

Recognizing Safety and Liveness*

Bowen Alpern
Fred B. Schneider

TR 86-727
January 1986
(Revised June 1986)

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work is supported, in part, by NSF Grant DCR-8320274 and Office of Naval Research contract N00014-86-K-0092.

Recognizing Safety and Liveness

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

January 8, 1986
Revised June 30, 1986

ABSTRACT

Formal characterizations for safety properties and liveness properties are given in terms of the structure of the Buchi automaton that specifies the property. The characterizations permit a property to be decomposed into a safety property and a liveness property whose conjunction is the original. The characterizations also give insight into techniques required to prove a large class of safety and liveness properties.

^{*}This work is supported, in part, by NSF Grant DCR-8320274 and Office of Naval Research contract N00014-86-K-0092.

1. Introduction

Informally, a *safety property* stipulates that “bad things” do not happen during execution of a program and a *liveness property* stipulates that “good things” do happen (eventually) [Lamport 77]. Distinguishing between safety and liveness properties is useful because proving that a program satisfies a safety property involves an invariance argument while proving that a program satisfies a liveness property involves a well-foundedness argument. Thus, knowing whether a property is safety or liveness helps when deciding how to prove that the property holds.

The relationship between safety properties and invariance arguments and between liveness properties and well-foundedness arguments has, until now, not been formalized or proved. Rather, it was supported by practical experience in reasoning about concurrent and distributed programs in light of the informal definitions of safety and liveness given above. This paper substantiates that experience by formalizing safety and liveness in a way that permits the relationship between safety and invariance and between liveness and well-foundedness to be demonstrated for a large class of properties. In so doing, we give new characterizations of safety and liveness and prove that they satisfy the formal definitions in [Alpern & Schneider 85a].

We proceed as follows. Section 2 describes an automata-theoretic approach for specifying properties. Section 3 contains automata-theoretic characterizations of safety and liveness. Section 4 shows how a property can be expressed as the conjunction of a safety property and a liveness property. Section 5 discusses the relationship between safety and liveness and various proof techniques. Section 6 discusses related work and section 7 summarizes our contributions.

2. Histories and Properties

A program π is assumed to be specified in terms of

- its set of atomic actions A_π , and
- a predicate $Init_\pi$ that describes its possible initial states.

An execution of π can be viewed as an infinite sequence σ of program states

$$\sigma = s_0 s_1 \dots,$$

which we call a *history*. State s_0 satisfies $Init_\pi$, and each following state results from executing a single enabled atomic action from A_π in the preceding state. For a terminating execution, an infinite sequence is obtained by repeating the final state. This corresponds to the view that a terminating execution is the same as a non-terminating execution in which after some finite time—once the program has terminated—the state remains fixed.

A *property* is a set of infinite sequences of program states. We write $\sigma \models P$ to denote that (infinite sequence) σ is in property P . A program *satisfies* a property P if for each of its histories h , $h \models P$.

A property is usually specified by a characteristic predicate on sequences rather than by enumeration. Formulas of temporal logic can be interpreted as predicates on infinite sequences of states, and various formulations of temporal logic have been used for specifying properties [Lamport 83] [Lichtenstein et al. 85] [Manna & Pnueli 81] [Wolper 83]. However, for our purposes, it will be convenient to specify properties using Buchi automata—finite-state automata that accept infinite sequences [Eilenberg 74]. Mechanical procedures exist to translate linear-time and branching-time first-order temporal formulas into Buchi automata [Alpern 86][Clarke et al. 86][Wolper 84], so using Buchi automata does not constitute a restriction. In fact, Buchi automata are more expressive than most temporal logic based specification languages—there exist properties that can be specified using Buchi automata but cannot be specified in (standard) temporal logics [Wolper 83].

A *Buchi automaton* [Eilenberg 74] m accepts the sequences of program states that are in $L(m)$, the property it specifies. Figure 2.1 is a Buchi automaton m_{tc} that accepts (i) all infinite sequences in which the first state satisfies a predicate $\neg Pre$ and (ii) all infinite sequences consisting of a state satisfying Pre , followed by a (possibly empty) sequence of states satisfying $\neg Done$, followed by an infinite sequence of states satisfying $Done \wedge Post$. Thus, m_{tc} specifies *Total Correctness* with precondition Pre and postcondition $Post$, where $Done$ holds if and only if the program has terminated.

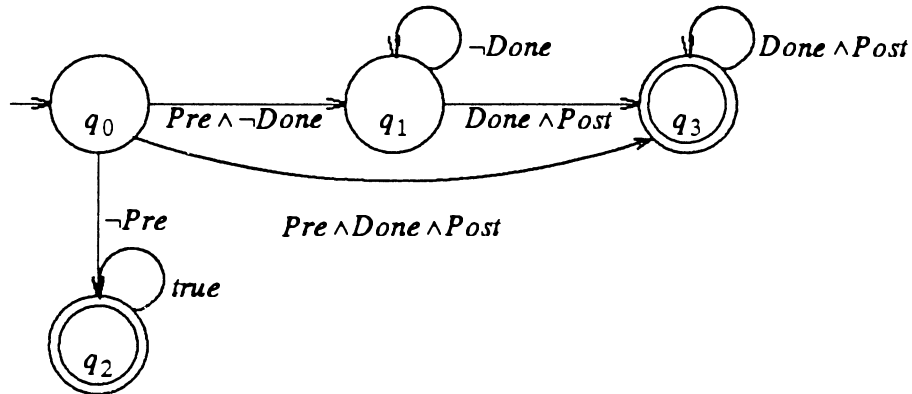


Figure 2.1. m_{tc}

Buchi automaton m_{tc} contains four *automaton states* labeled q_0 , q_1 , q_2 , and q_3 . The *start state* (q_0) is denoted by an arc with no origin and *accepting states* (q_2 and q_3) by concentric circles. An infinite sequence is accepted by a Buchi automaton if and only if it causes the recognizer to be infinitely often in accepting states.

Arcs between automaton states are labeled by program-state predicates called *transition predicates*. These define transitions between automaton states based on the next symbol read from the input. For example, because there is an arc labeled $\neg Pre$ from q_0 to q_2 in m_{ic} , whenever m_{ic} is in q_0 and the next symbol read is a program state satisfying $\neg Pre$, then a transition to q_2 is made. If the next symbol read by a Buchi automaton satisfies no transition predicate on an arc emanating from the current automaton state, the input is rejected; in this case, we say the transition is *undefined* for that symbol. This is used in m_{ic} to ensure that an infinite sequence that starts with a state satisfying Pre ends in an infinite sequence of states that satisfy $Done \wedge Post$ —once m_{ic} enters q_3 , every subsequent program state read must satisfy $Done \wedge Post$ or an undefined transition occurs.

A Buchi automaton is *reduced* if from every state there is a path to an accepting state. Thus, m_{ic} is an example of a reduced Buchi automaton. Given an arbitrary Buchi automaton, an equivalent reduced Buchi automaton can always be obtained by deleting every state from which no accepting state is reachable.

When there is more than one start state or more than one transition is possible from some automaton state for some input symbol, the automaton is *non-deterministic*; otherwise it is *deterministic*. Thus, m_{ic} is deterministic because it has a single start state and because disjoint transition predicates label the arcs that emanate from each automaton state. Although any set of finite sequences recognizable by a non-deterministic (ordinary) finite-state automaton can be recognized by some deterministic (ordinary) finite-state automaton [Hopcroft & Ullman 79], Buchi automata do not enjoy this equivalence. Some sets of infinite sequences can be recognized by non-deterministic Buchi automata but by no deterministic one [Eilenberg 74]. For example, consider m_{mono} of Figure 2.2, which accepts all sequences with an infinite suffix of program states that satisfy P .¹

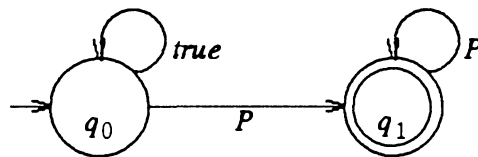


Figure 2.2. m_{mono}

No deterministic Buchi automaton accepts this set of sequences. The standard subset construction for transforming a non-deterministic (ordinary) finite-state automaton to a deterministic one does not work; when it is applied to m_{mono} , m_{det} of Figure 2.3 results, and no combination of q_0 and $q_{\{0,1\}}$ as accepting states causes m_{det} to accept the same set of

¹In linear-time temporal logic, this property is characterized by $\Diamond \Box P$.

sequences as for m_{mono} .

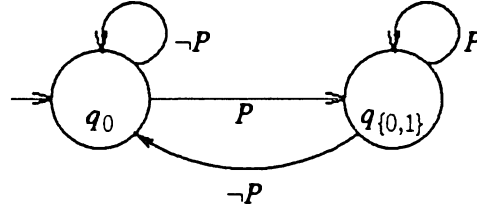


Figure 2.3. m_{det}

Formally, a Buchi automaton m for a property of a program π is a five-tuple $\langle S, Q, Q_0, Q_\infty, \delta \rangle$, where

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

Transition predicates are derived from δ as follows. T_{ij} , the transition predicate associated with the arc from automaton state q_i to q_j , is the predicate that holds for all program states s such that $q_j \in \delta(q_i, s)$. Thus, T_{ij} is *false* if no symbol can cause a transition from q_i to q_j .

In order to formalize when m accepts a sequence, some notation is required. For any sequence $\sigma = s_0 s_1 \dots$,

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

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

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

A *run* of m for an infinite sequence σ is a sequence of automaton states that m could be in while reading σ . Thus, for ρ to be a run for σ , $\rho[0] \in Q_0$, and $(\forall i: 0 < i < |\sigma|: \rho[i] \in \delta(\rho[i-1], \sigma[i-1]))$. Let $\Gamma_m(\sigma)$ be the set of runs of m on σ . (This set has only one element if m is deterministic.) Define $INF_m(\sigma)$ to be the set of automaton states that appear infinitely often in any element of $\Gamma_m(\sigma)$. Then, σ is accepted by m if and only if $INF_m(\sigma) \cap Q_\infty \neq \emptyset$.

Examples of Properties

A Buchi automaton m_{pc} that specifies *Partial Correctness* is shown in Figure 2.4. As in m_{ic} (Figure 2.1), Pre is a transition predicate that holds for states satisfying the given precondition, $Done$ holds for states in which the program has terminated, and $Post$ holds for states satisfying the given postcondition. Thus, m_{pc} accepts all sequences in which (i) the first state satisfies $\neg Pre$, as well as those (ii) consisting of a state satisfying Pre , followed by an infinite sequence of states satisfying $\neg Done$, and those (iii) consisting of a state satisfying Pre , followed by a (possibly empty) finite sequence of states satisfying $\neg Done$, followed by an infinite sequence of states satisfying $Done \wedge Post$.

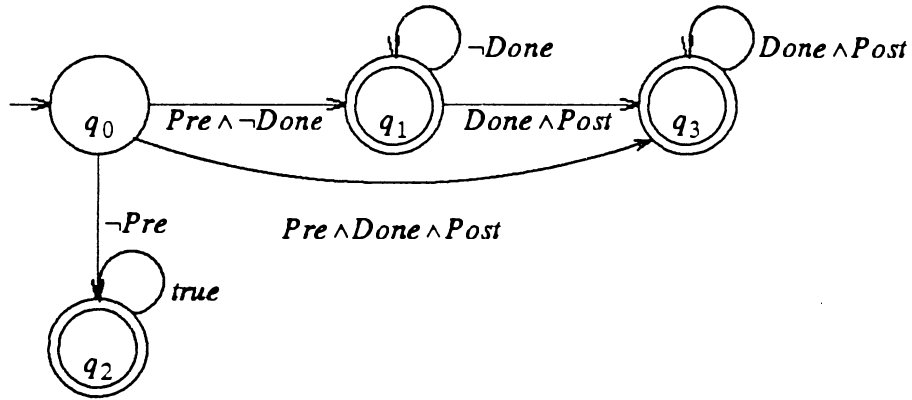


Figure 2.4. m_{pc}

A Buchi automaton m_{mutex} for *Mutual Exclusion* of two processes is given in Figure 2.5. We assume transition predicate CS_ϕ (CS_ψ) holds for any state in which process ϕ (ψ) is executing in its critical section.

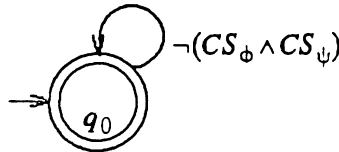


Figure 2.5. m_{mutex}

Starvation Freedom for a process using a mutual exclusion protocol is specified by m_{starv} of Figure 2.6. A process ϕ becomes enabled when its state satisfies the predicate $Request_\phi$, which characterizes the state of ϕ whenever it attempts to enter its critical section, and ϕ makes progress when its state satisfies the predicate $Served_\phi$, which holds whenever ϕ enters its critical section.

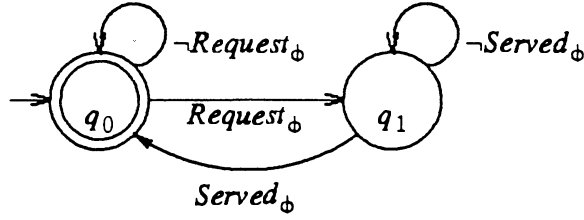


Figure 2.6. m_{starv}

3. Safety and Liveness

Just as properties can be viewed in terms of proscribed “bad things” and prescribed “good things”, so can Buchi automata. When a “bad thing” (“good thing”) of the property occurs, we would expect a “bad thing” (“good thing”) to happen in the recognizer for that property. The “bad thing” for a Buchi automaton is attempting an undefined transition, because if such a “bad thing” happens (in every run) while reading an input, the Buchi automaton will not accept that input. The “good thing” for a Buchi automaton is entering an accepting state infinitely often, because we require this “good thing” to happen for an input to be accepted. Having isolated these “bad things” and “good things”, it is possible to give an automata-theoretic characterization of safety and liveness.

Since every Buchi automaton is equivalent to some reduced Buchi automaton, it suffices to consider only reduced Buchi automata.

Recognizing Safety

To give an automata-theoretic characterization of safety properties, we require the following formal definition of safety from [Alpern & Schneider 85a]. Consider a property P that stipulates that some “bad thing” does not happen. If a “bad thing” happens in an infinite sequence σ , then it must do so after some finite prefix and must be irremediable. Thus, if $\sigma \not\models P$, there is some prefix of σ (that includes the “bad thing”) for which no extension to an infinite sequence will satisfy P . Taking the contrapositive of this, we get the formal definition of a safety property P :

$$\text{Safety: } (\forall \sigma: \sigma \in S^\omega: \sigma \models P \Leftrightarrow (\forall i: 0 \leq i: (\exists \beta: \beta \in S^\omega: \sigma[..i] \beta \models P))), \quad (3.1)$$

where S is the set of program states, S^* the set of finite sequences of states, S^ω the set of infinite sequences of states, and juxtaposition is used to denote catenation of sequences.

For a reduced Buchi automaton m , define its *closure* $cl(m)$ to be the corresponding Buchi automaton in which every state has been made into an accepting state. For example, $Safe(m_{tc})$ (Figure 4.1) is the closure of m_{tc} (Figure 2.1) and m_{mutex} (Figure 2.5) is its own

closure. The closure of m can be used to determine whether the property specified by m is a safety property. This is because $cl(m)$ accepts a safety property—it never rejects an input by failing to enter accepting states (lack of a “good thing”); it rejects only by attempting an undefined transition (a “bad thing”). Thus, if a Buchi automaton m does not specify a safety property then $cl(m)$ accepts a superset of the inputs accepted by m . Therefore, if m and $cl(m)$ accept the same language then m recognizes a safety property.

Theorem 1: A reduced Buchi automaton m specifies a safety property if and only if $L(m) = L(cl(m))$.²

Proof. First, assume m specifies a safety property. Since $cl(m)$ is obtained from m by making all states accepting, every (infinite) sequence α accepted by m is also accepted by $cl(m)$. Hence, $L(m) \subseteq L(cl(m))$. To show $L(cl(m)) \subseteq L(m)$, we first prove that if $\alpha \in L(cl(m))$ then

$$(\forall i: 0 \leq i: (\exists \beta: \beta \in S^\omega: \alpha[..i]\beta \in L(m))). \quad (3.2)$$

Let $\alpha \in L(cl(m))$. Choose any prefix $\alpha[..i]$ and suppose $\alpha[..i]$ puts m in state q_i . Since m is reduced, q_i precedes an accepting state. Thus, there exists a sequence of program states β_0 that takes m from q_i to some accepting state q_{a1} . Moreover, there also exists a sequence of program states β_1 that takes m from q_{a1} to some accepting state q_{a2} , a sequence of program states β_2 that takes m from q_{a2} to some accepting state q_{a3} , etc. Define $\beta = \beta_0\beta_1\beta_2\dots$. Since $\alpha[..i]\beta$ causes m to be in accepting states infinitely often, $\alpha[..i]\beta \in L(m)$, so (3.2) is satisfied. Now, from (3.2) we can conclude $\alpha \in L(m)$ due to (3.1), because by assumption $L(m)$ is a safety property. Thus, $\alpha \in L(cl(m)) \Rightarrow \alpha \in L(m)$, hence $L(cl(m)) \subseteq L(m)$.

Next, assume $L(m) = L(cl(m))$. To show that m specifies a safety property, we show that definition (3.1) is satisfied. Trivially,

$$\sigma \in L(m) \Rightarrow (\forall i: 0 \leq i: (\exists \beta: \beta \in S^\omega: \sigma[..i]\beta \in L(m)))$$

since we can choose $\beta = \sigma[i+1\dots]$. Thus, it suffices to prove

$$\sigma \notin L(m) \Rightarrow \neg(\forall i: 0 \leq i: (\exists \beta: \beta \in S^\omega: \sigma[..i]\beta \in L(m))). \quad (3.3)$$

Since $L(m) = L(cl(m))$, (3.3) is equivalent to

$$\sigma \notin L(cl(m)) \Rightarrow (\exists i: 0 \leq i: (\forall \beta: \beta \in S^\omega: \sigma[..i]\beta \notin L(cl(m)))).$$

Suppose $\sigma \notin L(cl(m))$. Thus, $cl(m)$ rejects σ . σ must be rejected by $cl(m)$ because an undefined transition is attempted since, by construction, every state in $cl(m)$ is accepting. Let this undefined transition occur upon reading $\sigma[k]$. Hence,

$$(\forall \beta: \beta \in S^\omega: \sigma[..k]\beta \notin L(cl(m)))$$

as required for m to specify a safety property according to definition (3.1). \square

²See [Sistla et al. 85] for an algorithm to test whether the languages accepted by two Buchi automata are equal.

For example, according to Theorem 1, m_{pc} (Figure 2.4), m_{mutex} (Figure 2.5), and $Safe(m_{tc})$ (Figure 4.1) all specify safety properties.

Recognizing Liveness

To give an automata-theoretic characterization of liveness properties, we require the following formal definition of liveness from [Alpern & Schneider 85a]. The thing to observe about a liveness property is that no partial execution is irremediable since if some partial execution were irremediable, then it would be a “bad thing”. We take this to be the defining characteristic of liveness. Thus, P is a liveness property if and only if

$$\text{Liveness: } (\forall \alpha: \alpha \in S^*: (\exists \beta: \beta \in S^\omega: \alpha\beta \models P)). \quad (3.4)$$

The closure of a Buchi automaton can also be used to determine if m specifies a liveness property. This is because $cl(m)$ can reject an input only by attempting an undefined transition—a “bad thing”. A liveness property never proscribes a “bad thing”, so if m specifies a liveness property, $cl(m)$ must accept every input.

Theorem 2: A reduced Buchi automaton m specifies a liveness property if and only if $L(cl(m)) = S^\omega$.

Proof. First, assume m specifies a liveness property. According to definition (3.4) of liveness,

$$(\forall \alpha: \alpha \in S^\omega: (\forall i: 0 \leq i: (\exists \beta: \beta \in S^\omega: \alpha[..i]\beta \in L(m)))).$$

Thus, m does not attempt an undefined transition when reading an input α . Since $cl(m)$ has the same transition function as m , $cl(m)$ does not attempt an undefined transition when reading α . Each of the states of $cl(m)$ is accepting, and thus $cl(m)$ accepts α . Hence, $(\forall \alpha: \alpha \in S^\omega: \alpha \in L(cl(m)))$, or equivalently $L(cl(m)) = S^\omega$, as required.

Next assume $L(cl(m)) = S^\omega$. Thus, $cl(m)$, hence m , does not attempt an undefined transition when reading any input α . Therefore, m does not attempt an undefined transition when reading a finite prefix $\alpha[..i]$ of an input. Suppose $\alpha[..i]$ leaves m in automaton state q_i . Since m is reduced, there exists a path from q_i to some accepting state q_{a1} , from q_{a1} to some accepting state q_{a2} , etc. Let β_0 be a sequence of program states that takes m from q_i to q_{a1} , let β_1 be a sequence of program states that takes m from q_{a1} to q_{a2} , etc. Thus, $(\exists \beta: \beta \in S^\omega: \alpha[..i]\beta \in L(m))$ since we can choose $\beta = \beta_0\beta_1\dots$. Definition (3.4) of liveness therefore holds, so $L(m)$ is a liveness property. \square

For example, according to Theorem 2, m_{mono} (Figure 2.2), m_{starv} (Figure 2.6), and $Live(m_{tc})$ (Figure 4.2) specify liveness properties.

4. Partitioning into Safety and Liveness

Given a Buchi automaton m , it is not difficult to construct Buchi automata $Safe(m)$ and $Live(m)$ such that $Safe(m)$ specifies a safety property, $Live(m)$ specifies a liveness property, and the property specified by m is the intersection of those specified by $Safe(m)$ and $Live(m)$. This proves that every property specified by a Buchi automaton is equivalent to the conjunction of a safety property and a liveness property, each of which can be specified by a Buchi automaton.

For $Safe(m)$, we use $cl(m)$.

Theorem 3: $Safe(m)$ specifies a safety property.

Proof. By construction, $cl(cl(m)) = cl(m)$. Hence, $L(cl(cl(m))) = L(cl(m))$, and the result follows from Theorem 1. \square

We will construct $Live(m)$ to be a Buchi automaton such that

$$L(Live(m)) = L(m) \cup (S^\omega - L(cl(m))). \quad (4.1)$$

If m is deterministic then for $Live(m)$ we use m augmented by an accepting trap state³ q_{trap} . Transition function $\delta_{Live(m)}$ is the transition function for m extended so that it causes every undefined transition of m to put $Live(m)$ in q_{trap} .

If m is non-deterministic then for $Live(m)$ we use $m \times env(m)$ and take as accepting states any automaton state in which m or $env(m)$ is in an accepting state. Buchi automaton $env(m)$ specifies $S^\omega - L(cl(m))$ and is defined as follows. The states of $env(m)$ include a trap state q_{trap} and the sets of automaton states of m ; q_{trap} is the only accepting state of $env(m)$. The transition function $\delta_{env(m)}$ is, for all sets q_i of the automaton states of m :

$$\delta_{env(m)}(q_i, s) = \begin{cases} q_{trap} & \text{if } \delta_m(q_i, s) = \emptyset \\ \delta_m(q_i, s) & \text{otherwise} \end{cases}$$

Lemma: $L(Live(m)) = L(m) \cup (S^\omega - L(cl(m)))$.

Proof. First, consider the case where m is deterministic. Assume $\alpha \in L(Live(m))$. Thus, $Live(m)$ enters accepting states infinitely often when reading α . If all of these accepting states are also (accepting) states of m then m will accept α , hence $\alpha \in L(m)$. Otherwise, one of the accepting states is q_{trap} . In this case, $\alpha \notin L(cl(m))$ because to enter q_{trap} , m makes an undefined transition. From $\alpha \notin L(cl(m))$, we conclude $\alpha \in (S^\omega - L(cl(m)))$.

³A trap state is one that has a transition to itself for all inputs.

Now assume $\alpha \in L(m) \cup (S^\omega - L(cl(m)))$. If $\alpha \in L(m)$ then, by construction, $\alpha \in L(Live(m))$. If $\alpha \in (S^\omega - L(cl(m)))$ then $\alpha \notin L(cl(m))$, hence $cl(m)$ attempts an undefined transition on input α . This means that m attempts an undefined transition on input α , so, by construction, $Live(m)$ will make a transition to q_{trap} and accept α . Hence, $\alpha \in L(Live(m))$.

Next, consider the case where m is non-deterministic. By construction of the accepting states of $Live(m)$, $L(Live(m)) = L(m) \cup L(env(m))$. Since $env(m)$ specifies $S^\omega - L(cl(m))$, the result follows. \square

We can now prove that $Live(m)$ specifies a liveness property. By construction, $Live(m)$ never rejects an input by attempting an undefined transition (a “bad thing”). Because m is reduced and $Live(m)$ accepts any input accepted by m , for each finite sequence of program states there is an infinite suffix that can be appended to obtain a sequence that will be accepted by $Live(m)$ —the defining characteristic of liveness.

Theorem 4: $Live(m)$ specifies a liveness property.

Proof. First, consider the case where m is deterministic. By construction, $Live(m)$ has no undefined transitions. Therefore $L(cl(Live(m))) = S^\omega$ and the result follows from Theorem 2.

Next, consider the case where m is non-deterministic. Due to the construction of $Live(m)$,

$$L(cl(Live(m))) = L(cl(m)) \cup L(cl(env(m))).$$

$L(cl(env(m))) \supseteq L(env(m))$, since $cl(env(m))$ differs from $env(m)$ by having more states accepting. Substituting,

$$L(cl(Live(m))) \supseteq L(cl(m)) \cup L(env(m)).$$

Since, by construction, $L(env(m)) = S^\omega - L(cl(m))$, we get

$$L(cl(Live(m))) \supseteq L(cl(m)) \cup (S^\omega - L(cl(m))),$$

or $L(cl(Live(m))) \supseteq S^\omega$. Since $L(cl(Live(m))) \subseteq S^\omega$, the result then follows from Theorem 2. \square

Finally, we can prove

Theorem 5: Given a reduced Buchi automaton m , $L(m) = L(Safe(m)) \cap L(Live(m))$.

Proof. Substituting for $L(Live(m))$ according to (4.1) and for $L(Safe(m))$ according to Theorem 1, we get

$$\begin{aligned}
L(Live(m)) \cap L(Safe(m)) &= (L(m) \cup (S^\omega - L(cl(m)))) \cap L(cl(m)) \\
&= (L(m) \cap L(cl(m))) \cup ((S^\omega - L(cl(m))) \cap L(cl(m))) \\
&= (L(m) \cap L(cl(m))) \cup \emptyset \\
&= L(m) \quad \square
\end{aligned}$$

The construction of Theorem 5 is now illustrated for m_{ic} of Figure 2.1, which specifies Total Correctness. $Safe(m_{ic})$ is given in Figure 4.1; $Live(m_{ic})$ is given in Figure 4.2.

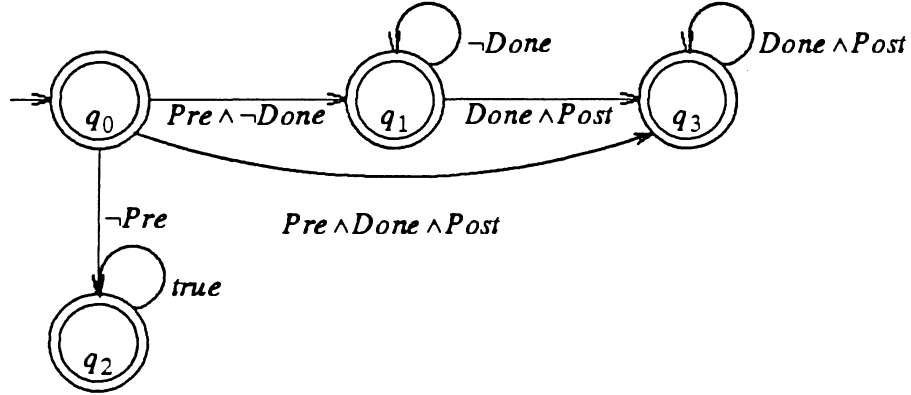


Figure 4.1. $Safe(m_{ic})$

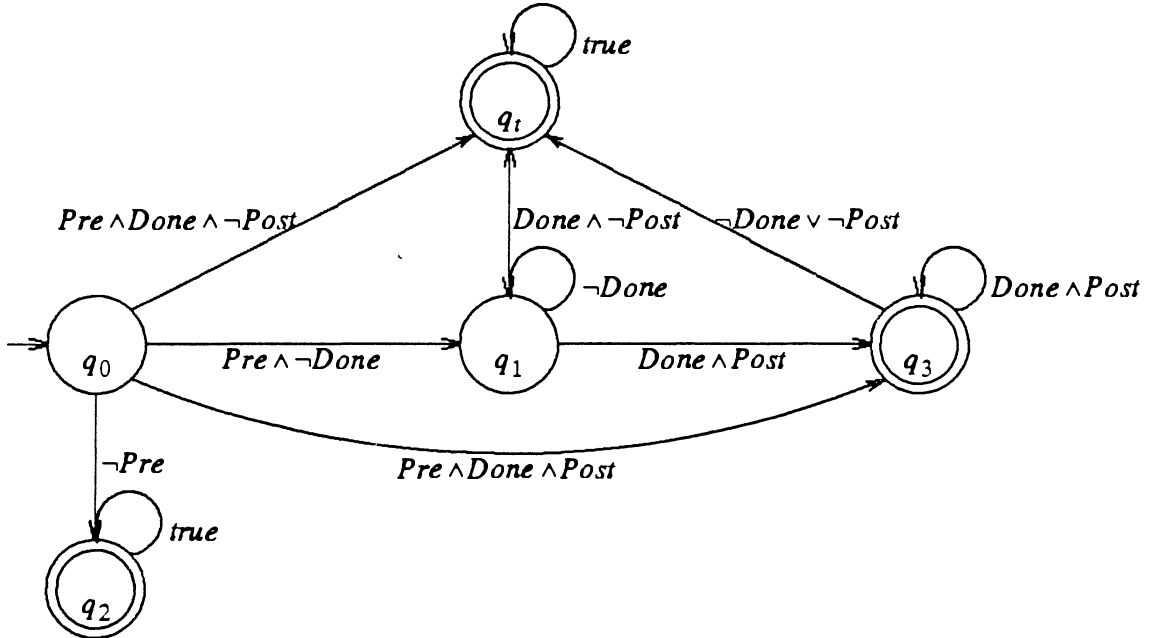


Figure 4.2. $Live(m_{ic})$

However, $Live(m_{ic})$ can be simplified by combining the three accepting states, resulting in the equivalent recognizer of Figure 4.3.

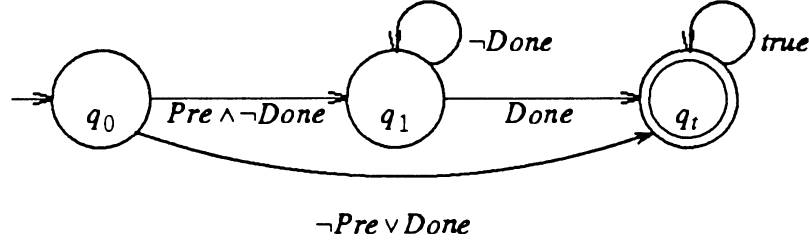


Figure 4.3. $Live(m_{ic})$ Simplified

Notice that $L(Safe(m_{ic})) = L(m_{pc})$ because $Safe(m_{ic})$ differs from m_{pc} only by having q_0 as an accepting state. Thus, we have shown that Partial Correctness (as specified by m_{pc} of Figure 2.4) is the safety component of Total Correctness. Also, observe that the simplified $Live(m_{ic})$ automaton specifies a property comprising sequences of program states in which either the first state satisfies $\neg Pre$ or eventually there is a state satisfying $Done$. Such a property might well be called *Termination*, since it requires the program to reach $Done$ if started in a state satisfying Pre . Termination is the liveness component of Total Correctness. We have therefore proved, by our construction, that Total Correctness is the intersection of Partial Correctness and Termination.

5. Proving Deterministic Safety and Liveness Properties

Given a deterministic Buchi automaton specification of a property, it is possible to extract proof obligations that must be satisfied by any program for which that property holds. This forms the basis for an approach to program verification proposed in [Alpern & Schneider 85b] and permits us to formalize the relationship of safety and liveness properties specified by deterministic Buchi automata to invariance and well-foundedness arguments.

To prove that every history of a program π is in a property specified by a deterministic Buchi automaton m , a set of *correspondence invariants* and a set of variant functions are constructed and shown to satisfy certain *proof obligations*. There is one correspondence invariant C_i for each automaton state q_i ; one variant function v_κ for each *reject knot* κ , where a reject knot is a maximal strongly connected subset of automaton states in m containing no accepting states.

The first two proof obligations ensure that C_i holds on a program state s if there exists a history of π containing s and m enters q_i upon reading s .

$$\text{Correspondence Basis: } (\forall j: q_j \in Q: (Init_\pi \wedge T_{0j}) \Rightarrow C_j). \quad (5.1)$$

$$\begin{aligned}
\text{Correspondence Induction: For all } \alpha: \alpha \in A_\pi: \\
\text{For all } i: q_i \in Q: \\
\{C_i\} \alpha \{ \bigwedge_{j: q_j \in Q} (T_{ij} \Rightarrow C_j) \}
\end{aligned} \tag{5.2}$$

The next two obligations ensure that m never attempts an undefined transition when reading a history of π .

$$\text{Transition Basis: } \text{Init}_\pi \Rightarrow \bigvee_{j: q_j \in Q} T_{0j} \tag{5.3}$$

$$\begin{aligned}
\text{Transition Induction: For all } \alpha: \alpha \in A_\pi: \\
\text{For all } i: q_i \in Q: \\
\{C_i\} \alpha \{ \bigvee_{j: q_j \in Q} T_{ij} \}
\end{aligned} \tag{5.4}$$

The final two obligations ensure that m does not loop forever in non-accepting states when reading a history of π .

$$\text{Knot Exit: For each reject knot } \kappa: (\forall i: q_i \in \kappa: (v_\kappa(q_i)=0) \Rightarrow \neg C_i) \tag{5.5}$$

$$\begin{aligned}
\text{Knot Variance: For each reject knot } \kappa: \\
\text{For all } \alpha: \alpha \in A_\pi: \\
\text{For all } q_i \in \kappa: \\
\{C_i \wedge 0 < v_\kappa(q_i) = V\} \alpha \{ \bigwedge_{j: q_j \in \kappa} ((T_{ij} \wedge C_j) \Rightarrow v_\kappa(q_j) < V) \}
\end{aligned} \tag{5.6}$$

First observe that, by construction, $\text{Safe}(m)$ can never have reject knots. Thus, (5.5) and (5.6) are trivially satisfied for such a recognizer, and proving that a program satisfies such a property never requires a variant function (or well-foundedness argument). The remaining proof obligations for $\text{Safe}(m)$ constitute an invariance argument. Moreover, if m specifies a safety property then $L(m) = L(\text{Safe}(m))$. Proving that π satisfies $\text{Safe}(m)$ is, therefore, sufficient in order to prove that π satisfies m . Thus, safety properties specified by deterministic Buchi automata can be proved using only invariance arguments.

Second, recall from Theorem 2 that m specifies a liveness property if $L(cl(m)) = S^\omega$. By construction, if $L(cl(m)) = S^\omega$ then it is not possible for m to attempt an undefined transition. Therefore, (5.3) and (5.4) are trivially satisfied when trying to prove that a program π satisfies a liveness property. Automata specifying liveness properties can have reject knots, so (5.5) and (5.6) must be proved—a variant function of well-foundedness argument is therefore required in proving a liveness property. In addition, an invariance argument is required to satisfy (5.1) and (5.2).

6. Related Work

The first formal definition of safety was given in [Lamport 85]. While that definition correctly captures the intuition for an important class of safety properties—those invariant under stuttering—it is inadequate for safety properties that are not invariant under stuttering. The formal definition of safety used in this paper, which was first proposed in [Alpern & Schneider 85a], is independent of stuttering; in [Alpern et al. 85] it is shown equivalent to Lamport's for properties that are invariant under stuttering. The definition of liveness used in this paper also first appeared in [Alpern & Schneider 85a] along with a proof that every property can be expressed as the conjunction of a safety property and a liveness property. That proof is based on a topological characterization of safety properties as closed sets and liveness properties as dense sets. The automata-theoretic characterizations and proofs in this paper more closely parallel the informal definitions of safety and liveness in terms of “bad things” and “good things”.

In [Sistla 85], an attempt is made to characterize syntactically safety and liveness properties that are expressed in linear-time temporal logic. Deductive systems are given for safety and liveness formulas in a temporal logic with *eventually* (\Diamond), but without *next* (\bigcirc), or *until*. In addition, deductive systems for full (propositional) temporal logic are given for a subset of the safety properties, called strong safety properties, and for a subset of the liveness properties, called absolute liveness properties. Finally, [Sistla 85] proves that the states of a deterministic Buchi automaton for a safety property can be partitioned into “good” and “bad” states, where “bad” states are never entered in an accepting run. That work is extended in [Sistla 86]. There, safety properties, safety properties that are invariant under stuttering, strong safety properties, absolute liveness properties, and liveness properties are characterized in terms of the form of the linear-time temporal logic formula that specifies each. Sistla also gives a syntactic characterization of the temporal logic specification of strong safety properties that can be defined using deterministic Buchi automata.

Another syntactic characterization of safety and liveness properties appears in [Lichtenstein et al. 85]. The definition of safety given there coincides with ours; the definition of liveness classifies some properties as liveness that our definition does not. We do not classify $p \text{ until } q$ as liveness because the occurrence of $\neg p$ before q constitutes a “bad thing” and therefore $p \text{ until } q$ has elements of safety, but [Lichtenstein et al. 85] consider it liveness. Another difference is that the definitions in [Lichtenstein et al. 85] are based on existing temporal logic inference rules (proof obligations) whereas our definitions are not. Independence from inference rules makes our results about the relationship between types of properties and proof techniques all the more interesting. Also, in contrast to the definitions in [Lichtenstein et al. 85], our characterizations of safety and liveness are independent of the notation used to express the properties and therefore apply to a larger class of properties.

7. Conclusions

This paper gives simple tests to determine whether a property specified by a (deterministic or nondeterministic) Buchi automaton is safety or liveness. We show how to extract automata $Safe(m)$ and $Live(m)$ from a Buchi automaton m , where $Safe(m)$ specifies a safety property, $Live(m)$ specifies a liveness property, and the property specified by m is the intersection of the ones specified by $Safe(m)$ and $Live(m)$. The extraction is illustrated by proving that Total Correctness is the conjunction of safety property Partial Correctness and liveness property Termination. Finally, we prove that for properties specified by deterministic Buchi automata, safety properties can be proved by use of an invariance argument while liveness properties also require a well-foundedness argument.

Acknowledgments

David Gries and Joe Halpern provided helpful comments on a draft of this paper.

References

- [Alpern 86] Alpern, B. Proving Temporal Properties of Concurrent Programs: A Non-Temporal Approach. Ph.D. Thesis. Department of Computer Science, Cornell University. January 1986.
- [Alpern et al. 85] Alpern, B., A.J. Demers, and F.B. Schneider. Safety without stuttering. Technical Report TR 85-708, Department of Computer Science, Cornell University, Oct. 1985.
- [Alpern & Schneider 85a] Alpern, B., and F.B. Schneider. Defining liveness. *Information Processing Letters* 21 (Oct. 1985), 181-185.
- [Alpern & Schneider 85b] 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.
- [Clarke et al. 86] Clarke, E.M., E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (April 1986), 244-263.
- [Eilenberg 74] Eilenberg, S. *Automata, Languages and Machines, Vol A*. Academic Press, New York, 1974.
- [Hopcroft & Ullman 79] Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.
- [Lamport 77] Lamport, L. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering* SE-3, 2 (March 1977), 125-143.
- [Lamport 83] Lamport, L. What good is temporal logic. *Information Processing '83*, (R.E.A. Mason, ed.) North-Holland Publishing Company, Amsterdam, 1983, 657-668.
- [Lamport 85] Lamport, L. Logical Foundation. In *Distributed Systems—Methods and Tools for Specification*, Lecture Notes in Computer Science, Vol 190. M. Paul and H.J. Siegart, eds. (1985), Springer-Verlag, Berlin.
- [Lichtenstein et al. 85] Lichtenstein, O., A. Pnueli, and L. Zuck. The glory of the past. *Proc. Workshop on Logics of Programs*, June 1985, Brooklyn, New York, Lecture Notes in Computer Science, Vol. 193. (R. Parikh, ed.), Springer-Verlag, Berlin, 196-218.
- [Manna & Pnueli 81] 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.
- [Sistla 85] Sistla, A.P. On characterization of safety and liveness properties in temporal logic. *Proc. 4th Sym.*

Principles of Distributed Computing, ACM, Minaki, Aug. 1985, 39-48.

- [Sistla 86] Sistla, A.P. On characterization of safety and liveness properties in temporal logic. Technical Report, GTE Laboratories Inc., Waltham, Mass. March 1986.
- [Sistla et al. 85] Sistla, A.P., M.Y. Vardi, P. Wolper. The complementation problems for Buchi automata with applications to temporal logic (extended abstract). *Proc. 12th International Colloquium on Automata, Languages and Programming, ICALP'85*, Nafplion, Greece, July 1985, Lecture Notes in Computer Science, Vol. 194, Springer Verlag, New York, 1985, 465-474.
- [Wolper 83] Wolper, P. Temporal logic can be more expressive. *Information and Control* 56, 1-2 (1983), 72-99.
- [Wolper 84] Wolper, P. The tableau method for temporal logic: An overview. Unpublished manuscript.