Verifying Temporal Properties Without Temporal Logic

Fred B. Schneider*
Bowen Alpern*

TR 87-848
(Revised July 1988)

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Research supported in part by the National Science Foundation under Grants DCR-8320274 and CCR-8701103, by the Office of Naval Research under contract N00014-86-K-0092, and by Digital Equipment Corporation..

Verifying Temporal Properties without Temporal Logic†

Bowen Alpern

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Fred B. Schneider*

Department of Computer Science
Cornell University
Ithaca, New York 14853

December 23, 1985
Revised July 1, 1987
Revised July 21, 1988

ABSTRACT

An approach to proving temporal properties of concurrent programs that does not use temporal logic as an inference system is presented. The approach is based on using Buchi automata to specify properties. To show that a program satisfies a given property, proof obligations are derived from the Buchi automata specifying that property. These obligations are discharged by devising suitable invariant assertions and variant functions for the program. The approach is shown to be sound and relatively complete. A mutual exclusion protocol illustrates its application.

Categories and Subject Descriptors D.2.1 [**Software Engineering**]: Requirements/Specifications--methodologies; D.2.4 [**Software Engineering**]: Program Verification--correctness proofs; D.3.1 [**Programming Languages**]: Formal Definitions and Theory--semantics; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs;

General Terms: Program verification, Temporal logic, Safety Properties, Liveness Properties, Buchi automata,

*Supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grants DCR-8320274 and CCR-8701103, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these sponsors.

†Some of the results in this paper were first presented at IEEE Symposium on Logic in Computer Science, Ithaca, New York, June 1987.

1. Introduction

Experience has shown that while it may be possible to convince oneself of the correctness of a sequential program by considering some subset of its executions, this is impossible for concurrent programs. Consequently, methods have been devised 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 very large, only a single program text need be analyzed.

An execution of a program can be viewed as an 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. Terminating executions are extended to infinite histories by repeating the final state. In a concurrent or distributed program, a history is the sequence of states that results from the interleaved execution of atomic actions of the processes.

A *property* is a set of sequences of states; a program *satisfies* a property if each of its histories is in the property. Specifying a property as a predicate on sequences allows the essence of that property to be made explicit. Formulas of temporal logic can be interpreted as predicates on sequences of states, and various formulations of such temporal logics have been used for specifying properties—called *temporal 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 temporal 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 that a program satisfies some property of interest by using these axioms along with program-independent axioms and inference rules of temporal logic [Manna & Pnueli 83b] to show that the temporal formula characterizing the property is a theorem of the logic. Thus, proving that a program satisfies a property is reduced to theorem-proving in a temporal logic.

This paper describes a different approach for proving temporal properties of (concurrent) programs. The approach is based on specifying a property as a Boolean combination of deterministic Buchi automata. Proof obligations are extracted from these automata. These obligations generalize the invariant and variant function used to prove partial correctness and termination of sequential programs and define verification conditions that must hold for any program satisfying the property. The verification conditions themselves can be formulated as Hoare triples [Hoare 69], so reasoning in temporal logic is not required.

We proceed as follows. Section 2 gives the semantics of the programs that we consider. Section 3 reviews Buchi automata and explains how they can be used to specify properties. Extraction of proof obligations from Buchi automata is discussed in section 4. Section 5 illustrates our method on a mutual exclusion protocol. Section 6 compares our approach to related work and section 7 is a summary.

2. Programs

A program Π is specified by

S_Π a countable set of program states,

$Init_\Pi \subseteq S_\Pi$ a set of possible initial states, and

$\mathcal{A}_\Pi$, a finite set of atomic actions.

An atomic action defines a set of pairs of program states and is therefore a subset of $S_\Pi \times S_\Pi$. Atomic action α is *enabled* in a state s provided $(\exists\, t\colon \langle s, t\rangle \in \alpha)$. The statement

$\langle \textbf{if}\ b \rightarrow C\ \textbf{fi} \rangle$

is used to specify an atomic action containing those elements $\langle s, t\rangle$ such that (predicate) b holds on s and t is the state produced by executing (assignment) C starting in state s.

A program is usually presented as a text, where statements or phrases in the text denote atomic actions. Rather than enumerating the atomic action of a program directly, it is frequently convenient to identify in such a text the control points that delimit atomic actions. In this paper, control points are denoted by marking and numbering them and an atomic action is described by the text between these marks. For example, program fragment of process A

$\ldots\ \{3{:}\}\ \ x := 23\ \ \{4{:}\}\ \ldots$

defines a single atomic action

$\alpha_3\colon\ \langle \textbf{if}\ pc_A{=}3 \rightarrow pc_A := 4;\ x := 23\ \textbf{fi} \rangle$

where pc_A simulates the program counter for process A. Thus, α_3 is enabled in any state in which $pc_A{=}3$.

Formally, a history of a program Π is any sequence of states from S_Π such that the first state is in $Init_\Pi$ and every subsequent state is the result of executing an enabled atomic action from $\mathcal{A}_\Pi$ on the previous state in the sequence. Notice that no restriction is made about the choice of an atomic action when more than one is enabled.[1] To ensure that all histories are infinite, we include in $\mathcal{A}_\Pi$ an atomic action that has no affect on the program state and is enabled when no other atomic action is.

An example program *MEP* is shown in Figure 2.1. It is a simplified version of the solution to the two-process critical section problem described in [Peterson 81].[2] Based on the control point annotations, we obtain set of atomic actions $\mathcal{A}_{MEP}$ of Figure 2.2. In those atomic actions, variable pc_A simulates the program counter for process A and pc_B simulates the program counter for process B. Finally, we have

$$Init_{MEP} \;=\; \{s \mid s \in S_{MEP} \wedge s \models (pc_A = pc_B = 1 \wedge (turn = A \vee turn = B))\}$$

because when execution is begun, both processes start at the beginning of their loops and *turn* is initialized.

3. Specifying Properties Using Buchi Automata

A property is a set of infinite sequences of program states. We restrict attention to properties that can be specified by formulas of some linear-time, temporal logic with first-order monadic predicates—that is, formulas composed of temporal operators, boolean connectives, and atoms that

```
MEP: cobegin
 A :: do true → {1: } ncs_A;
                {2: } active_A := true;
                {3: } turn := B;
                {4: } do active_B ∧ turn=B → {5: } skip od;
                {6: } cs_A;
                {7: } active_A :=false
      od
 //
 B :: do true → {1: } ncs_B;
                {2: } active_B := true;
                {3: } turn := A;
                {4: } do active_A ∧ turn=A → {5: } skip od;
                {6: } cs_B;
                {7: } active_B :=false
      od
 coend.
```

Figure 2.1. Peterson's Mutual Exclusion Protocol

[1] Fairness is discussed in Section 5.2.

[2] To simplify the presentation, we have assumed that the non-critical section (*ncs*) always terminates. However, the algorithm and our proof do not rely on this assumption.

$$
\begin{aligned}
\mathcal{A}_{MEP} = \{ & \alpha_1^A: \langle \textbf{if } pc_A=1 \rightarrow pc_A := 2;\ ncs_A \textbf{ fi}\rangle,\\
& \alpha_2^A: \langle \textbf{if } pc_A=2 \rightarrow pc_A := 3;\ active_A := true \textbf{ fi}\rangle,\\
& \alpha_3^A: \langle \textbf{if } pc_A=3 \rightarrow pc_A := 4;\ turn := B \textbf{ fi}\rangle,\\
& \alpha_4^A: \langle \textbf{if } pc_A=4 \wedge active_B \wedge turn=B \rightarrow pc_A := 5 \textbf{ fi}\rangle,\\
& \alpha_5^A: \langle \textbf{if } pc_A=4 \wedge \neg(active_B \wedge turn=B) \rightarrow pc_A := 6 \textbf{ fi}\rangle,\\
& \alpha_6^A: \langle \textbf{if } pc_A=5 \rightarrow pc_A := 4 \textbf{ fi}\rangle,\\
& \alpha_7^A: \langle \textbf{if } pc_A=6 \rightarrow pc_A := 7;\ cs_A \textbf{ fi}\rangle,\\
& \alpha_8^A: \langle \textbf{if } pc_A=7 \rightarrow pc_A := 1;\ active_A := false \textbf{ fi}\rangle,\\
& \alpha_1^B: \langle \textbf{if } pc_B=1 \rightarrow pc_B := 2;\ ncs_B \textbf{ fi}\rangle,\\
& \alpha_2^B: \langle \textbf{if } pc_B=2 \rightarrow pc_B := 3;\ active_B := true \textbf{ fi}\rangle,\\
& \alpha_3^B: \langle \textbf{if } pc_B=3 \rightarrow pc_B := 4;\ turn := A \textbf{ fi}\rangle,\\
& \alpha_4^B: \langle \textbf{if } pc_B=4 \wedge active_A \wedge turn=A \rightarrow pc_B := 5 \textbf{ fi}\rangle,\\
& \alpha_5^B: \langle \textbf{if } pc_B=4 \wedge \neg(active_A \wedge turn=A) \rightarrow pc_B := 6 \textbf{ fi}\rangle,\\
& \alpha_6^B: \langle \textbf{if } pc_B=5 \rightarrow pc_B := 4 \textbf{ fi}\rangle,\\
& \alpha_7^B: \langle \textbf{if } pc_B=6 \rightarrow pc_B := 7;\ cs_B \textbf{ fi}\rangle,\\
& \alpha_8^B: \langle \textbf{if } pc_B=7 \rightarrow pc_B := 1;\ active_B := false \textbf{ fi}\rangle \ \}
\end{aligned}
$$

Figure 2.2. Atomic Actions $\mathcal{A}_{MEP}$

are first-order predicates of the program states. Such logics are slightly more expressive than propositional temporal logics where the atoms are propositions. However, our temporal formulas can be treated as if they were propositional temporal formulas over different sequences. The elements of these sequences are the equivalence classes of the program states under the monadic predicates.

A Buchi automaton is a finite-state machine that accepts or rejects infinite sequences of input symbols [Eilenberg 74]. Such an automaton m can be used to specify the property containing those sequences of program states accepted by m. Procedures exist to translate propositional temporal formulas into Buchi automata where automaton state transitions are defined in terms of the atoms of the temporal formula [Emerson & Sistla 83] [Alpern 86]. Therefore, restricting consideration to properties that can be specified by Buchi automata—as we do in this paper—is not an additional restriction. Moreover, Buchi automata have natural diagrammatic representations and this is sometimes a convenient way to specify a property.

An example of a Buchi automaton m_{ae} is given in Figure 3.1. It accepts infinite sequences in which after a finite prefix each state satisfies the program state predicate p. In temporal logic, this property is specified as $\Diamond\Box p$. Automaton m_{ae} contains three *automaton states* labeled q_0, q_1, and q_2. The *start state*, q_0, is denoted by an edge with no origin, and *accepting state*, q_1, is denoted by two concentric circles. A Buchi automaton accepts a sequence σ if and only if it enters an accepting state infinitely often while reading σ (assuming non-deterministic choices are resolved in favor of acceptance). Notice, there is no way in m_{ae} to get from q_2 to an accepting state. Such states are called *dead states*. If an automaton is in a dead state, it cannot accept its input.

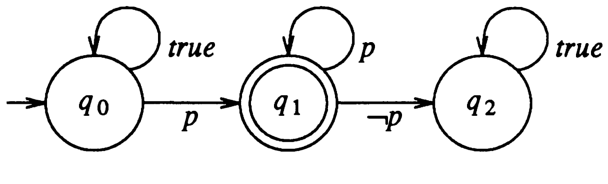

Figure 3.1. m_{ae}

Edges between automaton states are labeled by program-state predicates that are called *transition predicates* and define transitions between automaton states. If a program state satisfies the transition predicate on an edge, then the edge is defined for that program state. For example, because there is an edge labeled p from q_0 to q_1 in m_{ae}, whenever m_{ae} is in q_0 and the next symbol read is a program state satisfying p, then a transition to q_1 can be made. We adopt the convention that there be at least one edge defined from each automaton state for each input symbol.

In order to define a Buchi automaton formally, the following notation will be useful. For any sequence $\sigma = s_0 s_1 \ldots$,

$$\begin{aligned}
\sigma[i] &\equiv s_i, \\
\sigma[..i] &\equiv s_0 s_1 \ldots s_{i-1}, \\
\sigma[i..] &\equiv s_i s_{i+1} \ldots, \\
|\sigma| &\equiv \text{the length of } \sigma \ (\omega \text{ if } \sigma \text{ is infinite}), \\
INF(\sigma) &\equiv \{s \mid s \text{ appears infinitely often in } \sigma\}.
\end{aligned}$$

A Buchi automaton m for a property of a program Π is a five-tuple $\langle S_\Pi, Q, Q_0, A, \delta \rangle$, where

S_Π is the (countable) set[3] of program states of Π,
Q is the (finite) set of automaton states of m,
$Q_0 \subseteq Q$ is the set of start states of m,
$A \subseteq Q$ is the set of accepting states of m,
$\delta \in (Q \times S) \rightarrow 2^Q - \varnothing$ is the *transition function* of m.

Transition function δ can be extended to handle finite sequences of program states in the usual way:

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

It is often convenient to represent δ by using transition predicates, as in the diagram above. The transition predicate t_{ij} associated with the edge from automaton state q_i to q_j, holds for a program state s if and only if $q_j \in \delta(q_i, s)$.

[3]Technically, this is the finite set of equivalence classes (under the monadic predicates of the temporal formula being specified) of program states.

A sequence of automaton states that m might occupy while reading σ is called a *run*. Thus, ρ is a run of m on σ if and only if

$$\rho[0] \in Q_0 \text{ and } (\forall i: 0<i<|\sigma| : \rho[i] \in \delta(\rho[i-1], \sigma[i-1])).$$

A Buchi automaton m accepts a sequence σ if and only if there is a run ρ of m on σ for which $INF(\rho) \cap A \neq \emptyset$.

Notice in m_{ae} (Figure 3.1) that two transitions are possible from q_0 for a program state satisfying p, because any program state that satisfies p also satisfies *true*. When there is more than one start state or more than one transition is possible from some automaton state for a given input symbol, the automaton is *non-deterministic*; otherwise, it is *deterministic*. Thus, m_{ae} is non-deterministic. Using a non-deterministic Buchi automaton, it is possible to specify a property that cannot be specified by a single deterministic Buchi automaton. However, any property that is specified by a non-deterministic automaton can be specified as a Boolean combination of properties each of which can be specified by a deterministic Buchi automaton. For example, $\Diamond\Box p$ the property specified by m_{ae} is the negation of the property $\Box\Diamond\neg p$ specified by the deterministic Buchi automaton m_{io} of Figure 3.2.

Examples of Property Specifications

A Buchi automaton m_{mutex} that specifies the property of *Mutual Exclusion* for two processes is given in Figure 3.3. Mutual Exclusion is the set of sequences in which there is no state where the program counters for two or more processes denote control points inside critical sections. In m_{mutex}, we assume cs_A (cs_B) holds for any state in which process A (B) is executing in its critical section.

The property *Partial Correctness for pre and post* includes all sequences of program states where, if the first state in the sequence satisfies *pre* then in any state where the program counter denotes the end of the program, *post* is satisfied. A Buchi automaton m_{pc} that specifies this property is shown in Figure 3.4. In it, *done* is a predicate that holds for program states in which the program

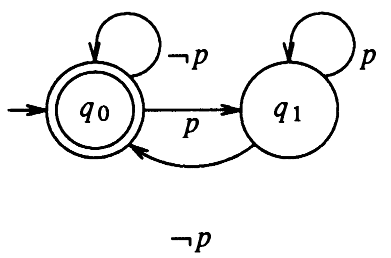

Figure 3.2. m_{io}

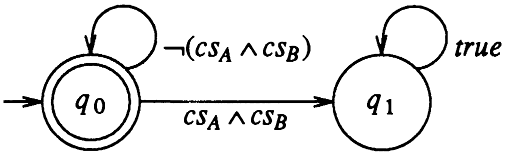

Figure 3.3. m_{mutex}

counter denotes the end of the program.

4. Proof Obligations

Every temporal property $\mathcal{P}$ is a Boolean combination of properties that can be specified by deterministic Buchi automata. Without loss of generality, we assume that this combination is in conjunctive normal form. Thus, $\mathcal{P}$ is the conjunction of *clauses* $\mathcal{P}_1$, $\mathcal{P}_2$, ... $\mathcal{P}_m$. To prove that a program Π satisfies $\mathcal{P}$, one proves separately that the program satisfies each of these clauses. This establishes that every history of program Π is in the property (i.e. set of sequences of program states) specified by each clause, so we can conclude that every history of Π is in the intersection of the properties specified by the clauses and the program satisfies $\mathcal{P}$. Thus, it suffices to derive proof obligations for a single clause.

The proof obligations for a single clause involve exhibiting three *proof instruments*. The first proof instrument, an invariant, handles the safety aspects of the proof; the second, a variant function,

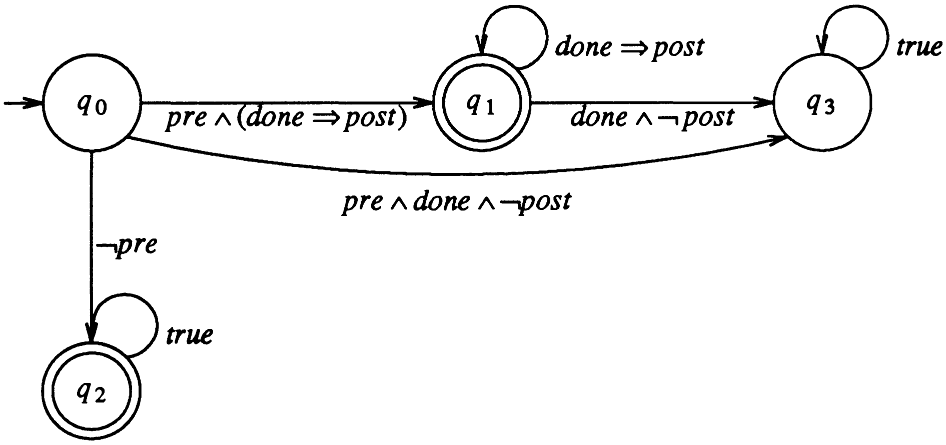

Figure 3.4. m_{pc}

The invariant relates program states in a history to automaton states occupied while reading that history. The candidate function identifies negative automata that might never again enter an accepting state. And, the variant function bounds the number of times that the candidate function can become empty before one of the positive automata enters an accepting state.

If the invariant, candidate function, and variant function satisfy the obligations below, then Π will satisfy $\mathcal{P}_i$. In these obligations, x and y denote elements of $JS(\mathcal{P}_i, \Pi)$, and predicate $x \rightarrow y$ abbreviates

$$x \in I \ \wedge \bigwedge_{1 \le i \le n+p} \delta_i(x^{(i)}, x^{(\Pi)}) = y^{(i)} \ \wedge \ (\exists \alpha\colon \alpha \in \mathcal{A}_\Pi\colon \langle x^{(\Pi)}, y^{(\Pi)} \rangle \in \alpha).$$

The obligations are:

O1: For all $x \in JS(\mathcal{P}_i, \Pi)$: $(\bigwedge_{1 \le i \le n+p} x^{(i)} = q_{i0} \ \wedge \ x^{(\Pi)} \in Init_\Pi) \Rightarrow x \in I$

O2: For all $x, y \in JS(\mathcal{P}_i, \Pi)$: $x \rightarrow y \Rightarrow y \in I$

O3: For all $x, y \in JS(\mathcal{P}_i, \Pi)$: $x \rightarrow y \Rightarrow (Pos(y) \ \vee \ v(x) \ge v(y))$

O4: For all $x, y \in JS(\mathcal{P}_i, \Pi)$: $x \rightarrow y \Rightarrow (Pos(y) \ \vee \ v(x) > v(y) \ \vee \ \emptyset \subset u(y) \subseteq u(x) - neg(y))$.

The first two obligations ensure that I holds throughout any joint history. O1 requires that the first state of a joint history be in I; O2 requires that if one state in a joint history is in I, then so must the next.

The third proof obligation requires that variant function v increase only upon entering a positive state. It ensures that either (i) some positive automaton enters an accepting state infinitely often (and the history is accepted) or (ii) after some point v never increases and no positive automaton subsequently enters an accepting state.

The final proof obligation, O4, ensures that a history not accepted by a positive automaton is rejected by some negative automaton. To see how, observe that after some initial prefix of a history that is not accepted by a positive automaton, there will be no positive state and v will be constant (since its range is well founded). In this case, O4 requires that after executing an atomic action, the candidate function on the new state

- contain only negative automata that were in the candidate function on the old state,
- not contain automata that are accepting in the new state, and
- not be empty.

Thus, unless a positive automaton accepts a history, there must be some negative automaton that, after some point in the history, is thereafter in the candidate function. This automaton rejects the history since it cannot be in an accepting state after the prefix.

We can now prove:

Soundness Theorem: If there are proof instruments I, v, and u that satisfy obligations O1 through O4 then program Π satisfies property $\mathcal{P}_i$.

Proof: Let ρ be a joint history of Π and $\mathcal{P}_i$. $\rho[0] \in I$ by construction (O1). By O2 and induction, $\rho[j] \in I$ for all $0<j$. We must show that ρ causes some positive automaton to be in an accepting state infinitely often or some negative automaton to be in an accepting state only finitely often. Assume that no positive automaton is in an accepting state infinitely often. Thus, there is an index l_1 such that $\rho[l_1..]$ contains no positive states. By O3, the variant function is non-increasing on $\rho[l_1..]$. Since its range is well-founded, there must be an index l_2 such that $l_1 \le l_2$ and the variant function is constant on $\rho[l_2..]$. By O4, there is a negative automaton m_k such that for all $j > l_2$,

$$k \in u(\rho[j]) \wedge k \notin neg(\rho[j]).$$

Therefore, m_k does not enter an accepting state after l_2. This means that m_k rejects ρ, so Π satisfies $\mathcal{P}_i$.

We now show that the method is relatively complete.

Completeness Theorem: If program Π satisfies property $\mathcal{P}_i$ then there exist proof instruments I, v, and u that satisfy obligations O1 through O4.

Proof: Form a directed graph where the nodes are the joint states and there is an edge from node x to y iff y is not a positive accepting state and x immediately precedes y in some joint history. Define transitive, antisymmetric, relation $\gg$ on the nodes of this graph such that $x \gg y$ iff $x \neq y$, there is a path from x to y, and an accepting state for each negative automaton appears somewhere on this path.

Relation $\gg$ is well-founded, as is shown by the following proof by contradiction. If $\gg$ were not well-founded then there would be an infinite descending chain $x_1 \gg x_2 \ldots$. By construction of the graph, this implies the existence of a joint history that includes x_1. Let σ_0 be a prefix of such a joint history that ends with x_1. For each x_i in the infinite descending chain, let σ_i be the path from x_i to x_{i+1} that includes an accepting state for every negative automaton. Such a path exists by definition, because $x_i \gg x_{i+1}$. Finally, let σ be $\sigma_0\sigma_1 \ldots$. Notice that σ is a joint history, that σ contains no positive accepting states after σ_0, and that there are infinitely many accepting states for each negative automaton in σ. Thus, σ does not satisfy $\mathcal{P}_i$. This is a contradiction, and we conclude that $\gg$ is well-founded.

Since $\gg$ is well-founded, the following ordinal function is well defined:

$$H(x) = \sup_{x \gg y}\{H(y)+1\}.$$

If there is no y such that $x \gg y$ then $H(x)=0$ by definition of sup, so H is total. Notice that if $x \gg y$ then $H(x) > H(y)$. Moreover, if there is any path from x to y in the graph then $H(x) \geq H(y)$.

The variant function will be constructed using H and the *level* $l(x)$ of a node x, defined as follows. Level $l(x)$ is the largest integer i such that for any collection of i negative automata there exists some node w with $H(w)=H(x)$ such that there is a path from w to x and there is an accepting state for each automaton in the collection somewhere on the path. Note that by definition of H, the level of a node will be less than or equal to n. (The equality will hold only if x itself is accepting for all negative automata.)

The three proof instruments can now be defined. Choose I to be the characteristic predicate for the set of joint states that appear in joint histories of Π and $\mathcal{P}_i$. Choose $v(x)$ to be ∞ if x is positive and $\langle H(x), n-l(x)\rangle$ otherwise. The range of v with lexicographic ordering of pairs with ∞ larger than any pair is well-founded because the ordinals are. Finally, choose $u(x)$ to be the set of negative automata that do not have accepting states on any path to x from any w such that $v(w)=v(x)$. (Note that $u(x)=\emptyset$ only if there is no such w different from x.)

Proof obligations O1 and O2 follow immediately from the definition of I.

To see that O3 holds, notice that if $x \rightarrow y$ and y is not positive then there is a path (of length 1) from x to y in the graph. Thus, $H(x) \geq H(y)$. Suppose $H(x)=H(y)$. Since any path to x can be extended to y, $l(x) \leq l(y)$. Therefore, $v(x) \geq v(y)$.

To see that O4 holds, suppose that $x \rightarrow y$, y is not positive, and $v(x)=v(y)$ holds. First, note that $neg(y) \cap u(y)$ is empty, since there is a trivial path from y to itself. Because there is a path from x to y, if m_k has an accepting state on a path from w to x, then m_k must have an accepting state on a path from w to y. Therefore, $u(y)$ is contained in $u(x)$. Further, since $x \neq y$ holds, $u(y) \neq \emptyset$. O4 follows immediately. This completes the proof.

4.2. Verification Conditions

Obligations O1 through O4 can be translated into verification conditions formulated as Hoare triples [Hoare 69] and predicate logic formulas. The Hoare triple $\{P\}\ \alpha\ \{Q\}$ for an atomic action α asserts that any execution of α started in a state satisfying P terminates in a state satisfying Q.[5] Thus, $\{P\}\ \alpha\ \{Q\}$ is valid iff $(\forall \langle s,t\rangle\colon \langle s,t\rangle \in \alpha\colon (s \models P) \Rightarrow (t \models Q))$.

[5] Since α is an atomic action, it *must* terminate once execution commences.

To reformulate O1 through O4 in terms of Hoare triples, we define slight variations of the three proof instruments. Define $PJS(\mathcal{P}_i)$, the projection of $JS(\mathcal{P}_i, \Pi)$ with respect to program states, as $PJS(\mathcal{P}_i) = Q_1 \times \cdots \times Q_{p+n}$. Elements of $PJS(\mathcal{P}_i)$ are called *projected joint states*. We write $\bar{x}$ to denote the projection of a joint state x. The projected joint state in which every automaton is in its start state is called the *projected joint start state* and denoted $q_{\bar{0}}$. A projected joint state $\bar{x}$ is considered *positive* if it is the projection of a positive state. For $\bar{x}$ the projection of x and $s \in S_\Pi$, we define:

$$
\begin{aligned}
I_{\bar{x}} &\equiv \{s \mid s \in S_\Pi \wedge \langle \bar{x}, s\rangle \in I\} \\
v_{\bar{x}}(s) &\equiv v(\langle \bar{x}, s\rangle) \\
u_{\bar{x}}(s) &\equiv u(\langle \bar{x}, s\rangle) \\
Pos_{\bar{x}} &\equiv Pos(x) \\
neg_{\bar{x}} &\equiv neg(x)
\end{aligned}
$$

Finally, we define a *projected joint state transition predicate* $t_{\bar{x}\bar{y}}$ to be a predicate that holds for any program state causing a transition from a projected joint state $\bar{x}$ to a projected joint state $\bar{y}$. $t_{\bar{x}\bar{y}}$ can be formed by taking the conjunction over all m_i of the transition predicates labeling the edge from $\bar{x}^{(i)}$ to $\bar{y}^{(i)}$.

Satisfying the following two verification conditions implies that O1 through O4 hold. The first one implies O1.

VC1: $Init_\Pi \Rightarrow I_{\bar{0}}$

The next implies O2 through O4.

VC2: For all $\bar{x}, \bar{y} \in PJS(\mathcal{P}_i)$ and $\alpha \in \mathcal{A}_\Pi$:

$$
\begin{aligned}
&\{I_{\bar{x}} \wedge v_{\bar{x}}{=}\mathrm{V} \wedge u_{\bar{x}}{=}\mathrm{U} \wedge t_{\bar{x}\bar{y}}\} \\
&\alpha \\
&\{I_{\bar{y}} \wedge (Pos_{\bar{y}} \vee v_{\bar{y}} < \mathrm{V} \vee (v_{\bar{y}}{=}\mathrm{V} \wedge \varnothing \subset u_{\bar{y}} \subseteq \mathrm{U} - neg_{\bar{y}}))\}
\end{aligned}
$$

To see that O2, O3, and O4 are implied by VC2, choose

$$x = \langle \bar{x}, s\rangle \text{ and } y = \langle \bar{y}, s'\rangle$$

and assume $x \rightarrow y$. Thus, there is an $\alpha \in \mathcal{A}_\Pi$ such that $\langle s, s'\rangle \in \alpha$, $s \models I_{\bar{x}}$, and $s \models t_{\bar{x}\bar{y}}$. Choose V and U such that $v_{\bar{x}}(s){=}\mathrm{V}$ and $u_{\bar{x}}(s){=}\mathrm{U}$ hold. From VC2, we conclude that postcondition

$$I_{\bar{y}} \wedge (Pos_{\bar{y}} \vee v_{\bar{y}} < \mathrm{V} \vee (v_{\bar{y}}{=}\mathrm{V} \wedge \varnothing \subset u_{\bar{y}} \subseteq \mathrm{U} - neg_{\bar{y}}))$$

holds in y. However, this is equivalent to

$$I_{\bar{y}} \wedge (Pos_{\bar{y}} \vee v_{\bar{y}} \leq \mathrm{V}) \wedge (Pos_{\bar{y}} \vee v_{\bar{y}} < \mathrm{V} \vee \varnothing \subset u_{\bar{y}} \subseteq \mathrm{U} - neg_{\bar{y}})).$$

O2 then follows from the first conjunct, O3 from the second, and O4 from the last.

Conversely, assume O1 through O4 hold. VC1 follows from O1. From O2 we get

$$\{I_{\bar{x}} \wedge t_{\overline{xy}}\}\ \alpha\ \{I_{\bar{y}}\}.$$

From O3 we get

$$\{I_{\bar{x}} \wedge v_{\bar{x}}=\mathrm{V} \wedge t_{\overline{xy}}\}\ \ \alpha\ \ \{Pos_{\bar{y}} \vee v_{\bar{y}} \le \mathrm{V}\}$$

Finally, from O4 we get

$$\{I_{\bar{x}} \wedge v_{\bar{x}}=\mathrm{V} \wedge u_{\bar{x}}=\mathrm{U}\}\ \ \alpha\ \ \{Pos_{\bar{y}} \vee v_{\bar{y}} < \mathrm{V} \vee \varnothing \subset u_{\bar{y}} \subseteq \mathrm{U} - neg_{\bar{y}}\}.$$

Together, these three Hoare triples imply VC2.

We have shown that obligations O1 through O4 are equivalent to verification conditions VC1 and VC2. If the underlying assertion language is expressive enough to capture the preconditions and postconditions of the Hoare triples of VC2, then the verification conditions can be expressed in this logic. Since Hoare Logic is semantically complete relative to the completeness of the assertion language, our proof technique is complete relative to the semantic and expressive completeness of this logic.

4.3. Eliminating the Candidate Function

The invariant and variant function above are generalizations of standard proof instruments used to prove partial correctness and termination of sequential programs. The candidate function is not standard. It can be a useful proof tool, as illustrated in the example of section 5.2, but—as we now show—is not necessary for proving temporal properties.

Although the set of properties that can be specified by deterministic Buchi automata is not closed under negation, it is closed under conjunction and disjunction. Thus, any property $\mathcal{P}_i$ can be written $\mathcal{M} \vee \neg\mathcal{N}$, where $\mathcal{M} = \mathcal{D}_1 \vee \ldots \vee \mathcal{D}_p$ and is accepted by deterministic Buchi automaton $m_{\mathcal{M}}$, and $\mathcal{N} = \mathcal{D}_{p+1} \wedge \ldots \wedge \mathcal{D}_{p+n}$ and is accepted by deterministic Buchi automaton $m_{\mathcal{N}}$. Having made this observation, the proof obligations for $\mathcal{P}_i$ can now be stated without using a candidate function because there exists only one negative automaton. Obligations O1–O3 remain as before. O4 becomes

O4': For all $x, y \in JS(\mathcal{P}_i, \Pi)$: $x \rightarrow y \Rightarrow (Pos(y)\ \vee\ (Neg(y) \Rightarrow v(x) > v(y)))$,

where $Neg(y)$ holds when $m_{\mathcal{N}}$ is in an accepting state in y.

The verification conditions can also be simplified. VC1 is unchanged, and VC2 becomes

VC2': For all $\bar{x}, \bar{y} \in PJS(\mathcal{P}_i)$ and $\alpha \in \mathcal{A}_\Pi$:

$$\{I_{\bar{x}} \wedge v_{\bar{x}}=\mathrm{V} \wedge t_{\overline{xy}}\}$$
$$\alpha$$
$$\{I_{\bar{y}} \wedge (Pos_{\bar{y}} \vee (\mathrm{V} \ge v_{\bar{y}} \wedge (Neg_{\bar{y}} \Rightarrow \mathrm{V} > v_{\bar{y}})))\}$$

where $Neg_{\bar{y}}$ holds when m_N is in an accepting state in $\bar{y}$.

5. Example: Peterson's Protocol

To illustrate our verification method, we prove two properties of Peterson's Protocol (Figure 2.1). First, we prove that it prevents two processes from concurrently executing in critical sections. Then, we prove that a process attempting to enter its critical section will succeed eventually.

5.1. Mutual Exclusion

To prove that the protocol satisfies Mutual Exclusion as specified by m_{mutex} (Figure 3.3), we use the following proof instruments:

$$I = \{\langle q_0, POL\rangle,\ \langle q_1, false\rangle\}$$

$$\begin{aligned}\text{where } POL = {} & (turn{=}A \lor turn{=}B) \land \\ & pc_A{=}3 \Rightarrow active_A \land \\ & pc_A{=}4 \Rightarrow active_A \land \\ & pc_A{=}5 \Rightarrow active_A \land \\ & pc_A{=}6 \Rightarrow active_A \land (turn{=}A \lor \neg active_B \lor pc_B{=}3) \land \\ & pc_A{=}7 \Rightarrow active_A \land (turn{=}A \lor \neg active_B \lor pc_B{=}3) \land \\ & pc_B{=}3 \Rightarrow active_B \land \\ & pc_B{=}4 \Rightarrow active_B \land \\ & pc_B{=}5 \Rightarrow active_B \land \\ & pc_B{=}6 \Rightarrow active_B \land (turn{=}B \lor \neg active_A \lor pc_A{=}3) \land \\ & pc_B{=}7 \Rightarrow active_B \land (turn{=}B \lor \neg active_A \lor pc_A{=}3)\end{aligned}$$

$$v = 0$$

$$u = \varnothing$$

We now show that verification conditions VC1 and VC2 are satisfied with these proof instruments.

To demonstrate VC1, we must show that $Init_\Pi \Rightarrow I_{\bar{0}}$ holds, where $I_{\bar{0}}$ is POL. This is trivial.

To show that VC2 is satisfied, a number of Hoare triples must be checked. For every atomic action α, they are:

$$\{I_{\bar{0}} \land t_{\overline{00}}\}\ \alpha\ \{I_{\bar{0}}\}$$

$$\{I_{\bar{0}} \land t_{\overline{01}}\}\ \alpha\ \{I_{\bar{1}}\}$$

$$\{I_{\bar{1}} \land t_{\overline{11}}\}\ \alpha\ \{I_{\bar{1}}\}$$

For the program of Figure 3.3,

$$cs_A \equiv 6 \le pc_A \le 7$$

$$cs_B \equiv 6 \le pc_B \le 7.$$

Because

$$t_{\overline{00}} \equiv \neg(cs_A \wedge cs_B)$$
$$t_{\overline{01}} \equiv cs_A \wedge cs_B$$
$$t_{\overline{11}} \equiv true$$

and $POL \Rightarrow \neg(cs_A \wedge cs_B)$ holds, these Hoare triples simplify to:

$\{POL\}\ \alpha\ \{POL\}$

$\{false\}\ \alpha\ \{false\}$

$\{false\}\ \alpha\ \{false\}$

There are 16 atomic actions in the program. Since $\{false\}\ \alpha\ \{false\}$ is trivially valid for any α, a total of 16 triples must, therefore, be checked. Establishing that these 16 triples are valid can be done by inspection.

Most of the work in this proof concerns invariant I and verifying its invariance. To make this easier to handle, I can be presented as a *property outline*. A property outline is a program that has been annotated by asserting at each control point the subset of the joint state space in I corresponding to that control point.[6] Subsets of the joint state space are described by enumeration or by a characteristic predicate, whichever is more convenient. A property outline equivalent to I above is given in Figure 5.1. Given such a property outline, it is usual possible to verify the triples for VC2 by inspection.

5.2. Starvation Freedom and Fairness

In addition to mutual exclusion, a solution to the critical section problem should ensure that processes attempting entry to critical sections actually do enter eventually. A process of *MEP* is said to *starve* if it trys to enter its critical section but never succeeds. Process A of *MEP* should eventually enter cs_A whenever

try_A: $pc_A = 4 \vee pc_A = 5$

holds,[7] and similarly process B should eventually enter cs_B whenever

try_B: $pc_B = 4 \vee pc_B = 5$

holds. The Non-Starvation Property $\mathcal{NS}_{MEP}$ asserts that neither process starves. This can be formalized in temporal logic as

$\mathcal{NS}_{MEP}$: $(\Box\Diamond\neg try_A) \wedge (\Box\Diamond\neg try_B)$.

---

[6]Proof outlines serve the same purpose in Hoare's Logic [Lamport & Schneider 84].

[7]This choice of *try* is stronger than it needs to be, but simplifies the proof.

MEP: **cobegin**

A :: **do** $true \rightarrow$ {1: $\langle q_0, true\rangle, \langle q_1, false\rangle$ }
ncs_A;
{2: $\langle q_0, true\rangle, \langle q_1, false\rangle$ }
$active_A := true$;
{3: $\langle q_0, active_A\rangle, \langle q_1, false\rangle$ }
$turn := B$;
{4: $\langle q_0, active_A\rangle, \langle q_1, false\rangle$ }
do $active_B \wedge turn = B \rightarrow$ {5: $\langle q_0, active_A\rangle, \langle q_1, false\rangle$ }
skip od;
{6: $\langle q_0, active_A \wedge (turn = A \vee \neg active_B \vee pc_B = 3)\rangle, \langle q_1, false\rangle$ }
cs_A;
{7: $\langle q_0, active_A \wedge (turn = A \vee \neg active_B \vee pc_B = 3)\rangle, \langle q_1, false\rangle$ }
$active_A := false$

od

//

B :: **do** $true \rightarrow$ {1: $\langle q_0, true\rangle, \langle q_1, false\rangle$ }
ncs_B;
{2: $\langle q_0, true\rangle, \langle q_1, false\rangle$ }
$active_B := true$;
{3: $\langle q_0, active_B\rangle, \langle q_1, false\rangle$ }
$turn := A$;
{4: $\langle q_0, active_B\rangle, \langle q_1, false\rangle$ }
do $active_A \wedge turn = A \rightarrow$ {5: $\langle q_0, active_B\rangle, \langle q_1, false\rangle$ }
skip od;
{6: $\langle q_0, active_B \wedge (turn = B \vee \neg active_A \vee pc_A = 3)\rangle, \langle q_1, false\rangle$ }
cs_B;
{7: $\langle q_0, active_B \wedge (turn = B \vee \neg active_A \vee pc_A = 3)\rangle, \langle q_1, false\rangle$ }
$active_B := false$

od

coend.

Figure 5.1. Property Outline for I

It is easy to see that MEP does not satisfy $\mathcal{NS}_{MEP}$ for some histories in which one or the other process is not given sufficient opportunity to execute. A *fairness assumption* asserts that an atomic action that is enabled "often enough" will be executed eventually. Let $\mathcal{F}_\alpha$ be a fairness assumption for atomic action α.[8] A fairness assumption for a program Π is the conjunction of fairness assumptions—one for each atomic action in $\mathcal{A}_\Pi$. Thus, $\mathcal{F}_{MEP}$ the fairness assumption for MEP is defined by

[8]See [Francez 86] for various types of fairness assumptions.

$\mathcal{F}_{MEP}: \bigwedge_{\alpha \in \mathcal{A}_{MEP}} \mathcal{F}_\alpha.$

To show that MEP satisfies $\mathcal{NS}_{MEP}$ for all histories satisfying $\mathcal{F}_{MEP}$, we must prove that MEP satisfies $\mathcal{F}_{MEP} \Rightarrow \mathcal{NS}_{MEP}$. Putting this property into conjunctive normal form, we get

$$\mathcal{FNS}_{MEP}: \Big(\bigvee_{\alpha \in \mathcal{A}_{MEP}} \neg\mathcal{F}_\alpha \vee \Box\Diamond\neg try_A\Big) \wedge \Big(\bigvee_{\alpha \in \mathcal{A}_{MEP}} \neg\mathcal{F}_\alpha \vee \Box\Diamond\neg try_B\Big)$$

Each of the two clauses (conjuncts) in $\mathcal{FNS}_{MEP}$ is proved separately. However, since the two clauses are symmetric, only the first one is proved here; the proof of the second is similar.

As a fairness assumption for atomic actions, we choose *weak fairness*, which asserts that an atomic action that becomes enabled is eventually executed or otherwise becomes disabled. This is expressed in temporal logic as

$$\mathcal{WF}_\alpha: \Box\Diamond\neg enabled(\alpha).$$

where $enabled(\alpha)$ holds in any program state s where atomic action α is enabled. Thus, the first conjunct of $\mathcal{FNS}_{MEP}$ is

$$\begin{aligned} \mathcal{FNS}^A_{MEP}: &\bigvee_{\alpha \in \mathcal{A}_{MEP}} \neg\mathcal{WF}_\alpha \vee \Box\Diamond\neg try_A \\ = &\bigvee_{\alpha \in \mathcal{A}_{MEP}} \neg\Box\Diamond\neg enabled(\alpha) \vee \Box\Diamond\neg try_A \end{aligned}$$

This clause contains seventeen disjuncts because $|\mathcal{A}_{MEP}| = 16$; all but one of these disjuncts is negated. Each disjunct is of the form $\Box\Diamond\neg p$, so the Buchi automaton specification for each is m_{i_o} of Figure 3.2.

To prove that $\mathcal{FNS}^A_{MEP}$ holds we use the following proof instruments.

$$I = JS(\mathcal{FNS}^A_{MEP}, MEP)$$

$$v = \begin{cases} \infty & \neg try_A \\ pc_A - 3 & try_A \wedge try_B \wedge turn = A \\ 2 + (4 - pc_B) \bmod 7 & try_A \wedge \neg try_B \\ 7 + pc_B - 3 & try_A \wedge try_B \wedge turn = B \end{cases}$$

$$u = \begin{cases} \varnothing & \neg try_A \\ \{\, i \mid enabled(\alpha_i^A)\} & try_A \wedge try_B \wedge turn = A \\ \{\, i \mid enabled(\alpha_i^B)\} & try_A \wedge (\neg try_B \vee turn = B) \end{cases}$$

Rather than checking that verification conditions VC1 and VC2 hold, we argue informally that obligations O1–O4 are satisfied by this choice of proof instruments.

O1 and O2 are trivially satisfied by the choice of invariant, since the invariant rules out no joint state.

O3 follows from the construction of the variant function, as follows. When $\neg try_A$ holds, then an automaton for a positive disjunct is in an accepting state, so O3 is satisfied in that case. If $try_A \wedge try_B \wedge turn = A$, then v is the number of atomic actions that A must execute before entering cs_A. Since $turn = A$, subsequent execution by B does not alter this value. If $try_A \wedge \neg try_B$ then v includes the number of atomic actions B must execute to establish the previous case $(try_A \wedge try_B \wedge turn = A)$. Clearly, execution by A does not alter this value. Finally, when $try_A \wedge try_B \wedge turn = B$, then v is the maximum value of v in the preceding case $(try_A \wedge \neg try_B)$ plus the number of atomic actions that B must execute to make $try_A \wedge \neg try_B$ hold.

O4 follows from the construction of the candidate function, which always contains the index of the automaton corresponding the fairness assumption for the enabled atomic action that will reduce v.

Note that this proof is an instance of the method of helpful directions for weak fairness [Francez 86]. Each process corresponds to a direction. Execution of an enabled atomic action in the helpful direction decreases the variant function. Execution of an enabled atomic action in some other direction does not increase the variant function nor does it disable any helpful atomic action. Thus, when $try_A \wedge try_B \wedge turn = A$, the helpful direction is process A; when $try_A \wedge (\neg try_B \vee turn = B)$ the helpful direction is process B. The weak fairness assumption guarantees that the variant function decreases eventually.

The method of helpful directions is a special case (for proving properties that assume weak fairness) of the technique given in this paper. To see this, observe that the helpful directions identify atomic actions that decrease the variant function. The stipulation that a non-helpful atomic action not increase the variant function guarantees O3. The value of the candidate function can be taken to correspond to any subset of the helpful atomic actions. The stipulation that a non-helpful atomic action leave helpful ones enabled guarantees O4.

6. Discussion

We have shown how to reduce a temporal property into proof obligations that can be formulated as formulas of predicate logic and Hoare's logic. The idea that temporal properties can be proved without temporal logic is not new. For example, [Manna & Pnueli 79] point out that it is possible to prove temporal properties using a partially interpreted first-order logic with operators that correspond roughly to the right-hand sides of the definitions of the temporal modalities. The use of invariance and well foundedness for proving temporal properties is suggested in [Lehmann et al. 81] and [Manna & Pnueli 83a]. And [Manna & Pnueli 84] advocate using temporal logic along with invariance and well-foundedness. What *is* new in this paper is a systematic method for reducing a temporal property to non-temporal proof obligations.

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 temporal properties are reduced to primitive formulas that resemble triples. That work, however, is concerned only with finite sequences (both as properties and programs) and therefore does not address the problem we are most concerned with.

Another, related, approach to 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* $\mathcal{K}_\Pi$. $\mathcal{K}_\Pi$ is a model for a temporal property $\mathcal{P}$ if and only if Π satisfies $\mathcal{P}$. To determine if Π satisfies $\mathcal{P}$ it suffices to check whether $\mathcal{K}_\Pi$ is a model for $\mathcal{P}$, and this amounts to checking each state in the state space to see which sub-formulas of $\mathcal{P}$ hold in that state. Thus, for programs with finite state spaces, it is possible to mechanically verify whether the program satisfies a given temporal property.

Recently, [Vardi & Wolper 86] observed that for programs with finite state spaces, $\mathcal{K}_\Pi$ can be viewed as a Buchi automaton that accepts exactly the histories of Π. From this automaton and one that recognizes sequences satisfying $\neg\mathcal{P}$, a Buchi automaton $m_{\Pi\wedge\neg\mathcal{P}}$ can be constructed that accepts all histories of Π not satisfying $\mathcal{P}$. The decision procedure for the emptyness problem for $m_{\Pi\wedge\neg\mathcal{P}}$ can then be used to determine if Π satisfies $\mathcal{P}$. A similar approach was developed independently by Kurshan [Kurshan 87a] [Kurshan 87b].

Model checking is restricted to programs with finite state spaces[9] but is algorithmic. Since it is algorithmic, it can be mechanized and does not require creativity in devising invariants, variant functions, or candidate functions. And, model checking is always guaranteed to get the correct answer. In contrast, the methods presented in this paper are not limited to finite-state programs. Unfortunately, the methods are, in general, undecidable. Moreover, they may require creativity in devising suitable proof instruments, although this might be viewed as an asset since the proof instruments can give insight into why a program works.

[9]However, proponents of the model checking approach have made progress in weakening the finite state assumption so that it applies only to certain key parts of the program [Clarke & Grumberg 87] [Sistla & German 87].

The first Buchi automaton based method for extracting first-order proof obligations for temporal properties was proposed by us in [Alpern & Schneider 85] [Alpern 86]. That work applied to those properties that can be specified using a single deterministic Buchi automaton. Formulated in the terminology of this paper, the method requires the program prover to exhibit an invariant I and a variant function v satisfying:

AS1: For all $x \in JS(\mathcal{P}, \Pi)$: $(x^{(1)}{=}q_{10} \wedge x^{(\Pi)} \in Init_{\Pi}) \Rightarrow x \in I$

AS2: For all $x, y \in JS(\mathcal{P}, \Pi)$: $x \rightarrow y \Rightarrow y \in I$

AS3: For all $x, y \in JS(\mathcal{P}, \Pi)$: $x \rightarrow y \Rightarrow (Pos(y) \vee v(y) < v(x))$

AS1 and AS2 are obligations O1 and O2 of the approach outlined in Section 4. Consider the remaining obligations (O3 and O4) of that approach. The property is a single clause that consists of a single, non-negated property, so there are no negative automata. Thus, $u(x) = \varnothing$ for every joint state x, so the final disjunct of O4 must be false. AS3 and O4 are therefore equivalent, and each implies O3. Thus, the two techniques yield essentially the same proof obligations when applied to properties that they both can handle.

The method in [Alpern & Schneider 85] is unsatisfactory for properties specified by non-deterministic Buchi automata. To use it to prove that a program Π satisfies such a property $\mathcal{P}$, a deterministic property $\mathcal{D}$ that is contained in $\mathcal{P}$ is found. Proof obligations are then extracted from the deterministic Buchi automaton for $\mathcal{D}$. If a (finite-state) program Π satisfies $\mathcal{P}$, an appropriate $\mathcal{D}$ always exists but may be big and difficult to find. Furthermore, the proof obligations for a non-deterministic property now depend on the program as well as on the Buchi automaton for the property to be proved. The approach of Section 4 does not suffer from these difficulties since every property that can be specified using a non-deterministic Buchi automaton can be specified as a Boolean combination of properties specified by deterministic ones [Eilenberg 74].

In [Manna & Pnueli 87], Manna and Pnueli concurrently and independently developed a different technique for extending the approach of [Alpern & Schneider 85] to obtain proof obligations for properties specified by non-deterministic Buchi automata. The approach is based upon a $\forall$-automaton for a property. Simplifying slightly,[10] a $\forall$-automaton is a Buchi automaton that accepts its input iff every run on that input eventually is restricted to accepting states. Using the parlance of Section 4, to show that every history of a program will be accepted by a $\forall$-automaton m, one must exhibit an invariant I that satisfies obligations O1 and O2 and a variant function v satisfying:

---

[10]We are ignoring the behavior of $\forall$-automata on finite sequences, the placement of transition predicates in states rather than on edges, and $\forall$-automata with recurrent states (which are shown to be equivalent $\forall$-automaton without such states in [Manna & Pnueli 87]).

MP1: For all $x, y \in JS(\mathcal{P}, \Pi)$: $x \rightarrow y \Rightarrow v(y) \leq v(x)$

MP2: For all $x, y \in JS(\mathcal{P}, \Pi)$: $(x \rightarrow y \wedge x^{(1)} \notin A_1) \Rightarrow v(y) < v(x)$

The $\forall$-automaton for a property is isomorphic to the Buchi automaton for the negation of that property.[11] This suggests there might be a connection between the proof obligations that are obtained from a $\forall$-automaton for the negation of a deterministic property and the proof obligations we obtain for a clause with a single negated property. And, there is. Since there are no positive joint states with a single negated property, O3 and MP1 are equivalent. We can choose the candidate function such that O4 and MP2 are equivalent—define u to be empty whenever the Buchi automaton is in an accepting state and to be $\{1\}$ when it is not. Therefore, the two techniques yield the same proof obligations for a property whose negation can be specified by a deterministic Buchi automaton.

The key insight underlying the approach of [Manna & Pnueli 87] is that proof obligations for the negation of a property can be extracted directly from a Buchi automaton for the property—whether or not this automaton is deterministic.[12] Given a property specified by a Buchi automaton, to extract proof obligations using the approach of [Manna & Pnueli 87], the Buchi automaton for the negation of the property is constructed. With the technique presented in this paper, the property is decomposed into a Boolean combination of properties where the non-negated terms must be specified by deterministic Buchi automata. Depending on the property, one or the other approach may be easier. An added advantage of our Boolean decomposition approach is that parts of the proof may be reusable since other properties might be constructed from these parts.

A final insight into the difference between the approach of [Manna & Pnueli 87] and our earlier approach in [Alpern & Schneider 85] is obtained by considering clauses of the form $\mathcal{N} \Rightarrow \mathcal{M}$, as was done in section 4.3. The technique in [Alpern & Schneider 85] is a restriction to the special case *true* $\Rightarrow \mathcal{M}$ and the technique in [Manna & Pnueli 87] treats the other special case, $\mathcal{N} \Rightarrow$ *false*. Of these "special cases", the second is general; the first is not.

7. Summary

We have described an approach to proving temporal properties of concurrent programs. This approach is based on using deterministic Buchi automata to specify properties. Such automata are quite expressive—any temporal property can be formulated as a Boolean combination of properties specified by them. Proof obligations for a property are extracted directly from the automata for that property. These proof obligations are discharged by devising suitable proof instruments. The

[11] To obtain a $\forall$-automaton for $\mathcal{P}$ from a Buchi automaton for $\neg\mathcal{P}$, exchange the accepting and non-accepting states.

[12] Consequently, when the approach of section 4 is used, the negative automata of $\mathcal{P}_i$ need not be deterministic.

adequacy of the proof instruments is established by verifying predicate logic formulas and triples. Thus, temporal inference is not necessary for proving temporal properties. The same techniques that prove total correctness of sequential programs can prove arbitrary temporal properties of concurrent ones.

Acknowledgments

D. Gries, N. Klarlund, L. Lamport, P. Panangaden, and A. Pnueli made helpful comments on an earlier draft of this paper. We are also grateful to all three referees, who provided extremely useful criticisms on the two earlier versions of this paper. Discussions with Starvos Cosmadakis and Sheryl Brady helped in formulating the completeness proof.

References

[Alpern 86] Alpern, B. Proving Temporal Properties of Concurrent Programs: A Non-temporal Approach. Ph.D. Thesis. Department of Computer Science, Cornell University. Feb. 1986.

[Alpern & Schneider 85] Alpern, B. and F.B. Schneider. Verifying Temporal Properties without using Temporal Logic. Technical Report TR 85-723, Department of Computer Science, Cornell University, Dec. 1985.

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

[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. of the 10th ACM Symposium on Principles of Programming Languages*, Austin, January 1983, 117-126.

[Clarke & Grumberg 87] Clarke, E.M. and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. *Proc. of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 1987, 294-303.

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

[Emerson & Sistla 83] Deciding branching time logic: A triple exponential decision procedure for CTL*. *Logics of Programs*, Lecture Notes in Computer Science, Vol. 164, Springer-Verlag, Berlin, 1983, 176-192.

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

[Francez 86] Francez, N. *Fairness*. Texts and Monographs in Computer Science, Springer-Verlag, Berlin, 1986.

[Gerth 84] Gerth, R. Transition logic. *Proc. Sixteenth Annual Symposium on Theory of Computing*, Washington, D.C., April 1984, 39-50.

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

[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.

[Kurshan 87a] Kurshan, R. Complementing deterministic Buchi automata in polynomial time. *JCSS 35* (August 1987), 59-71.

[Kurshan 87b] Kurshan, R. Reducibility in analysis of coordination. *Discrete Event Systems, Lecture Notes in Control and Information Sciences* IIASA, Vol. 103 (1987), 19-39.

[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 & Schneider 84] Lamport, L. and F.B. Schneider. The 'Hoare Logic' of CSP, and All That. *ACM Transactions on Programming Languages and Systems 6*, 2 (April 1984), 281-296.

[Lehmann et al. 81] Lehmann, D., A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. *Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 115, Springer-Verlag, Berlin, 1981, 264-277.

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

[Manna & Pnueli 79] The modal logic of programs. *Proc. 6th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 71, Springer-Verlag, Berlin, 1979, 385-409.

[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*, Lecture Notes in Computer Science, Vol. 131, Springer-Verlag, Berlin, 1981, 200-252.

[Manna & Pnueli 83a] Manna, Z. and A. Pnueli. Verification of concurrent programs: A temporal proof system. In *Foundations of Computer Science IV, Distributed Systems: Part 2*, J.W. DeBakkar, J. Van Leuwen (eds.). Mathematical Centre Tracts 159. Amsterdam, 1983, 163-255.

[Manna & Pnueli 83b] Manna, Z, and A. Pnueli. How to cook a temporal proof system for your pet language. *Proc. of the Symposium on Principles of Programming Languages*, ACM, Austin, Jan. 1983.

[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 & Pnueli 87] Manna, Z. and A. Pnueli. Specification and Verification of Concurrent Programs by ∀-Automata. *Proc. 14th Symposium Principles of Programming Languages*, ACM, Munich, Jan. 1987, 1-12.

[Misra et al. 82] Misra, J., K.M. Chandy, and T. Smith. Proving safety and liveness of communicating processes with examples. *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982, 157-164.

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

[Owicki & Lamport 82] Owicki, S.S. and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems 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.

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

[Sistla & German 87] Sistla, A.P. and S.M. German. Reasoning with many processes. *Proc. Symposium on Logic in Computer Science* Ithaca, New York, June 1987, IEEE, 138-152.

[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, August 1984.

[Vardi & Wolper 86] Vardi, M.Y. and P. Wolper. An automata-theoretic approach to automatic program verification. *Proc. Symposium on Logic in Computer Science*, Boston, Mass, June 1986, IEEE.

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