Recognizing Safety and Liveness

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 safety and liveness properties.
Recognizing Safety and Liveness*

Bowen Alpern
Fred B. Schneider

TR 86-727
January 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work is supported, in part, by NSF Grant DCR 8320374 and the Office of Naval Research.
Recognizing Safety and Liveness*

Bowen Alpern
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

January 3, 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 safety and liveness properties.

This work is supported, in part, by NSF Grant DCR-8320274 and the Office of Naval Research.
1. Introduction

Informally, a safety property stipulates that some "bad thing" does not happen during execution of a program and a liveness property stipulates that some "good thing" does happen (eventually) [Lamport 77]. Distinguishing between safety and liveness properties has merit because proving that a program satisfies a safety property involves an invariance argument [Lamport & Schneider 84] while proving that a program satisfies a liveness property involves a well-foundedness argument [Manna & Pnueli 84]. Thus, knowing whether a property is safety or liveness suffices for deciding on a technique 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. In so doing, we give new formal characterizations of safety and liveness and show that they satisfy the formal definitions in [Alpern & Schneider 85a]; we also give a new constructive proof that every property can be expressed as the conjunction of a safety and a liveness property.

We proceed as follows. In section 2, an automata-theoretic approach for specifying properties is described. Section 3 contains our new characterizations of safety and liveness. Section 4 shows that every property can be expressed as the conjunction of a safety property and a liveness property. The relationship between safety and liveness and various proof techniques is discussed in section 5. Section 6 discusses related work.

2. Histories and Properties

An execution of a program can be viewed as an infinite sequence \( \sigma \) of program states
\[
\sigma = s_0, s_1, \ldots
\]
which we call a history. State \( s_0 \) is an initial state of the program, and each following state results from executing a single atomic action 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—one the program has terminated—the state remains fixed.

A property is a set of infinite sequences of program states. For an infinite sequence \( \sigma \), we write \( \sigma \models P \) to denote that \( \sigma \) 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 any temporal formula into a corresponding Buchi automata [Alpern 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 (most) temporal logics [Wolper 83].

A Buchi automaton accepts those sequences of program states that are in the property it specifies. Figure 2.1 is an example of a Buchi automaton \( m_c \) that accepts (i) all infinite sequences in which the first state satisfies a predicate ~Pre and (ii) all infinite sequences in which the first state satisfies Pre, a possibly empty sequence of states follows in which each satisfies ~Done, and each state in the remaining infinite suffix satisfies Done \& Post. Thus, \( m_c \) specifies Total Correctness with precondition Pre, postcondition Post, where Done holds if and only if the program has terminated.

![Buchi automaton](image)

**Figure 2.1.** \( m_c \)

Buchi automaton \( m_c \) contains four automaton states labeled \( q_0, q_1, q_2, \) and \( q_3 \). The start state is denoted by an arc with no origin and infinite-accepting states 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 some infinite-accepting state. In \( m_c, q_0 \) is the start state, and both \( q_2 \) and \( q_3 \) are infinite-accepting states.

Arrows 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, the arc labeled $Pre$ from $q_0$ to $q_2$ in $m_c$ means that whenever $m_c$ is in $q_0$ and the next symbol read is a program state satisfying $Pre$, 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 $m_c$ to ensure that an infinite sequence that starts with a state satisfying $Pre$ ends in an infinite sequence of states that each satisfy $Done \land Post$—once $m_c$ enters $q_3$, every subsequent program state read must satisfy $Done \land Post$ or an undefined transition occurs.

When there is more than one start state or there is more than one transition possible from some automaton state for some input symbol, the automaton is non-deterministic; otherwise it is deterministic. Thus, $m_c$ is deterministic because it has a single start state and disjoint transition predicates label the arcs that emanate from each automaton state.

Formally, a Buchi automaton $m$ for a property of a program $\pi$ is a five-tuple $(S, Q, Q_0, Q_\infty, \delta)$, where

- $S$ is the set of program states of $\pi$,
- $Q$ is the set of automaton states of $m$,
- $Q_0 \subseteq Q$ is the set of start states of $m$,
- $Q_\infty \subseteq Q$ is the set of infinite-accepting states of $m$,
- $\delta : (Q \times S) \to 2^Q$ is the transition function of $m$.

Transition predicates are derived from $\delta$ 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 $\sigma$ such that $q_j \in \delta(q_i, \sigma)$. 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 definitions are required. For any sequence $\sigma = s_0 s_1 \ldots$,

- $\sigma[1] = s_1$,
- $\sigma[..i] = s_0 s_1 \ldots s_i$,
- $\sigma[..i] = s_i s_{i-1} \ldots$,
- $|\sigma|$ = the length of $\sigma$ ($\omega$ if $\sigma$ is infinite).

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

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

A run of $m$ for an infinite sequence $\sigma$ is a sequence of automaton states that $m$ could be in while reading $\sigma$. Thus, for $\rho$ to be a run for $\sigma$, $\rho(0) \in Q_0$, 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 $\sigma$. (It is a set because $m$ might be non-deterministic.) Define $INF_m(\sigma)$ to be the set of automaton states
that appear infinitely often in any element of \( T_m(\sigma) \). Then, \( \sigma \) is accepted by \( m \) if and only if
\[
\text{INF}_m(\sigma)^* Q_{inf} \neq \emptyset.
\]

Any set of finite sequences that can be recognized by a non-deterministic, finite-state automaton can be recognized by some deterministic, finite-state automaton [Hopcroft & Ullman 79]. Unfortunately, Buchi automata do not enjoy this equivalence—there are sets of infinite sequences that can be recognized by non-deterministic Buchi automata but by no deterministic one [Eilenberg 74]. However, for our purposes it suffices to restrict attention to properties specified by deterministic Buchi automata because [Alpern & Schneider 85b] proves the following for a program \( \pi \) that satisfies a property \( ND \) specified by a non-deterministic Buchi automaton \( m_{ND} \): if \( \pi \) has a finite state space then there exists a property \( D \) such that \( D \subseteq ND \), \( D \) is specified by a deterministic Buchi automaton \( m_D \), and \( \pi \) satisfies \( D \).

### Examples of Properties

A Buchi automaton \( m_{pc} \) that specifies Partial Correctness is shown in Figure 2.2. As in \( m_c \) (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 the first state satisfies \( \neg Pre \), as well as all sequences in which the first state satisfies \( Pre \) and every subsequent state satisfies \( Done \Rightarrow Post \).

![Figure 2.2. \( m_{pc} \)](image)

A Buchi automaton \( m_{max} \) for Mutual Exclusion of two processes is given in Figure 2.3. We assume transition predicate \( CS_\phi \) (\( CS_\psi \)) holds for any state in which process \( \phi \) (\( \psi \)) is executing in its critical section.
Starvation Freedom for a mutual exclusion protocol is specified by $m_{\text{starv}}$ of Figure 2.4. A process $\phi$ becomes enabled when its state satisfies the predicate $\text{Request}_\phi$, which characterizes the state of $\phi$ whenever it attempts to enter its critical section, and $\phi$ makes progress when its state satisfies the predicate $\text{Served}_\phi$, which holds whenever $\phi$ enters its critical section. Notice that $m_{\text{starv}}$ exploits the fact that in a mutual exclusion protocol $\phi$ will make but a single request for each entry into the critical section.

3. Recognizers for 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 making 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 infinite-accepting state, because we require this "good thing" to happen infinitely often 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.

Recognizing Safety

Define a safety recognizer to be a deterministic Buchi automaton in which

SR: Every cycle contains an infinite-accepting state.

In a safety recognizer, "good things" are inevitable, unless they become impossible due to an
undefined transition, which is a "bad thing". Both m_{pe} (Figure 2.2) and m_{max} of (Figure 2.3) are examples of safety recognizers.

There is a natural correspondence between safety recognizers and safety properties. To prove this, we require the following formal definition of a safety property [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 σ\notin 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 a formal definition of a safety property P:

Safety: \( (\forall \sigma: \sigma \in S^\omega \land \sigma \models P \Rightarrow (\forall i: (\exists \beta: \beta \in S^\omega: \sigma[i] \beta \models P))) \), \hspace{1cm} (3.1)

where \( S \) is the set of program states, \( S^\omega \) the set of finite sequences of states, \( S^\omega \) the set of infinite sequences of states, and juxtaposition is used to denote concatenation of sequences.

Now we can prove that safety recognizers and safety properties specified by deterministic Buchi automata are equivalent.

**Theorem 1:** Safety recognizers specify only safety properties.

**Proof.** Assume \( m_{\text{safe}} \) is a safety recognizer for a property Safe. We must show that Safe satisfies (3.1).

Let \( \sigma \) be an infinite sequence not accepted by \( m_{\text{safe}} \). Thus, \( \sigma \notin \text{Safe} \), and according to (3.1) we must show

(\exists i: 0 \leq i: (\forall \beta: \beta \in S^\omega: \sigma[i] \beta \models \text{Safe})). \hspace{1cm} (3.2)

Since \( \sigma \) is not accepted by \( m_{\text{safe}} \), because \( m_{\text{safe}} \) is a safety recognizer it must attempt an undefined transition upon reading some finite prefix \( \sigma[i] \). Consequently \( m_{\text{safe}} \) rejects any sequence beginning with \( \sigma[i] \), and

(\forall \beta: \beta \in S^\omega: \sigma[i] \beta \models \text{Safe})).

Showing that (3.2) \( \Rightarrow \sigma \notin \text{Safe} \) is trivial, so \( \text{Safe} \) satisfies (3.1) and we conclude that \( \text{Safe} \) is a safety property. \( \square \)

**Theorem 2:** Any safety property specified by a deterministic Buchi automaton can be specified by a safety recognizer.

**Proof.** Let \( P \) be a safety property specified by a deterministic Buchi automaton \( m_P \) with initial state \( q_0 \). Construct \( m_{\text{safe}(P)} \) with transition function \( \delta_{\text{safe}(P)} \) from \( m_P \) as follows.


(1) Delete all states from which no infinite-accepting state is reachable.

(2) Make all remaining states infinite-accepting.

The resulting automaton satisfies SR, so it is a safety recognizer. Let Safe(P) be the property specified by m_{Safe(P)}.

Notice that P \subseteq Safe(P). This is because the states deleted in step (1) of the construction of m_{Safe(P)} cannot be reached in an accepting run of m_p and step (2) in the construction cannot cause a sequence accepted by m_p to be rejected by m_{Safe(P)}.

It remains to show that Safe(P) \subseteq P. Suppose \sigma \models Safe(P); we must show \sigma \models P. For any arbitrary \tau, let q = \delta_{Safe(P)}(q_0, \sigma[\tau..]). By construction of m_{Safe(P)}, there must exist a sequence of program states \beta_0 and an infinite-accepting state q_1 of m_p such that \delta_{Safe(P)}(q, \beta_0) = q_1. We can now construct a series of finite sequences \beta_1, \beta_2, ..., where each \beta_j causes m_p to enter an infinite-accepting state when started in the infinite-accepting state that it is left in by \beta_{j-1}. This is possible due to step (1) in the construction of m_{Safe(P)}, which ensures that an infinite-accepting state is reachable from every automaton state. Define \beta = \beta_0 \beta_1 .... Clearly, \sigma[\tau..] \beta \models P because \sigma[\tau..] \beta causes m_p to enter an infinite-accepting state infinitely often. Since P is a safety property, we conclude \sigma \models P due to (3.1). □

Recognizing Liveness

Define a liveness recognizer to be a deterministic Buchi automaton in which

LR1: All states have transitions defined for every program state.

LR2: There is a path from every automaton state to an infinite-accepting state.

LR1 ensures that "bad things" are not possible for a liveness recognizer; LR2 ensures that a "good thing" is always possible. Buchi automaton m_{liveness} of Figure 2.4 is an example of a liveness recognizer.

There is a natural correspondence between liveness recognizers and liveness properties. To prove this, we require the following formal definition of liveness properties [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

Liveness: (\forall \alpha: \alpha \in S^*: (\exists \beta: \beta \in S^*: \alpha \beta \models P)) \tag{3.3}

Now we can prove that liveness recognizers and liveness properties specified by deterministic Buchi automata are equivalent.
Theorem 3: Liveness recognizers specify only liveness properties.

Proof. Assume $m_{\text{Live}}$ is a liveness recognizer for a property $\text{Live}$. We must show that $\text{Live}$ satisfies (3.3).

Let $\sigma$ be a finite sequence. To show that (3.3) holds, we must show that there is an infinite sequence $\beta$ such that $\sigma\beta=\text{Live}$. Due to LR1, $m_{\text{Live}}$ cannot attempt an undefined transition upon reading $\sigma$. Thus, $\sigma$ leaves $m_{\text{Live}}$ in some automaton state $q$. Due to LR2, there is a path of automaton states from $q$ to some infinite-accepting state $q'$. Let $\beta_0$ be a finite input that takes $m_{\text{Live}}$ from $q$ to $q'$. Again, by LR2, there must be a path from $q'$ to an infinite-accepting state $q''$. Let $\beta_1$ be a finite input that takes $m_{\text{Live}}$ from $q'$ to $q''$. This argument can be repeated, resulting in an infinite sequence $\beta = \beta_0\beta_1\ldots$. Moreover, $\sigma\beta$ causes $m_{\text{Live}}$ to be in some infinite-accepting state infinitely often. Thus, $\sigma\beta$ is accepted by $m_{\text{Live}}$, and so $\sigma\beta=\text{Live}$ and (3.3) holds. $\square$

Theorem 4: Any liveness property specified by a deterministic Buchi automaton can be specified by a liveness recognizer.

Proof. Let $P$ be a liveness property specified by a deterministic Buchi automaton $m_P$ with transition function $\delta_P$ and initial state $q_0$. Construct $m_{\text{Live}(P)}$ with transition function $\delta_{\text{Live}(P)}$ from $m_P$ as follows.

1. Delete states from which no infinite-accepting state is reachable.
2. Add a new infinite-accepting state $q_i$ that has a transition to itself on all input symbols.
3. For every state $q$ that has an undefined transition on any input symbol $s$, add a transition from $q$ to $q_i$ under $s$.

The resulting automaton satisfies LR1 and LR2, hence it is a liveness recognizer. Let $\text{Live}(P)$ be the property specified by $m_{\text{Live}(P)}$.

Notice that $P \subseteq \text{Live}(P)$. This is because the states deleted in step (1) of the construction of $m_{\text{Live}(P)}$ cannot be reached in an accepting run of $m_P$ and steps (2) and (3) in the construction cannot cause a sequence accepted by $m_P$ to be rejected by $m_{\text{Live}(P)}$.

It remains to show that $\text{Live}(P) \subseteq P$. Suppose $\sigma \text{Live}(P)$ and, by way of contradiction, $\sigma \not\in P$. Since $\sigma \not\in P$, we conclude that $\sigma_i$ appears infinitely often in the run of $m_{\text{Live}(P)}$ on $\sigma$. Let $i$ be the smallest integer such that $\delta_{\text{Live}(P)}(q_0, \sigma[i]) = q_i$. Since $\sigma \not\in P$, due to the construction of $m_{\text{Live}(P)}$, $\delta_P(q_0, \sigma[i])$ is undefined or there is no path in $m_P$ from $\delta_P(q_0, \sigma[i])$ to an infinite-accepting state. In either case, $m_P$ will reject infinite sequence $\sigma[i]\beta$ for any $\beta \in S^\omega$. Thus, $P$ does not satisfy (3.3). This contradicts the assumption that $P$ is a liveness
4. Partitioning into Safety and Liveness

Given a deterministic Buchi automaton, it is not difficult to construct a safety recognizer and a liveness recognizer that specify properties whose intersection is the original property. This shows that every property that is specified by a deterministic Buchi automaton is equivalent to the conjunction of a safety property and a liveness property that can each be specified by deterministic Buchi automata.

**Theorem 5:** Given a property \( P \) specified by a deterministic Buchi automaton \( m_P \), there are properties \( P_{\text{Safe}} \) and \( P_{\text{Live}} \) with recognizers \( m_{\text{Safe}} \) and \( m_{\text{Live}} \) such that

(i) \( m_{\text{Safe}} \) is a safety recognizer,
(ii) \( m_{\text{Live}} \) is a liveness recognizer, and
(iii) \( P = P_{\text{Safe}} \cap P_{\text{Live}} \).

**Proof.** Construct safety recognizer \( m_{\text{Safe}} \) as in the proof of Theorem 2. Construct liveness recognizer \( m_{\text{Live}} \) as in the proof of Theorem 4. It remains to show that

\( P = P_{\text{Safe}} \cap P_{\text{Live}} \).

Suppose an infinite sequence \( \sigma \) is accepted by \( m_P \). To show that \( P \subseteq P_{\text{Safe}} \cap P_{\text{Live}} \), we must show that \( \sigma \) is accepted by both \( m_{\text{Safe}} \) and \( m_{\text{Live}} \). Step (2) in the construction of \( m_{\text{Safe}} \) and steps (2) and (3) in the construction of \( m_{\text{Live}} \) cannot cause a sequence accepted by \( m_P \) to be rejected by either recognizer. The states deleted in step (1) of both constructions cannot be reached in an accepting run of \( m_P \). So, deleting them will not cause a sequence accepted by \( m_P \) to be rejected by either \( m_{\text{Safe}} \) or \( m_{\text{Live}} \). Thus, both \( m_{\text{Safe}} \) and \( m_{\text{Live}} \) accept \( \sigma \).

Now suppose an infinite sequence \( \sigma \) is not accepted by \( m_P \). We must show that either \( m_{\text{Safe}} \) or \( m_{\text{Live}} \) rejects \( \sigma \). Since \( m_P \) rejects \( \sigma \), either (i) it makes an undefined transition on \( \sigma \), or (ii) \( m_P \) does not enter an infinite-accepting state after some finite prefix of \( \sigma \). In case (i), \( m_{\text{Safe}} \) does not accept \( \sigma \). In case (ii), on reading \( \sigma \), \( m_p \) loops in non-infinite-accepting states. Either all of these non-infinite-accepting states were deleted from \( m_{\text{Safe}} \) in step (1) of its construction, in which case \( \sigma \) will be rejected by \( m_{\text{Safe}} \), or else they were not deleted in either \( m_{\text{Safe}} \) or \( m_{\text{Live}} \) (since step (1) is the same for both) and therefore \( m_{\text{Live}} \) will reject \( \sigma \).

The construction of Theorem 5 is now illustrated for \( m_{\text{nc}} \) of Figure 2.1 which specifies Total Correctness. The safety recognizer is:
The liveness recognizer is:

However, $m_{\text{Live}(c)}$ can be simplified by combining the three infinite-accepting states, resulting in the equivalent liveness recognizer:
5. Proof Obligations for Safety and Liveness

One can think of a deterministic Buchi automaton $m$ that specifies a property $P$ as simulating—in an abstract way—any program $\pi$ that satisfies $P$. This forms the basis for an approach to program verification described in [Alpern & Schneider 85b]. In that approach, a program $\pi$ is specified in terms of

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

To prove that every history of $\pi$ is in $P$, i.e., $\pi$ satisfies $P$, a set of assertions, called correspondance 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 $q_i$ and one variant function $v_\kappa$ for each reject knot $\kappa$, where a reject knot is a maximal strongly connected subset of automaton states in $m$ containing no infinite-accepting states.

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

**Correspondence Basis:** \(\forall j: (Init_\pi \land T_{0j}) \Rightarrow C_j\). \hspace{1cm} (5.1)

**Correspondence Induction:** For all $\alpha: \alpha \in A_\pi$:

For all $i$: \(q_i \in Q:\)

\(\{C_i\} \leftrightarrow \{\land_{j: q_j \in Q} (T_{ij} \Rightarrow C_j)\}\) \hspace{1cm} (5.2)

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

**Transition Basis:** $Init_\pi \Rightarrow \bigvee_{j: q_j \in Q} T_{0j}$ \hspace{1cm} (5.3)
Transition Induction: For all $\alpha$: $\alpha \in A_{\omega}$
For all $i$: $q_i \in Q$
\[ \{C_i\} \alpha \{ \forall j : q_j \in Q \} \] (5.4)

The final two obligations ensure that $\pi$ does not loop forever in non-infinite accepting states when reading a history of $\pi$.

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

Knot Variance: For each reject knot $k$:
For all $\alpha$: $\alpha \in A_{\omega}$
For all $q_i \in k$
\[ \{C_i \land \neg v_{\pi}(q_i)=\forall \} \alpha \{ \forall j : q_j \in k \land (T_{ij} \land C_j) \Rightarrow v_{\pi}(q_j)<\forall \} \] (5.6)

Soundness and relative completeness of the approach is proved in [Alpern & Schneider 85b].

Returning to safety recognizers, observe that due to SR a safety recognizer has no reject knots. Thus, (5.5) and (5.6) are trivially satisfied by a safety recognizer. This means that proving that a program satisfies a safety property never requires a variant function (or well-foundedness argument). The remaining proof obligations for a safety recognizer constitute an invariance argument. We, therefore, conclude that safety properties are proved using only invariance arguments.

Returning to liveness recognizers, observe that, due to LR1, undefined transitions are not possible, so (5.3) and (5.4) are trivially satisfied when trying to prove that a program $\pi$ satisfies a property specified by a liveness recognizer. A liveness recognizer 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 because (5.1) and (5.2) must be satisfied.

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 appeared in [Alpern & Schneider 85a]. In addition, in [Alpern & Schneider 85a], we proved that every property can be expressed as the conjunction of a safety property and a liveness property. That proof is based on a topology in which safety properties correspond to the closed sets and liveness properties to the dense sets. The automata-
theoretic proof of this paper more closely parallels the informal definitions of safety and liveness in terms of "bad things" and "good things".

In [Sistla 85], an attempt is made to give syntactic characterizations for safety and liveness properties that are expressed in temporal logic. Deductive systems are given for safety and liveness formulas in a temporal logic with "eventually", but without "next", or "until". However, 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 Buchi automaton for a safety property can be partitioned into "good" and "bad" states, where "bad" states are never entered in an accepting run. This result is equivalent to Theorem 2 of the current paper.

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; [Lichtenstein et al. 85] consider it liveness. The definitions in [Lichtenstein et al. 85] are based on existing temporal logic inference rules (proof obligations) whereas our definitions are independent of proof techniques. This 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 apply to a larger class of properties.

Acknowledgments

As usual, David Gries provided helpful comments on a draft of this paper.

References


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


[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.


