Proving Boolean Combinations of Deterministic Properties

Bowen Alpern
Watson Research Center
Box 216
Yorktown Heights, NY 10598

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract This paper gives a method for proving that a program satisfies a temporal property that has been specified in terms of Buchi automata. The method permits extraction of proof obligations for a property formulated as the Boolean combination of properties, each of which is specified by a deterministic Buchi automaton, directly from the individual automata. The proof obligations can be formulated as Hoare triples. The method is proved sound and relatively complete. A simple example illustrates application of the method.

1. Introduction

Over the past 20 years, there has been increasing interest in ways to deduce properties of program behavior from the program text itself. One technique that is attracting attention is based on specifying a temporal property \( P \) with a Buchi automata that accepts every sequence of program states satisfying \( P \).\(^1\) Proof obligations are then extracted from this automaton. These obligations are generalizations of the invariant and variant function used to prove partial correctness and termination of a while loop. They define verification conditions that must hold for any program satisfying \( P \). Since the verification conditions themselves are formulated as Hoare triples [Hoare 69], all reasoning can be done in first-order predicate logic, and temporal logic is not required.

Extraction of proof obligations from Buchi automata was first proposed in [Alpern & Schneider 85] for deterministic properties—those properties that can be specified by deterministic Buchi automata. Subsequently, [Manna & Pnueli 87] generalized the approach to handle properties that can be specified by non-deterministic Buchi automata. Both [Alpern & Schneider 85] and [Manna & Pnueli 87], however, treat a property as a monolithic entity. Doing so has some practical disadvantages. First, the Buchi automaton specifying a property that is the Boolean combination of simpler properties can be exponentially bigger than the combined size of the automata specifying its components [Sistla et al 85]. Since the amount of work required to prove correctness is linear in the size of the automata specifying the property being proved, working with the combined automata can be impractical. Second, combining properties requires complex automata constructions and simplifying properties requires automata transformations, both of which are error prone processes. By contrast, when, for example, temporal logic is used to specify properties, logical connectives permit properties to be combined and the laws of logic allow properties to be simplified.

This paper shows how to avoid these disadvantages by giving a way that proof obligations for a property formulated as the Boolean combination of deterministic properties can be extracted directly from the automata specifying those components. The method also unifies the approaches in [Alpern & Schneider 85] and [Manna & Pnueli 87]. The method of [Alpern & Schneider 85] extracts obligations for \( P \) from a deterministic Buchi automata that accepts \( P \); the method of [Manna & Pnueli 87] is equivalent to extracting obligations for \( \neg P \) from a Buchi automata that accepts \( P \).

In our new approach, a property is presented as a conjunction of clauses, each of which is proved separately and is formulated as the disjunction of deterministic properties and negations of deterministic properties. To prove that a program satisfies a clause, three proof instruments must be exhibited. An invariant handles the safety aspects of the proof; a variant function handles the liveness aspects; and a candidate function, arbitrates among the automata specifying the disjunctions of a clause to ensure that any program execution will be accepted by at least one automata representing a deterministic property or rejected by at least one automata representing the negation of a deterministic property. The three instruments define verification conditions, which can be formulated as Hoare triples.

In section 2, a simple concurrent programming language and its semantics is defined. Section 3 reviews Buchi automata for specifying properties. Proof instruments for a single clause are discussed in section 4, sound-
ness and completeness proofs are also given there. The method is illustrated in section 5 by proving that assuming weak fairness, Peterson’s protocol for mutual exclusion [Peterson 81] satisfies a non-starvation property. Section 6 relates our approach to other techniques for proving properties of concurrent programs.

2. Programs

A program π is specified by a predicate Initn describing possible initial states and a set of atomic actions An. An execution of π is modeled by an infinite sequence of program states, called a history, in which the first state satisfies Initn and every subsequent state results from executing an element of An 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 processes as they are executed.

An atomic action α defines a set of pairs of program states. Execution of α is enabled in a state s, denoted enabled(α, s), provided (3 t: (s, t) ∈ α). The statement

(∈ b → S fi)

specifies an atomic action containing those elements (s, t) such that b holds on s and t is the state produced by executing S starting in state s.

Rather than enumerating the atomic actions of a program directly, it is frequently convenient to annotate a program with control points that delimit its atomic actions. We will denote control points by marking and numbering them. For example, program fragment of process π

.. (3) S (4) ..

defines a single atomic action

(∈ pC = 3 → pC = 4; S fi)

where pC simulates the program counter.

All histories are assumed to be infinite. One way to achieve this is by including in An an atomic action that has no effect on the program state and is enabled when no other atomic action is. Thus, terminating executions are extended to infinite histories by repeating the final state.

An example program MEP is shown in Figure 2.1. It is a solution to the two-process critical section problem described in [Peterson 81]. Based on the control point annotations, we obtain set of atomic actions A MEP of Figure 2.2. In those atomic actions, variable pC simulates the program counter for process A and variable pIB simulates the program counter for process B. Finally, we have

Init MEP: pC = 1 ∧ pIB = 1

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

3. Specifying Properties using Buchi Automata

A Buchi automaton is a finite state machine that accepts infinite sequences. Since a property of a program can be viewed as a definition of a set of infinite sequences of program states, a Buchi automaton m can be used to specify the property containing exactly those sequences accepted by m. Mechanical procedures exist to translate linear-time first-order temporal formulas into Buchi automata where state transitions are defined in terms of first-order program state predicates [Wolper et al 83] [Alpern 86]. Moreover, Buchi automata have natural diagrammatic representations and this is sometimes convenient.

MEP: cobegin
A := do true → [1]: ncsA;
[2]: activeA := true;
[3]: turn := B;
[4]: do activeA ∧ turn = A → [5]: skip od;
[6]: csA;
[7]: activeA := false
od;

B := do true → [1]: ncsB;
[2]: activeB := true;
[3]: turn := A;
[4]: do activeB ∧ turn = B → [5]: skip od;
[6]: csB;
[7]: activeB := false
end
cend

Figure 2.1. Peterson’s Mutual Exclusion Protocol

A MEP = {α1:
(∈ pC = 1 → pC := 2; ncsA fi),
α2:
(∈ pC = 2 → pC, activeA := 3, true fi),
α3:
(∈ pC = 3 → pC, turn := 4, B fi),
α4:
(∈ pC = 4 ∧ activeA ∧ turn = B → pC := 5 fi),
α5:
(∈ pC = 5 → pC := 4 fi),
α6:
(∈ pC = 6 → pC := 7; csA fi),
α7:
(∈ pC = 7 → pC, activeA := 1, false fi),
α8:
(∈ pC = 1 → pC := 2; ncsB fi),
α9:
(∈ pC = 2 → pC, activeB := 3, true fi),
α10:
(∈ pC = 3 → pC, turn := 4, A fi),
α11:
(∈ pC = 4 ∧ activeB ∧ turn = A → pC := 5 fi),
α12:
(∈ pC = 5 → pC := 6 fi),
α13:
(∈ pC = 6 → pC := 7; csB fi),
α14:
(∈ pC = 7 → pC, activeB := 1, false fi)
}

Figure 2.1. Atomic Actions A MEP

Figure 3.1 is an example of a Buchi automaton m. It accepts an infinite sequence in which after a finite prefix each state satisfies the program state predicate P. In temporal logic, this property is given by ∃qP. Automaton m contains three automaton states labeled q0, q1, and q2. The start state, q0, is denoted by an edge with no origin, and accepting state, q1, is denoted by two concentric circles. A Buchi automaton accepts a sequence α if and only if it enters an accepting state infinitely often while reading α. Notice, there is no way in m to get from q2 to an accepting state. Such states are called dead states. If an automaton is in a dead state, it cannot accept its input.
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 that edge is defined for that program state. For example, because there is an edge labeled $P$ from $q_0$ to $q_1$ in $m_{se}$ whenever $m_{se}$ 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 an edge defined from each automaton state for each input symbol.

In order to define Buchi automata formally, the following notation will be useful. For any sequence $\sigma = s_0 s_1 \ldots$, $\sigma[i] = s_i$, $\sigma[..] = s_0 s_1 \ldots s_i$, $\sigma[i..] = s_i s_{i+1} \ldots$, $|\sigma| = \text{the length of } \sigma$ ($\omega$ if $\sigma$ is infinite), $INF(\sigma) = \{ s \in \sigma \text{ appears infinitely often in } \sigma \}$.

A Buchi automaton $m$ for a property of a program $\pi$ is a five-tuple $(S_m, Q, Q_0, A, \delta)$, where $S_m$ is the (countable) set of program states of $\pi$, $Q$ 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 $m$, $\delta : (Q \times S) \rightarrow 2^Q - \emptyset$ is the transition function of $m$.

Transition predicate $T_\pi$ associated with the edge from automaton state $q_i$ to $q_j$ holds for a program state $s$ if and only if $q_j \in \delta(q_i, s)$. (Although $S_m$ may be infinite, we only consider properties for which the transition predicates have finite description as a first-order formula involving program variables.) Transition function $\delta$ can be extended to handle finite sequences of program states in the usual way:

$$\delta(q, \sigma) = \begin{cases} \{q\} & \text{if } |\sigma| = 0 \\ \{q\} \cup \exists q' : \sigma ' \in \delta(q, s_{|\sigma|-1}) : \\ q ' \in \delta(q', s_{|\sigma|-1}) & \text{if } q < |\sigma| < \omega \end{cases}$$

A sequence of automaton states that $m$ might occupy while reading $\sigma$ is called a run. Thus, $\rho$ is a run of $m$ on $\sigma$ if and only if

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

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

Notice in $m_{se}$ (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, $m_{se}$ is non-deterministic. However, $L(m_{se})$ the language accepted by $m_{se}$ is the negation of the deterministic property $\square \neg P$ accepted by $m_{se}$ of Figure 3.2.

4. Proof Instruments from Buchi Automata

Let property $P$ be the conjunction of clauses $P_1, P_2, \ldots$, $P_n$. To prove that a program $\pi$ satisfies $P$, we prove separately that $\pi$ satisfies each of the clauses. Thus, it suffices to derive proof instruments for a single clause $P_i$ of the form $D_1 \lor \ldots \lor D_p \lor \neg D_{p+1} \lor \ldots \lor \neg D_{p+s}$, where each $D_k$ is specified by $m_k$, a deterministic Buchi automaton. Any property that can be specified by a non-deterministic Buchi automaton can be put in this form [Eilenberg 74]. We call $m_i$, through $m_n$ the positive automata of $P_i$ and $m_{p+1}$ through $m_{p+s}$, the negative automata. Let $m_b$ be defined by $(S_b, Q_b, (q_0, A_1, A_2))$.

To formulate proof instruments for $\pi$ and $P$, define a joint state $x$ to be an element of $JS(P_i, \pi)$ the joint state space $Q_i \times \ldots \times Q_{p+s} \times \sigma_\pi$. Let $x^{(k)}$ denote the $k^{th}$ element of a joint state $x$ and let $x^{(k+1)}$ abbreviate $x^{(k+1)}(x^{(k)}(A_1))$. A joint state $x$ is positive, denoted Pos$(x)$, iff for some positive automata $m_j$, $x^{(j)}(A_1)$. Thus, $x$ is positive if and only if $m_j$ is in an accepting state. The set of negative automata that are in an accepting state in a joint state $x$ is $neg(x) = \{ k | 1 \leq k \leq p + s \land x^{(k)}(A_1) \}$. For a sequence $\rho$ of joint states, define $rho^{(k)}$ to be the projection of $\rho$ onto its $k^{th}$ element; i.e., $rho^{(k)} = rho[0][k][1][k]$. A sequence of joint states $\rho$ is a joint history of $\pi$ and $P$ if $\rho^{(k)}$ is a history of $\pi$ and for all $1 \leq k \leq p + s$, $rho^{(k)}$ is a run of $m_b$ on $\rho^{(k)}$.

A program $\pi$ satisfies the property specified by a clause $P_i$, iff every history $\sigma$ of $\pi$ is either accepted by one of the positive automata—and therefore satisfies one of the properties $D_1$ through $D_p$ or is rejected by one of the negative automata—and therefore satisfies one of the $\neg D_{p+1}$ through $\neg D_{p+s}$. Thus, $\pi$ satisfies $P_i$, iff for every history $\sigma$ there is a joint history $\rho$ such that $\sigma = \rho^{(k)}$ and either (i) there is a positive automaton $m_j$ such that $INF(\rho^{(k)}) \cap A_2 = \emptyset$ or (ii) there is a negative automaton $m_k$ such that $INF(\rho^{(k)}) \cap A_2 = \emptyset$.

To prove that $\pi$ satisfies the property specified by a clause $P_i$, three proof instruments must be exhibited:

- an invariant $I \subseteq JS(P_i, \pi)$,
- a candidate function $u : JS(P_i, \pi) \rightarrow 2^{(p+1..p+s)}$, and
- a variant function $v : JS(P_i, \pi) \rightarrow W$, where $W$ is a well-founded set.

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 following obligations, then $\pi$ will satisfy $P_i$. In these obligations, $x$ and $x'$ denote elements of $JS(P_i, \pi)$.
The first two obligations ensure that \( I \) holds throughout any joint history.

O1: If \( x = \rho[0] \) for some joint history of \( w \) and \( P \), then \( x \in I \).

O2: If \( x \in I \) and \( (x^n, x^{n+1}) \in q \) then \( x' \in I \).

O1 constitutes the base case and O2 induction step for an argument that execution of each atomic action produces a joint state in \( I \). Choosing an invariant \( I \) permits unreachable joint states to be ignored in the obligations that follow.

The next obligation requires that a variant function \( v \) increase only in or immediately after processing a joint state in which a positive automata is in an accepting state.

O3: If \( x \in I \) and \( (x^n, x^{n+1}) \in q \) then
\[ Pos(x) \lor Pos(x') \lor v(x) < v(x') \lor \emptyset \subseteq u(x') \subseteq u(x). \]

Thus, O3 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.

Since the range of \( v \) is a well-founded set, if \( v \) never increases then eventually its value must become constant. The next obligation implies that under these circumstances, throughout the rest of the history, the candidate function is a non-empty set and contains the index of some negative automaton \( m_k \).

O4: If \( x \in I \) and \( (x^n, x^{n+1}) \in q \) then
\[ Pos(x) \lor Pos(x') \lor v(x) < v(x') \lor \emptyset \subseteq u(x') \subseteq u(x). \]

The final obligation ensures that a negative automata that enters an accepting state cannot be a candidate unless a positive state is entered or the variant function decreases. Thus, if no positive state is entered and the variant function does not decrease then \( m_k \) does not enter an accepting state for the remainder of the history, hence it rejects the history.

O5: If \( x \in I \) and \( (x^n, x^{n+1}) \in q \) then
\[ Pos(x) \lor Pos(x') \lor neg(x') \cap u(x') = \emptyset. \]

Since \( m_k \) rejects the history, the history satisfies \( \neg D_k \) and therefore satisfies \( P \).

We can now prove:

Soundness Theorem: If there are proof instruments \( I, v \), and \( u \) that satisfy obligations O1 through O5 then \( \pi \) satisfies \( P \).

Proof: Let \( \rho \) be a joint history of \( \pi \) and \( P \), \( \rho[0] \in I \) by O1. By O2 induction, \( \rho[j] \in I \) for all \( j \geq 0 \). We must show that \( \rho \) 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 \( l_i \) such that \( \rho[l_i] \) contains no positive states. By O3, the variant function is non-increasing on \( \rho[l_i] \). Since its range is well-founded, there must be an index \( l_2 \) such that \( l_2 \leq l_i \) and the variant function is constant on \( \rho[l_2] \). By O4, there is a negated automaton \( m_k \) such that \( k \in u[l_2] \). Also by O4, for all \( l_2 < l, k \notin u[l] \). Therefore, by O5, \( m_k \) does not enter an accepting state after \( l_2 \). This means that \( m_k \) rejects \( \rho \), so \( \pi \) satisfies \( P \).

We now show that our technique is relatively complete.

Completeness Theorem: If \( \pi \) satisfies \( P \) then there exist proof instruments \( I, v \), and \( u \) that satisfy obligations O1 through O5.

Proof: Form a directed graph where the nodes are the joint states appearing in joint histories 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 partial order \( \Rightarrow \) on the nodes of this graph such that \( x \Rightarrow y \) iff there is a path with non-zero length from \( x \) to \( y \) and somewhere on this path appears an accepting state for each negative automaton.

Partial order \( \Rightarrow \) is well-founded, as is shown by the following proof by contradiction. If \( \Rightarrow \) were not well-founded then there would be an infinite descending chain \( x_1 \Rightarrow x_2 \Rightarrow \ldots \). By construction of the graph, this implies the existence of a joint history that includes \( x_1 \). Let \( \sigma_0 \) be a prefix of such a joint history that ends with \( x_1 \). For each \( x_i \) in the infinite descending chain, let \( \sigma_i \) 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 \( x_i \Rightarrow x_{i+1} \). Finally, let \( \sigma \) be \( \sigma_0 \sigma_1 \ldots \). Notice that \( \sigma \) is a joint history, that \( \sigma \) contains no positive accepting states after \( \sigma_0 \) and that there are infinitely many accepting states for each negative automaton in \( \sigma \). Thus, \( \sigma \) does not satisfy \( P \). This is a contradiction, and we conclude that \( \Rightarrow \) is well-founded.

Since \( \Rightarrow \) is well-founded, the following recursive function is well-defined:
\[ H(x) = \sup\{H(y) + 1 \mid x \Rightarrow y \}. \]

\( H \) is a function from the joint state space to the ordinals. If there is no \( y \) such that \( x \Rightarrow y \) then \( H(x) = 0 \) by definition of \( \sup \), so the function is total. Notice that if \( x \Rightarrow y \) 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 \( l(x) = i \) iff for any collection of \( i \) and no more than \( i \) negative automata, there exists some node \( w \) with \( H(w) = H(x) \) such that there is a path from \( x \) to \( w \) 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 \( n \).

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 \( \pi \) and \( P \). Choose \( u(x) \) to be \( \{H(x), n - l(x)\} \). The range of \( v \) with lexicographic ordering of pairs as the ordering relation 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 from \( w \) to \( x \) for any \( w \neq x \) such that \( u(w) = u(x) \).

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

To see that O3 holds, notice that if neither \( x \) nor \( x' \) is positive then there is a path (of length 1) from \( x \) to \( x' \) in the graph. Thus, \( H(x) = H(x') \). Suppose \( H(x) = H(x') \). Since any path to \( x \) can be extended to \( x' \), \( l(x) \leq l(x') \). Therefore, \( u(x) \supseteq u(x') \).

To see that O4 holds, suppose that neither \( x \) nor \( x' \) is positive and that \( u(x) = u(x') \). By construction of \( u(x) \), both \( x \) and \( x' \) must be at the same level. By the construction of \( u(x) = u(x') \), since there is a path of
length 1) from \( x \) to \( x' \). Also by construction of \( u, u(x') \in \emptyset. \)

If there exists a node \( w \) such that \( u(w) = u(x') \) and there is a path from \( w \) to \( x' \) then \( \text{neg}(x') \cap u(x') = \emptyset \) since \( x' \) is (trivially) included on the path to \( x' \). If there is no such \( w \) then \( u(x) > u(x'). \) So, 05 holds. This completes the proof.

Obligations 01 through 05 can be translated into verification conditions formulated as Hoare triples and first-order predicate logic formulas. A Hoare triple \( \{ P \} a \{ Q \} \) for an atomic action \( a \) asserts that any execution of \( a \) started in a state satisfying \( P \) terminates in a state satisfying \( Q. \)³ Thus, \( \{ P \} a \{ Q \} \) is valid iff \( \forall (v(s,t), (s,t) \in a; (s,t) \in Q) \).

To reformulate 01 through 05 in terms of Hoare triples, we define slight variations of the three proof instruments. Define \( PJS(P, \pi) \) the projection of \( JS(P, \pi) \) with respect to program states as \( PJS(P, \pi) = Q_1 \times \cdots \times Q_{\pi + 1}. \) Elements of \( PJS(P) \) are called projected joint states. The projected joint state in which every automaton is in its initial state is called the projected joint state. A projected joint state \( y \) is considered positive if it is the projection of a positive state. (Similarly, if \( y \) is a projection of a positive state then \( \text{neg}(y) = \text{neg}(x) \).) We now define projections of the three proof instruments with respect to projected joint states. For an invariant \( I, I_y \) is the projection of \( I \) with respect to a projected joint state \( y \); i.e., \( I_y = \{ s | s \in S_y \land y(s) \in I \} \). For a variant function \( u, \) projection \( u_y \) is a function from \( S_y \) to \( W \) such that \( u_y(s) = u(s) \) for \( s \in S_y \). And, for a candidate function \( u, \) define \( u_y \) such that \( u(y, s) = u(s) \) for \( s \in S_y \).

Finally, we define a projected joint state transition predicate \( T_y \) to be a predicate that holds for any program state causing a transition from a projected joint state \( y \) to a projected joint state \( z \). \( T_y \) can be formed by taking the conjunction for all \( m \) of the transition predicates labeling the edge from \( y^{(m)} \) to \( z^{(m)} \). \( T_y \) is the projected joint state transition predicate from the joint state to \( z \).

Verifying each of the following formulas implies that 01 through 05 hold. The first verification condition implies 01.

**VC1:** \( (\mu y, (T_y) \Rightarrow I_y) \)

The next implies 02 through 05.

**VC2:** For all projected joint states \( y \) and \( z \) and atomic actions \( \alpha \in \mathcal{A}, \)
(i) If \( \mu \) or \( \nu \) is positive then \( \{ I_y \} a \{ T_y \Rightarrow I_z \}. \)
(ii) Otherwise,
\[
\{ I_y \land u_y = \mu \land u_z = \nu \Rightarrow (T_y \Rightarrow \{ I_z \land (\mu < Y \lor (\nu = \mu \land \bigcirc C u_z \subseteq U \land \text{neg}(z)))\})
\]

02 is implied by \( T_y \Rightarrow (I_y \land (\mu < Y \lor (\nu = \mu \land \bigcirc C u_z \subseteq U \land \text{neg}(z)))) \).

03 through 05 follow from VC1 and VC2, choose
\( x = (y, s) \) and \( x' = (s, s') \).

and assume \( x \in I \) and \( (s, s') \in \alpha. \) It follows immediately from 02 that \( x \in I \).

We conclude that \( x \) satisfies \( T_y \) directly from the definition of \( T_y \). If \( y \) or \( z \) is positive, then 03, 04, and 05 are trivially satisfied. Suppose neither \( y \) nor \( z \) is positive and let \( \mu(x) < Y \) and \( u(z) = U \). From the postcondition of VC2(iii), it follows that either \( \mu(x') < Y \) or \( u(x') = V \). Thus, 03 holds. If \( \mu(x') < Y \) does not hold then, from the postcondition of VC2(iii), \( \bigcirc C u(z) \subseteq U \land u(x') \subseteq U \land \text{neg}(x') \). Thus, 04 and 05 hold.

Conversely, assume 01 through 05 hold. VC1 follows from 01. From 02 we get
\[
\{ I_y \land u_y = \mu \land u_z = \nu \Rightarrow (T_y \Rightarrow (\mu < Y \lor \bigcirc C u_z \subseteq U)). \}
\]

Thus, VC2(i) holds. Assume that neither \( y \) nor \( z \) is positive. From 03 we get
\[
\{ I_y \land u_y = \mu \land u_z = \nu \Rightarrow (T_y \Rightarrow (\mu < Y \lor \bigcirc C u_z \subseteq U \land \text{neg}(z))). \}
\]

From 04 we get
\[
\{ I_y \land u_y = \mu \land u_z = \nu \Rightarrow (T_y \Rightarrow (\mu < Y \lor \bigcirc C u_z \subseteq U \land \text{neg}(z))). \}
\]

Finally, from 05 we get
\[
\{ I_y \land u_y = \mu \land u_z = \nu \Rightarrow (T_y \Rightarrow (\mu < Y \lor \bigcirc C u_z \subseteq U \land \text{neg}(z))). \}
\]

Together, these four Hoare triples imply VC2(ii).

We have shown that obligations 01 through 05 are equivalent to verification conditions VC1 and VC2. If the underlying first-order predicate logic 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 underlying first-order predicate logic [Apt 81], our proof technique is complete relative to the semantic and expressive completeness of this logic.

### 5. Example: Proving Non-Starvation Assuming 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 (Figure 2.1) is said to starve if it tries to enter its critical section but never succeeds. To formalize this, let \( \rho \subseteq (p, \alpha) \) be a variable containing the value of the program counter for process \( P \). Process \( A \) of \( MEP \) should eventually enter \( \epsilon[A] \) whenever
\[
\text{TRY}_A : \rho[A] = 4 \lor \rho[A] = 5
\]
holds, and similarly process \( B \) should eventually enter \( \epsilon[B] \) whenever
\[
\text{TRY}_B : \rho[B] = 4 \lor \rho[B] = 5
\]
holds. The Non-Starvation Property \( NS_{MEP} \) asserts that neither process starves. This can be formalized in temporal logic as
\[
NS_{MEP} = (\Box \forall \rightarrow \text{TRY}_A) \land (\Box \forall \rightarrow \text{TRY}_B)
\]

It is easy to see that \( MEP \) does not satisfy \( NS_{MEP} \) for 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 \( F \) be a fairness assumption for atomic action \( \alpha. \) A fairness assumption for a program \( \pi \) is the conjunction of fairness assumptions—one for each atomic action in \( A. \) Thus, \( F_{MEP} \) the fairness assumption for \( MEP \) is defined by

³See [Francoz 86] for various types of fairness assumptions.
To show that MEP satisfies NS_MEP for all histories satisfying F MEP, we must prove that MEP satisfies F MEP ⇒ NS MEP. Putting this property into conjunctive normal form, we get

\[ F_{\text{MEP}}: \quad A \land F_A. \]

Each of the two clauses (conjuncts) in 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. This is expressed in temporal logic as

\[ WF_A: \quad \Box \neg \text{Before}_A. \]

where \( \text{Before}_A(s) \) is the predicate \( (\exists t: (s, t) \in o) \). Thus, the first conjunct of FNS MEP property to be proven is

\[ F_{\text{NS}}^A: \quad A \land WF_A \land \Box \neg TRY_A. \]

This clause contains seventeen disjuncts because \( |A_{\text{MEP}}| = 16 \); all but one of these disjuncts are negated. Each disjunct is of the form \( \Box \neg P \), so the Buchi automaton specification for each literal is \( m_a \) of Figure 3.2.

To prove that FNS MEP holds we use the following proof instruments.

\[ I = JS(F_{\text{NS}}^A_{\text{MEP}}, \text{MEP}) \]

\[ v = \begin{cases} 0 & \neg TRY_A \\ p_A - 3 & TRY_A \land TRY_B \land \text{turn} = A \\ 2 + (4 - p_B) \mod 7 & TRY_A \land \neg TRY_B \\ 7 + p_B - 3 & TRY_A \land TRY_B \land \text{turn} = B \end{cases} \]

where \( \text{enbl}(a) \) holds in any program state \( s \) where atomic action \( a \) is enabled (i.e., \( \text{enabled}(a, s) \)).

Rather than checking that verification conditions VC1 and VC2 hold, we argue (informally) that obligations O1–O5 are satisfied by this choice of proof instruments.

Clearly, O1 and O2 are 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 \( \neg TRY_A, v = 0 \), but an automaton for a positive disjunct is in an accepting state, so O3 is satisfied in that case. If \( TRY_A \land \neg TRY_B \land \text{turn} = A \), then \( v \) is the number of atomic actions that \( A \) must execute before entering \( c_A \). Since \( \text{turn} = A \), subsequent execution by \( B \) does not alter this value. If \( TRY_A \land \neg TRY_B \), then \( v \) includes the number of atomic actions \( B \) must execute to establish the previous case \( (TRY_A \land TRY_B \land \text{turn} = A) \).

Clearly, execution by \( A \) does not alter this value. Finally, when \( TRY_A \land \neg TRY_B \land \text{turn} = B \), then \( v \) is the maximum value of \( v \) in the preceding case \( (TRY_A \land \neg TRY_B) \) plus the number of atomic actions that \( B \) must execute to make \( TRY_A \land \neg TRY_B \) hold.

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

O5 holds because \( m_a \) for an enabled atomic action is not in an accepting state.

Note that this proof is an instance of the method of helpful directions for weak fairness (Francese 66). 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 a helpful atomic action. Thus, when \( TRY_A \land TRY_B \land \text{turn} = A \), the helpful direction is process \( A \); when \( TRY_A \land \neg TRY_B \) or \( TRY_A \land \neg TRY_B \land \text{turn} = B \), 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 developed in this paper. To see this observe that the candidate function identifies the helpful directions. The stipulation that execution of a helpful atomic action decrease the variant function guarantees O5. The stipulation that a non-helpful atomic action not increase the variant function guarantees O3. And, the stipulation that a non-helpful atomic action leave a helpful one enabled guarantees O4.

6. Discussion

Formulated in the terminology of this paper, our previous method ([Alpern & Schneider 85]) for extracting proof obligations from a deterministic Buchi automaton requires the program prove to exhibit an invariant \( I \) and a variant function \( v \) satisfying:

\[ AS1: \text{If } x \text{ is the first state of a joint history then } x \in I. \]

\[ AS2: \text{If } x \in I \text{ and } (x \leftarrow x' \in I). \alpha \text{ then } x' \in I. \]

\[ AS3: \text{If } x \in I \text{ and } (x \leftarrow x' \in I). \alpha \text{ then } Pos(x) \lor Pos(x') \lor v(x') < v(x). \]

\[ AS1 \text{ and } AS2 \text{ are obligations } O1 \text{ and } O2 \text{ of the approach outlined in Section 4. Consider the remaining obligations } O3 \text{ through } O5 \text{ of that approach. The deterministic property is a single clause that consists of a single, non-negated property, so there are no negative automata. Thus, } u(x) = \emptyset \text{ for every joint state } x \text{ and } O5 \text{ is trivially satisfied because its final disjunct is. Moreover, the final disjunct of } O4 \text{ must be false. AS3 and } O4 \text{ are therefore equivalent, and each implies } O3. \text{ Thus, the two techniques yield essentially the same proof obligations when applied to a deterministic property.} \]

The method in [Alpern & Schneider 85] is unsatisfactory for properties specified by non-deterministic Buchi automata. To use it to prove that a program \( \pi \) satisfies such a property \( P \), a deterministic property \( D \) that is contained in \( P \) is found. Proof obligations are then extracted from the deterministic Buchi automata for \( D \). If a (finite-
An added advantage of the Boolean decomposition approach is that parts of the proof may be reusable since other properties might be constructed from these parts.

Acknowledgments

Discussions with Stergos Cosmadakis and Sheryl Brady were helpful, especially in formulating the completeness proof.

References


---

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

6 To obtain a v-automaton for h from a Buchi automaton for \( \neg h \), exchange the accepting and non-accepting states.