of property triples is valid. The soundness of this approach for establishing property outline validity is based on the same argument as for proof outline validity.

Based on the interpretation of property assertions, note that:

$$\begin{aligned}((q_0{\sim}P_0 \mid \dots \mid q_n{\sim}P_n) \wedge (q_0{\sim}R_0 \mid \dots \mid q_n{\sim}R_n))\\ \equiv \ (q_0{\sim}P_0{\wedge}R_0 \mid \dots \mid q_n{\sim}P_n{\wedge}R_n)\end{aligned} \tag{5.3}$$

Thus, it suffices to be able to prove the validity of property triples like (5.1), since using (5.3), those like (5.2) can always be transformed to be like (5.1). We, therefore, turn to the problem of proving validity of property triples.

Consider a property triple

$$\{\bar{P}\colon (q_0{\sim}P_0 \mid \dots \mid q_n{\sim}P_n)\} \ \alpha \ \{\bar{Q}\colon (q_0{\sim}Q_0 \mid \dots \mid q_n{\sim}Q_n)\}. \tag{5.4}$$

To prove the validity of (5.4), it suffices to prove the following partial correctness logic triples.

$$\{P_0\} \ \alpha \ \{(T_{00} \wedge Q_0) \vee \dots \vee (T_{0n} \wedge Q_n)\} \tag{5.5}$$

$$\dots$$

$$\{P_n\} \ \alpha \ \{(T_{n0} \wedge Q_0) \vee \dots \vee (T_{nn} \wedge Q_n)\} \tag{5.6}$$

The first, (5.5), establishes that every atomic action enabled in a (program) state satisfying P_0 terminates in a (program) state satisfying $T_{0j} \wedge Q_j$, for some j. From this, we conclude that execution of an enabled atomic α in a program state satisfying P_0 with m in automaton state q_0 terminates in a state s' satisfying Q_j and m will make a transition to automaton state q_j upon reading s'. Thus, $\bar{Q}$ holds for the case that m is started in q_0. Repeating this argument for the remaining triples, we find that no matter what automaton state m is in, $\bar{Q}$ will hold after the execution of an enabled atomic action. Thus, (5.5)−(5.6) together imply that in a state satisfying $\bar{P}$ an enabled atomic action will terminate in a state satisfying $\bar{Q}$, hence $\{\bar{P}\}\ \alpha\ \{\bar{Q}\}$.

We illustrate this approach for proving validity of a property outline, on the one in Figure 5.1. There are two Sequential Correctness obligations:

$$\{1\}\ \alpha_1\ \{2\} \tag{5.7}$$

$$\{3\}\ \alpha_2\ \{4\} \tag{5.8}$$

And, there are four Interference Freedom obligations:

$$\{1\ \wedge\ 3\}\ \alpha_1\ \{3\} \tag{5.9}$$

$$\{1\ \wedge\ 4\}\ \alpha_1\ \{4\} \tag{5.10}$$

$$\{3\ \wedge\ 1\}\ \alpha_2\ \{1\} \tag{5.11}$$

$$\{3\ \wedge\ 2\}\ \alpha_2\ \{2\} \tag{5.12}$$

The details for only one of these property triples will be given; the remaining ones are left to the energetic reader. Property triple (5.7) is:

$$\begin{array}{l}
\{1\!: \ q_0 \sim \mathit{False} \\
\quad \mid\ q_1 \sim (pc_\eta = 3 \Rightarrow x = 0) \wedge (pc_\eta = 4 \Rightarrow x = 1) \\
\quad \mid\ q_2 \sim \mathit{True} \\
\quad \mid\ q_3 \sim \mathit{False}\} \\
\alpha_1\!: \ x := x + 1 \\
\{2\!: \ q_0 \sim \mathit{False} \\
\quad \mid\ q_1 \sim (pc_\eta = 3 \Rightarrow x = 1) \wedge pc_\eta \neq 4 \\
\quad \mid\ q_2 \sim \mathit{True} \\
\quad \mid\ q_3 \sim pc_\mu = 2 \wedge pc_\eta = 4 \wedge x = 2\}
\end{array}$$

Decomposing this into partial correctness logic triples we get:

$$\begin{array}{l}\{\mathit{False}\} \\ \alpha_1 \\ \{\ (\mathit{Pre} \wedge \neg \mathit{Done} \wedge (pc_\eta = 3 \Rightarrow x = 1) \wedge pc_\eta \neq 4) \\ \ \ \vee (\neg \mathit{Pre}) \\ \ \ \vee (\mathit{Pre} \wedge \mathit{Done} \wedge \mathit{Post} \wedge pc_\mu = 2 \wedge pc_\eta = 4 \wedge x = 2)\}\end{array} \tag{5.13}$$

$$\begin{array}{l}\{(pc_\eta = 3 \Rightarrow x = 0) \wedge (pc_\eta = 4 \Rightarrow x = 1)\} \\ \alpha_1 \\ \{\ (\neg \mathit{Done} \wedge (pc_\eta = 3 \Rightarrow x = 1) \wedge pc_\eta \neq 4) \\ \ \ \vee (\mathit{Done} \wedge \mathit{Post} \wedge pc_\mu = 2 \wedge pc_\eta = 4 \wedge x = 2)\}\end{array} \tag{5.14}$$

$$\{\mathit{True}\}\ \alpha_1\ \{\mathit{True}\} \tag{5.15}$$

$$\{\mathit{False}\}\ \alpha_1\ \{\mathit{False}\} \tag{5.16}$$

Partial correctness logic triples (5.13), (5.15), and (5.16) are trivially valid; (5.14) follows from the Assignment Axiom and the Rule of Consequence.

Certain notational conventions will be useful in presenting property outlines. Terms of the form $q_i \sim \mathit{False}$ will be omitted from property assertions. Terms of the form $q_i \sim \mathit{True}$ will be abbreviated q_i. Two terms with the same predicate, $q_i \sim P$ and $q_j \sim P$, will be coalesced as $q_i,\ q_j \sim P$.

5.3. Proof Obligations and Property Outlines

The proof obligations of Chapter 3 are based on using correspondence invariants that link program states and property recognizer states. Therefore, to show that π satisfies m using a property outline PO for m and π, we must be able to extract from PO the correspondence invariant for each automaton state of m. Doing this turns out to be trivial, due to the way property assertions are defined. Each property assertion in a property outline contains a piece of every correspondence invariant. These pieces are labelled by the automaton state to which they correspond (by the "$q\sim$") and are exactly the part of the correspondence invariant that must hold whenever a program counter denotes the control point to which the property assertion is attached.

Suppose the property assertion attached to control point l in a valid property outline for π and m is of the form $q_0{\sim}P_0^l \mid \ldots \mid q_n{\sim}P_n^l$. Then, choose

$$C_i \equiv \bigwedge_{\phi \in \mathcal{P}_\pi} \bigwedge_{l \in \mathcal{L}_\phi} (pc_\phi = l \Rightarrow P_i^l), \tag{5.17}$$

where $\mathcal{P}$ is the set of processes of π, as the correspondence invariant for automaton state q_i. The advantage of making this choice is that it is not necessary to demonstrate Simulation Induction (4.7)—this obligation is subsumed by having established validity of the property outline.

To see that Simulation Induction (4.7) holds for a correspondence invariant extracted according to (5.17), consider an atomic action from a process ϕ,

$$\alpha\colon \langle \textbf{if } pc = l \rightarrow \bar{x}, l := \bar{e}, l' \textbf{ fi} \rangle$$

where $\bar{x}$ is a vector of the program variables changed by executing α and $\bar{e}$ is a vector of expressions whose values are assigned to those variables. Simulation Induction (4.7) requires that we prove, for each automaton state q_i,

$$\{C_i\} \;\; \alpha \;\; \{(T_{i0} \wedge C_0) \vee \ldots \vee (T_{in} \wedge C_n)\}$$

According to the Await Rule and Rule of Consequence, this is implied by

$$\begin{array}{l} \{C_i \wedge pc_\phi = l\} \\ \bar{x},\, pc_\phi := \bar{e},\, l' \\ \{pc_\phi = l' \wedge ((T_{i0} \wedge C_0) \vee \ldots \vee (T_{in} \wedge C_n))\}. \end{array} \tag{5.18}$$

The precondition and postcondition of (5.18) can be simplified because $(C_i \wedge pc_\phi = l) = (P_i^l \wedge C_i^{-\phi} \wedge pc_\phi = l)$, where

$$C_i^{-\phi} \equiv \bigwedge_{\substack{\psi \in \mathcal{P}_\pi \\ \psi \neq \phi}} \bigwedge_{l \in \mathcal{L}_\psi} (pc_\psi = l \Rightarrow P_i^l),$$

so we have

$$\begin{array}{l} \{P_i^l \wedge C_i^{-\phi} \wedge pc_\phi = l\} \\ \bar{x},\, pc_\phi := \bar{e},\, l' \\ \{pc_\phi = l' \ \wedge\ ((T_{i0} \wedge P_0^l \wedge C_0^{-\phi}) \vee \ldots \vee (T_{in} \wedge P_n^l \wedge C_n^{-\phi}))\}. \end{array} \tag{5.19}$$

Therefore, due to the Conjunction Rule and the fact that transition predicates are disjoint, it suffices to prove

$$\begin{array}{l} \{P_i^l\} \\ \bar{x},\, pc_\phi := \bar{e},\, l' \\ \{pc_\phi = l' \ \wedge\ ((T_{i0} \wedge P_0^{cp'}) \vee \ldots \vee (T_{in} \wedge P_n^{l'}))\}, \end{array} \tag{5.20}$$

and

$$\begin{array}{l} \{P_i^l \wedge C_i^{-\phi} \wedge pc_\phi = l\} \\ \bar{x},\, pc_\phi := \bar{e},\, l' \\ \{pc_\phi = l' \wedge ((T_{i0} \wedge C_0^{-\phi}) \vee \ldots \vee (T_{in} \wedge C_n^{-\phi}))\}. \end{array} \tag{5.21}$$

Notice, (5.20) is exactly what was proved in the Sequential Correctness step of establishing validity of PO. Now we prove (5.21). Using the Conjunction Rule and the definition of $C_i^{-\phi}$, it suffices to prove:

For all ψ: $\psi \neq \phi \ \wedge\ \psi \in \mathcal{P}_\pi$

  For all k: $k \in \mathcal{L}_\psi$:

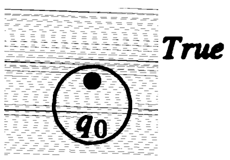
    $\{P_i^l \wedge P_i^k \wedge pc_\phi = l\}$ $\bar{x},\, pc_\phi := \bar{e},\, l'$ $\{pc_\phi = l' \ \wedge\ ((T_{i0} \wedge P_0^k) \vee \ldots \vee (T_{in} \wedge P_n^k))\}$

And, these partial correctness logic triples are exactly what was proved in the Interference Freedom step of establishing validity of PO.

Thus, given a valid property outline for m and π, in order to prove that π satisfies m, the correspondence invariants are first extracted from the property outline, and then only Simulation Basis (4.6), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10) must be proved—Simulation Induction (4.7) follows immediately from validity of the property outline. The property outline can also be used to help establish Simulation Basis and Finite Acceptance. If the property assertion associated with the first control point in each process follows from $Init_\pi$ then Simulation Basis holds. If each predicate corresponding to a non-finite-accepting state implies $\neg Blocked_\pi$ then Finite Acceptance holds.

5.4. Proof Outlines Revisited

Proof outlines for partial correctness logic are closely related to property outlines. Let PO_{pcl} be a valid proof outline for a concurrent program π. To obtain valid property outline PO_{prop} for π and the trivial property recognizer



replace each assertion P^l in PO_{pcl} with the property assertion $q_0 \sim P^l$. Validity of PO_{prop} follows from the partial correctness logic triples for Sequential Correctness and Interference Freedom used to establish validity of PO_{pcl}.

CHAPTER 6

Critical Section Examples

Solving the critical section problem involves devising protocols to ensure that two or more processes do not execute in *critical sections* at the same time. A good solution to the critical section problem must not only satisfy this Mutual Exclusion property, but should ensure—Guaranteed Service—that a process attempting to enter a critical section eventually does so assuming no process remains forever in its critical section. We might also require that a protocol satisfy First-come First-serve, which asserts that requests to enter a critical section are not served out-of-order.

In this section, we prove that a program *mxp* based on the two-process mutual exclusion protocol in [Peterson 81] satisfies Mutual Exclusion, Guaranteed Service, and First-come First-serve. The interested reader might wish to compare our proofs with the operational proofs for Mutual Exclusion and Guaranteed Service in [Peterson 81] and the temporal logic proofs for those properties in [Pnueli 86] and for First-come First-serve in [Pnueli & Manna 83].

A control-point annotation for the program we analyze is given in Figure 6.1. Assume that initially $active_\phi = active_\psi = False$, since neither ϕ nor ψ is initially executing in its critical section and that *turn* is initialized to ϕ or ψ. Thus,

$$Init_{mxp} \;\equiv\; pc_\phi=1 \wedge pc_\psi=8 \wedge \neg active_\phi \wedge \neg active_\psi \wedge (turn=\phi \vee turn=\psi)$$

```
mxp: cobegin
       φ: {1:} do True → {2:}
                            non critical section;
                            {3:}
                            active_φ := True;
                            {4:}
                            turn := ψ;
                            {5:}
                            ⟨if ¬ active_ψ ∨ turn=φ → skip fi⟩;
                            {6:}
                            critical section;
                            {7:}
                            active_φ := False
               od
     //
       ψ: {8:} do True → {9:}
                            non critical section;
                            {10:}
                            active_ψ := True;
                            {11:}
                            turn := φ;
                            {12:}
                            ⟨if ¬ active_φ ∨ turn=ψ → skip fi⟩;
                            {13:}
                            critical section;
                            {14:}
                            active_ψ := False;
               od
     coend
```

Figure 6.1. Peterson's Protocol

$$Blocked_{mxp} \equiv pc_{\phi}=5 \wedge pc_{\psi}=12 \wedge active_{\phi} \wedge active_{\psi} \wedge turn \neq \phi \wedge turn \neq \psi.$$

6.1. Mutual Exclusion

A property outline for process ϕ of *mxp* and property recognizer m_{mutex} (see Figure 3.2) appears in Figure 6.2; the property outline for ψ is symmetric. The only non-trivial part of showing that Figure 6.2 is valid property outline is showing Interference Freedom—in particular, showing that execution of ψ cannot invalidate the property assertion at control point 6, since this is the only property assertion in ϕ that mentions variables altered by execution of ψ. Execution of $active_\psi := True$ by ψ (at control point 10) invalidates $\neg active_\psi$ but establishes $pc_\psi=11$, and execution of $turn := \phi$ by ψ (at control point 11) invalidates

ϕ: {1: q_0}
    do *True* $\rightarrow$ {2: q_0}
        non critical section;
        {3: q_0}
        $active_\phi := True$;
        {4: $q_0 \sim active_\phi$}
        $turn := \psi$;
        {5: $q_0 \sim active_\phi$}
        $\langle$**if** $\neg active_\psi \vee turn=\phi \rightarrow$ **skip fi**$\rangle$;
        {6: $q_0 \sim active_\phi \wedge (turn=\phi \vee \neg active_\psi \vee pc_\psi=11)$}
        critical section;
        {7: q_0}
        $active_\phi := False$
    od

Figure 6.2. Mutual Exclusion Property Outline

$pc_\psi = 11$ but establishes $turn = \phi$. Thus, the property assertion is not interfered with.

To prove that π satisfies the property accepted by m_{mutex}, we must first define Cs_ϕ and Cs_ψ in terms of the program state:

$$
\begin{aligned}
Cs_\phi &= 6 \le pc_\phi \le 7 \\
Cs_\psi &= 13 \le pc_\psi \le 14
\end{aligned}
$$

(These definitions allow our proof to cover programs for which a critical section is not just a single atomic action as is assumed in *mxp*.) Next, we must prove Simulation Basis (4.6), Simulation Induction (4.7), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10). We can use (5.17) to extract from that property outline, a correspondence invariant for automaton state q_0:

$$
\begin{aligned}
C_0 = \; & pc_\phi = 4 \Rightarrow active_\phi \wedge pc_\phi = 5 \Rightarrow active_\phi \wedge \\
& pc_\phi = 6 \Rightarrow (active_\phi \wedge (turn = \phi \vee \neg active_\psi \vee pc_\psi = 11)) \wedge \\
& pc_\psi = 11 \Rightarrow active_\psi \wedge pc_\psi = 12 \Rightarrow active_\psi \wedge \\
& pc_\psi = 13 \Rightarrow (active_\psi \wedge (turn = \psi \vee \neg active_\phi \vee pc_\phi = 4))
\end{aligned}
$$

Simulation Basis requires that we prove

$$Init_{mxp} \Rightarrow (\neg (Cs_\phi \wedge Cs_\psi) \wedge C_0) \qquad (6.1)$$

Substituting and simplifying, we find that (6.1) is valid. Simulation Induction (4.7) follows because the property outline of Figure 6.2 is valid. Finite Acceptance, Knot Exit, and Knot Variance are vacuously satisfied because the single automaton state of m_{mutex} is both finite-accepting and infinite-accepting state.

6.2. Guaranteed Service

In Peterson's mutual exclusion protocol, process ϕ makes a request to enter its critical section by reaching control point 5; its request is serviced when it reaches control point 6. Thus, to use property recognizer m_{gs} (Figure 3.3) to show Guaranteed Service for ϕ, we choose transition predicates:

$$\begin{aligned} \mathit{Request}_\phi &\equiv pc_\phi = 5 \\ \mathit{Serve}_\phi &\equiv pc_\phi = 6 \end{aligned}$$

A valid property outline for the protocol and m_{gs} is given in Figure 6.3. Proving Sequential Correctness and Interference Freedom is a simple matter and is omitted here.

We extract correspondence invariants from the property outline using (5.17):

$$\begin{aligned} C_0 &\equiv pc_\phi \neq 5 \Rightarrow I_0 \wedge pc_\phi = 4 \Rightarrow \mathit{active}_\phi \wedge pc_\phi = 5 \Rightarrow \mathit{False} \\ C_1 &\equiv pc_\phi \neq 5 \Rightarrow \mathit{False} \wedge pc_\phi = 5 \Rightarrow (\mathit{active}_\phi \wedge I_0)) \wedge \\ &\quad\; pc_\psi \neq 12 \Rightarrow (I_1) \wedge pc_\psi = 12 \Rightarrow pc_\phi = 5 \end{aligned}$$

To prove Simulation Basis (4.6) we show that

$$\mathit{Init}_{mxp} \Rightarrow (\neg \mathit{Request}_\phi \wedge C_0) \vee (\mathit{Request}_\phi \wedge C_1)$$

is valid. This simplifies to

$$\begin{aligned} &pc_\phi = 1 \wedge pc_\psi = 8 \wedge \neg \mathit{active}_\phi \wedge \neg \mathit{active}_\psi \wedge (\mathit{turn} = \phi \vee \mathit{turn} = \psi) \\ &\Rightarrow (pc_\phi \neq 5 \wedge C_0) \vee (pc_\phi = 5 \wedge C_1) \end{aligned}$$

which is valid.

Simulation Induction follows from the validity of the proof outline of Figure 6.3.

mxp: **cobegin**
ϕ: $\{1:\ q_0 \sim I_0: turn = \phi \vee turn = \psi\}$
do *True* $\rightarrow \{2:\ q_0 \sim I_0\}$
non critical section;
$\{3:\ q_0 \sim I_0\}$
$active_\phi := True$;
$\{4:\ q_0 \sim active_\phi \wedge I_0\}$
$turn := \psi$;
$\{5:\ q_1 \sim active_\phi \wedge I_0\}$
$\langle$**if** $\neg\, active_\psi \vee turn = \phi \rightarrow$ **skip fi**$\rangle$;
$\{6:\ q_0 \sim I_0\}$
critical section;
$\{7:\ q_0 \sim I_0\}$
$active_\phi := False$
od
//
ψ: $\{8:\ q_0 \mid q_1 \sim I_1: pc_\phi = 5 \wedge turn = \psi\}$
do *True* $\rightarrow \{9:\ q_0 \mid q_1 \sim I_1\}$
non critical section;
$\{10:\ q_0 \mid q_1 \sim I_1\}$
$active_\psi := True$;
$\{11:\ q_0 \mid q_1 \sim I_1\}$
$turn := \phi$;
$\{12:\ q_0 \mid q_1 \sim pc_\phi = 5\}$
$\langle$**if** $\neg\, active_\phi \vee turn = \psi \rightarrow$ **skip fi**$\rangle$;
$\{13:\ q_0 \mid q_1 \sim I_1\}$
critical section;
$\{14:\ q_0 \mid q_1 \sim I_1\}$
$active_\psi := False$
od
coend

Figure 6.3. Guaranteed Service Property Outline

Next, we prove Finite Acceptance (4.8). There is only one non-finite-accepting state, q_1. Thus, Finite Acceptance (4.8) requires that we show that

$$C_1 \Rightarrow \neg Blocked_{mxp}$$

is valid. It is.

There is one reject knot $\kappa = \{q_1\}$ in m_{gs}. Choose the following as a variant function v_κ for the knot.

$$v_\kappa(q) \;=\; \begin{cases} 0, \text{ if } pc_\phi \neq 5 \\ 1, \text{ if } pc_\phi = 5 \wedge pc_\psi = 12 \wedge turn = \phi \\ 2 + ((11 - pc_\psi) \; mod \; 6), \text{ if } pc_\phi = 5 \wedge turn = \psi \end{cases}$$

To satisfy Knot Exit (4.9), we must prove

$$(v_\kappa(q_1) = 0) \Rightarrow Blocked_{mxp} \vee \neg C_1.$$

This follows because $v_\kappa(q_1) = 0 \Rightarrow pc_\phi \neq 5$ and $pc_\phi \neq 5 \Rightarrow \neg C_1$.

To satisfy Knot Variance (4.10), we must show that for every atomic action α:

$$\begin{array}{l} \{C_1 \wedge 0 < v_\kappa(q_1) = V\} \\ \alpha \\ \{(\neg Serve_\phi \wedge C_1) \Rightarrow v_\kappa(q_1) < V\} \end{array} \tag{6.2}$$

Since $v_\kappa(q_1) = 1 \Rightarrow (pc_\phi = 5 \wedge pc_\psi = 12 \wedge turn = \phi)$, and $(pc_\phi = 5 \wedge C_1) \Rightarrow active_\phi$, it suffices to prove

$$\begin{array}{l} \{active_\phi \wedge pc_\phi = 5 \wedge pc_\psi = 12 \wedge turn = \phi\} \\ \alpha \\ \{(\neg Serve_\phi \wedge C_1) \Rightarrow v_\kappa(q_1) < 1\} \end{array} \tag{6.3}$$

for each atomic action α. Only the atomic actions at control points 5 and 12 are potentially enabled in the precondition of (6.3), and from $active_{\phi} \wedge turn=\phi$, we conclude the one at 12 is not enabled. Since $\neg Serve_{\phi}$ is *False* after the atomic action at control point 5 executes, the postcondition of (6.3) is *true* and the triple is valid.

Next, we show that (6.2) is valid if $v_{\kappa}(q_1)=2$. From $v_{\kappa}(q_1)=2$, we infer $pc_{\phi}=5 \wedge turn=\psi$ and since $2+((11-11)\ mod\ 6=2$, $pc_{\psi}=11$. Thus, it suffices to show that

$$\begin{array}{l} \{pc_{\phi}=5 \ \wedge \ turn=\psi \ \wedge \ pc_{\psi}=11\} \\ \alpha \\ \{(\neg Serve_{\phi} \wedge C_1) \Rightarrow v_{\kappa}(q_1)<2\} \end{array} \tag{6.4}$$

is valid. Only the atomic action at control points 5 and 11 are enabled in the precondition of (6.4), so they are the only ones for which (6.4) is not trivially valid. Executing the atomic action at control point 5 makes $pc_{\phi}=6$, hence the postcondition of (6.4) is true and the triple valid; executing the atomic action at control point 11 makes $pc_{\psi}=12 \wedge turn=\phi$, which decreases $v_{\kappa}(q_1)$ to 1.

Finally, we show that (6.2) is valid if $v_{\kappa}(q_1)>2$. If $v_{\kappa}(q_1)>2$ then the atomic action at control point 5, as well as an action at 9, 10, 12, 13, or 14 must be enabled. As already argued, executing the atomic action at 5 decreases $v_{\kappa}(q_1)$ to 0. Executing the other enabled atomic action also decreases $v_{\kappa}(q_1)$, since by reaching the next control point, the value of $2+((11-pc_{\psi})\ mod\ 6)$ is decreased. Thus, *mxp* satisfies Guaranteed Service.

Notice that our proof relies on the fact that the critical section for ψ at 13 is a single atomic action. (If it is enabled then its execution will decrease v_κ.) If this critical section were more complicated then a different variant function would have to be found. This will be possible, provided ψ is guaranteed to exit its critical section after entering it.

6.3. First-come First-serve

A property recognizer for First-come First-serve is given in Figure 3.4. Transition predicates $Request_\phi$ and $Serve_\phi$ are as defined above for Guaranteed Service; the remaining two transition predicates used in m_{fcfs} are:

$$Request_\psi \equiv pc_\psi = 12$$
$$Serve_\psi \equiv pc_\psi = 13$$

The proof obligations Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10) are vacuously true because every automaton state in m_{fcfs} is both finite-accepting and infinite-accepting.

It remains to give correspondence invariants for the automaton states and to show Simulation Basis (4.6) and Simulation Induction (4.7). Informally, the correspondence invariants can be characterized as follows:

C_0: either ϕ does not have a pending request or ψ has a prior request pending.

C_1: ϕ has a pending request and ψ does not.

```
mxp: cobegin
    ϕ: {1: q0}
        do True → {2: q0}
                  non critical section;
                  {3: q0}
                  activeϕ := True;
                  {4: q0~activeϕ}
                  turn := ψ;
                  {5: q0~activeϕ ∧ Requestψ ∧ turn=ψ
                     | q1~activeϕ ∧ ¬Requestψ
                     | q2~activeϕ ∧ Requestψ ∧ turn=ϕ}
                  ⟨if ¬ activeψ ∨ turn=ϕ → skip fi⟩;
                  {6: q0}
                  critical section;
                  {7: q0}
                  activeϕ := False
        od
  //
    ψ: {8: q0, q1~I: Requestϕ⇒activeϕ}
        do True → {9: q0, q1~I}
                  non critical section;
                  {10: q0, q1~I}
                  activeψ := True;
                  {11: q0, q1~I}
                  turn := ϕ;
                  {12: q0~I ∧ Requestϕ⇒turn=ψ
                      | q2~activeϕ ∧ turn=ϕ}
                  ⟨if ¬ activeϕ ∨ turn=ψ → skip fi⟩;
                  {13: q0, q1~I}
                  critical section;
                  {14: q0, q1~I}
                  activeψ := False
        od
  coend
```

Figure 6.4. First-come First-serve Property Outline

C_2: both ϕ and ψ have pending requests and the one from ϕ was prior to the one from ψ.

More formal definitions for these invariants can be found in the property outline for *mxp* and m_{fcfs} in Figure 6.4. Simulation Basis is trivially valid. Simulation Induction follows from the validity of the proof outline (which is straightforward).

CHAPTER 7

Verifying Non-Deterministic Properties

The proof obligations of Chapter 4 concern properties specified by deterministic property recognizers. We now address the problem of proving that a program π satisfies a property N specified by a non-deterministic property recognizer m_N. Adapting the proof obligations of Chapter 4, results in sufficient, but not necessary, conditions to prove π satisfies N. Consequently, in this chapter, we show how to construct a deterministic property recognizer m_D that accepts every history of π accepted by m_N (but not necessarily every sequence in N). Necessary and sufficient proof obligations can be extracted from m_D. As an example of this technique, we show how fairness assumptions can be used to prove Guaranteed Service of a modification of the mutual exclusion protocol (3.3). Finally, we show that this technique can always be employed, provided π has a finite state space.

7.1. Refining Non-Deterministic Property Recognizers

Given m_N and π, if one were to prove (4.1) and (4.3) for each $q_0 \in Q_0$, (4.2), (4.4), Finite Acceptance (4.8), Knot Exit (4.9), and Knot Variance (4.10), then every run of m_N on any history of π would be accepting.[1] Thus, these proof

[1] In Chapter 4, (4.1) is merged with (4.3) to form Simulation Basis (4.6), and (4.2) is merged with (4.4) to form Simulation Induction (4.7). This merging is justified because

obligations are sufficient to prove π satisfies N. They are not, however, necessary. Recall that a non-deterministic property recognizer accepts its input if even only one of its runs is accepting. The proof obligations require that all runs be accepting. There are properties N and programs π such that π satisfies N but the proof obligations for m_N and π from Chapter 4 do not hold. In this chapter we shall see how to prove such properties.

To prove that π satisfies N, it suffices to be able to construct m_D, a *deterministic refinement* of m_N, such that π satisfies D and D is a subset of N. The construction of m_D involves repeatedly modifying m_N, using the techniques given below, so that it becomes progressively more deterministic. A modifications is *legitimate* if and only if the resulting property recognizer accepts only sequences accepted by the original. Satisfying the proof obligations for the deterministic refinement ensures that all histories of the program are accepted by the original property recognizer m_N as well as its deterministic refinement.

Legitimate modifications fall into two classes:

- those that result in an automaton that accepts the same sequences as the original; and
- those that result in an automaton that accepts fewer sequences than the original.

---

transition predicates for deterministic property recognizers are disjoint. These predicates need not be disjoint for non-deterministic property recognizers.

The second class of modification is necessary since some non-deterministic property recognizers do not have deterministic equivalents. By removing transitions from m_N, the resulting property recognizer is more deterministic and can accept no sequence that would not have been accepted by m_N. An example of such a modification is shown in Figure 7.1.

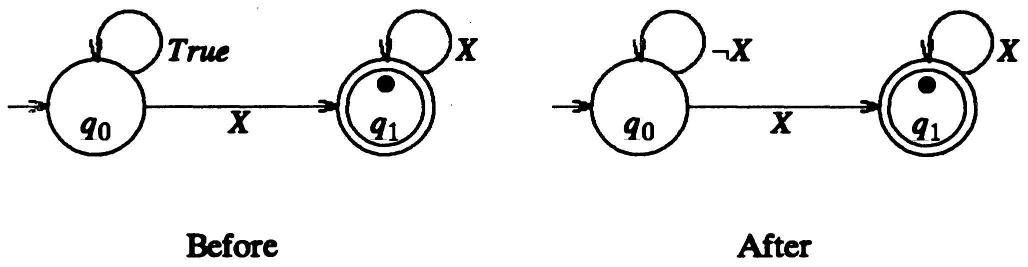
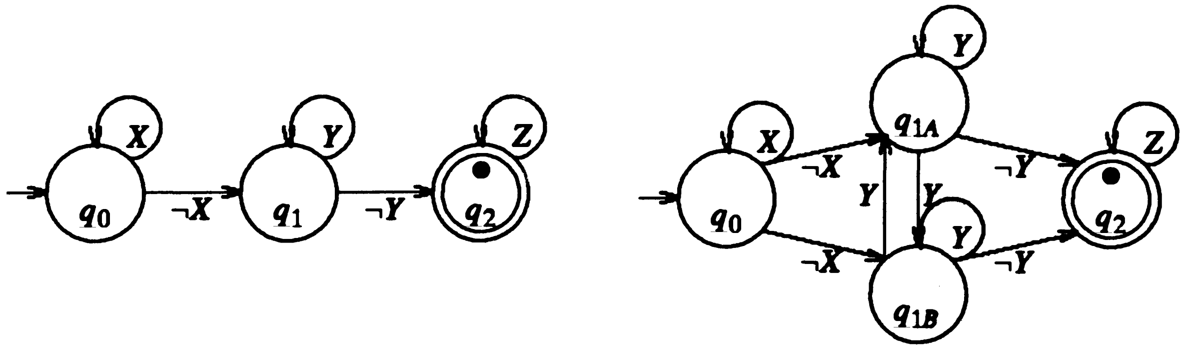


Figure 7.1. Pruning Transitions

Here, the transitions from q_0 to itself under symbols that satisfy X have been removed. Frequently, transition predicates are strengthened based on knowledge of the program state. This can only cause the resulting automaton to reject sequences accepted by the original. Similarly, removing a state from Q_0, Q_F or Q_∞ cannot cause the resulting automaton to accept sequences not accepted by the original.

An automaton state may serve many roles. By splitting such a state into several copies, we can separate these roles. This modification is illustrated in Figure 7.2.



Before

After

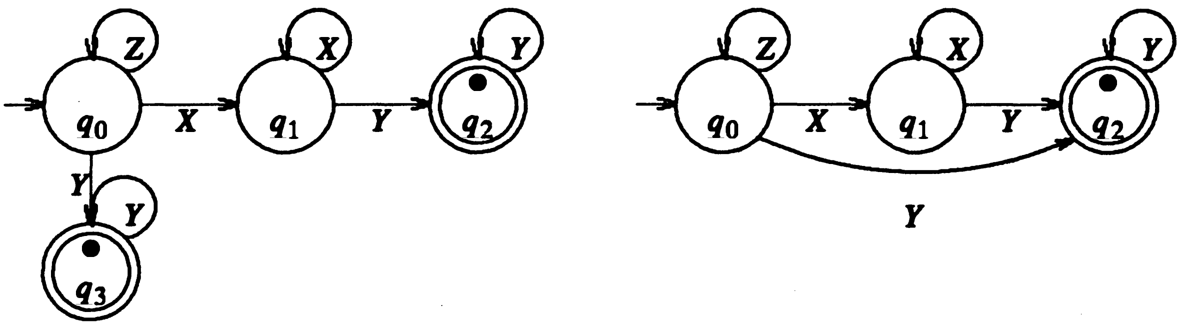
Figure 7.2. Splitting States

Here, state q_1 is split into two states. After a state has been split into several copies, transitions can be selectively pruned based on the role ascribed to each copy. Splitting states is a modification that always results in an equivalent automaton.

Other modifications are only legitimate in certain circumstances. A non-finite-accepting state q may be promoted to being finite accepting, if for every run that reaches q there is another run on the same input that reaches a finite-accepting state. A non-infinite-accepting state q may be promoted to being infinite-accepting, if for every run that contains q infinitely often there is a run (perhaps the same one) on the same input that contains some infinite-accepting state infinitely often. Two states may be combined if they are congruent. States q_1 and q_2 are *congruent* if and only if

- neither or both q_1 and q_2 are finite-accepting,
- neither or both q_1 and q_2 are infinite-accepting, and
- if there is a transition under s from q_1 to q_3 then there is a transition under s from q_2 to some state congruent to q_3.

An example of combining congruent states is illustrated in Figure 7.3.



Before After

Figure 7.3. Combining Congruent States

It is always legitimate to replace a property recognizer by an equivalent one. While not all Buchi automata have deterministic equivalents, there is a decision procedure [Landweber 69] to determine which do. Since this decision procedure is constructive, it may be used to construct a deterministic property recognizer equivalent to a given non-deterministic property recognizer in those cases for which a deterministic equivalent does exist.

7.2. Example: Using Fairness Assumptions

The mutual exclusion protocol of Figure 6.1 uses atomic actions at control points 5 and 12 to delay processes from entering their critical sections. Such a delay operation is usually implemented by using a *spin lock*—a loop that repeatedly tests some awaited condition until it is found to be *False*. In Figure 7.4, the atomic action at control point 12 of *mxp* has been replaced with a spin lock. (The labelling of control points is somewhat non-standard. This will facilitate developing a variant function later.)

The program of Figure 7.4 satisfies Guaranteed Service, provided an atomic action is guaranteed to execute if it is continuously enabled. The property that an atomic action that is continuously enabled is guaranteed to execute is called ***Weak Fairness*** [Francez 86]. To prove Guaranteed Service for the case where the delay in ψ is replaced by a spin lock, we construct a property recognizer for the property $WF \Rightarrow GS$, where WF denotes weak fairness and GS denotes Guaranteed Service.

A recognizer for Guaranteed Service is shown in Figure 3.3. The transition predicates for this machine are $Request_\phi$ $(pc_\phi = 5)$ and $Serve_\phi$ $(pc_\phi = 6)$ (and their negations). To keep subsequent figures from becoming overly cluttered, we shall abbreviate these predicates R and S respectively. With these abbreviations, m_{gs} the recognizer for Guaranteed Service, is

```
mxp2: cobegin
        φ: {1:} do True → {2:}
                          non critical section;
                          {3:}
                          active_φ := True;
                          {4:}
                          turn := ψ;
                          {5:}
                          ⟨if ¬ active_ψ ∨ turn=φ → skip fi⟩;
                          {6:}
                          critical section;
                          {7:}
                          active_φ := False
                od
      //
        ψ: {8:} do True → {9:}
                          non critical section;
                          {10:}
                          active_ψ := True;
                          {11:}
                          turn := φ;
                          {12:}
                          do ⟨active_φ ∨ turn=ψ⟩ →
                                                   {14:}
                                                   skip;
                                                   {13:}
                          od;
                          {15:}
                          critical section;
                          {16:}
                          active_ψ := False;
                od
      coend
```

Figure 7.4. A Second Version of Peterson's Protocol

CHAPTER 10

Conclusion

A new approach to proving temporal properties of concurrent programs was presented. The approach is based on specifying properties using automata, called property recognizers. Property recognizers are quite expressive—any linear-time temporal logic formula can be formulated as a property recognizer. Proof obligation for a deterministic property can be extracted directly from the automata for that property. The proof obligations are predicate logic formulas and partial correctness logic triples. Thus, temporal inference is not necessary for proving temporal properties; the same techniques that work for proving Total Correctness of sequential programs can be used to prove temporal properties of concurrent ones: when proving Total Correctness of a loop in a sequential program, a loop invariant and variant function must be devised and checked. When our method is used to prove that a temporal property holds for a concurrent program, correspondence invariants and variant functions must be devised and checked. Property outlines were proposed as a succinct way to represent a program and its correspondence invariants for a given property recognizer. The result is a Floyd-style method [Floyd 67] of proving deterministic properties of concurrent programs.

Our approach works less well for non-deterministic properties. While it is possible to use the refinement techniques of Chapter 7 to prove such properties, these techniques are much less elegant than extracting proof obligations directly from the property recognizer. Devising a direct method of proving non-deterministic properties remains an open problem.

In addition to their use in verification, property recognizers also provide insight into the nature of safety and liveness. The duality between "bad things" (or "good things") in a property and in its property recognizer first led us to formal definitions for safety and liveness. Using these definitions, we proved that every property is the conjunction of a safety property and a liveness property. We showed how to construct a safety recognizer for the safety aspect of a property and a liveness recognizer for the liveness aspect of a property. These constructions allowed us to explain why the proof of a safety property requires an invariance argument and to realize that the proof of a liveness property requires both variance arguments and an invariance argument.

References

[Alpern & Schneider 86]
Alpern, B, F.B. Schneider. Recognizing safety and liveness. In preparation.

[Arnold 83]
Arnold, A. Topological characterizations of infinite behaviors of transition systems. *Proc. 10th Col. Automata, Languages and Programming* Lecture Notes in Computer Science Vol. 154, Springer-Verlag, Berlin, 1983, 490-510.

[Barringer et al. 84]
Barringer, H, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. *Proc. 16th Sym. Theory of Computing*, Washington, D.C., April 1984, 51-63.

[Chandra et al. 80]
Chandra A.K., D.C. Kozen, and L.J. Stockmeyer. Alternation. *JACM 28*, 1 (Jan 1980) 114-133.

[Clarke & Emerson 81]
Clarke, E.M. and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logic of Programs* (D. Kozen ed.), Lecture Notes in Computer Science Vol. 131, Springer Verlag, Berlin, 1981, 52-71.

[Clarke et al. 83]
Clarke, E.M., E.A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. *Proc. 10th Sym. Principles of Programming Languages*, ACM, Austin, Jan. 1983, 117-126.

[Dijkstra 76]
Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[Eilenberg 74]
Eilenberg, S. *Automata, Languages and Machines, Vol A*. Academic Press, New York, 1974.

[Emerson & Lei 85]
Emerson, E.A., and C-L. Lei. Modalities for model checking: Branching time strikes back. *Proc. 12th Sym. Principles of Programming Languages*, ACM, New Orleans, Jan. 1985, 84-96.

[Francez 86]
Francez, N. *Fairness*. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1986, To appear.

[Gerth 84]
Gerth, R. Transition logic. *Proc. 16th Sym. Theory of Computing*, Washington, D.C., April 1984, 39-50.

[Gries 81]
Gries, D. *The Science of Programming*. Springer-Verlag, Berlin, 1981.

[Hoare 69]
Hoare, C.A.R. An axiomatic basis for computer programming. *CACM 12*, 10 (Oct. 1969), 576-580.

[Hoare 78]
Hoare, C.A.R. Communicating Sequential Processes. *CACM 21*, 8 (Aug. 1978), 666-677.

[Hopcroft & Ullman 79]
Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.

[Jones 83]
Jones, C.B. Specification and design of (parallel) programs. *Information Processing '83*, (R.E.A. Mason, ed.) North-Holland Publishing Company, Amsterdam, 1983, 321-332.

[Lamport 77]
Lamport, L. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering SE-3*, 2 (March 1977), 125-143.

[Lamport 80]
Lamport, L. The 'Hoare logic' of concurrent programs. *Acta Informatica 14*, 1 (1980), 21-37.

[Lamport 83a]
Lamport, L. What good is temporal logic? *Information Processing '83*, (R.E.A. Mason, ed.) North-Holland Publishing Company, Amsterdam, 1983, 657-668.

[Lamport 83b]
Lamport, L. Specifying concurrent program modules. *ACM TOPLAS 6*, 2 (April 1983), 190-222.

[Lamport 85]
Lamport, L. Logical Foundation. In *Distributed Systems—Methods and Tools for Specification*, Lecture Notes in Computer Science, Vol 190. M. Paul and H.J. Siegert, eds. (1985), Springer-Verlag, Berlin.

[Lamport & Owicki 81]
Lamport, L. and S.S. Owicki. Program logics and program verification. *Logic of Programs* (D. Kozen ed.), Lecture Notes in Computer Science Vol. 131, Springer Verlag, Berlin, 1981, 197-199.

[Lamport & Schneider 84]
Lamport, L. and F.B. Schneider. The 'Hoare Logic' of CSP, and all that. *ACM TOPLAS 6*, 2 (April 1984), 281-296.

[Landweber 69]
Landweber, L.H. Decision problems for ω-automata. *Math. System Theor. 3* (1969), 376-384.

[Lichtenstein & Pnueli 85]
Lichtenstein, O. and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. *Proc. 12th Sym. Principles of Programming Languages*, ACM, New Orleans, Jan. 1985, 97-107.

[Manna & Pnueli 81a]
Manna, Z. and A. Pnueli. Verification of concurrent programs: The temporal framework. *The Correctness Problem in Computer Science* (R.S. Boyer and J.S. Moore, eds.), International Lecture Series in Computer Science, Academic Press, London, 1981, 141-154.

[Manna & Pnueli 81b]
Manna, Z. and A. Pnueli. Verification of concurrent programs: Temporal proof principles. *Logic of Programs* (D. Kozen ed.), Lecture Notes in Computer Science, Vol. 131, Springer-Verlag, Berlin, 1981, 200-252.

[Manna & Pnueli 83]
Manna, Z, and A. Pnueli. How to cook a temporal proof system for your pet language. *Proc. 10th Sym. Principles of Programming Languages*, ACM, Austin, Jan. 1983, 141-154.

[Manna & Pnueli 84]
Manna, Z. and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming 4* (1984), 257-289.

[Manna & Wolper 84]
Manna, Z and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM TOPLAS 6*, 1 (Jan. 1984), 68-93.

[Misra et al. 82]
Misra, J., K.M. Chandy, and T. Smith. Proving safety and liveness of communicating processes with examples. *Proc. 1st Sym. Principles of Distributed Computing*, ACM, Ottawa, Aug. 1982, 157-164.

[Nguyen et al. 85]
Nguyen, V., D. Gries, S. Owicki. A model and temporal proof system for networks of processes. *Proc. 12th Sym. Principles of Programming Languages*, ACM, New Orleans, Jan. 1985, 121-131.

[Owicki & Gries 76]
Owicki, S.S. and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica 6* (1976), 319-340.

[Owicki & Lamport 82]
Owicki, S.S. and L. Lamport. Proving liveness properties of concurrent programs. *ACM TOPLAS 4*, 3 (July 1982), 455-496.

[Peterson 81]
Peterson, G.L. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3 (June 1981), 115-116.

[Plotkin 83]
Plotkin, G. Private communication.

[Plotkin 84]
Plotkin, G. Private communication.

[Pnueli 77]
Pnueli, A. The temporal logic of programs. *Proc. 18th Sym. Foundations of Computer Science*, IEEE, Providence, Nov. 1977, 46-57.

[Pnueli 86]
Pnueli, A. In transition from global to modular temporal reasoning about programs. In *Current Trends in Concurrency*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1986, To appear.

[Pnueli & Manna 83]
Pnueli, A. and Z. Manna. Proving precedence properties: The temporal way. *Proc. Col. Automata, Languages and Programming 10.* Lecture Notes in Computer Science Vol. 154, Springer-Verlag, Berlin, 1983, 490-510.

[Schneider & Andrews 86]
Schneider, F.B and G. Andrews. Concepts for concurrent programming. In *Current Trends in Concurrency*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1986, To appear.

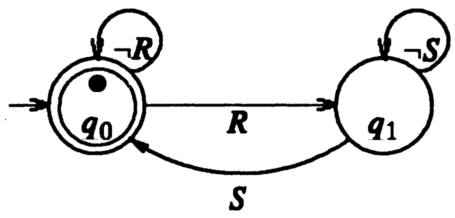
[Sistla 85]
Sistla, A.P. On characterization of safety and liveness properties in temporal logic. *Proc. 4th Sym. Principles of Distributed Computing*, ACM, Minaki, Aug. 1985, 39-48.

[Stark 84]
Stark, E.W. Foundations of a theory of specification for distributed systems. Ph.D. Thesis, M.I.T. Laboratory for Computer Science. MIT/LCS/TR-342, Aug. 1984.

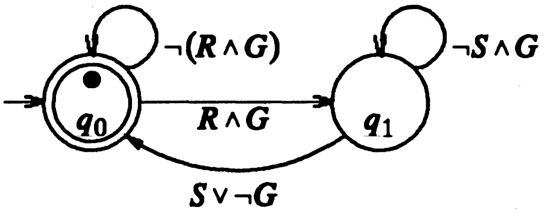
[Vardi & Wolper 85]
Vardi, M.Y. and P. Wolper. Applications of temporal logic: An automata-theoretic perspective. In preparation.

[Wolper 83]
Wolper, P. Temporal logic can be more expressive. *Information and Control* *56*, 1-2 (1983), 72-99.

[Wolper 84]
Wolper, P. The tableau method for temporal logic: An overview. Unpublished manuscript.

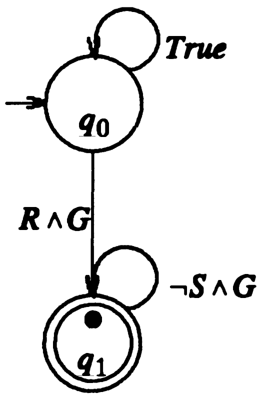


Strictly speaking, *WF* is the conjunction of a collection of Weak Fairness properties—one for each atomic action. To prove Guaranteed Service for process ϕ, we shall need only the Weak Fairness property for the atomic action at control point 5. The property recognizer m_{wf} for *WF* is



where G is the guard at control point 5: $\neg active_{\psi} \vee turn = \phi$.

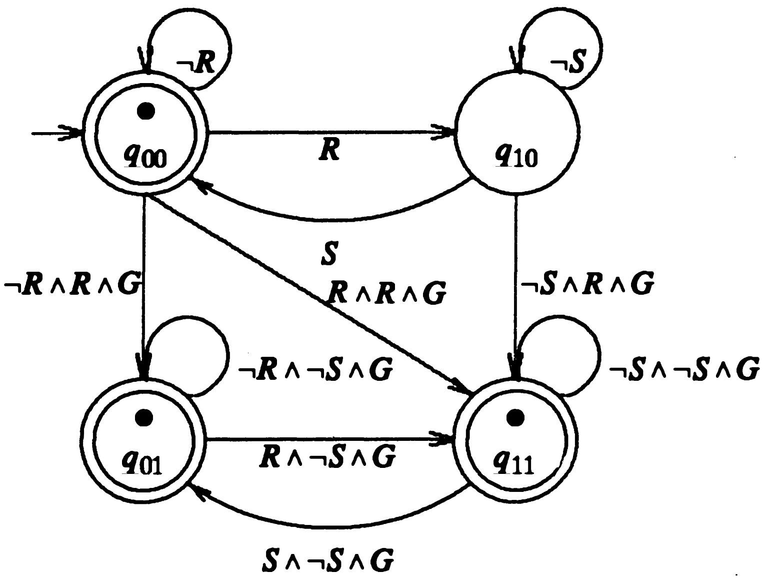
To construct a property recognizer for $WF \Rightarrow GS$ we need to take the disjunction of m_{gs} a recognizer for the negation of *WF*. This recognizer m_{nwf} is



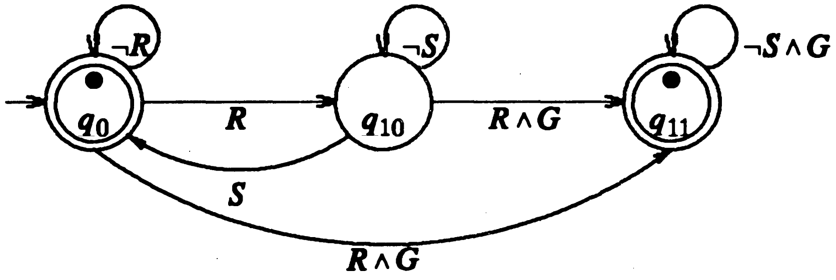
To see that m_{nwf} accepts exactly those sequences rejected by m_{wf}, notice that m_{wf}

accepts all sequences except those that after some finite prefix always satisfy $\neg S \wedge G$. Notice, also, that m_{nwf} accepts just such sequences.

The property recognizer for Guaranteed Service assuming Weak Fairness, $WF \Rightarrow GS$ is formed from the cross product of the states of m_{gs} and the states of m_{nwf}:

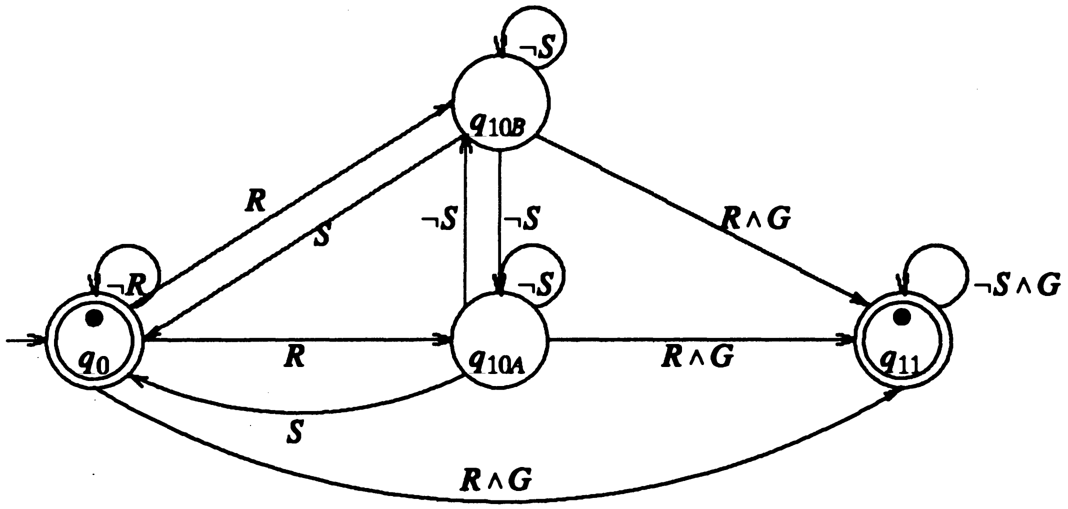


This automaton can be simplified. Since the predicates on all arcs into q_{01} are *False*, both the arcs and this state can be eliminated. Since $R \Rightarrow \neg S$, the predicate on the arc from q_{10} to q_{11} is equivalent to $R \wedge G$. The simplified recognizer m_1 for $WF \Rightarrow GS$ is

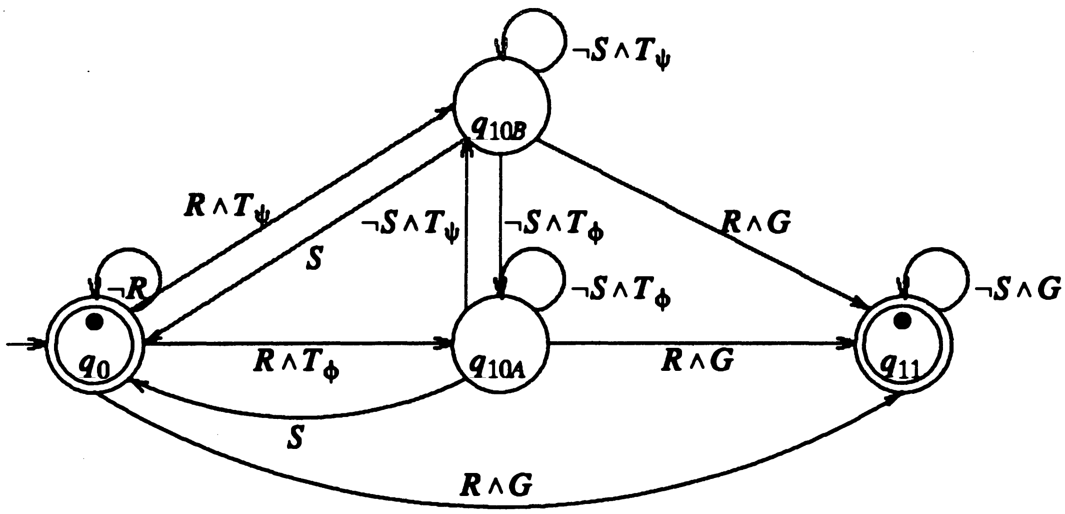


This is a non-deterministic property recognizer because from q_{00} (or q_{10}) under a program state that satisfies $R \wedge G$ (and $\neg S$) m_1 could enter either q_{10} or q_{11}. We shall refine it to a deterministic property recognizer that accepts every history of *mxp*2.

If q_{10} and q_{11} could be merged into a single accepting state then the non-determinism would be resolved. Unfortunately, the resulting automaton would not be equivalent to m_1. We shall split q_{10} so as to obtain a state that can be merged with q_{11}. The resulting automaton m_2 is

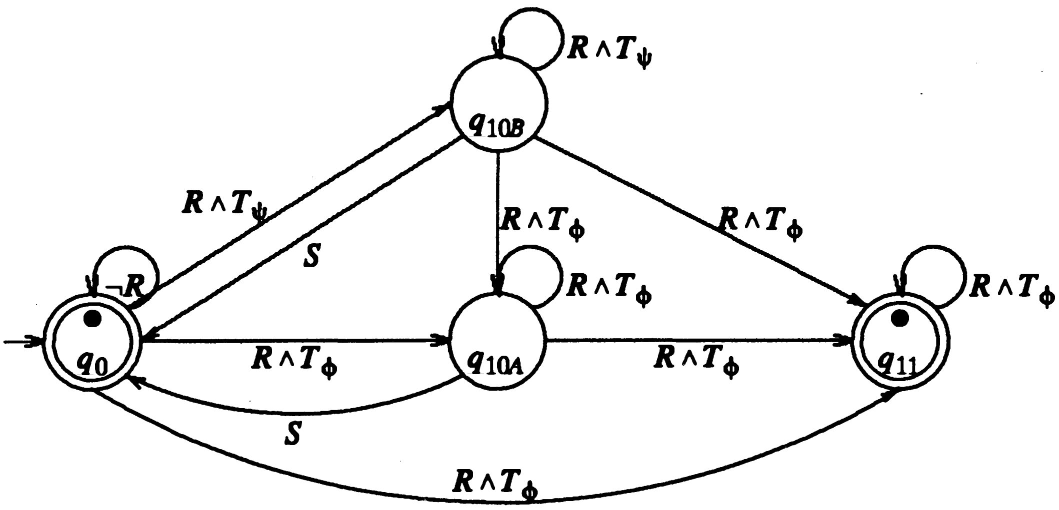


Splitting q_{10} introduces a great deal of non-determinacy. We can use the program variable *turn* to resolve some of this. We can prune the transitions of m_2 such that the resulting automaton m_3 enters q_{10A} if $turn = \phi$ and q_{10B} if $turn = \psi$:

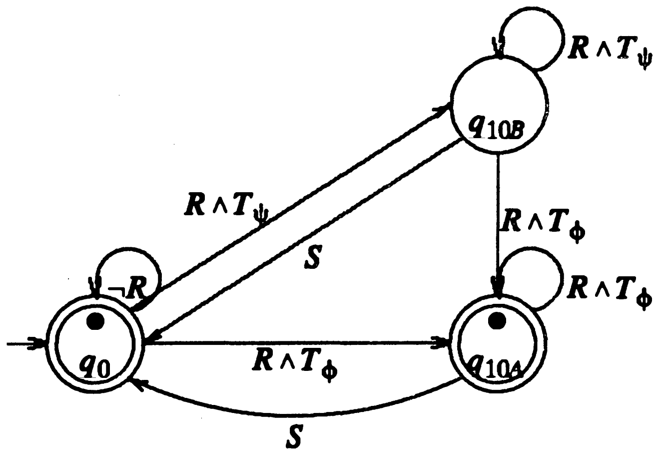


where T_ϕ denotes $turn=\phi$ and T_ψ denotes $turn=\psi$.

Next, note that the transition from q_{10A} to q_{10B} is superfluous because *turn* never changes from ϕ to ψ when ψ is "spinning" (see Figure 7.4) and $turn=\phi$ when m_3 is in q_{10A}. Thus, this transition can be removed. Now, notice if m_3 enters q_{11} then it can never get back to its other states, but whenever it is possible to enter q_{11} from one of these other states, it is also possible to enter q_{10A} or q_{10B}. Thus, q_{11} need only be entered if the predicate on the transition from it to itself will always hold. That is, if the atomic action a control point 5 is always enabled but never executes. If this happens ψ will eventually "spin" forever. For ψ to "spin," $active_\psi$ must hold, so the predicate G of this transition can be strengthened to T_ϕ. Finally, note that $\neg S$ ($pc_\phi \neq 6$) can be strengthened to R ($pc_\phi = 5$) everywhere. The resulting property recognizer m_4 is



Notice that whenever a run of m_4 is in q_{10A}, there is another run of m_4 that is in q_{11}, and *vice versa*. Since q_{11} is a finite-accepting state, q_{10A} can be also. If a run contains q_{10A} infinitely often, then either it contains the infinite-accepting q_0 infinitely often or eventually the run never leaves q_{10A}. In the latter case, there is another run that never leaves the infinite-accepting state q_{11}. In either case, the input is accepted. Thus, q_{10A} can be promoted to an infinite-accepting state. Further, q_{11} is now superfluous and can be deleted. The resulting property recognizer m_5 is



Since m_5 is equivalent to m_4, proving that every history of $mxp2$ is accepted by m_5 suffices to prove that mxp satisfies $WF \Rightarrow GS$.

Since m_5 is deterministic, we can now use the technique of Chapter 4. Take

$$
\begin{aligned}
C_0 &\equiv pc_\phi \neq 5, \\
C_{1B} &\equiv pc_\phi = 5 \wedge turn = \psi, \\
C_{1A} &\equiv pc_\phi = 5 \wedge turn = \phi, \text{ and} \\
v &\equiv (12 - pc_\psi) \, mod \, 8.
\end{aligned}
$$

We leave it to the reader to check that these correspondence invariants and this variant function meet the proof obligations for m_5. Since m_5 is a deterministic refinement of the property recognizer for $WF \Rightarrow GS$, that will complete the proof that the program satisfies Guaranteed Service assuming Weak Fairness.

7.3. Soundness and Relative Completeness

The Soundness Theorem for non-deterministic property recognizers below shows that constructing a deterministic refinement suffices for proving the non-deterministic property of interest. The Soundness Theorem for deterministic property recognizers then allows us to conclude that satisfying the proof obligations extracted from this deterministic refinement is sufficient. Completeness for non-deterministic property recognizers involves showing that if a program π satisfies a property specified by a non-deterministic property recognizer m_N, then it is possible to construct a deterministic refinement of m_N that π satisfies, provided that the state space for π is finite.

Soundness Theorem 7.1: If a program π satisfies a property D with a property recognizer m_D, and if m_D is a deterministic refinement of the recognizer m_N of a property N then π satisfies N.

Proof: Assume π satisfies D and m_D is a deterministic refinement of m_N. Let σ be a history of π. Since π satisfies D, σ is accepted by m_D. By definition of deterministic refinement, m_N must also accept σ. Thus, π satisfies N. □

Relative Completeness Theorem 7.2: If program π has a finite state space and satisfies some property N with property recognizer m_N, then there exists a deterministic refinement m_D of m_N that π satisfies.

Proof: First, we construct a deterministic property recognizer m_π that accepts H_π, the histories of π. Define m_π to be

$$\langle S_\pi,\ S_\pi \cup \{start\},\ \{start\},\ S_\pi,\ Blocked_\pi,\ \delta_\pi \rangle,$$

where S_π is the set of program states of π and

$\delta_\pi(start,\ s) = s$ iff $Init_\pi(s)$, and
$\delta_\pi(s,\ s') = s'$ iff there is an $\alpha \in A_\pi$ enabled in s that terminates in s'.

Clearly, m_π accepts exactly the histories of π.

We can use m_π to refine $m_N = \langle S_\pi, Q, Q_0, Q_\infty, Q_F, \delta_N \rangle$. Let $m_{N\times\pi}$ be

$\langle S_\pi, Q\times(S_\pi \cup \{start\}), Q_0\times\{start\}, Q_\infty\times S_\pi, Q_F\times Blocked_\pi, \delta_{N\times\pi}\rangle$,

where $\langle q', s'\rangle \in \delta_{N\times\pi}(\langle q, s\rangle, s')$ iff $q' \in \delta_N(q, s')$ and $\delta_\pi(s, s')=s'$. Note that $m_{N\times\pi}$ can be obtained by splitting each state of m_N, into one copy for each state of m_π and then pruning away extra transitions.

$m_{N\times\pi}$ accepts exactly those sequences that are histories of π (hence, accepted by m_π) and accepted by m_N. Since π satisfies N, every history of π is accepted by m_N. Thus, $m_{N\times\pi}$ recognizes the same set of sequences as m_π. We can replace $m_{N\times\pi}$ with m_π. Since m_π is deterministic and accepts every history of π, we have shown that a deterministic refinement for m_N that accepts π exists, if π satisfies N. □

CHAPTER 8

Metaproperties: Safety and Liveness

Two classes of properties are described in [Lamport 77] and different techniques are used to prove properties in each class. A proof that a program satisfies a *safety property* rests on an invariance argument; The proof that a program satisfies a *liveness property* depends primarily on a variance (well-foundness) argument. In this chapter, we give formal definitions for safety and liveness of properties of infinite sequences, and show that every such property is the intersection of a safety property and a liveness property. Using these formal definitions we demonstrate that the proof of a deterministic safety property can never require a variance argument, while the proof of a deterministic liveness property will, in general, require an invariance argument. Finally, we extend our formal definitions of safety and liveness to cover properties of both finite and infinite sequences. Until the end of the chapter, we restrict our attention to properties of infinite sequences.

8.1. Safety Properties

Informally, a safety property stipulates that no "bad thing" happen during execution [Lamport 77]. Examples of safety properties include Mutual Exclusion, Partial Correctness, and First-come First-serve. In Mutual Exclusion, the proscribed "bad thing" is two processes executing in critical sections at the same time. In Partial Correctness, it is terminating in a state not satisfying the postcondition when execution is started in a state that satisfies the precondition. In First-come First-serve, it is servicing a request before a prior request is serviced.

If a "bad thing" happens in a history, then it must do so in some finite prefix of that history. It is irremediable when it happens; no further execution of the program can repair the damage. A finite sequence is *live* for a property if it can be extended to satisfy the property. More formally, a finite sequence σ is live for a property P if and only if

Live: $(\exists \tau\colon \tau \in S^{\omega}\colon \sigma\tau \models P)$. (8.1)

If a finite sequence is live then no "bad thing" happens in it.

Safety properties require (only) that no "bad thing" happen. So, if P is a safety property then $\sigma \models P$ if and only if every prefix of σ is live for P. We take this to be the defining characteristic of safety. A property P is a *safety property* if and only if

Safety: $(\forall \sigma\colon \sigma \in S^{\omega}\colon \sigma \models P \Leftrightarrow (\forall i\colon 0 \le i\colon (\exists \tau\colon \tau \in S^{\omega}\colon \sigma[..i]\tau \models P)))$. (8.2)

One direction of the biconditional is trivial; if $\sigma \models P$ then every prefix of σ is automatically live to P.

There are two things to notice about this definition. First, it requires that a "bad thing" be *discrete*—if it happens during an execution then there is an identifiable point at which it happens. Otherwise, no restriction is placed on what a "bad thing" can be. Second, a safety property can never require that something happen sometime, as opposed to always. Thus, the definition merely stipulates that a safety property unconditionally prohibits such a "bad thing" from occurring and if it does occur there is an identifiable point at which this can be recognized.

A slightly different formalization of safety is given in [Lamport 85]. Here, P is a safety property if and only if

$$(\forall \sigma:\ \sigma \in S^{\omega}:\ \sigma \models P \Leftrightarrow (\forall i:\ 0 \leq i:\ \sigma[..i]\sigma[i]^{\omega} \models P)). \qquad (8.3)$$

A property P that meets (8.3) is satisfied by a history σ if and only if every prefix of σ—extended to an infinite sequence by repeating its last state—also satisfies P. (Extension of the prefix $\sigma[..i]$ to an infinite sequence is necessary because we are only considering properties of infinite sequences.) For (8.3), this extension must be made by repeating the final state; for Safety (8.2), any extension suffices.

For some properties, extending a finite sequence by repeating the final state causes problems. Consider the property

$$CP:\ \Box((clock=N) \Rightarrow \bigcirc(clock=N+1)).$$

This property stipulates that the variable *clock* is incremented whenever an instruction is executed. It is a safety property; the "bad thing" is *clock* having the same value in two successive states. However, CP does not meet (8.3), because for no history σ—even if $\sigma \models CP$—will the value of *clock* change after the i^{th} state of $\sigma[..i]\,\sigma[i]^{\omega}$.

Let us consider the relationship between definitions (8.2) and (8.3) in more detail. The difficulty encountered with *CP*—that replicating the final state of a prefix *causes* the resulting sequence not to be accepted—arises only for properties that are not invariant under stuttering. A property is ***invariant under stuttering*** if and only if whenever a history satisfies the property, the history with every state repeated zero or more times also satisfies the property, and vice versa. More formally, a property P is invariant under stuttering if and only if

Invariance Under Stuttering:

$$(\forall f\colon f \in \mathbf{N} \to \mathbf{N}\colon\ \sigma \models P \Leftrightarrow \sigma[0]^{f(0)+1} \ldots \sigma[i]^{f(i)+1} \ldots \models P). \tag{8.4}$$

Properties that are invariant under stuttering are well suited for hierarchical specification and verification [Lamport 83a], because, by permitting states to be repeated, meaningful statements can be made about the system at various levels of abstraction. For example, execution of a higher-level operation that is implemented by a sequence of lower-level operations can be viewed as a sequence of repeated, identical, higher-level states where there is one state for every lower-level instruction executed but the last, which produces a new higher-level state.

The relationship between Safety (8.2) and (8.3) is given by the following theorems.

Theorem 8.1: (8.3) implies Safety (8.2).

Proof: Assuming P meets (8.3), we must show:

$$\sigma \models P \Leftrightarrow (\forall i\colon 0 \le i\colon (\exists \tau\colon \tau \in S^{\omega}\colon \sigma[..i]\tau \models P)).$$

$$
\begin{aligned}
& (\forall i\colon 0 \le i\colon (\exists \tau\colon \tau \in S^\omega\colon \sigma[..i]\tau \models P)) \\
\Leftrightarrow\ & (\forall i\colon 0 \le i\colon (\exists \tau\colon \tau \in S^\omega\colon \sigma[..i]\sigma[i]^\omega \models P)) \\
& \qquad \sigma[..i]\tau \models P \text{ and } P \text{ meets (8.3)} \\
\Leftrightarrow\ & (\forall i\colon 0 \le i\colon \sigma[..i]\sigma[i]^\omega \models P) \quad \text{by predicate logic} \\
\Leftrightarrow\ & \sigma \models P \quad \text{since } P \text{ meets (8.3).} \ \square
\end{aligned}
$$

Theorem 8.2: Safety (8.2) and Invariant Under Stuttering (8.4) implies (8.3).

Proof: Assuming P is a safety property invariant under stuttering, we must show:

(1) $\sigma \models P \Rightarrow (\forall i\colon 0 \le i\colon \sigma[..i]\sigma[i]^\omega \models P)$
(2) $(\forall i\colon 0 \le i\colon \sigma[..i]\sigma[i]^\omega \models P) \Rightarrow \sigma \models P$

First, we prove (1):

$$
\begin{aligned}
& \sigma \models P \\
\Leftrightarrow\ & (\forall i\colon 0 \le i\colon (\exists \tau\colon \tau \in S^\omega\colon \sigma[..i]\tau \models P)) \quad \text{since } P \text{ is a safety property} \\
\Leftrightarrow\ & (\forall i\colon 0 \le i\colon (\exists \tau\colon \tau \in S^\omega\colon (\forall n\colon 0 \le n\colon \sigma[..i]\sigma[i]^n\tau \models P))) \\
& \qquad \text{since } P \text{ is invariant under stuttering.} \\
\Rightarrow\ & (\forall i\colon 0 \le i\colon (\forall n\colon 0 \le n\colon (\exists \tau\colon \tau \in S^\omega\colon \sigma[..i]\sigma[i]^n\tau \models P))) \\
& \qquad \text{by predicate logic} \\
\Leftrightarrow\ & (\forall i\colon 0 \le i\colon (\forall n\colon 0 \le n\colon (\exists \tau\colon \tau \in S^\omega\colon (\sigma[..i]\sigma[i]^\omega)[..i+n]\tau \models P))) \\
& \qquad \text{since } \sigma[..i]\sigma[i]^n = (\sigma[..i]\sigma[i]^\omega)[..i+n] \\
\Leftrightarrow\ & (\forall i\colon 0 \le i\colon (\forall j\colon 0 \le j\colon (\exists \tau\colon \tau \in S^\omega\colon (\sigma[..i]\sigma[i]^\omega)[..j]\tau \models P))) \\
& \qquad \text{by predicate logic} \\
\Leftrightarrow\ & (\forall i\colon 0 \le i\colon \sigma[..i]\sigma[i]^\omega \models P) \quad \text{since } P \text{ is a safety property.}
\end{aligned}
$$

Next, we prove (2):

$$
\begin{aligned}
& (\forall i\colon 0 \le i\colon \sigma[..i]\sigma[i]^\omega \models P) \\
\Rightarrow\ & (\forall i\colon 0 \le i\colon (\exists \tau\colon \tau \in S^\omega\colon \sigma[..i]\tau \models P)) \quad \text{use } \tau = \sigma[i]^\omega \\
\Leftrightarrow\ & \sigma \models P \quad \text{since } P \text{ is a safety property.} \ \square
\end{aligned}
$$

For properties that are invariant under stuttering, the two proposed definitions of safety coincide. For properties like CP that are not, our formalization

(8.2) captures the intuition of safety better than the formalization of Lamport (8.3).

8.2. Liveness Properties

Informally, liveness properties stipulate that "good things" happen eventually during execution [Lamport 77]. Examples of liveness properties include Starvation Freedom, Termination, and Guaranteed Service. In Starvation Freedom, the "good thing" is making progress. In Termination, the "good thing" is being blocked. Finally, in Guaranteed Service, the "good thing" is receiving service.

The thing to observe about a liveness property is that no partial history is irremediable: it always remains possible for the required "good thing" to occur in the future.[1] This is taken to be the defining characteristic of liveness since if some partial history were irremediable then it (this partial history) would be a "bad thing"; liveness properties cannot proscribe a "bad thing", they can only prescribe a "good thing". Thus, a *liveness property* is one for which every finite sequence is live. More formally, P is a liveness property if and only if

Liveness: $$(\forall \rho\colon\ \rho \in S^{*}\colon\ (\exists \tau\colon\ \tau \in S^{\omega}\colon\ \rho\tau \models P)). \qquad (8.5)$$

Again, there are two things to notice about this definition. First, while safety properties prohibit a single occurrence of a "bad thing", liveness properties may require that infinitely "good things" occur. The "good thing" for Starvation Freedom—progress—must occur infinitely often to satisfy the property. Again,

[1]"While there's life there's hope." — *Cicero*.

no restriction is placed on what a "good thing" can be. Second, a liveness property cannot require that something *always* happen, only that it eventually happen.

No definition of liveness can be more permissive than the one given above. Suppose, by way of contradiction, that P is a liveness property that does not satisfy this definition. There must be some finite sequence ρ, such that

$$(\forall \tau\colon\ \tau \in S^{\omega}\colon\ \rho\tau \not\models P).$$

Clearly, ρ is a "bad thing" proscribed by P. Thus P is in part a safety property and not a liveness property, as was assumed.

Obviously, definitions for liveness more restrictive than ours are possible. One investigated candidate is

$$\textbf{Uniform Liveness:}\qquad (\exists \tau\colon\ \tau \in S^{\omega}\colon\ (\forall \rho\colon\ \rho \in S^{*}\colon\ \rho\tau \models P)). \tag{8.6}$$

P is a uniform-liveness property if and only if there is a single sequence (τ) that can be appended to every finite sequence (ρ) so that the resulting history is in P. Another definition is proposed in [Sistla 85].

$$\begin{aligned}\textbf{Absolute Liveness:}\quad & (\exists \gamma\colon\ \gamma \in S^{\omega}\colon\ \gamma \models P)\\ & \wedge\ (\forall \tau\colon\ \tau \in S^{\omega}\colon\ \tau \models P \Rightarrow (\forall \rho\colon\ \rho \in S^{*}\colon\ \rho\tau \models P)).\end{aligned} \tag{8.7}$$

P is an absolute-liveness property if and only if it is non-empty and any history (τ) in P can be appended to any finite sequence (ρ) to obtain a history in P.

It is instructive to contrast these formal definitions. L is a liveness property if any finite sequence ρ can be extended by some sequence τ so that $\rho\tau$ is in L—

the choice of τ may depend of ρ. U is a uniform-liveness property if there is a single sequence τ that extends any finite sequence ρ such that $\rho\tau$ is in U. And, A is an absolute-liveness property if it is non-empty and any history τ in A can be used to extend all finite sequences ρ. Any absolute-liveness property is a uniform-liveness property and any uniform-liveness property is a liveness property.

While absolute liveness (8.7) characterizes an interesting class of properties, it does not seem to include all properties that should be considered liveness. In particular, Guaranteed Service is not an absolute-liveness property. To see this, consider a history τ in which neither requests nor service occurs. This sequence satisfies Guaranteed Service. However, appending τ to a finite sequence consisting of a single request yields a sequence that does not satisfy the property.

Uniform liveness (8.6) does not seem to capture the intuition for liveness, either. An example of a liveness property that is not a uniform-liveness property is characterized by

$$(A \Rightarrow \Box B) \wedge (\neg A \Rightarrow \Box \neg B).$$

This is a liveness property, because it requires some "good thing" (either $\Box B$ or $\Box \neg B$) to happen eventually. It is not a uniform-liveness property since there is no *single* sequence that can successfully extend all finite sequences.

8.3. Composite Properties

Many properties are neither safety nor liveness. Total correctness is the intersection of safety properties (Partial Correctness and Abortion Freedom) and a liveness property (Termination). In fact, every property is the intersection of a safety property and a liveness property. The proof of this (below) depends on a topological characterization of safety and liveness suggested in [Plotkin 84].

There is a natural topology of S^ω [Landweber 69] in which safety properties are exactly the closed sets and liveness properties are exactly the dense sets. The basic open sets of this topology are the sets of all sequences that share a common prefix. As usual, an open set is the union of basic open sets, a closed set is the complement of an open set, and a dense set is one that intersects every non-empty open set. It is now possible to prove

Theorem 8.3: Every property P is the intersection of a safety property and a liveness property.

Proof: Let $\bar{P}$ be the smallest safety property containing P and let L be $\neg(\bar{P}-P)$. Then,

$$\begin{aligned} L \cap \bar{P} &= \neg(\bar{P}-P) \cap \bar{P} \\ &= (\neg\bar{P} \cup P) \cap \bar{P} \\ &= (\neg\bar{P} \cap \bar{P}) \cup (P \cap \bar{P}) \\ &= P \cap \bar{P} \\ &= P. \end{aligned}$$

It remains to show that L is dense, and hence a liveness property. By way of contradiction, suppose there is a non-empty open set O contained in $\neg L$ and thus L is not dense. Then, $O \subseteq (\bar{P} - P)$. Consequently, $P \subseteq (\bar{P} - O)$. The intersection of two closed sets is closed, so $\bar{P} - O$ is closed and thus a safety property. This contradicts the hypothesis that $\bar{P}$ is the smallest safety property containing P. □

An obvious corollary of this is

Corollary 8.4: If a notation Σ for expressing properties is closed under complement, intersection, and topological closure then any Σ-expressible property is the intersection of a Σ-expressible safety property and a Σ-expressible liveness property.

Thus, in order to establish that every temporal logic expressible property P can be given as the conjunction of a safety property and a liveness property expressed in temporal logic, it suffices to show that the smallest safety property containing P is also expressible in temporal logic.

Plotkin has shown that any property that can be expressed in temporal logic can be written as the conjunction of two temporal logic expressible liveness properties [Plotkin 83]. In fact, a more general result can be proven.

Theorem 8.5: If $|S|>1$ then any property P is the intersection of two liveness properties.

Proof: By hypothesis, there are two states a and b in S. Let L_a (L_b) be the set of all histories that end in an infinite sequence of a's (b's). Both L_a and L_b are liveness properties and $L_a \cap L_b = \varnothing$. Now,

$$(P \cup L_a) \cap (P \cup L_b) \;=\; (P \cap P) \cup (P \cap L_a) \cup (P \cap L_b) \cup (L_a \cap L_b) \;=\; P.$$

The union of any set and a dense set is dense, so $P \cup L_a$ and $P \cup L_b$ are liveness properties and the theorem is proven. □

As before, there is an obvious corollary.

Corollary 8.6: If a notation Σ for expressing properties is closed under union and intersection and there exist Σ-expressible liveness properties with empty intersection then any Σ-expressible property is the intersection of two Σ-expressible liveness properties.

Topology also provides a convenient framework for investigating the closure of safety and liveness under Boolean operations. Safety properties (closed sets) are closed under union and intersection. Liveness properties (dense sets) are closed only under union. Neither is closed under complement. Finally, the only property that is both safety and liveness is S^ω itself.

8.4. Proof Obligations for Safety and Liveness

Just as properties can be viewed in terms of proscribed "bad things" and prescribed "good things," so can a property recognizer[2]. When a "bad thing" ("good thing") of a property occurs, we would expect a "bad thing" ("good thing") to happen in the property recognizer for that property. The "bad thing" for a property recognizer is making an undefined transition because if such a

[2] Since we are only concerned with properties of infinite sequences, these property recognizers are exactly Buchi automata.

"bad thing" happens (in every run) while reading an input, the automaton will not accept that input. The "good thing" for a property recognizer is entering an infinite-accepting state, because this "good thing" must happen infinitely often for the automaton to accept an input.

Having identified these "bad things" and "good things" for property recognizers, it is now possible to characterize a class of property recognizers that always specifies safety properties and another class that always specifies liveness properties. Define a *safety recognizer* to be a deterministic property recognizer in which

SR: Every cycle contains an infinite-accepting state.

In a safety recognizer, "good things" are inevitable, unless they become impossible due to an undefined transition, a "bad thing." Ignoring finite-accepting states, m_{pc} (Figure 3.5), m_{mutex} (Figure 3.2) and m_{fcfs} (Figure 3.4) are examples of safety recognizers.

There is a natural correspondence between safety recognizers and safety properties.

Theorem 8.7: Safety recognizers specify only safety properties.

Proof: Assume m_S is a safety recognizer for a property S. We must show that S satisfies Safety (8.2).

Let σ be an infinite sequence not accepted by m_S. Thus, $\sigma \not\models S$, and according to (8.2) we must show

$$(\exists i\colon 0 \le i\colon (\forall \tau\colon \tau \in S^{\omega}\colon \sigma[..i]\tau \not\models S)). \tag{8.8}$$

Since σ is not accepted by m_S, because m_S is a safety recognizer it must attempt an undefined transition after some finite prefix $\sigma[..i]$. Consequently, m_S must reject any sequence beginning with $\sigma[..i]$, and

$$(\forall \tau\colon \tau \in S^{\omega}\colon \sigma[..i]\tau \not\models S)).$$

Showing that $(8.8) \Rightarrow \sigma \not\models S$ is trivial, so S satisfies Safety (8.2). □

Notice that due to SR, a safety recognizer has no reject knots. Thus, it is unnecessary to prove Knot Exit (4.9) and Knot Variance (4.10) when showing that a program satisfies a property specified by a safety recognizer. This means that proving a safety property never requires a variant function (or well-foundedness argument). The remaining proof obligations for a safety recognizer constitute an invariance argument. We therefore conclude that safety properties are proved using only invariance arguments.

A *liveness recognizer* is a deterministic property recognizer in which

LR1: All states have transitions defined for every input symbol.
LR2: There is a path from every state to an infinite-accepting state.

LR1 ensures that "bad things" are not possible for a liveness recognizer; LR2 ensures that a "good thing" is always possible, as it should be according to Liveness (8.5). Ignoring finite-accepting states, property recognizer m_{gs} of Figure 3.3 is an example of a liveness recognizer. There is a natural correspondence between liveness recognizers and liveness properties.

Theorem 8.8: Liveness recognizers specify only liveness properties.

Proof: Assume m_L is a liveness recognizer for a property L. We must show that L satisfies Liveness (8.5).

Let σ be a finite sequence. To show that (8.5) holds, we must show that there is a infinite sequence τ such that $\sigma\tau \models L$. Due to LR1, m_L cannot attempt an undefined transition upon reading σ. Thus, σ leaves m_L in some automaton state q. Due to LR2, there is a path of automaton states from q to some infinite-accepting state q'. Let τ_0 be a finite input that takes m_L from q to q'. Again, by LR2, there must be a path from q' to an infinite-accepting state q''. Let τ_1 be a finite input that takes m_L from q' to q''. This argument can be repeated, resulting in an infinite sequence $\tau = \tau_0\tau_1 \ldots$. Moreover, $\sigma\tau$ causes m_L to be in some infinite-accepting state infinitely often. Thus, $\sigma\tau$ is accepted by m_L, and so $\sigma\tau \models L$ and (8.5) holds. □

Returning to our proof obligations, notice that due to LR1, undefined transitions are not possible and so (4.3) and (4.4) always hold for a liveness recognizer. Simulation Basis (4.6) and Simulation Induction (4.7) then degenerate to proving that the correspondence invariants are correct, using (4.1) and (4.2). A liveness recognizer can have reject knots, so Knot Exit (4.9) and Knot Variance (4.10) must be proved. We therefore conclude that liveness properties are proved using variance (well-foundedness) arguments in concert with an invariance argument ((4.1) and (4.2)). The proof obligations for a liveness property will generally be simple, since it is not necessary to show that undefined transitions are not taken.

Given a deterministic property of infinite sequences, it is not difficult to construct a safety recognizer and a liveness recognizer that specify properties whose intersection is the original property. The construction shows how the safety aspects and liveness aspects of a property can be isolated and how they manifest themselves as proof obligations.

Theorem 8.9: Given a property P recognized by property recognizer m_P, there are properties $P_{S(P)}$ and $P_{L(P)}$ with recognizers $m_{S(P)}$ and $m_{L(P)}$ such that

(i) $m_{S(P)}$ is a safety recognizer,
(ii) $m_{L(P)}$ is a liveness recognizer, and
(iii) $P = S(P) \cap L(P)$.

Proof: Construct $m_{S(P)}$ from m_P as follows.

(1) Delete states from which no infinite-accepting state is reachable, until there are no such states.

(2) Make all remaining states infinite-accepting.

The resulting automaton satisfies SR, hence it is a safety recognizer.

Construct $m_{L(P)}$ from m_P as follows.

(1) Delete states from which no infinite-accepting state is reachable, until there are no such states.

(2) Add a new infinite-accepting state q_t that has a transition to itself on all input symbols.

(3) For every state q that has an undefined transition on any input symbol s, add a transition from q to q_t under s.

The resulting automaton satisfies LR1 and LR2, hence it is a liveness recognizer.

It remains to show that $P = S(P) \cap L(P)$. Suppose an infinite sequence σ is accepted by m_P. To show that $P \subseteq S(P) \cap L(P)$, we must show that σ is accepted by both $m_{S(P)}$ and $m_{L(P)}$. Steps (2) in the construction of $m_{S(P)}$ and steps (2) and (3) in the construction of $m_{L(P)}$ cannot cause a sequence accepted by m_P to be rejected by either property recognizer. The states deleted in step (1) of both constructions cannot be reached in an accepting run of m_P. So, deleting them will not cause a sequence accepted by m_P to be rejected by either $m_{S(P)}$ or $m_{L(P)}$. Thus, both $m_{S(P)}$ and $m_{L(P)}$ accept σ.

Now suppose an infinite sequence σ is not accepted by m_P. We must show that either $m_{S(P)}$ or $m_{L(P)}$ rejects σ. Since m_P rejects σ either (i) it makes an undefined transition on σ, or (ii) m_P does not enter an infinite-accepting state after some finite prefix of σ. In case (i), $m_{S(P)}$ does not accept σ. In case (ii), on reading σ, m_P loops in a reject knot. Either all of the states in this reject knot were deleted from $m_{S(P)}$ in step (1) of its construction, in which case σ will be rejected by $m_{S(P)}$, or else the reject knot was not deleted in either $m_{S(P)}$ or $m_{L(P)}$ (since step (1) is the same for both) and therefore $m_{L(P)}$ will reject σ. □

The property recognizer $m_{inf\text{-}tc}$, that specifies the Total Correctness property for infinite sequences, and its safety and liveness recognizers $m_{S(inf-tc)}$ and $m_{L(inf-tc)}$ are shown in Figure 8.1. States q_t, q_2, and q_3 of $m_{L(inf-tc)}$ are

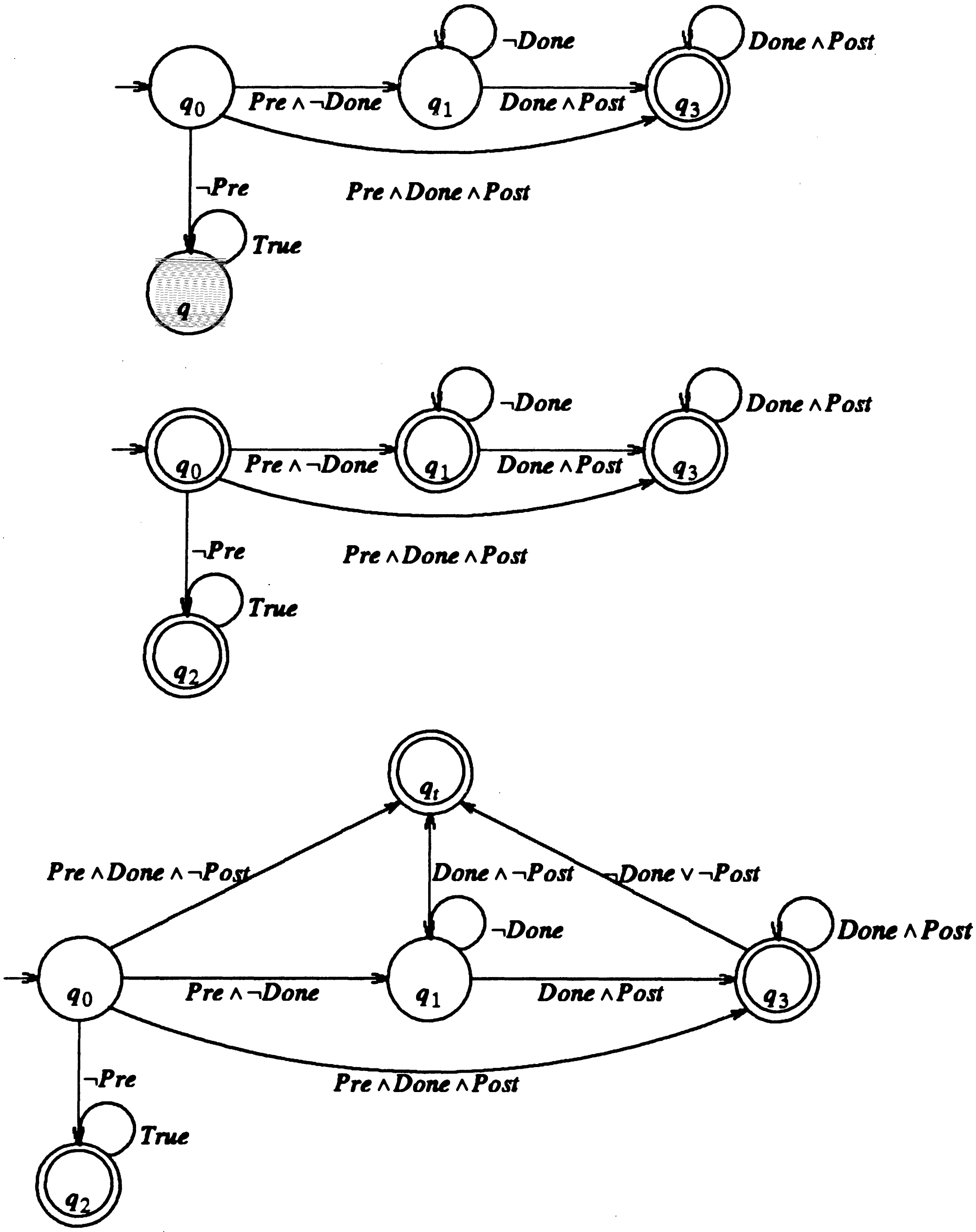


Figure 8.1. $m_{inf\text{-}tc}$, $m_{S(inf-tc)}$, and $m_{L(inf-tc)}$

congruent and can be combined. The resulting property recognizer is depicted in Figure 8.2.

8.5. Formal Definitions for Finite Sequences

We can extend the formal definitions of safety and liveness to handle properties including finite sequences by resorting to our topological characterization. The basic open sets of our topology are now the singleton sets containing each finite sequence and all sets of sequences that share a common prefix. Safety properties are the closed sets of this topology and liveness properties are its dense sets.

The definition of safety given by (8.2) characterizes the closed sets in the topology because any collection of finite sequences (isolated closed points of the topology) can be added to a closed set of infinite sequences and the result remains

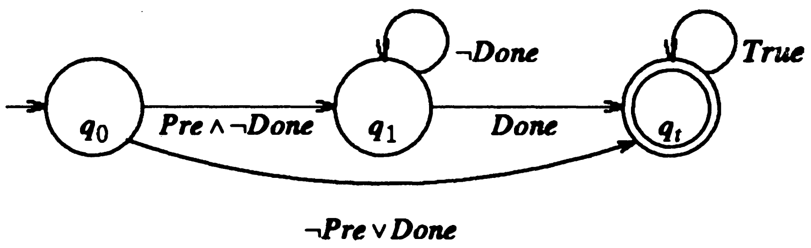


Figure 8.2. $m_{L(inf-tc)}$ simplified

closed. Thus, we can use (8.2) as a formal characterization of safety for properties involving finite as well as infinite sequences.

Since each finite sequence is an open set, any dense set must contain all of the finite sequences. Therefore, in order to be able to continue using the dense sets as a characterization of liveness, we take P to be a liveness property if and only if

$$\textbf{Liveness: } (\forall \sigma \colon \sigma \in S^{*} \colon \sigma \models P \land (\exists \tau \colon \tau \in S^{\omega} \colon \sigma\tau \models P)). \qquad (8.9)$$

This definition asserts that a liveness property must include all finite sequences. If a program execution terminates before a "good thing" has happened—corresponding to some history leaving a property recognizer in a non-finite-accepting state—then a "bad thing" has happened. Since a (pure) liveness property cannot prohibit "bad things", it cannot prohibit any given finite sequence. It must, therefore, include all such sequences.

It may be disturbing that Guaranteed Service as specified by m_{gs} is not a pure liveness property because there are finite sequences not accepted by m_{gs}. The liveness property in Guaranteed Service is that no request is infinitely delayed, but Guaranteed Service has a safety aspect as well. The safety property is that execution never ends with a request pending. Processing a pending request is a "good thing" and execution ending with a pending request is a "bad thing." Although the "good thing" not happening has the same net effect as the "bad thing" happening, they are quite different phenomena.

CHAPTER 9

Related Work

We have shown how to decompose a (temporal) property into proof obligations. We break up the task of proving that a program satisfies one temporal formula—the property—into showing that it satisfies a number of simpler temporal formulas—the proof obligations. Notice that Simulation Basis (4.6), Finite Acceptance (4.8), and Knot Exit (4.9) are temporal formulas because they are predicate logic formulas. The remaining two proof obligations, Simulation Induction (4.7) and Knot Variance (4.10), can be formulated in temporal logic, as

Temporal Simulation Induction: For all i: $q_i \in Q$:

$$\Box(C_i \Rightarrow \bigcirc(\bigvee_{j:q_j \in Q}(T_{ij} \wedge C_j))), \tag{9.1}$$

Temporal Knot Variance: For all reject knots κ and all $q_i \in \kappa$:

$$\Box((C_i \wedge 0 < v_\kappa(q_i) = V) \Rightarrow \bigcirc(\bigwedge_{j:q_j \in \kappa}((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < V))) \tag{9.2}$$

Other investigations into decomposing temporal properties include [Barringer et al. 84], [Gerth 84], [Jones 83], [Misra et al. 82], [Nguyen et al. 85] and [Stark 84]. Most of that work is concerned with decomposing various classes of global temporal properties of a system into local properties of the system components, resulting in so-called compositional proof systems. The work in [Gerth 84] is most similar to ours in that the primitive formulas into which temporal properties are decomposed resemble triples. That work, however, is concerned only with finite sequences (both as properties and programs) and therefore does not address most of the problems that concern us.

We chose to express the proof obligations as partial correctness logic triples rather than as temporal logic formulas because our experience is that people have less trouble understanding and manipulating triples. Moreover, the relation between triples and the program text is always clear—when a proof obligation formulated as a triple cannot be proved, there is little question where to start looking in the program. This is not the case for formulas of temporal logic, because they do not explicitly mention the program. Finally, we hope to integrate our approach with methods to develop a program and its proof of correctness hand-in-hand, as discussed in [Dijkstra 76] [Gries 81]. These methods can be formulated in terms of triples, so it made sense for us to remain in that framework.

Considering our proof obligations from a temporal viewpoint does offer some insights. Temporal Knot Variance (9.2) requires that execution of *every* atomic action cause the value of a variant function to decrease, thereby ensuring progress is made towards accepting the history. Without making assumptions about fairness, this is the only way to ensure that all infinite histories leave a reject knot because atomic actions that do not decrease any variant function, can be repeated indefinitely, resulting in a history that is not accepted by the property recognizer. Thus, while we would be happy to establish

$$\Box((C_i \wedge 0 < \nu_\kappa(q_i) = V) \Rightarrow \Diamond(\bigwedge_{j:q_j \in \kappa}((T_{ij} \wedge C_j) \Rightarrow \nu_\kappa(q_j) < V))), \tag{9.3}$$

without making fairness assumptions, we are forced to demonstrate

$$\Box((C_i \wedge 0 < \nu_\kappa(q_i) = V) \Rightarrow \bigcirc(\bigwedge_{j:q_j \in \kappa}((T_{ij} \wedge C_j) \Rightarrow \nu_\kappa(q_j) < V))). \tag{9.4}$$

However, if we can make assumptions about fairness, then we need not prove (9.4), in order to establish (9.3). Instead, it suffices to prove that certain *helpful processes* that do decrease the variant function are eventually executed and that executing other processes does not increase the variant function. This method is formalized as temporal logic inference rules in [Manna & Pnueli 84]—one rule for each type of fairness (e.g. weak fairness, strong fairness)—and can be adapted to our approach by replacing Knot Variance (4.10) with the hypotheses of the appropriate inference rule. These hypotheses are easily formulated as predicate logic formulas and partial correctness logic triples. This, then, provides a second way in our approach to prove a property P under a fairness assumption F. The first (see Chapter 7), was to construct the property recognizer for $F \Rightarrow P$ and show that the proof obligations it defines are satisfied; the second, is to construct a property recognizer for P and extract proof obligations from it, except with the Knot Variance (4.10) obligation replaced by the hypotheses from the appropriate temporal logic inference rule.

A third way to prove properties assuming fairness might be possible. To show that a program π satisfies a property P, one needs to show that certain partial correctness logic triples are satisfied. We would like to be able to read *proof* *assumptions* from the property recognizer for F. These assumptions could be used in the proofs of the triples for P to show that π satisfies P assuming F. How to extract proof assumptions from a property recognizer remains an open problem.

Another method of verifying that a program satisfies a property is *model checking* [Clarke et al. 83] [Emerson & Lei 85] [Lichtenstein & Pnueli 85]. Here, a program π is viewed as specifying a *Kripke structure* K_π. To determine if π satisfies P it suffices to check whether K_π is a model for P. This amounts to checking each state in the state space to see which sub-formulas of P hold in that state. Determining whether K_π is a model for P requires time linear in both the length of P and the size of the program state space.

Recently, [Vardi & Wolper 85] observed that K_π can be viewed as a Buchi automaton that accepts exactly the histories of π. From this automaton and one that recognizes sequences satisfying $\neg P$, a Buchi automaton $m_{\pi \wedge \neg P}$ can be constructed that accepts all histories of π not satisfying P. The decision procedure for the emptiness problem for $m_{\pi \wedge \neg P}$ can then be used to determine if π satisfies P; the decision procedure is exponential in the length of P and linear in the size of the program state space.

The drawback to both these methods is that they require time linear in the size of the state space. (The fact that the second method is exponential in the length of P is inconsequential due to the relative size of the program state space.) They are practical only for those applications where the program state space is of a manageable size. In our approach, rather than check every state in the state space, the state space is partitioned into equivalence classes defined by the correspondence invariants. The number of correspondence invariants is exponential in the length of P, since there is one for each state in m_P; the number of proof obligations is linear in the size of the program. Thus, with our method, the number of proof obligations incurred for a deterministic property is exponential in the length of P and linear in the size of the program. Since the size of the program is likely to be substantially smaller than the size of the state space, our approach is rather attractive.[1] Even for non-deterministic properties, the number of proof obligations incurred with our approach is bounded by the size of the state space. Thus, our approach is comparable to the model checking for this case.

Of course, verification is only necessary if synthesis is not possible. Techniques to synthesize the synchronization portion of a finite-state concurrent program from a propositional temporal logic specification are given in [Clarke & Emerson 81] and [Manna & Wolper 84]. The latter technique is most closely related to the work of this paper, since it is based on linear time temporal logic. In it, a *model graph* for a property P is constructed and then converted into a program. This model graph is just the safety recognizer, $m_{S(P)}$.

To see how this works, suppose we are given a property P recognized by m_P and a collection of process names Π. The atomic formulas of P are *communication*

[1]We assume that the cost of deciding the validity of a Hoare triple is constant. This is reasonable for purposes of comparison because in the model checking approach the ability to decide the validity of an implication in constant time follows from the restriction to propositional temporal logic.

predicates of the form $\langle \phi, X, \psi \rangle$, where ϕ and ψ are process names and X is a value. A communication predicate holds on a program state if and only if in that program state process ϕ communicates X to process ψ. We need to assume there is at most one communication predicate on each arc of m_P and that each communication predicate involves (as its source or destination) a distinguished *synchronization process* χ. Our goal is to synthesize sequential program skeletons, for each of the process names in Π such that every concurrent execution of these program skeletons satisfied P. A program skeleton includes code for all of the interactions between processes but ignores the strictly internal actions of a process.

The first step in the synthesis is to eliminate the liveness aspects of P. We eliminate reject knots. This can be done using any of the modification techniques discussed in Chapter 7. The resulting recognizer m_S recognizes a safety property S that is a subset of P. If the synthesized program satisfies S then it will satisfy P.

The next step is to particularize m_S for each process name ϕ in Π. Let two automaton states q and q' be *equivalent with respect to* ϕ if and only if there is a path from q to q' or from q' to q along the arcs of m_S that does not contain an arc labelled with a communication predicate involving ϕ. Let m_S^{ϕ} be m_S with all equivalent states collapsed together. Notice that each arc in m_S^{ϕ} is labelled with a communication predicate involving ϕ. Notice also that m_S^{χ} is m_S.

The final step of the synthesis is to create program skeletons from the m_S^{ϕ}'s. Each of these skeletons will consist of a single repetitive command. The arms of this command will have the form

$$mstate_\phi = q_i \wedge c_{ij} \rightarrow mstate_\phi := q_j,$$

where $mstate_\phi$ is a local variable of process ϕ, q_i and q_j are states of m_S^ϕ, and c_{ij} is the *communication guard* [Hoare 78] derived from the transition predicate T_{ij}. If T_{ij} is $\langle \phi, X, \psi \rangle$ then c_{ij} is $\psi!X$, and if T_{ij} is $\langle \psi, X, \phi \rangle$ then c_{ij} is $\psi?X$.

To see that the resulting program satisfies P, let the correspondence predicate for state q_i of m_S be:

$$C_i \equiv mstate_\chi = q_i.$$

(Since all of the reject knots of m_P were eliminated in going to m_S, there is no need for any variant functions.)

The synthesis technique of [Manna & Wolper 84] suggests the following approach to deriving concurrent programs from their specifications. If one (magically) were given variant functions for the reject knots then one would not have to restrict m_P to m_S. One would have to guarantee that the synthesized code decreased these variant functions appropriately. Similarly, if one (magically) were given correspondence invariants then one might be able to do away with the synchronization process and to loosen up the control structure of the other processes. Again, one would have to guarantee that the synthesized code maintained these correspondence invariants.

We do not believe one will ever be able to deduce automatically the necessary correspondence invariants and variant functions from a temporal property.

However, it may be possible to amass a body of techniques that can be used to guide the development of variant functions and correspondence invariants and the transformation of these into program text. This, a methodology for deriving concurrent programs from their specifications, is the principal open problem in our work.

Turning from the issues of program verification and derivation, let us consider safety and liveness. Recently, the problem of syntactically characterizing those temporal formulas that denote safety properties and those that denote liveness properties has been explored. Sistla gives complete deductive systems for generating both safety and liveness properties that are expressable in a temporal logic where the only temporal connectives are $\Box$ and $\Diamond$ [Sistla 85]. This approach is not entirely syntactic since it depends upon a decision procedure for the validity of atomic formulas.

A syntactic characterization for a temporal logic that allows quantification over the past as well as the future is claimed in [Lichtenstein et al. 85]. However, the property $P \mathcal{U} Q$ is held to be a liveness property. This is intuitively implausable. Surely, the occurrence in a history of a state satisfying $\neg P$ before a state satisfying Q is a "bad thing." Therefore, we cannot accept this characterization of liveness.

An automata-theoretic characterization of safety is given in [Sistla 85]. It is shown that the states of a property recognizer for a safety property can be partitioned into "good" states and "bad" states; the "bad" states are never entered in an accepting run. Entering a "bad" state is thus the "bad thing" of the safety property. Notice that if the "bad" states are removed from the property recognizer (a legitimate modification, since no accepting run contains these states) then entering a "bad" state becomes making an undefined transition. Sistla's result is, in effect, a converse to Theorem 8.7. The analogous converse to Theorem 8.8 appears in [Alpern & Schneider 86].

Finally, let us consider the topological characterization of properties. We have shown that safety properties are closed sets and liveness properties are dense sets. Landweber shows that deterministic Buchi automata (hense deterministic property recognizers) recognize the $\mathcal{G}_\delta$ sets (that is, countable intersections of open sets) [Landweber 69]. In [Arnold 83], various transition systems are given topological characterization. General Buchi automata are shown to recognize the Souslin sets.