Verifying Safety Properties Using Non-deterministic Infinite-state Automata*

Nils Klarlund**
Fred B. Schneider

TR 89-1036
September 1989

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

* This material is based on work supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under grant no. CCR-870113, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.
** Supported by a grant from the University of Aarhus, Denmark.

Verifying Safety Properties Using Non-deterministic Infinite-state Automata*

Nils Klarlund† Fred B. Schneider

September 8, 1989

Abstract

A new class of infinite-state automata, called safety automata, is introduced. Any safety property can be specified by using such an automaton. Sound and complete proof obligations for establishing that an implementation satisfies the property specified by a safety automaton are given.

1 Introduction

A central problem in program verification is establishing that an implementation satisfies some specification of interest. Various ways to solve this problem have been explored [7]. One recent and promising direction is to extract proof obligations for an implementation directly from an automaton formulation of that specification. This approach was first introduced in [3] for a limited class of specifications and was subsequently extended in [5] and [14] to handle any specification that could be expressed using finite-state (Büchi) automata.

*This material is based on work supported in part by the Office of Naval Research under contract N00014-86-K-0092, the National Science Foundation under Grant No. CCR-8701103, and Digital Equipment Corporation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

†Supported by a grant from the University of Aarhus, Denmark.

However, the class of specifications that can be expressed using Büchi-automata does not include all properties of interest. There exist important properties that only can be expressed using infinite-state automata. Properties described by first-order temporal formulas with universal quantification over global symbols [13] are examples of such properties. These properties include real-time response as well as specifications involving unbounded buffers. The methods in [5] and [14] also suffer from another limitation: they do not directly deal with properties expressed using non-deterministic automata. Yet, in some cases, using non-determinism allows more natural and more concise specifications because different assumptions about future courses of events can be considered independently. In contrast, determinism requires all such assumptions to be considered at the same time and can lead to a state-explosion in the automaton.

This paper extends automaton-based approaches for verification to include properties expressible by a class of non-deterministic infinite-state automata, called *safety automata*, that are powerful enough to specify any safety property. Safety properties assert that some "bad thing" does not happen during execution of an implementation. Examples of safety properties include deadlock-freedom, where the bad thing is occurrence of deadlock; mutual exclusion, where the bad thing is simultaneous access of several processes to a shared resource; and real-time response properties such as "a reply is received within 5 seconds", where the bad thing is that no reply is received by the 6th second.

Safety automata cannot specify liveness properties, properties asserting that some "good thing" does happen during execution. The automata used in [5] and [14] could express certain liveness properties. Thus, at the expense of not handling liveness properties, the approach described in this paper extends automaton-based verification to the class of all safety properties.

The main contribution of this paper is to give sound and complete proof obligations for deducing that a given implementation satisfies the property specified by a safety automaton. These obligations are presented in two forms. First, they are given in terms of an invariant relating states of the implementation and of the automaton. Second, the obligations are given in a style similar to that of Hoare's programming logic [9].

The organization of this paper is as follows. In section 2, we introduce a notation for describing infinite-state, non-deterministic automata.

Section 3 defines safety automata and proves that the properties they can specify is exactly the class of safety properties. Automata to model implementations are described in section 4. Next, in section 5, we give sound and complete proof obligations to verify that an implementation satisfies a property specified by a safety automaton. In section 6, these proof obligations are reformulated as Hoare triples; and, sound and complete decomposition principles that allow triples to be broken into simpler ones are given. The method is illustrated in section 7 with an example. Finally, relation to other work and conclusions appear in section 8 and section 9.

2 Properties and Automata

Formally, a *property* defines a set of infinite sequences of *events*. Events may be states of an implementation or actions of an implementation—our results are independent of the view taken, hence the neutral term "event". An *implementation* Π is regarded as a generator of events, and the property $L(\Pi)$ defined by Π is the set of sequences of events generated by executing Π. A *specification* Σ also defines a property. This property, $L(\Sigma)$, consists of all event sequences that satisfy the specification. An implementation Π *satisfies* a specification Σ if and only if $L(\Pi) \subseteq L(\Sigma)$, that is every event sequence generated by Π satisfies Σ.

2.1 Automata Diagrams

An automaton defines the property consisting of exactly those sequences of events that it accepts. As an example, for a system whose interface consists of a register[1] `N` and a green light, consider the property[2] $\mathcal{P}$ containing all sequences in which the first event is loading `N` with some value n such that subsequently the green light is flashed at most n times. If events model actions, then $\mathcal{P}$ concerns two types of actions, "loading register `N`" and "flashing the green light", although an implementation might perform other actions as well. If events model states, then $\mathcal{P}$ concerns states, where a state, among other things, assigns a value to `N` and indicates whether the green light is illuminated.

[1]Typewriter font is used for variables in programs or specifications.

[2]Caligraphic letters $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$ and $\mathcal{S}$ are used for properties.

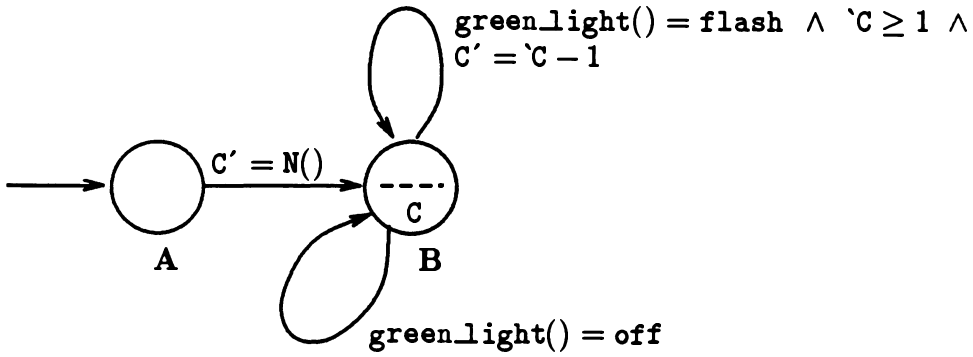

Figure 1: $A_{\mathcal{P}}$, an example automaton diagram.

We employ *automaton diagrams* to define the infinite-state automata we use to specify properties. Figure 1 is an example of an automaton diagram. It represents an automaton $A_{\mathcal{P}}$ for property $\mathcal{P}$, assuming events are actions. An automaton diagram is a directed graph, not unlike the usual representation of a finite state automaton [10]. However, in order to describe an infinite state space, the diagram includes *node variables*. This permits a single node to denote a set of automaton states. For example, the diagram of Figure 1 has two nodes[3] A and B, where node B introduces a node variable C by naming it under a dashed line. Node variable C is used as a counter that bounds the maximum number of times the green light may still flash in a given execution.

Each node in an automaton diagram contributes a set of automaton states. In Figure 1, automaton states defined by node B have the form $(\mathbf{B}, \langle C \rangle)$, where C is an integer. Node A defines a single state, $(\mathbf{A}, \langle\rangle)$, abbreviated $(\mathbf{A})$. Thus, the set of automaton states for $A_{\mathcal{P}}$ is $\{(\mathbf{A})\} \cup \{(\mathbf{B}, \langle C \rangle) \mid \text{integer}(C)\}$. In general, a node P that introduces node variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$ defines automaton states of the form $(\mathbf{P}, \langle X_1, \ldots, X_n \rangle)$, where $\langle X_1, \ldots, X_n \rangle$, also denoted $\vec{X}$, is a *value assignment* that associates the value X_i with node variable $\mathtt{X}_i$. We say that an automaton is *in a node* P if the automaton is in a state of the form $(\mathbf{P}, \langle \ldots \rangle)$.

[3]Boldface capital letters are used to designate nodes of a automaton diagram.

Edges in an automaton diagram are labeled by *transition predicates.*[4] A transition predicate labeling an edge between a node `P and a node P′ may mention node variables of `P, node variables of P′, and *event functions.* An event function f(e) maps an event e to a value. The value of an event function f() appearing in a transition predicate is f(e), where e is the event on which the transition is taken.

In a transition predicate, we write `X to denote a node variable X that is introduced by `P and write Y′ to denote a node variable Y introduced by P′. Transition predicates define automaton transitions in the expected way: an event e may cause a transition from a state (`P,$\vec{`X}$) to a state (P′,$\vec{X}'$) if there is an edge from node `P to node P′ in the automaton diagram and this edge is labeled by a transition predicate satisfied by $\vec{`X}$ and $\vec{X}'$ and e. For example, consider the self-loop in Figure 1 labeled

$$\texttt{green\_light}() = \text{flash} \wedge \texttt{`C} \geq 1 \wedge \texttt{C}' = \texttt{`C} - 1.$$

It asserts that if an event occurs for which the event function green_light() has value "flash" and the automaton is in a state (B,`$C$) with `$C \geq 1$, then a transition is possible to state (B,C') where $C' =$ `$C - 1$ holds. By convention, an explicit predicate need to be not given for a variable that does not change its value during a transition. Thus in Figure 1, the predicate `C = C′ is assumed for the transition corresponding to the self-loop labeled green_light() = off.

For automaton states `$q$ and $q'$, we write `$q \xrightarrow{e} q'$ if the automaton may enter state q' from state `$q$ when event $e$ occurs. The relation $\rightarrow$ is called the *transition relation* for the automaton. When `$q \xrightarrow{e} q'$ holds, q' is said to be a *successor state* of `q on e.

For example, the transitions of $A_{\mathcal{P}}$ from node B to node B are

$$(\mathrm{B},\texttt{`}C) \xrightarrow{e} (\mathrm{B},C')$$

if and only if

$$(\texttt{green\_light}(e) = \text{flash} \wedge C' = \texttt{`}C - 1 \wedge \texttt{`}C \geq 1)$$
$$\vee\ (\texttt{green\_light}(e) = \text{off} \wedge C' = \texttt{`}C).$$

[4] Although here we define automata by means of predicate logic, our verification method is not dependent on any assumptions about recursiveness of transitions and state spaces.

An *initial node* P of an automaton diagram is a node that has an incoming edge with no origin node. Such an edge may be labeled with a predicate restricting values of the *initial states* that correspond to node P. The set of all initial states is the set of states of initial nodes. In Figure 1, there is only one edge without an origin node. This edge terminates at A, so the set of initial states for $A_{\mathcal{P}}$ is $\{(\mathrm{A})\}$.

In general, an infinite-state automaton A is defined by a four-tuple $(\mathcal{E}, Q, Q^0, \rightarrow)$ where

- $\mathcal{E}$ is the (finite or countable) set of events,
- Q is the (finite or countable) set of automaton states,
- $Q^0 \subseteq Q$ is the set of initial states, and
- $\rightarrow$ is the transition relation.

An automaton A defines a property $L(A)$, consisting of all words *accepted* or *generated* by the automaton. An infinite event sequence, $w = e_0, e_1, \ldots$ in $\mathcal{E}^\omega$ is accepted by A if and only if there is a *run* of A over w, where a run of A over w is a sequence of automaton states $q_0, q_1, \ldots$ such that q_0 is an initial state (i.e. $q_0 \in Q^0$), and $q_0 \xrightarrow{e_0} q_1 \xrightarrow{e_1} q_2 \ldots$ holds.[5] An automaton is *deterministic* if it has only one inititial state and if for all states $`q$ and all events $e$ there is at most one state $q'$ such that $`q \xrightarrow{e} q'$. $A_{\mathcal{P}}$ of Figure 1 is a deterministic automaton.

Finally, we give some notation that will be required later. Consider the automaton $A = (\mathcal{E}, Q, Q^0, \rightarrow)$. For $u = e_0, \ldots, e_n \in \mathcal{E}^*$ and $`q, q' \in Q$, we write $`q \xrightarrow{u} q'$ instead of

$$\exists q_0, \ldots, q_{n+1} \in Q : q_0 \xrightarrow{e_0} \cdots \xrightarrow{e_n} q_{n+1} \ \wedge \ q_0 = `q \ \wedge \ q_{n+1} = q'.$$

Note that if u is the empty string, then $`q \xrightarrow{u} q'$ if and only if $`q = q'$. We say that automaton state q is *reachable by* u if $q_0 \xrightarrow{u} q$ for some $q_0 \in Q^0$. State q is *reachable* if it is reachable by u for some $u \in \mathcal{E}^*$.

[5] $\mathcal{E}^\omega$ is the set of all infinite words over $\mathcal{E}$.

3 Safety automata

The informal characterization of safety—that no "bad thing" happens—can be formalized as follows [2]. A property $\mathcal{S}$ is a *safety property* if and only if every infinite sequence of events $w \notin \mathcal{S}$ has some finite prefix u such that no infinite extension $v \in \mathcal{E}^\omega$ exists that makes $u \cdot v \in \mathcal{S}$ hold.[6] Here, prefix u defines the "bad thing". In terms of topology, a safety property defines a closed set of words under the natural topology on $\mathcal{E}^\omega$. This topology is induced by defining a basic open set to have the form $u \cdot \mathcal{E}^\omega$ for $u \in \mathcal{E}^*$.[7] An open set, the countable union of basic open sets, can be written $\bigcup_i u_i \cdot \mathcal{E}^\omega$, where each u_i is in $\mathcal{E}^*$. Thus, an open set is a set of words that have common prefixes from a set of finite words. A safety property $\mathcal{S}$, being a closed set and therefore the complement of an open set, has the form $\overline{\bigcup_i u_i \cdot \mathcal{E}^\omega}$. Here, the u_i's represent the "bad things" proscribed by the safety property.

As shown below, safety properties are exactly the properties specifiable by a class of infinite-state automata that we call *safety automata*. Safety automata are infinite-state automata that have a finite set of initial states and a finitely branching transition relation (i.e. for all states s and all events e, the set of successor states on e, $\{s' \mid s \xrightarrow{e} s'\}$, is finite). All deterministic automata are safety automata, and all finite-state automata are safety automata. For example, $A_{\mathcal{P}}$ from section 2 is a safety automaton because it is a deterministic automaton.

The following two propositions generalize results in [4] from finite to infinite-state automata.

Proposition 1 *Safety properties are exactly those definable by deterministic infinite-state automata.*

Proof First, we prove that the property defined by a deterministic infinite-state automaton is a safety property. For an automaton $A = (\mathcal{E}, Q, Q^0, \rightarrow)$, define a *partial run* over a finite word $e_0, \ldots, e_n$ to be a sequence of states $q_0, \ldots, q_{n+1}$ such that $q_0 \in Q^0$ and $q_0 \xrightarrow{e_0} \cdots \xrightarrow{e_n} q_{n+1}$. Let $A_{\mathcal{S}}$ be a deterministic infinite-state automaton, and let the set $\{u_i \mid i \geq 0\}$ be the set of strings u_i in $\mathcal{E}^*$ such that there is no partial run of $A_{\mathcal{S}}$ along u_i. Let $\mathcal{S} = \overline{\bigcup_i u_i \cdot \mathcal{E}^\omega}$, so by construction, $\mathcal{S}$ is a safety property. We now show that $L(A_{\mathcal{S}}) = \mathcal{S}$.

[6] $u \cdot v$ is the sequence obtained by concatenating u with v.

[7] For V a set of sequences, define $u \cdot V$ to be $\{u \cdot v | v \in V\}$.

$L(A_S) \subseteq S$: If w is accepted by A_S, then from the definition of $\{u_i | i \geq 0\}$, w can contain no prefix from among the u_i's. Therefore, $w \in S$.

$S \subseteq L(A_S)$: Let w be in S. There are partial runs of A_S along all prefixes of w. (Otherwise, by definition of S, w would not be in S.) Because the automaton is deterministic, these partial runs are ordered under the prefix ordering (either any two runs are equal or one is a prefix of the other). Their limit defines a run over w. Hence, w is accepted by A_S.

Next, we prove that every safety property can be specified by a deterministic automaton. Consider a safety property $S = \overline{\bigcup_i u_i \cdot \mathcal{E}^\omega}$, where the u_i's are in $\mathcal{E}^*$. To construct an automaton A_S such that $L(A_S) = S$, we proceed as follows. Let $A_S = (\mathcal{E}, \{u_i | i \geq 0\}, \{\epsilon\}, \rightarrow_S)$, where $u \xrightarrow{e}_S v$ if and only if $u \cdot e = v$.[8] This defines a deterministic automaton that checks that its current state—the past sequence of events—does not contain a prefix among the u_i's. Thus, $w \in L(A_S)$ if and only if w does not have a prefix that is in $\{u_i | i \geq 0\}$ and therefore, $w \in L(A_S)$ if and only if $w \in S$. □

Properties defined by non-deterministic, infinite-state automata need not be safety properties. To see this, consider a property asserting that a green light will flash eventually. This property is not a safety property because any prefix is remediable: simply extend the prefix with an event in which the green light flashes. A non-deterministic automaton can recognize this property—the automaton starts by guessing when the green light will flash and counts down to that event, checking that the green light has flashed before the counter reaches 0. Because the set of initial states for this automaton is infinite, the automaton is not a safety automaton. In fact, the restrictions in the definition of safety automata limit their expressiveness to that of deterministic infinite-state automata:

Proposition 2 *All safety automata express safety properties.*

Proof A construction, similar to the classic subset construction for finite-state automata [10], can be applied to construct a deterministic automaton A_d from a finitely branching automaton $A = (\mathcal{E}, Q, Q^0, \rightarrow)$ such that $L(A) = L(A_d)$. Let the deterministic automaton A_d be $(\mathcal{E}, F(Q), \{Q^0\}, \rightarrow_d)$, where $F(Q)$ denotes the set of finite subsets of Q and $M \xrightarrow{e} N$ if and only

[8] ϵ denotes the empty string.

if $N = \{s' | \exists `s \in M \text{ s.t. } `s \xrightarrow{e} s'\} \neq \emptyset$. The result now follows from Proposition 1. □

Virtues of Non-Determinism

Although non-deterministic safety automata are no more expressive than deterministic ones, non-determinism allows some safety properties to be specified more naturally. This is because non-determinism is often used implicitly when we reason about the future. For example, the property

$\mathcal{R}$: (a) p is true and then q will always be true, or
(b) r is true and then s will always be true

is expressed in a way that views the future as a disjunction of two different courses of events, namely (a) and (b). Here, p, q, r and s are (not necessarily disjoint) predicates about events, and each course of events contains expectations about the future that are not implied by the present. For example, "p is true and then q will always be true" is a course of events containing the implicit assumption that q will be true for the future, although this is not necessarily the case just because p is true of the present. Having p true of the present does not necessarily ensure this course of events.

Property $\mathcal{R}$ is specified both using a non-deterministic automaton $A_{\mathcal{R}}$ and a deterministic automaton $A_{\mathcal{R}_d}$ in Figure 2. Observe that for $A_{\mathcal{R}}$, each of the two initial nodes corresponds to a different course of events—P corresponds to course (a) and Q corresponds to course (b).

Intuitively, a deterministic automaton like $A_{\mathcal{R}_d}$ will be more complicated than its non-deterministic counterpart because each state in $A_{\mathcal{R}_d}$ must incorporate information about possible courses of events. Each state of a non-deterministic automaton (like $A_{\mathcal{R}}$) need not take into account many possible courses because the state itself represents an assumption about the course. For example, if both p and r are true for the first event, $A_{\mathcal{R}_d}$ makes a transition from PR to QS because both a continuation of course (a) and of course (b) are possible. In contrast, $A_{\mathcal{R}}$ can "guess" P or R, depending on the course that will transpire.

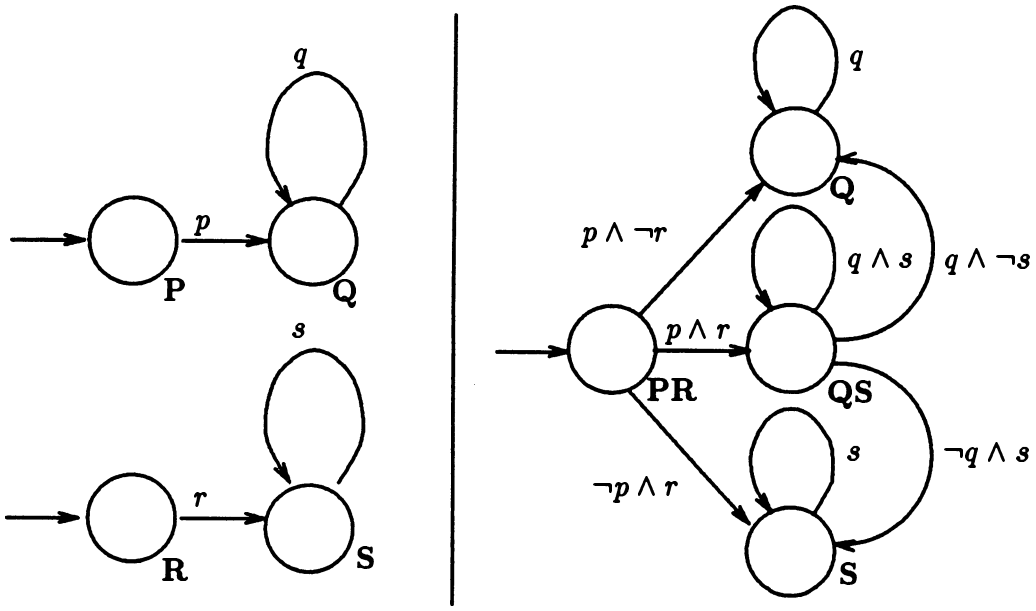

Figure 2: $A_{\mathcal{R}}$ (left) and $A_{\mathcal{R}_d}$ (right).

4 Implementation automata

Just as we can use safety automata to describe specifications, we use *implementation automata* to describe implementations. An implementation automaton is an infinite-state automaton. Since dead states can be deleted from an infinite-state automaton without altering the language accepted by that automaton [8], we assume that implementation automata have no dead states.[9] An implementation automaton is a safety automaton if its set of initial states is finite and its transition relation is finitely branching. In the absence of unbounded non-determinism, all programs can be represented as safety automata. The verification method presented in this paper, however, works even for implementation automata that are not safety automata.

An implementation automaton $A_\Pi = (\mathcal{E}, Q_\Pi, Q^0_\Pi, \rightarrow_\Pi)$ for a program Π is defined as follows. The event set $\mathcal{E}$ is the set of program states; the set Q_Π of automaton states is also the set of program states; the set Q^0_Π of initial automaton states is the set of initial program states; and the transition relation is defined such that $`p \xrightarrow{p'}_\Pi p'$ if and only if some atomic action of the program transforms the program state from $`p$ to p'.

Figure 3 is an example of program Π_{ex} that runs forever. The program state is defined by a program counter and two variables, a and b. Thus, a program state is a pair $p = (\ell, \vec{x})$, where ℓ denotes the value of the program counter, and $\vec{x} = \langle a, b \rangle$ is a value assignment. To formulate the implementation automaton

$$A_{\Pi_{ex}} = (\mathcal{E}, Q_{\Pi_{ex}}, Q^0_{\Pi_{ex}}, \rightarrow_{\Pi_{ex}})$$

for this program, let the domain of program labels be denoted $\mathcal{L}$ and the domain of value assignments be denoted $\mathcal{V}$. Thus, every program state p is an element of $\mathcal{L} \times \mathcal{V}$. Define $\mathcal{E}$ and state space $Q_{\Pi_{ex}}$ to be $\mathcal{L} \times \mathcal{V}$; define $Q^0_{\Pi_{ex}} = \{(\ell_0, \vec{x}) | \vec{x} \in \mathcal{V}\}$; and define transition relation $\rightarrow_{\Pi_{ex}}$ so that, for example, it includes $(\ell_2, \langle 15, 30 \rangle) \xrightarrow{(\ell_3, \langle 16, 30 \rangle)}_{\Pi_{ex}} (\ell_3, \langle 16, 30 \rangle)$.

[9] A *dead state* in an automaton is a reachable state from which it is not possible to extend any partial run.

```
ℓ0 :  ⟨a, b := 0, 0⟩
ℓ1 :  do true →
ℓ2 :     ⟨a := a+1⟩
ℓ3 :     ⟨b := b+2⟩
      od
```

Figure 3: Program Π_{ex}.

5 Proof Obligations

We now turn to the question of establishing that an implementation satisfies a specification. In particular, we develop proof obligations for demonstrating that $L(A_\Pi) \subseteq L(A_\Sigma)$, where $A_\Pi = (\mathcal{E}, Q_\Pi, Q^0_\Pi, \rightarrow_\Pi)$ is an implementation automaton and $A_\Sigma = (\mathcal{E}, Q_\Sigma, Q^0_\Sigma, \rightarrow_\Sigma)$, called the *specification automaton*, is a safety automaton. Satisfaction of these proof obligations allow us to conclude that implementation Π satisfies specification Σ.

The proof obligations relate states of the implementation automaton to sets of states of the specification automaton and ensure that this correspondence is maintained during each transition of the implementation automaton. To simplify the exposition, we refer to states of the specification automaton as *specification states* and states of the implementation automaton as *implementation states*. Define a *configuration* to be any finite set of specification states. Configurations are used to describe sets of reachable states of the specification (safety) automaton. Transitions on an event e between configurations $\mathcal{C}$ and C', denoted $\mathcal{C} \xrightarrow{e} C'$, occur if and only if

(i) $C' \neq \emptyset$
(ii) $\forall s' \in C' : \exists s \in \mathcal{C} : s \xrightarrow{e} s'$

Thus, $\mathcal{C} \xrightarrow{e} C'$ means that (i) transition to some new state is possible and (ii) each new state is a successor state of some old state.

The correspondence between implementation states and configurations is described by a predicate $\mathcal{I}$ called an *invariant* that associates a set of configurations with each implementation state. We write $\mathcal{I}(p, C)$ to denote

that configuration C satisfies invariant $\mathcal{I}$ on implementation state p. Two proof obligations suffice to demonstrate that the correspondence asserted by the invariant is maintained.

O1: $p \in Q_\Pi^0 \;\Rightarrow\; (\exists C : \mathcal{I}(p,C) \;\wedge\; \emptyset \subsetneq C \subseteq Q_\Sigma^0\,)$

O2: $\grave{p} \xrightarrow{e} p' \;\wedge\; \mathcal{I}(\grave{p},\grave{C}) \;\Rightarrow\; (\exists C' : \grave{C} \xrightarrow{e} C' \;\wedge\; \mathcal{I}(p',C'))$

Obligation O1 ensures that for every possible initial implementation state, there must be a corresponding non-empty set of initial specification states. O2 ensures that if the implementation automaton can make a transition from $\grave{p}$ to p' on e and $\grave{C}$ is a possible configuration corresponding to $\grave{p}$, there must be some configuration C' such that $\grave{C} \xrightarrow{e} C'$ and C' corresponds to implementation state p'.

Demonstrating the correspondence between implementation states and configurations using O1 and O2 is necessary and sufficient to establish that $L(A_\Pi) \subseteq L(A_\Sigma)$ holds. That is, if an invariant can be found satisfying O1 and O2, then $L(A_\Pi) \subseteq L(A_\Sigma)$ holds. And, whenever $L(A_\Pi) \subseteq L(A_\Sigma)$ holds, there is an invariant satisfying O1 and O2.

Proposition 3 *Let A_Π be an implementation automaton and A_Σ be a safety automaton.*
(Soundness) *If there is an invariant $\mathcal{I}$ satisfying* O1 *and* O2, *then $L(A_\Pi) \subseteq L(A_\Sigma)$.*
(Completeness) *If $L(A_\Pi) \subseteq L(A_\Sigma)$, then there is an invariant $\mathcal{I}$ satisfying* O1 *and* O2.

Proof (Soundness) Let $w = e_0, e_1, \ldots$ be accepted by A_Π. Thus, there is a run $p_0 \xrightarrow{e_0} p_1 \xrightarrow{e_1} \cdots$ of A_Π over w. Since p_0 belongs to Q_Π^0, by condition O1 there is a configuration C_0 such that $\mathcal{I}(p_0, C_0)$ and $C_0 \subseteq Q_\Sigma^0$. By condition O2, there is a configuration $C_1 \neq \emptyset$ such that $\mathcal{I}(p_1, C_1)$ and for all $s_1 \in C_1$ there is an $s_0 \in C_0$ and $s_0 \xrightarrow{e_0} s_1$. By repeating this argument, we obtain $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \cdots$. Thus, for any n and any $s \in C_n$ there exist $s_0 \in C_0, \ldots, s_{n-1} \in C_{n-1}$, such that $s_0 \xrightarrow{e_0} \cdots s_{n-1} \xrightarrow{e_{n-1}} s$. We now construct a forest of trees generated as follows. The nodes of the trees are of the form $s_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{n-1}} s_n$ such that for all $0 \leq i \leq n, s_i \in C_i$ holds. The edges are of the form

$$(s_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{n-1}} s_n, s_0 \xrightarrow{e_0} \cdots \xrightarrow{e_n} s_{n+1}),$$

and the roots of the trees are all s_0 such that $s_0 \in C_0$.

This forest defines a run of A_Σ over w, as follows. By definition of a safety automaton Q^0_Σ is finite and $\rightarrow_\Sigma$ is finitely branching. Therefore, the forest consists of a finite collection of finitely branching trees. Hence, by König's lemma, there is an infinite path through one of the trees. This path defines the desired run of A_Σ over w. Hence, w is accepted by A_Σ.

(Completeness) Assume $L(A_\Pi) \subseteq L(A_\Sigma)$. Define $\mathcal{I}$ so that $\mathcal{I}(p, C)$ is true if and only if $C \neq \emptyset$ and p is reachable by u for some finite word $u = e_0, \ldots, e_{n-1}$ (possible the empty word) such that

$$C = \{s | \exists e_0, \ldots, e_{n-1} \text{ and a run } s_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{n-1}} s_n \text{ of } A_\Sigma \text{ with } s_n = s\}$$

Since Q^0_Π is finite and A_Σ is finitely branching, all C's such that $\mathcal{I}(p, C)$ holds are finite.

To prove that O1 is satisfied by this $\mathcal{I}$, assume that $p \in Q^0_\Pi$. A_Π has no dead states because it is an implementation automaton. Therefore, there is an infinite word w that is accepted by A_Π.

Since, by assumption $L(A_\Pi) \subseteq L(A_\Sigma)$, we conclude that $L(A_\Sigma)$ is also non-empty, and $Q^0_\Sigma \neq \emptyset$ holds. By the definition of $\mathcal{I}$, $\mathcal{I}(p, Q^0_\Sigma)$ is true, so O1 is satisfied.

To prove that O2 holds, assume its antecedent—that there exist $\grave{p}$, e, p' and $\grave{C}$ such that $\grave{p} \xrightarrow{e} p'$ and $\mathcal{I}(\grave{p}, \grave{C})$—is true. Thus, there is a finite word u such that $\grave{p}$ is reachable by u and $\grave{C}$ is exactly the set of states that A_Σ might be in after having read u. Define

$$C' = \{s' \mid \exists \grave{s} \in \grave{C} : \grave{s} \xrightarrow{e} s'\}$$

Evidently, p' is reachable by $u \cdot e$, so C' is exactly the set of states A_Σ might be in after having read $u \cdot e$. Thus, by the definition of $\mathcal{I}$, we conclude $\mathcal{I}(p', C')$ holds and it remains to prove $\grave{C} \xrightarrow{e} C'$. To prove $\grave{C} \xrightarrow{e} C'$, we show that conditions (i) and (ii) of the definition of $\rightarrow$ for configurations are satisfied. Condition (ii) is satisfied due to the way C' was just defined. To show that condition (i) is satisfied, note that because p' is reachable by $u \cdot e$ and A_Π contains no dead states, there is an infinite word w such that $u \cdot e \cdot w \in L(A_\Pi)$. By the assumption that $L(A_\Pi) \subseteq L(A_\Sigma)$, we conclude that $u \cdot e \cdot w \in L(A_\Sigma)$.[10] Therefore, the set C' can not be empty and so (i) is satisfied. □

[10]This is the reason we have assumed implementation automata have no dead states.

6 A logic of automaton diagrams

Proof obligation O2 can also be formulated using triples similar to those of Hoare's logic [9]. We require the following notation for this.

An *automaton proof outline* is a program annotated with *automaton assertions*. The assertions define the invariant used in obligations O1 and O2. A *configuration descriptor* $[\mathbf{P}_1,\ldots,\mathbf{P}_r;p_1,\ldots,p_r]$ consists of nodes $\mathbf{P}_1,\ldots,\mathbf{P}_r$ and *configuration predicates* $p_1,\ldots,p_r$ (with $r \geq 1$). Each p_h is a predicate over the values of the program variables $\vec{x}$ and the node variables $\vec{X}_h$ of node $\mathbf{P}_h$. For a program state $(\ell,\vec{x})$, the configurations of the configuration descriptor $[\mathbf{P}_1,\ldots,\mathbf{P}_r;p_1,\ldots,p_r]$ are all sets $\{(\mathbf{P}_1,\vec{X}_1),\ldots,(\mathbf{P}_r,\vec{X}_r)\}$ such that $p_1 \wedge \ldots \wedge p_r$. Configuration descriptors are used to define sets of possible nodes and value assignments that can occur during an execution. Hence, the configuration $\{(\mathbf{P}_1,\vec{X}_1),\ldots,(\mathbf{P}_r,\vec{X}_r)\}$ means that the automaton can be in any node $\mathbf{P}_h$ with value assignment X_h. To make the notation clearer, we write $(\mathbf{P}_1,\ldots,\mathbf{P}_r;\vec{X}_1,\ldots,\vec{X}_r)$ for the configuration $\{(\mathbf{P}_1,\vec{X}_1),\ldots,(\mathbf{P}_r,\vec{X}_r)\}$.

An automaton assertion at a program label ℓ is used to define the invariant associated with program states in which the value of the program counter is ℓ. Such an assertion is given as a set of m configuration descriptors ($m > 0$),

$$
\begin{array}{ll}
 & \{[\mathbf{P}^1_1,\ldots,\mathbf{P}^1_{r_1};p^1_1,\ldots,p^1_{r_1}], \\
(1) & \quad\ldots \\
 & \quad[\mathbf{P}^m_1,\ldots,\mathbf{P}^m_{r_m};p^m_1,\ldots,p^m_{r_m}]\}\,.
\end{array}
$$

This assertion defines $\mathcal{I}((\ell,\vec{x}),C)$, the invariant at ℓ, to hold if and only if $p^i_1 \wedge \ldots \wedge p^i_{r_i}$, where $C = (\mathbf{P}^i_1,\ldots,\mathbf{P}^i_{r_i};\vec{X}^i_1,\ldots,\vec{X}^i_{r_i})$ for some i $(1 \leq i \leq m)$. Thus, C is a *configuration of* assertion (1)—or C *satisfies* the assertion—if and only if it is a configuration of one of the m configuration descriptors in the assertion.

Obligations O1 and O2 can be expressed in terms of automaton assertions of an automaton proof outline. To express O1, assume that the

---

If A_Π had dead states then it might be possible for A_Π to continue from a dead state to a dead state (although not *ad infinitum*) without the specification automaton being able to continue. In such cases, it will not be possible to associate a configuration with a dead state, since rewriting requires configurations to be non-empty.

assertion corresponding the initial program label has form (1). Then using the definition of the invariant given above, we can rewrite O1 as

$$\text{O1}'\text{:}\quad \forall \vec{x}:\ \exists i:\ \exists \vec{X}^i_1,\ldots,\vec{X}^i_{r_i}:\ p^i_1 \wedge \cdots \wedge p^i_{r_i} \\ \wedge\ (\mathbf{P}^i_1, \vec{X}^i_1) \in Q^0_\Sigma\ \wedge \cdots \wedge\ (\mathbf{P}^i_{r_i}, \vec{X}^i_{r_i}) \in Q^0_\Sigma$$

where Q^0_Σ is the set of states determined by the initial nodes.

Obligation O2 can also be expressed in terms of automaton assertions. For any two program labels ℓ_α and ℓ_β such that that ℓ_β can be reached from ℓ_α by executing a single atomic action $\langle \mathrm{S} \rangle$, we can formulate O2 as the *automaton triple*

$$(2)\qquad \begin{array}{ll} & \{[\mathbf{P}^1_1,\ldots,\mathbf{P}^1_{r_1}; p^1_1,\ldots,p^1_{r_1}], \\ & \ \ \ldots, \\ & \ [\mathbf{P}^m_1,\ldots,\mathbf{P}^m_{r_m}; p^m_1,\ldots,p^m_{r_m}]\} \\ \ell_\alpha: & \langle \mathrm{S} \rangle \\ & \{[\mathbf{Q}^1_1,\ldots,\mathbf{Q}^1_{s_1}; q^1_1,\ldots,q^1_{s_1}], \\ & \ \ \ldots, \\ & \ [\mathbf{Q}^n_1,\ldots,\mathbf{Q}^n_{s_n}; q^n_1,\ldots,q^n_{s_n}]\} \\ \ell_\beta: & \end{array}$$

In an automaton triple, the assertion preceding the atomic action is called the *precondition* and the assertion following it is called the *postcondition.* Informally, the meaning of triple (2) is that if C is a configuration satisfying the precondition, then after execution of $\langle \mathrm{S} \rangle$, there is a configuration C' satisfying the postcondition and $C \rightarrow C'$ holds.

The meaning of (2) is a conjuntion of the meanings of triples with somewhat simpler preconditions:

$$(3)\qquad \begin{array}{ll} & \{[\mathbf{P}^i_1,\ldots,\mathbf{P}^i_{r_i}; p^i_1,\ldots,p^i_{r_i}] \\ \ell_\alpha: & \langle \mathrm{S} \rangle \\ & \{[\mathbf{Q}^1_1,\ldots,\mathbf{Q}^1_{s_1}; q^1_1,\ldots,q^1_{s_1}], \\ & \ \ \ldots, \\ & \ [\mathbf{Q}^n_1,\ldots,\mathbf{Q}^n_{s_n}; q^n_1,\ldots,q^n_{s_n}]\} \\ \ell_\beta: & \end{array}$$

Thus, we first formalize the meaning of (3). This triple is valid if and only if whenever the automaton configuration and program state together satisfy the configuration descriptor in the precondition, then executing $\langle \mathrm{S} \rangle$

produces a new program state such that there is some new configuration defined by one of the configuration descriptors of the postcondition and there is a transition from the old configuration to this new configuration on the new program state. More formally, (3) is valid if and only if given a program state $(\ell_\alpha, \vec{x})$ and a new program state $(\ell_\beta, \vec{x}')$ that results from executing $\langle S \rangle$ and value assignments $\vec{X}_1^i, \ldots, \vec{X}_{r_i}^i$ that constitute a configuration of the configuration descriptor in the precondition of (3), then there is some configuration descriptor

(4) $$\{[\mathbf{Q}_1^j, \ldots, \mathbf{Q}_{s_j}^j; q_1^j, \ldots, q_{s_j}^j\}]$$

of the postcondition of (3) and some value assignments $\vec{X}_1'^j, \ldots, \vec{X}_{s_j}'^j$ such that

$$(\mathbf{Q}_1^j, \ldots, \mathbf{Q}_{s_j}^j; \vec{X}_1'^j, \ldots, \vec{X}_{s_j}'^j)$$

is a configuration of (4), and

$$(\mathbf{P}_1^i, \ldots, \mathbf{P}_{r_i}^i; \vec{X}_1^i, \ldots, \vec{X}_{r_i}^i) \xrightarrow{(\ell_\beta, \vec{x}')} (\mathbf{Q}_1^j, \ldots, \mathbf{Q}_{s_j}^j; \vec{X}_1'^j, \ldots, \vec{X}_{s_j}'^j)$$

holds.[11]

Thus, the meaning of (3) is given by

$Trpl_i$:

(5)
$$(\ell_\alpha, \vec{x}) \rightarrow_\Pi (\ell_\beta, \vec{x}') \wedge \bigwedge_{1 \le v \le r_i} p_v^i \Rightarrow$$
$$\exists 1 \le j \le n : \exists \vec{X}_1'^j, \ldots, \vec{X}_{s_j}'^j :$$
$$(\mathbf{P}_1^i, \ldots, \mathbf{P}_{r_i}^i; \vec{X}_1^i, \ldots, \vec{X}_{r_i}^i) \xrightarrow{(\ell_\beta, \vec{x}')} (\mathbf{Q}_1^j, \ldots, \mathbf{Q}_{s_j}^j; \vec{X}_1'^j, \ldots, \vec{X}_{s_j}'^j) \wedge \bigwedge_{1 \le w \le s_j} q_w'^j$$

[11] Here, $\rightarrow$ is the transition relation for configurations defined in section 5:

$$(\mathbf{P}_1^i, \ldots, \mathbf{P}_{r_i}^i; \vec{X}_1^i, \ldots, \vec{X}_{r_i}^i) \xrightarrow{(\ell_\beta, \vec{x}')} (\mathbf{Q}_1^j, \ldots, \mathbf{Q}_{s_j}^j; \vec{X}_1'^j, \ldots, \vec{X}_{s_j}'^j)$$

holds if and only if

$$\forall 1 \le k \le s_j :$$
$$\exists 1 \le h \le r_i :$$
$$(\mathbf{P}_h^i, \vec{X}_h^i) \xrightarrow{(\ell_\beta, \vec{x}')} (\mathbf{Q}_k^j, \vec{X}_k'^j).$$

where $(l_\alpha, \grave{\vec{x}}) \rightarrow_\Pi (l_\beta, \vec{x}')$ denotes the transitions defined by $\langle \mathsf{S} \rangle$ and where the predicate p with all its variables $\vec{\mathbf{x}}$ and $\vec{\mathbf{X}}$ marked with grave accents is denoted $\grave{p}$. Similarly, p' is the predicate p with its variables marked with acute accents.

The meaning of (2) is then

O2′: $\bigwedge_{1 \leq i \leq m} Trpl_i$.

Triple Decomposition

Triples like (2) at first may appear intimidating. Fortunately, such triples can always be decomposed into *simple* triples of the form:

$$\begin{array}{ll} & \{[\mathbf{P}; p]\} \\ \ell_\alpha: & \langle \mathsf{S} \rangle \\ & \{[\mathbf{Q}; q]\} \\ \ell_\beta: & \end{array}$$

According to (5), the meaning of such a triple is

(6) $\quad (\ell_\alpha, \grave{\vec{x}}) \rightarrow_\Pi (\ell_\beta, \vec{x}') \wedge \grave{p} \Rightarrow \exists \vec{X}' : (\mathbf{P}; \grave{\vec{X}}) \rightarrow (\mathbf{Q}; \vec{X}') \wedge q'.$

Decomposition of general triples into simple triples is accomplished by using the following propositions. The first follows directly from the formal definition of (2) as the conjunction of triples having form (3).

Proposition 4 *To verify a triple like (2), it is necessary and sufficient to verify for each* $1 \leq i \leq m$ *that*

$$\begin{array}{ll} & \{[\mathbf{P}^i_1, \ldots, \mathbf{P}^i_{r_i}; p^i_1, \ldots, p^i_{r_i}] \\ \ell_\alpha: & \langle \mathsf{S} \rangle \\ & \{[\mathbf{Q}^1_1, \ldots, \mathbf{Q}^1_{s_1}; q^1_1, \ldots, q^1_{s_1}], \\ & \quad \ldots, \\ & \quad [\mathbf{Q}^n_1, \ldots, \mathbf{Q}^n_{s_n}; q^n_1, \ldots, q^n_{s_n}]\} \\ \ell_\beta: & \end{array}$$

holds.

The next proposition is a case analysis and permits a triple like (3) to be reformulated as a collection of triples, each of which has only one configuration descriptor for its postcondition.

Proposition 5 *The triple*

$$
(7)\qquad
\begin{array}{ll}
 & \{[\mathbf{P}_1,\ldots,\mathbf{P}_r;p_1,\ldots,p_r]\} \\
\ell_\alpha: & \langle \mathsf{S} \rangle \\
 & \{[\mathbf{Q}^1_1,\ldots,\mathbf{Q}^1_{s_1};q^1_1,\ldots,q^1_{s_1}], \\
 & \ldots, \\
 & [\mathbf{Q}^n_1,\ldots,\mathbf{Q}^n_{s_n};q^n_1,\ldots,q^n_{s_n}]\} \\
\ell_\beta: &
\end{array}
$$

holds if and only if there exist predicates c_j *with free variables* $\vec{x},\vec{x}',\vec{X}_1,\ldots,\vec{X}_r$, *where* $1 \leq j \leq n$ *such that*

$$
(8)\qquad \bigvee_{1\leq j\leq n} c_j = true
$$

and for all $1 \leq j \leq n$

$$
(9)\qquad c_j \Rightarrow
\begin{array}{ll}
 & \{[\mathbf{P}_1,\ldots,\mathbf{P}_r;p_1,\ldots,p_r]\} \\
\ell_\alpha: & \langle \mathsf{S} \rangle \\
 & \{[\mathbf{Q}^j_1,\ldots,\mathbf{Q}^j_{s_j};q^j_1,\ldots,q^j_{s_j}]\} \\
\ell_\beta: &
\end{array}
$$

Proof Let *Tpl* be the triple (7) and for $1 \leq j \leq n$, let Tpl_j be the triple in the corresponding consequent of (9). It follows directly from (5) that

$$
(10)\qquad Tpl \equiv \bigvee_{1\leq j\leq n} Tpl_j.
$$

(If) Since, by (9), $c_j \Rightarrow Tpl_j$ holds for $1 \leq j \leq n$, it follows that $(\bigvee_j c_j) \Rightarrow (\bigvee_j Tpl_j)$. Therefore, by (8), $\bigvee_j Tpl_j$ holds and by (10), triple (7) holds.

(Only if) Assume that (7) holds and define for $1 \leq j \leq n$

$$
c_j \equiv Tpl_j.
$$

By (10) and the definition of c_j, $Tpl \equiv \bigvee_j c_j$. Then, by the assumption that (7) holds, (8) is satisfied. Condition (9) is trivially satisfied. □

The next proposition is used to decompose a configuration descriptor in a postcondition so that nodes can be considered one at a time.

Proposition 6 *The triple*

$$\begin{array}{ll} & \{[\mathbf{P}_1,\ldots,\mathbf{P}_r;p_1,\ldots,p_r]\} \\ \ell_\alpha: & \langle \mathsf{S} \rangle \\ & \{[\mathbf{Q}_1,\ldots,\mathbf{Q}_s;q_1,\ldots,q_s]\} \\ \ell_\beta: & \end{array}$$

holds if and only if for all $1 \leq k \leq s$

$$\begin{array}{ll} & \{[\mathbf{P}_1,\ldots,\mathbf{P}_r;p_1,\ldots,p_r]\} \\ \ell_\alpha: & \langle \mathsf{S} \rangle \\ & \{[\mathbf{Q}_k;q_k]\} \\ \ell_\beta: & \end{array} \tag{11}$$

Proof Follows directly from (5). □

Finally, the configuration descriptor in the precondition can be decomposed according to the case analysis principle below.

Proposition 7 *The triple*

$$\begin{array}{ll} & \{[\mathbf{P}_1,\ldots,\mathbf{P}_r;p_1,\ldots,p_r]\} \\ \ell_\alpha: & \langle \mathsf{S} \rangle \\ & \{[\mathbf{Q};q]\} \\ \ell_\beta: & \end{array} \tag{12}$$

holds if and only if there are predicates d_h *with free variables* $\vec{x}$, $\vec{x}'$, $\vec{X}_h$, *for* $1 \leq h \leq r$ *such that*

$$\bigwedge_{1\leq h\leq r} p_h \Rightarrow \bigvee_{1\leq h\leq r} d_h \tag{13}$$

and for all $1 \leq h \leq r$*:*

$$d_h \Rightarrow \begin{array}{ll} & \{[\mathbf{P}_h;p_h]\} \\ \ell_\alpha: & \langle \mathsf{S} \rangle \\ & \{[\mathbf{Q};q]\} \\ \ell_\beta: & \end{array} \tag{14}$$

Proof Let *Tpl* be (12), and for $1 \leq h \leq r$, define Tpl_h to be the triple in the corresponding consequent of (14). It follows from (5) that

(15) $\quad ((\bigwedge_{1\le h\le r} p_h) \wedge Tpl) \equiv ((\bigwedge_{1\le h\le r} p_h) \wedge \bigvee_{1\le h\le r} Tpl_h)).$

(If) Assume that (13) and (14) hold. According to (5), it is sufficient to show that $(\bigwedge_h p_h) \Rightarrow Tpl$. By (13) we have $(\bigwedge_h p_h) \Rightarrow ((\bigwedge_h p_h) \wedge (\bigvee_h d_h))$, so it follows from (14) that $(\bigwedge_h p_h) \Rightarrow ((\bigwedge_h p_h) \wedge (\bigvee_h Tpl_h))$. Thus, $(\bigwedge_h p_h) \Rightarrow (\bigvee_h Tpl_h)$ holds, and by (15), we obtain $(\bigwedge_h p_h) \Rightarrow Tpl$, as desired.

(Only if). Assume that (12) holds, i.e. Tpl is valid, and define $d_h = Tpl_h$ for $1 \le h \le r$. By the assumption, $(\bigwedge_h p_h) \Rightarrow ((\bigwedge_h p_h) \wedge Tpl)$ trivially holds. Then, by (15), $(\bigwedge_h p_h) \Rightarrow (\bigwedge_h p_h) \wedge (\bigvee_h Tpl_h)$ holds, and by the definition of c_j, it follows that $(\bigwedge_h p_h) \Rightarrow (\bigwedge_h p_h) \wedge (\bigvee_h d_h)$ holds. Thus, $(\bigwedge_h p_h) \Rightarrow (\bigvee_h d_h)$ holds. Hence, (13) holds. Condition (14) is trivially satisfied. □

The above decomposition propositions are based on having configurations defined by r configuration predicates, $p_1, \ldots, p_r$—one for each node P_r in the descriptor—rather than by a single predicate. Having these r predicates permits the decomposition of Proposition 6, which is something that would not have been possible if only a single predicate were used.

7 Example

To illustrate the method of section 6, we consider a simple but somewhat contrived example.

Define the property $p;q$ to hold at the present moment if p holds now and q holds at the next instant. Such a property is not uncommon when reasoning about concurrent systems and is formulated in temporal logic as $p \wedge \circ q$. Now, consider the more complicated property

$\mathcal{T}$: Property $p;q$ holds at least every 5 instants.

This property is easy to specify using a non-deterministic automaton, as shown in Figure 4. (Specifying $\mathcal{T}$ using a deterministic automaton is more difficult because $p;q$ might hold in overlapping intervals if p and q are not mutually exclusive. The automaton must keep track of all guesses of such intervals.) A program, $\Pi_{\mathcal{T}}$, purported to satisfy $\mathcal{T}$ with $p \equiv$ p() and $q \equiv$ q() is depicted in Figure 5. There, braces ⟨ and ⟩ in the **if** statement assert that execution of an atomic step from ℓ_1 consists of either executing p:=*true*

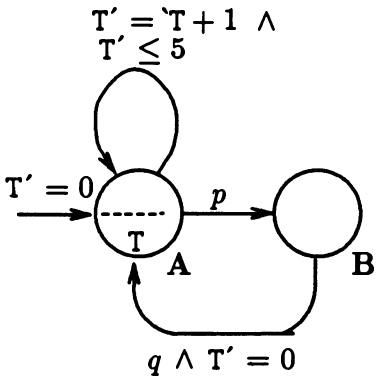

Figure 4: $A_{\mathcal{T}}$.

and setting the program counter to ℓ_2, or executing q:=*true* and setting the program counter to ℓ_0. We now demonstrate that $\Pi_{\mathcal{T}}$ indeed satisfies $\mathcal{T}$. In order to this, the automaton proof outline $PO(\Pi_{\mathcal{T}})$ in Figure 6 is used. (The notation "_" is used there to denote the configuration predicate *true*.) We must show that the proof outline satisfies obligations O1′ and O2′.

Obligation O1′ is valid because the assertion at ℓ_0, $\{[\mathrm{A}; T = 0]\}$, is satisfied by the initial state (A;0) of $A_{\mathcal{T}}$, for any initial program state.

The triple

$$\begin{array}{ll} & \{[\mathrm{A}; T = 0]\} \\ \ell_0: & \langle \mathtt{p},\mathtt{q} := \mathit{true}, \mathit{false} \rangle; \\ & \{[\mathrm{A},\mathrm{B}; T = 1, \_]\} \\ \ell_1: & \end{array}$$

is proved as follows. Using Proposition 6, it is decomposed into the simple triples

$$\begin{array}{ll} & \{[\mathrm{A}; T = 0]\} \\ \ell_0: & \langle \mathtt{p},\mathtt{q} := \mathit{true}, \mathit{false} \rangle; \\ & \{[\mathrm{A}; T = 1]\} \\ \ell_1: & \end{array} \tag{16}$$

and

```
        do
ℓ0 :      ⟨p,q:= true, false⟩;
ℓ1 :      ⟨if
           true  →
               p:=true⟩;
ℓ2 :          ⟨q:=true⟩;
           true  →
               q:=true⟩;
          fi
        od
```

Figure 5: Program Π_T

```
        do  {inv: [A;  T = 0]}
ℓ0 :      ⟨p,q:= true, false⟩;
          {[A,B;T = 1,_]}
ℓ1 :      ⟨if
           true  →
               p:=true⟩;
               {[B;  _]}
ℓ2 :          ⟨q:=true⟩;
           true  →
               q:=true⟩;
          fi
        od
```

Figure 6: Proof outline $PO(\Pi_T)$

$$
\begin{array}{lll}
 & & \{[\mathbf{A}; T = 0]\} \\
(17) & \ell_0: & \langle \mathtt{p,q:=}\ \mathit{true},\ \mathit{false} \rangle; \\
 & & \{[\mathbf{B}; \_]\} \\
 & \ell_1: &
\end{array}
$$

According to (6), the meaning of triple (16) is

$$
\begin{array}{ll}
(18) & ((p' = \mathit{true} \wedge q' = \mathit{false}) \ \wedge \ \grave{T} = 0) \ \Rightarrow \\
 & \quad (\exists T' : (T' = \grave{T} + 1 \wedge T' \leq 5) \ \wedge \ T' = 1).
\end{array}
$$

Here, $(p' = \mathit{true} \wedge q' = \mathit{false})$ is the formulation of the program transition $\rightarrow_\Pi$ defined by the assignment statement at ℓ_0. The predicate $\grave{T} = 0$ is obtained from precondition $\{[\mathbf{A}; T = 0]\}$ by marking the configuration predicate with grave accents; the automaton transition predicate $T' = \grave{T} + 1 \ \wedge \ T' \leq 5$ is obtained from the self-loop of node **A** in Figure 4; and the predicate $T' = 1$ is obtained by marking the configuration predicate in postcondition $\{[\mathbf{A}; T = 1]\}$ with acute accents.

Similarly, the meaning of triple (17) is

$$
(19) \qquad ((p' = \mathit{true} \wedge q' = \mathit{false}) \ \wedge \ \grave{T} = 0) \ \Rightarrow \ (p').
$$

Triples (16) and triple (17) are valid because (18) and (19) are tautologies.

The triple corresponding to the first branch of the **if** statement,

$$
\begin{array}{lll}
 & & \{[\mathbf{A}, \mathbf{B}; T = 1, \_]\} \\
 & \ell_1: & \langle \mathbf{if} \\
 & & \quad \mathit{true} \ \rightarrow \\
 & & \quad\quad \mathtt{p:=}\mathit{true} \rangle; \\
 & & \quad\quad \{[\mathbf{B}; \_]\} \\
 & \ell_2: &
\end{array}
$$

is verified by decomposing it using Proposition 7 and choosing *true* for d_1 and *false* for d_2. Hence, we need to verify

$$
\begin{array}{lll}
 & & \{[\mathbf{A}; T = 1]\} \\
 & \ell_1: & \langle \mathbf{if} \\
 & & \quad \mathit{true} \ \rightarrow \\
\mathit{true} \ \Rightarrow & & \quad\quad \mathtt{p:=}\mathit{true} \rangle; \\
 & & \quad\quad \{[\mathbf{B}; \_]\} \\
 & \ell_2: &
\end{array}
$$

and

$$
\begin{array}{lll}
 & & \{[\mathbf{B};\_]\} \\
 & \ell_1: & \langle \mathbf{if} \\
 & & \quad \mathit{true} \ \rightarrow \\
\mathit{false} \ \Rightarrow & & \quad\ \mathrm{p}:=\mathit{true}\rangle; \\
 & & \quad \{[\mathbf{B};\_]\} \\
 & \ell_2: &
\end{array}
$$

The first formula is equivalent to

$$\mathit{true} \ \Rightarrow \ (((p' = \mathit{true} \wedge q' = \grave{}q) \ \wedge \ \grave{}T = 1) \ \Rightarrow \ (p')),$$

which is a tautology. The second formula is trivially valid.

The triple corresponding to the second branch of the if statement,

$$
\begin{array}{ll}
 & \{[\mathbf{A},\mathbf{B}; T = 1,\_]\} \\
\ell_1: & \langle \mathbf{if} \\
 & \mathit{true} \ \rightarrow \\
 & \quad \mathrm{q}:=\mathit{true}\rangle; \\
\ell_0: & \{[\mathbf{A}; T = 0]\}
\end{array}
$$

is decomposed using Proposition 7 with *false* for d_1 and *true* for d_2, so it is necessary only to verify:

$$
\begin{array}{ll}
 & \{[\mathbf{B};\_]\} \\
\ell_1: & \langle \mathbf{if} \\
 & \mathit{true} \ \rightarrow \\
 & \quad \mathrm{q}:=\mathit{true}\rangle; \\
\ell_0: & \{[\mathbf{A}; T = 0]\}
\end{array}
\qquad (20)
$$

which, according to (6), is

$$((p' = \grave{}p \wedge q' = \mathit{true})) \ \Rightarrow \ (\exists T' : (q' \wedge T' = 0) \ \wedge \ T' = 0),$$

a tautology. Finally, the triple

$$
\begin{array}{ll}
 & \{[\mathbf{B};\_]\} \\
\ell_2: & \langle \mathrm{q}:=\mathit{true}\rangle; \\
 & \{[\mathbf{A}; T = 0]\} \\
\ell_0: &
\end{array}
$$

is the same as (20) except for the labeling.

8 Relation to other work

Use of automata on infinite words for program specification has been studied in different contexts. In [19], Wolper used finite-state automata to define temporal logic operators. Alpern and Schneider [3] demonstrated how invariants could be used to verify that an implementation satisfies a specification expressed by a finite-state deterministic Büchi-automaton. This work was generalized in [5] to Boolean combinations of finite-state deterministic Büchi-automata, and in [14] to $\forall$-automata, finite-state automata that accept a word only if all runs over the word are accepting.

Vardi gave a recursion theoretic view of the verification problem for non-deterministic automata and defined verification conditions in terms of computation trees and product automata [18]. Sistla proved that the verification problem for unbounded non-deterministic automata is Π^1_2-complete [15].

For languages over finite alphabets, relationships between automata and topology are studied in [6]. Results similar to those of section 3 but formulated for stuttering automata, were developed independently in [1].

In [17], Stark presented proof obligations based on automata for temporal logic formulas that have two types of variables: program variables and logical variables. Stated in our terminology, Stark's proof obligations rely on an invariant relation that associates a set of specification states (instead of a set of sets as done in this paper) with each program state. This method is not complete, because it requires all specification states associated with a program state to have succesors on any event. In fact, only some of these specification states might have successors, and therefore, Stark's method cannot deal directly with all non-deterministic specifications. For example, the method cannot be used to prove correctness of the program in section 7 unless the specification is first reformulated as an equivalent deterministic automaton.

Lynch and Tuttle used automata for hierarchical proofs of program correctness in [12]. They employed mappings between implementation and specification automata in a way similar to that of Stark. The method, therefore, suffers from the same incompleteness. Jonsson's mappings [11] are also similar to those of Stark, and therefore also incomplete.

The method of Abadi and Lamport [1] handles more general specifications than ours, because auxiliary liveness properties can be attached to

automata. A simulation function f is found that associates a specification state $f(p)$ with each implementation state p. The method is indirect, as it relies on changing the implementation automaton by adding *history* and *prophecy* variables so that the implementation automaton simulates the specification automaton. Enlarging the implementation automaton with information about the specification automaton ensures the existence of the simulation function f.

Abadi and Lamport's work uses stuttering automata, although this is not essential to their results. A stuttering automaton is one in which repetition of (what we call) events is considered a single event. Since non-stuttering automata can both restrain stuttering and allow time-bounded and unbounded stuttering when needed, we prefer these automata.

Using a similar approach as in [1], Sistla developed proof obligations for the same automata that we consider [16]. He observed that by adding only a history component to the implementation automaton, sound and complete proof obligations can be obtained. These obligations use invariant relations that define multi-valued functions from implementation states to specification states. Adding history information to the implemention automaton makes each implementation state reachable by only one event sequence; hence, only one configuration needs to be associated with each implementation state. In our method, we circumvent the apparent need for a history component by associating multiple configurations with each implementation state.

9 Conclusion

This paper describes a method for verifying that an implementation satisfies any property specified by a safety automaton. Even though all safety properties can be specified by deterministic safety automata, we believe that there are advantages to using non-determinism in specifying safety properties. In particular, the use of non-determinism permits the writer of a specification to make different sets of assumptions about the future. This results in simpler specifications, because the disjunctive nature of non-determinism makes separation between several possible courses of events possible—even before it is clear which one is the actual course.

The paper also describes the first direct method of verification for non-

deterministic automaton specifications. Further, we have showed how the method can be formulated in terms of proof outline assertions. Thus, verification is similar to that of conventional Hoare-style logics.

Acknowledgments

We would like to thank B. Alpern and A. Zwarico for their very helpful comments on an earlier version of this paper.

References

[1] Abadi, M., and Lamport, L. The Existence of Refinement Mappings. *Proc. 2. Symp. on Logic in Computer Science*, IEEE, 1988.

[2] Alpern, B. and F.B. Schneider. Defining liveness. *Information Processing Letters 21*, Oct. 1985, pp. 181–185.

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

[4] Alpern, B. and F.B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, Vol. 2, 1987, pp. 117–126.

[5] Alpern, B. and F.B. Schneider. Proving Boolean Combinations of Deterministic properties. *Proc. Symp. on Logic in Computer Science*, IEEE, 1987.

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

[7] de Bakker, J.W. de Roever,W.-P. and Rozenberg, G. (Eds.). *Current Trends in Concurrency Overviews and Tutorials*. Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, Berlin, 1986.

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

[9] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12, 10, Oct. 1969, pp. 576-580.

[10] Hopcroft, J. and Ullman, J. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[11] Jonsson, B. Modular Verification of Asynchronous Networks. *Proc. Sixth Symp. on the Principles of Distributed Computing*, ACM, 1987, pp. 152–166.

[12] Lynch, N. and Tuttle, M. Hierarchical Correctness Proof for Distributed Algorithms. *Proc. Sixth Symp. on the Principles of Distributed Computing*, ACM, 1987, pp. 137–151.

[13] Manna, Z. and Pnueli, A. Verification of Concurrent Programs: A Temporal Proof System. *Foundations of Computer Science IV, Distributed Systems Part*, J.W. DeBakker and J. Van Leeuwen (eds.), Mathematical Centre Tracts 159, Amsterdam 1983, 163-255.

[14] Manna, Z. and Pnueli, A. Specification and Verification of Concurrent Programs by ∀-automata. *Proc. Fourteenth Symp. on the Principles of Programming Languages*, ACM, 1987, pp. 1–12.

[15] Sistla, A.P. *On Verifying that a Concurrent Program Satisfies a Nondeterministic Specification*, Information Processing Letters, Vol. 32, No 1, July 1989, pp. 17–24.

[16] Sistla, A.P. *A Complete Proof System for Proving Correctness of Nondeterministic Safety Specifications*, Computer and Intelligent Systems Laboratory, GTE Laboratories Inc., 1989.

[17] Stark, E. Proving Entailment Between Conceptual State Specifications. *Theoretical Computer Science*, Vol. 56, North-Holland, 1988, pp. 135–154.

[18] Vardi, M. Verification of Concurrent Programs: The Automata-Theoretic Framework. *Proc. Symp. on Logic in Computer Science*, IEEE, 1987.

[19] Wolper, P. Temporal Logic Can Be More Expressive. *Information and Control 56*, 1–2, 1983, pp. 72–99.