

**Specification and Verification of Concurrent Programs by  
 $\forall$ -automata**

by

**Zohar Manna and Amir Pnueli**

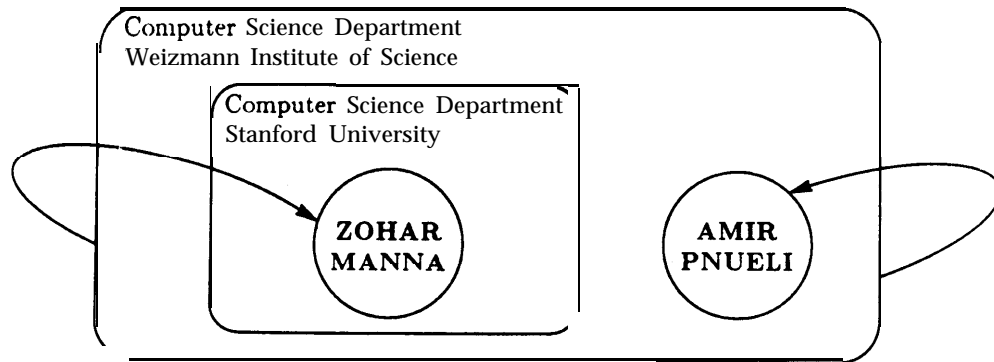
**Department of Computer Science**

**Stanford University  
Stanford, California 94305**





**SPECIFICATION AND VERIFICATION  
OF CONCURRENT PROGRAMS  
BY V-AUTOMATA**



## ABSTRACT

V-automata are non-deterministic finite-state automata over infinite sequences. They differ from conventional automata in that a sequence is accepted if *all* runs of the automaton over the sequence are accepting. These automata are suggested as a formalism for the specification and verification of temporal properties of concurrent programs. It is shown that they are as expressive as extended temporal logic (ETL), and, in some cases, provide a more compact representation of properties than temporal logic. A structured diagram notation is suggested for the graphical representation of these automata. A single sound and complete proof rule is presented for proving that all computations of a program have the property specified by a V-automaton.

## 1. INTRODUCTION

As the field of formal specification and verification of concurrent systems grows more mature, increasing attention should be directed to making the suggested techniques convenient and natural. In earlier stages of the research in this field, the emphasis was put on maximal expressibility of the specification language and universal applicability (completeness) of the verification method.

One approach that was developed during these earlier stages consists of the specification language of temporal *logic* and its associated verification system. It gave a satisfactory answer to the requirements of expressibility and relative completeness of the proof system. However, it soon became apparent that

---

An abbreviated version of this paper appeared in the Proceedings of the 14th Symp. on Principles of Programming Languages (January 1981).

This research was supported by the National Science Foundation under Grant DCR-84-13230 and by the Defense -Advanced Research Projects Agency under Contract N00039-84-C-021 1.

detailed proofs of program properties, using temporal logic, are sometimes tedious to read and follow. Several suggestions were made in order to remove some of the tedious detail and give a more compact representation of such proofs, highlighting the creative elements, which usually are the invariants and the convergence functions used. Notable among these suggestions are several high-level rules, such as the chain rule discussed in [MP2], and the representation of proofs by diagrams ([OL], [MP2]).

Another somewhat unsatisfactory point about temporal proof systems is that so far they have not provided a general and direct reduction of the proof of a temporal property into a set of non-temporal *verification conditions* (also known as *proof obligations*). Traditionally, all the proposed proof systems for sequential programs (e.g., [F], [Ho], [D]) can be summarily described as a reduction of a program property, such as partial or total correctness, into a set of first-order verification conditions expressible in the underlying assertion language. For special temporal-logic formulas that are boolean combinations of the four basic forms  $\Box p$ ,  $\Diamond p$ ,  $\Box \Diamond p$ , and  $\Diamond \Box p$ , where  $p$  is a stake-formula (i.e., a non-temporal formula), a direct reduction to verification conditions over the underlying assertion language is provided by the proof rules presented in [MP2]. However, for more general formulas  $\varphi$ , we have to use *temporal reasoning*, i.e., general theorem-proving methods within temporal logic, to obtain the validity of  $\varphi$  from the validity of simpler formulas of the four basic forms.

One of the often suggested alternatives to the verification of concurrent systems by temporal logic is the use of finite-state automata. Since one of the unique features offered by temporal logic is its ability to deal with infinite computations, the appropriate version to consider is automata over infinite words. As was proved by Wolper ([W]), the expressive power of such automata exceeds that of common temporal logic, and corresponds to a stronger temporal logic called ETL. He also provided a transformation from a propositional temporal formula into an equivalent automaton over infinite inputs. Further research on verification by automata (see [W], [WVS], [VW]) concentrated on the cases of *finite-state* programs, in which the verification problem is decidable. The approach recommended in those papers is to use temporal logic for specification and then, in order to verify a program property, translate the temporal formula into the equivalent automaton and apply automata-theoretic methods to solve the translated verification problem.

A significant improvement in the utilization of automata for the specification and verification of concurrent systems has been recently suggested by Alpern and Schneider ([AS]). They recommend using automata also for specification. Then, they suggest an approach to verification that applies to the general case of programs with possibly infinitely many states. The actual verification method is based on the introduction of invariants and convergence functions, and is very similar to the general verification methods suggested in [OL] and [MP2] for a restricted set of temporal formulas.

The approach of [AS] is based on deterministic *Büchi* Automata. The expressive power of a *single* deterministic Büchi automaton (DBA) is rather limited, and there are interesting program properties which cannot be expressed by such an automaton. Consequently [AS] consider a specification to be presented as a boolean combination of deterministic Büchi automata. For the verification of a property presented in this way, over a given program, they provide several proof rules; one rule handles a single DBA, and the other rule handles the complement of a DBA. Since such a combination is known to express any property specifiable by automata over infinite inputs, the approach attains maximal expressibility.

In this paper we present an alternative uniform approach to the specification and verification of concurrent programs, using finite-state automata, over infinite inputs. The approach is based on a new

type of non-deterministic automata, called  $\forall$ -*automata*. Such an automaton accepts a given input if *all* its possible runs over this input are accepting. We will show that **V-automata** are maximally expressive, i.e., as expressive as ETL. Our automata also lead to a very natural verification method based on invariants and convergence functions, in a style similar to [OL], [MP2], and [AS].

We would also like to promote in this paper a wider use of graphical representation of automata and proof diagrams. It has been our experience, which we would like to share with the readers, that a well-structured diagram often represents the major ideas in a proof in a more concise and lucid form than a string of textual lemmas. A common objection against graphical representation by transition diagrams is that, beyond a very modest size, they become so entangled as to be unreadable. We counter this objection by using a *structured-diagram* notation as used in the proof lattices of [OL] and Statecharts of [Hal].

## 2. VARIABLES AND ASSERTIONS

The following elements will be used both in the programs and in the specification formalism:

$V = \{u_1, u_2, \dots\}$  — A countable set of *variables*. Some of these variables represent *data* variables which can be modified by assignments of the program. Other variables are *control* variables and may represent, for example, the location of the next statement to be executed by the program. We assume that each variable ranges over an appropriate domain, e.g., a data variable may range over the non-negative integers, a control variable may range over a finite set of locations.

$L$  — A language of *assertions*. The language includes all the first-order formulas over the variables in  $V$ . We assume a fixed interpretation of the predicate, function, and constant symbols over the appropriate domains. (To achieve completeness, it is not sufficient to consider a first-order language, and a stronger assertion language is needed. For that purpose, we will introduce fixpoint operators into our language.)

## 3. PROGRAMS AND COMPUTATIONS

With no commitment to a syntax of a particular programming language, we associate with each program  $P$  the following elements:

$\Sigma$  — A set of *program states*. Each program state  $s \in \Sigma$  is a mapping from the variables in  $V$  to their domains. We use the notation  $s[u]$  to denote the value that  $s$  assigns to  $u \in V$ . More generally, for an expression  $e$  over the variables in  $V$ , we denote by  $s[e]$  the value of  $e$  in  $s$ . Similarly, for an assertion  $\varphi$  and a program state  $s$ , we say that  $s$  *satisfies*  $\varphi$ , and write  $s \models \varphi$ , if  $s[\varphi] = \top$ , that is, evaluating  $\varphi$  over  $s$  yields the truth value  $\top$ . In this case we refer to  $s$  as a *p-state*.

$T$  — A finite set of *transitions*. In our model, each transition  $t \in T$  is represented by an enabling condition  $p_t$ , which is a quantifier-free assertion, and a transformation  $\bar{u}_t := \bar{e}_t$ , assigning to a finite list of variables  $\bar{u}_t$  a finite list of expressions  $\bar{e}_t$ . If  $s$  and  $s'$  are two program states such that  $s \models p_t$  and  $s' = (s; \bar{u}_t : s[\bar{e}_t])$ , that is,  $s$  satisfies  $p_t$  and  $s'$  is obtained from  $s$  by reassigning the values  $s[\bar{e}_t]$  to

the variables  $\bar{u}_t$ . then we say that,  $s'$  is the  $t$ -successor of  $s$  and write  $s' = t(s)$ . If  $s \models p_t$ , we say that  $t$  is *enabled* otherwise  $t$  is *disabled* on  $s$ , and then  $t(s)$  is undefined.

$\Theta$  A *precondition*. This is an assertion specifying initial values for some variables, and conditions **that other** variables satisfy in the first program state.

$F$  -- A finite set of *fairness requirements*. Each requirement is a pair  $\langle \varphi, \psi \rangle$  of two quantifier-free assertions. The intended meaning of the fairness requirement  $\langle \varphi, \psi \rangle$  is that a computation that has infinitely many  $\varphi$ -states should also have infinitely many  $\psi$ -states. In a typical application,  $\varphi$  may state that a transition  $t$  is enabled, while  $\psi$  states that it is activated. In this case, the requirement  $\langle \varphi, \psi \rangle$  states that if  $t$  is enabled infinitely many times, it must, also be activated infinitely many times.

A program state  $s$ , such that  $t(s)$  is disabled for all  $t \in T$  is called *terminal* for  $P$ .

A *computation* of a program  $P$  specified by the above elements is a finite or infinite sequence of states

$$\sigma : s_0, s_1, s_2, \dots$$

such that the following requirements are met:

(1) *Initialzty*:  $s_0 \models \Theta$ .

(2) *Consecution*: For each  $i$ ,  $0 \leq i < |\sigma|$ , there exists a transition  $t \in T$  such that  $s_{i+1} = t(s_i)$ .

(3) *Termination*: Either  $\sigma$  is infinite, or it terminates in a state  $s_k$  that is terminal for  $P$ .

(4) *Fairness*: For each fairness requirement  $\langle \varphi, \psi \rangle \in F$ , either  $\sigma$  contains only finitely many  $\varphi$ -states, or  $\sigma$  contains infinitely many  $\psi$ -states.

For a finite computation  $\sigma = s_0, s_1, \dots, s_k$ , we denote by  $|\sigma|$  the index of the last state in  $\sigma$ . We write  $|\sigma| = \omega$  to denote the fact that  $\sigma$  is infinite.

## 4. V-AUTOMATA

A V-automaton  $A$  is specified by the following elements:

$Q$  — A finite set of *automaton states*.

$R \subseteq Q$  — A set of *recurrent* states. These are states that some good runs are expected to visit infinitely many times.

$S \subseteq Q$  — A set of *stable* states. These are states that some good runs visit exclusively, from a certain point on.

$E$  — A finite set of *entry conditiona*. With each  $q \in Q$ , we associate an assertion  $e(q) \in E$  that characterizes the condition under which the automaton may start its activity in  $q$ .

$C$  — A finite set of *transition conditions*. With each pair  $q, q' \in Q$ , we associate an assertion  $c(q, q') \in C$  that characterizes the condition under which the automaton may move from  $q$  to  $q'$ .

Any automaton states  $q \in Q$ , such that  $\epsilon(q) = F$ , can never appear as the first automaton state in a run. To those states  $q \in Q$  whose entry condition  $\epsilon(q)$  is different from  $F$  we refer as *initial* states, implying that they can appear as the first automaton state in a run.

The two sets  $R$  and  $S$  are the generalization of the notion of accepting states to the case of infinite inputs. For convenience, we denote by  $B = Q - R - S$  the set of *non-accepting (bad)* states. The sets  $R$  and  $S$  may have a nonempty intersection.

$\forall$ -automata are intended to specify computations of programs. Therefore, we define the notion of a  $\forall$ -automaton  $A$  accepting a computation.

Let  $\sigma$  be a computation. A *run* of  $A$  over  $\sigma$  is a sequence of automaton states

$$r : q_0, q_1, q_2, \dots$$

such that:

- The first program state  $s_0$  satisfies the entry condition associated with  $q_0$ , i.e.,  $s_0 \models e(q_0)$ .
- For each  $i$ , such that  $0 \leq i < |r| - 1$ ,  $s_{i+1} \models c(q_i, q_{i+1})$ .
- Either  $|r| = |\sigma|$ , i.e., the run and the computation have equal lengths, or  $|r| = i < |\sigma|$ , but then  $s_{i+1} \not\models c(q_i, q)$  for every  $q \in Q$ , which means that the run cannot be extended beyond  $q_{|r|}$  to an automaton state consistent with the program state  $s_{i+1}$ .

We refer to runs for which  $|r| = |\sigma|$  as *complete* runs, and to runs for which  $|r| < |\sigma|$  as *incomplete*.

We can describe the behavior of the automaton  $A$ , when generating a run  $r$  over a computation  $\sigma$ , as follows. Initially, it chooses an automaton state  $q_0 \in Q$  such that  $s_0$ , the first program state in  $\sigma$ , satisfies  $\epsilon(q_0)$ . The automaton state  $q_0$  is the first state in the run. Everafter, let the automaton be at automaton state  $q_i$ , at position  $i = 0, 1, \dots$  in the run. It then reads the next program state  $s_{i+1}$  from  $\sigma$  and non-deterministically chooses to move to a next automaton state  $q_{i+1}$ , provided  $s_{i+1} \models c(q_i, q_{i+1})$ .

In the case that there is no  $q_{i+1}$  such that  $s_{i+1} \models c(q_i, q_{i+1})$ , the run cannot be extended, and we obtain an incomplete run.

We define now the notion of a run  $r$  of  $A$  over a computation  $\sigma$  being accepting.

- An incomplete run is never accepting.
- A finite complete run  $r$  of  $A$  is defined to be *accepting* if its last (automaton) state belongs to  $R \cup S$ .
- For an infinite complete run  $r$ , let  $Inf(r) \subseteq Q$  denote the set of (automaton) states that appear infinitely many times in  $r$ . The infinite run  $r$  is defined to be *accepting* if:
  - $Inf(r) \cap R \neq \emptyset$ , i.e., some of the states appearing infinitely many times in  $r$  belong to  $R$ , or
  - $Inf(r) \subseteq S$ , i.e., *all* the states appearing infinitely many times in  $r$  belong to  $S$  (equivalently, from a certain point on, only states belonging to  $S$  appear in  $r$ ).

Note that if we define  $Inf(r)$  for a finite run  $r$  to be a singleton set consisting of the last state of  $r$ , then the definition of acceptance given above applies to both finite and infinite complete runs.

A V-automaton  $A$  *accepts* a computation  $\sigma$  if *all* the possible runs of  $A$  over  $\sigma$  are accepting. This definition embodies the main difference between V-automata and conventional finite automata (which can be called  $\exists$ -automata) in the way they treat non-determinism.

A run which is not accepting is called rejecting. If a computation  $\sigma$  has at least one rejecting run  $r$ , then the automaton  $A$  does not accept the computation  $\sigma$ . In this case, we say that  $A$  rejects the computation  $\sigma$ .

Two V-automata  $A$  and  $\tilde{A}$  are defined to be equivalent if they accept precisely the same set of computations.

If a V-automaton  $A$  accepts all computations of the program  $P$ , we say that  $A$  is *valid* over  $P$ .




Clearly, the automaton  $A$  is valid over a program  $P$ , if for each computation  $\sigma$  of  $P$ ,

- (a) All complete runs of  $A$  over  $\sigma$  are accepting.
- (b)  $A$  has no incomplete runs over  $\sigma$ .

We define  $A$  to be weakly *valid* over  $P$ , if clause (a) above holds for every computation of  $P$ . Thus, weak validity allows some rejecting runs over computations of  $P$ , provided they are incomplete.

## 5. REPRESENTATION BY DIAGRAMS

It is useful and illuminating to represent V-automata by diagrams. The basic conventions for such representations are the following:

- The automaton states are represented by nodes in a directed graph.
- Each initial state is marked by a small arrow, called the *entry* edge, pointing to it 
- Directed edges, drawn as arrows, connect some of the states.
- Each state belonging to  $R$  is represented by a diamond shape inscribed within a circle 
- Each state belonging to  $S$  is represented by a square inscribed within a circle 

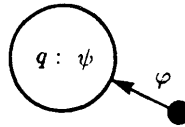
The diagram contains assertions that label both nodes and edges (i.e., entry edges and edges between nodes). Unlabeled nodes and edges are implicitly labeled with the assertion  $\top$ .

The assertions labeling nodes and edges in the diagram define the set of entry conditions and transition conditions of the associated automaton as follows:

- Let  $q \in Q$  be a node in the diagram corresponding to an initial automaton state. Let  $v$  be the



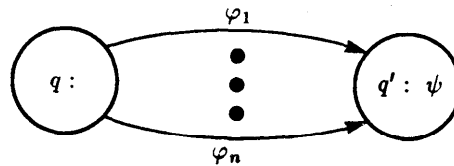
assertion labeling the node  $q$ , and  $\varphi$  be the assertion labeling the entry edge.



Then, the entry condition  $e(q)$  is given by:

$$e(q) = \varphi \wedge \psi.$$

- Let  $q, q'$  be two nodes in the diagram corresponding to automaton states. Let  $\psi$  be the assertion labeling node  $q'$  and  $\varphi_1, \dots, \varphi_n$  the assertions labeling *all* the edges connecting  $q$  to  $q'$



Then, the transition condition  $c(q, q')$  is given by:

$$c(q, q') = (\varphi_1 \vee \varphi_2 \vee \dots \vee \varphi_n) \wedge \psi.$$

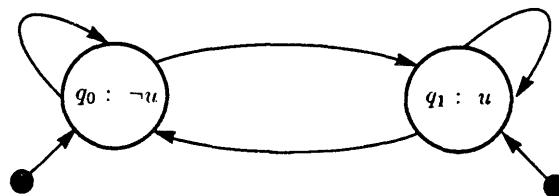
Note that this convention allows putting into the label of  $q'$  any conjunctive factor  $\psi$  that is common to the labels of the edges entering  $q'$ . Note also that if there is no edge connecting  $q$  to  $q'$ , then  $c(q, q') = F$ .

Since V-automata and their diagram representations are suggested as a specification language, we list below several examples of simple temporal-logic formulas and their representation by diagrams.

**Example 1:** The following automaton specifies the temporal property

$$\diamond \square u,$$

i.e.,  $u$  holds continuously from a certain point on:

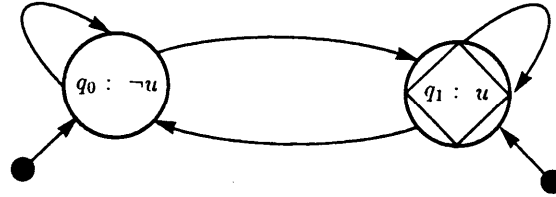


This automaton is actually deterministic. It accepts a computation iff from a certain point on all program states satisfy  $u$ . This is obvious since such a computation leads to a (unique) run that stays in  $q_1$  from a certain point on. Clearly, for this automaton  $R = 0, S = \{q_1\}$ .  $\blacksquare$

**Example 2:** As our next example, consider an automaton specifying the temporal property

$$\square \diamond u,$$

i.e.,  $u$  holds infinitely many times:

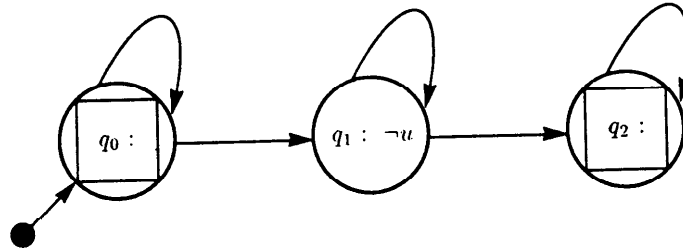


This automaton is similar to the previous one, but differs in its acceptance sets  $R$  and  $S$ . For the present automaton  $R = \{q_1\}$ ,  $S = \emptyset$ , and hence a computation is accepted iff it has infinitely many 'u-states, causing the automaton to visit  $q_1$  infinitely many times.  $\lrcorner$

Actually, from a theoretical point of view, the set  $R$  of recurrent states is redundant. This is stated in the following proposition.

**Proposition:** For every V-automaton  $A$ , there exists an equivalent V-automaton  $\tilde{A}$ , effectively derivable from  $A$ , such that  $\tilde{R} = \emptyset$ .

**Example 3:** We first illustrate the proposition by presenting an  $R$ -less automaton for the temporal property of Example 2,  $\square \diamond u$ , which seems to use the set  $R$  in an essential way (having  $S = \emptyset$ ).



This automaton is non-deterministic.

Consider first a computation  $\sigma$  that contains infinitely many  $u$ -states. One possible run over  $\sigma$  stays forever in  $q_0$  and is accepting. Any run over  $\sigma$  that enters  $q_1$  will eventually reach a later  $u$ -state in  $\sigma$ , which will force it to proceed to  $q_2$  and remain there forever. Hence all runs over  $\sigma$  are accepting, and therefore the automaton accepts  $\sigma$ .

Consider next an infinite computation  $\sigma'$  which has only finitely many  $u$ -states. For such a computation we can devise a run  $r$  that stays in  $q_0$  until the last  $u$ -state is passed. It then proceeds to  $q_1$  and stays there forever. This run is obviously rejecting, and hence the automaton rejects the computation  $\sigma'$ .  $\lrcorner$

**Proof of the Proposition:** Let  $A$  consist of the components  $Q$ ,  $R$ ,  $S$ ,  $E$ , and  $C$ . Without loss of generality we may assume that the acceptance sets  $R$  and  $S$  are disjoint, i.e.,  $R \cap S = \emptyset$ . If they are

not disjoint, it can be shown that the automaton  $A'$ , which is identical to  $A$  in all components except for the acceptance sets that are given by  $R' = R$  and  $S' = S - R$ , is equivalent to  $A$ . Trivially, for the automaton  $A'$ ,  $R' \cap S' = \emptyset$ .

Let  $Q' = \{q' | q \in Q\}$ ,  $Q'' = \{q'' | q \in Q\}$  be two disjoint copies of the set  $Q$ . In general for any subset  $K \subseteq Q$ , we denote by  $K'$  and  $K''$  the subsets  $\{q' | q \in K\}$  and  $\{q'' | q \in K\}$ , respectively, referring to the corresponding copies of  $K$  in  $Q'$  and  $Q''$ .

The general idea of the construction is to let  $A$  consist of two copies of the automaton  $A$ , whose sets of states are denoted, respectively, by  $Q'$  and  $Q''$ . The first copy has similar entry and transition conditions as  $A$ . In addition, for each pair of states  $q_1, q_2 \in Q$ , we allow a transition between  $q'_1$  and  $q''_2$ , whose transition condition is  $c(q'_1, q''_2) = c(q_1, q_2)$ . This allows a run to proceed for awhile in the first copy, and then, non-deterministically, to switch and continue within the second copy. The second copy has a structure which is essentially similar to that of  $A$ , except that we set all transition conditions  $c(q''_1, q''_2)$ , for  $q_1 \in R$ , to  $\mathbf{F}$ . This causes the  $R''$ -states to become traps, i.e., once a run enters such a state it cannot continue.

As acceptance sets for the automaton  $\tilde{A}$ , we take  $\tilde{R}$  to be empty and  $\tilde{S}$  to consist of  $Q' \cup R'' \cup S''$ . The intention of the construction is that for an infinite run  $\tilde{r}$  over  $A$  to be rejecting, it must eventually switch to the second copy, where it never visits an  $R''$ -state and visits infinitely many times some non- $S''$ -states.

We define the following components of  $\tilde{A}$ :

$\tilde{Q} = Q' \cup Q''$ , that is,  $\tilde{Q}$  consists of the two disjoint copies of  $Q$ .

$\tilde{R} = \emptyset$ , as stated by the proposition.

$\tilde{S} = Q' \cup R'' \cup S''$ . Thus the stable states in  $\tilde{A}$  are all the states in  $Q'$  and all the accepting states of both types in  $Q''$ .

The set  $\tilde{E}$  of entry conditions is defined as follows: For each  $q \in Q$ ,

$$\tilde{e}(q') = \tilde{e}(q'') = e(q).$$

The set  $\tilde{C}$  of transition conditions is defined by the following cases: For each  $q_1, q_2 \in Q$ ,

$$\tilde{c}(q'_1, q'_2) = c(q_1, q_2)$$

$$\tilde{c}(q'_1, q''_2) = c(q_1, q_2)$$

$$\tilde{c}(q''_1, q''_2) = \begin{cases} c(q_1, q_2) & \text{if } q_1 \notin R \\ \mathbf{T} & \text{if } q_1 = q_2 \in R \\ \mathbf{F} & \text{otherwise} \end{cases}$$

$$\tilde{c}(q''_1, q'_2) = \mathbf{F}.$$

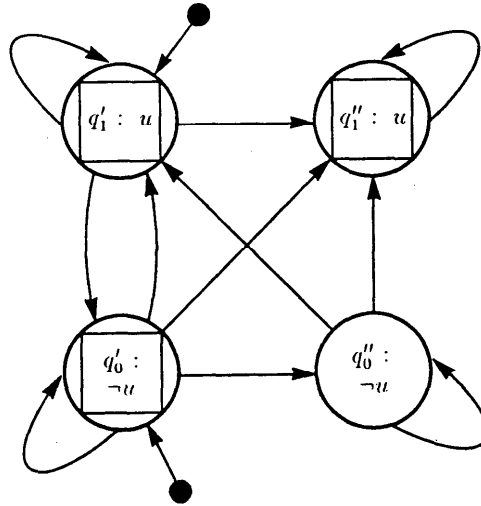
Thus, transitions within  $Q'$ , or from  $Q'$  to  $Q''$ , have conditions identical to the corresponding transitions in  $A$ . The same holds for all transitions within  $Q''$  that do not depart from  $R''$ -states. All the  $R''$ -states are trap states, in the sense that once a run reaches such a state, it remains there forever. No transitions are allowed from  $Q''$  back to  $Q'$ .

We claim that a computation is rejected by  $A$  iff it is rejected by  $\tilde{A}$ . This will establish that  $A$  and  $\tilde{A}$  are equivalent.

We will show only the case of infinite computations. Assume that a computation  $\sigma$  is rejected by A. This means that there exists a rejecting computation  $r$  which, from a certain point on (say after step  $k \geq 0$ ), never visits an  $R$ -state and visits infinitely many times a non- $s$ -state. We can construct a run  $\tilde{r}$  of A that simulates  $r$  in the first copy  $Q'$  up to step  $k$ , where it switches to  $Q''$  and continues the simulation there. It is easy to see that  $\tilde{r}$  visits infinitely many non- $S''$ -states and does not get trapped in an  $R''$ -state. Consequently,  $\tilde{r}$  is a rejecting run of A, causing A to reject  $\sigma$ .

Similarly, given a rejecting run  $\tilde{r}$  of  $\tilde{A}$ , it is easy to simulate it by a run  $r$  of A, that moves to  $q_i$  whenever  $\tilde{r}$  moves to  $q'_i$  or to  $q''_i$ . Since a rejecting run of  $\tilde{A}$  must eventually move to  $Q''$ , it will visit infinitely many non- $S''$ -states and only finitely many  $R'$ - and  $R''$ -states. Consequently  $r$  is also rejecting.  $\blacksquare$

The automaton presented in Example 3 is an improved version of the general construction. Literally applying the construction, described in the proof, to the automaton of Example 2 yields the following equivalent automaton:



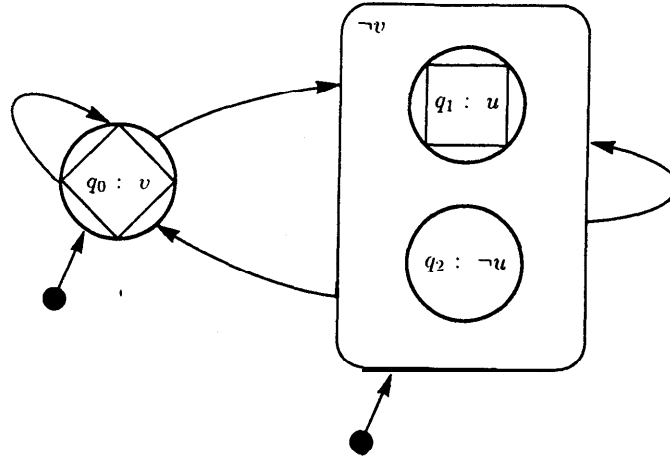
Following the syntax of State&arts [Hal], we introduce two additional conventions that lead to more compact and structured representation of diagrams. We introduce the notion of *super-states* represented as boxes containing other states. As a general rule we interpret any construct associated with a super-state to be associated with every contained state. The two applications of this general rule are:

- An edge connecting super-states  $\hat{q}$  to  $\hat{q}'$  is equivalent to a set of edges connecting each  $q \in \hat{q}$  to each  $q' \in \hat{q}'$ .
- An assertion  $\psi$ , labeling a super-state  $\hat{q}$ , should be added as a conjunct to the label of each contained state  $q \in \hat{q}$ .

**Example 4:** Consider the following diagram representation of the temporal property

$$\diamond \square u \vee \square \diamond v.$$

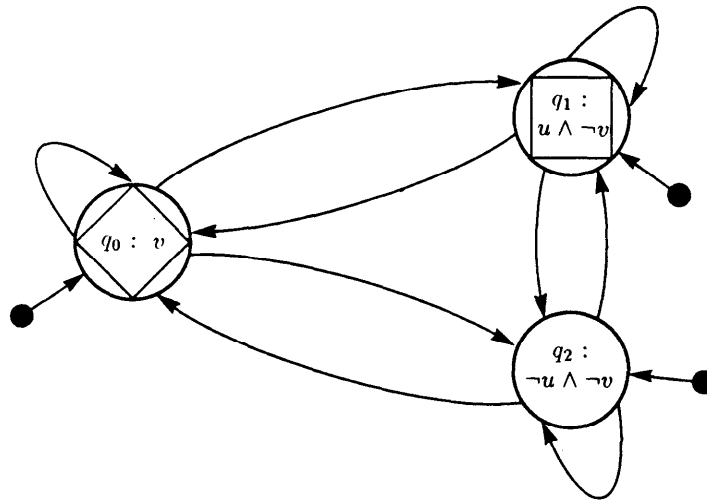
i.e.,  $u$  holds continuously from a certain point on or  $v$  holds infinitely many times:



We show that this deterministic automaton accepts a computation iff it has the required property.

Obviously a run  $r$  over a computation  $\sigma$  is accepting iff either  $r$  visits  $q_0$  infinitely many times or it is restricted to  $q_1$  from a certain point on. The first case is possible iff infinitely many program states in  $\sigma$  satisfy  $v$ . The second case is possible iff all program states in  $\sigma$  beyond a certain point satisfy  $u \wedge \neg v$ . That is,  $r$  is accepting iff it satisfies the given property.

This structured representation is equivalent to the following flat representation (not using any of the structured conventions):



A  $t/\tau$ -automaton is called *complete* if the following requirements are met:

- (a)  $(\bigvee_{q \in Q} e(q)) \equiv \top$ .
- (b) For every  $q \in Q$ ,  $(\bigvee_{q' \in Q} c(q, q')) \equiv \top$ .

These two requirements guarantee that all runs are complete. Since any partial run over  $\sigma$  can always be extended to the full length of  $\sigma$ . Clearly, a complete automaton is valid over a program  $\mathcal{P}$  iff it is weakly valid over  $\mathcal{P}$ .

In many cases, we will restrict ourselves to complete automata. This is not a real restriction since any automaton  $A$  can be transformed to an equivalent complete automaton  $A'$ .

To see this, consider an incomplete automaton  $A$ . To construct  $A'$ , we add to  $Q$ , the set of states of  $A$ , an additional error state  $q_E$ , which is not included in either  $R'$  or  $S'$ . Thus, we define

$$Q' = Q \cup \{q_E\}$$

$$R' = R, \quad S' = S$$

For every  $q, \tilde{q} \in Q$ , we define  $e'(q) = e(q)$  and  $c'(q, \tilde{q}) = c(q, \tilde{q})$ .

In addition, we define the entry condition for  $q_E$  by

$$e'(q_E) = \neg \left( \bigvee_{q \in Q} e(q) \right),$$

and the transition conditions by

$$c'(q_E, q_E) = \text{T},$$

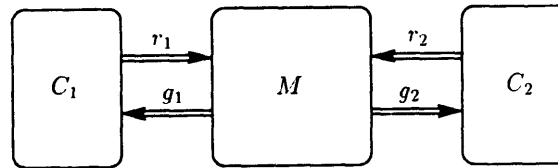
$$c'(q_E, q) = \text{F} \quad \text{for each } q \in Q,$$

$$c'(q, q_E) = \neg \left( \bigvee_{q' \in Q} c(q, q') \right) \quad \text{for each } q \in Q.$$

Thus, an incomplete run that has nowhere else to go can proceed to  $q_E$ , but then must reject.

**Example 5** (Resource Manager):

As a more extensive example consider a system consisting of a *resource manager*  $M$  and two customers,  $C_1$  and  $C_2$ .



The customers communicate with the manager by shared boolean variables  $r_i$  and  $g_i$ ,  $i = 1, 2$ . The protocol of communication between the manager and customer  $C_i$  can be expressed by the following cycle:

$r_i := \text{T} \dashv\vdash C_i$  sets  $r_i$  to T, signaling a request for the resource.

$g_i := \text{T} \dashv\vdash M$  sets  $g_i$  to T, signaling  $C_i$  that the resource is granted.

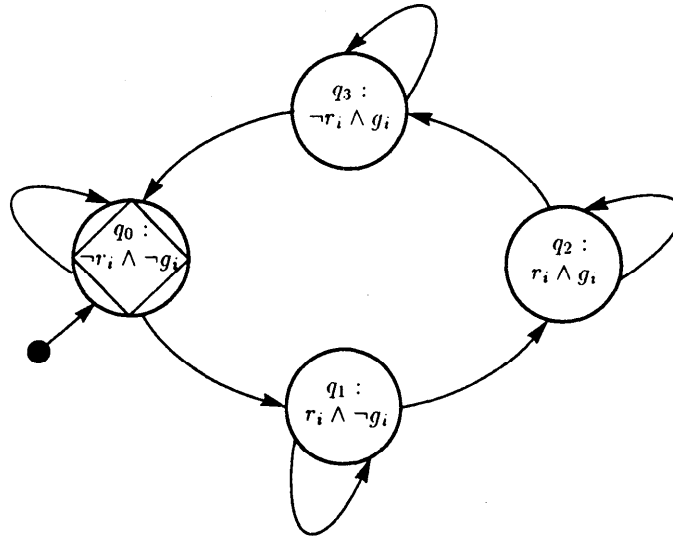
$r_i := \text{F} \dashv\vdash C_i$  resets  $r_i$  to F, signaling a release of the resource.

$g_i := \text{F} \dashv\vdash M$  resets  $g_i$  to F, acknowledging the release.

Under the assumption that there is only one resource, it is required that the resource is never granted to more than one customer at a time.

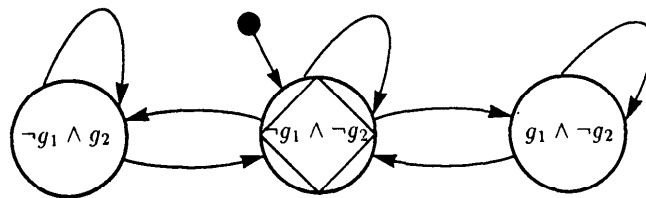
We present a specification of this system by a *set* of automata. The system satisfies the specification if each of its computations is accepted by every automaton in the set.

The first automaton specifies that the communication between the manager and the customer  $C_i$  (for  $i = 1, 3$ ) precisely follows the protocol described above.



Note that this automaton is incomplete. Consequently, whenever the automaton observes a program state satisfying  $g_i$ ,  $\neg r_i$ ,  $\neg g_i$  or  $r_i$ , while being at the automaton state  $q_0$ ,  $q_1$ ,  $q_2$  or  $q_3$ , respectively, it generates an incomplete run. Such a computation is therefore rejected. In addition to the safety requirement, that the four events of setting and resetting the communication variables follow the periodical sequence described above, this automaton also contains a liveness requirement, by which the state  $q_0$  should be visited infinitely many times. This implies that the computation cannot stay forever in any of  $q_1$ ,  $q_2$ ,  $q_3$ , and forces the eventual occurrence of the next event in the protocol.

The other automaton specifies the integrity of the resource, expressed by the requirement that it is always granted to at most one customer.



Note that in addition to the safety property  $\Box \neg(g_1 \wedge g_2)$ , this automaton specifies the liveness property that no  $C_i$  holds the resource forever.

For comparison, let us consider the temporal specification of the property expressed by the first

automaton. This property can also be expressed by the following temporal formula:

$$\begin{aligned}
& (\neg r_i \wedge \neg g_i) \wedge \\
& \square \left[ r_i \rightarrow (\neg g_i) \mathcal{U} (\neg g_i \wedge r_i) \right] \wedge \\
& \square \left[ r_i \rightarrow (r_i) \mathcal{U} (r_i \wedge g_i) \right] \\
& \square \left[ g_i \rightarrow (g_i) \mathcal{U} (g_i \wedge \neg r_i) \right] \\
& \square \left[ (\neg r_i) \rightarrow (\neg r_i) \mathcal{U} (\neg r_i \wedge \neg g_i) \right].
\end{aligned}$$

where  $\mathcal{U}$  is the *unless* operator (also called *weak until*), whose relation to the *until* operator  $\mathcal{U}$  is given by:

$$\alpha \mathcal{U} \beta = (\square \alpha \vee \alpha \mathcal{U} \beta).$$

We consider this to be one of the examples where specification by automata appears to be more lucid and concise than the equivalent temporal specification.  $\blacksquare$

## 6. VERIFICATION

Let  $P$  be a program and  $A$  a complete V-automaton defining a temporal property. We would like to verify that  $A$  is valid over  $P$ , i.e., all computations of  $P$  are accepted by  $A$  and hence satisfy the temporal property. This requires showing, for each run  $r$  over every computation  $\sigma$  of  $P$ , that

$$Inf(r) \cap R \neq \emptyset \quad \text{or} \quad Inf(r) \subseteq S.$$

We introduce a single proof rule by which the validity of complete automata over programs can be established.

In the proof rule we use the notion of well-founded relations. A binary relation  $(W, \prec)$  is called well-founded if there does not exist an infinite descending sequence of elements  $w_i$  of  $W$ , that is,

$$w_0 \succ w_1 \succ w_2 \succ \dots$$

For a transition  $t \in T$ , associated with the enabling condition  $p_t$  and the transformation  $\bar{u}_t := \bar{e}_t$ , and assertions  $\varphi, \psi$ , we write

$$\{\varphi\} t \{\psi\}$$

to denote the verification condition

$$(\varphi \wedge p_t) \rightarrow \psi[\bar{e}_t/\bar{u}_t].$$

The formula  $\psi[\bar{e}_t/\bar{u}_t]$  is obtained from  $\psi$  by substituting the expressions  $\bar{e}_t$  for all the (free) occurrences of the variables  $\bar{u}_t$ . It obviously holds over a state  $s$  iff  $\psi$  holds over the state  $s'$  obtained by applying the transformation  $\bar{u}_t := \bar{e}_t$  to  $s$ . The validity of this verification condition implies that every t-successor of a v-state satisfies  $\psi$ . We write

$$\{\varphi\} \mathcal{P} \{\psi\}$$

to denote that  $\{\varphi\} t \{\psi\}$  holds for all transitions  $t \in T$  in  $\mathcal{P}$ .



V-rule (Validity of A over P)

**To** show that, a complete automaton  $\mathcal{A}$  accepts all computations of a program  $P$  that has an empty set of fairness requirements:

(I) Associate with each automaton state  $q \in Q$  an assertion  $\alpha_q$ , called the *invariant* at  $q$ , such that the following requirements are satisfied:

(11) *Initiality*

$$[\Theta \ A \ e(q)] \rightarrow \alpha_q \text{ for each } q \in Q.$$

(12) *Consecution*

$$\{\alpha_q\} P \{c(q, q') \rightarrow \alpha_{q'}\} \text{ for each } q, q' \in Q.$$

(13) *Termination*

$$\alpha_q \rightarrow \left( \bigvee_{t \in T} p_t \right) \text{ for each } q \in B.$$

(R) Find a well-founded relation  $(TV, \prec)$ . Associate with each automaton state  $q \in Q$ , a (partial) *ranking function*  $\rho_q: \Sigma \rightarrow W$ , mapping program states into elements of  $W$ , such that the following requirements are satisfied:

(R1) *Definedness*

$$\alpha_q \rightarrow (\rho_q \in TV) \text{ for each } q \in Q.$$

• (R2) *Non-increase*

$$\{\alpha_q \wedge (\rho_q = w)\} P \{c(q, q') \rightarrow (\rho_{q'} \preceq w)\} \text{ for each } q \in Q, q' \in S.$$

(R3) *Decrease*

$$\{\alpha_q \wedge (Pq = w)\} P \{c(q, q') \rightarrow (\rho_{q'} \prec w)\} \text{ for each } q \in Q, q' \in B.$$

The intended meaning of the invariants  $\alpha_q$  is that in any run  $r$  over a computation  $\sigma$ , whenever  $r$  visits the automaton state  $q$  in response to reading the program state  $s$ , then  $s \models \alpha_q$ .

The intended meaning of the ranking functions  $\rho_q$  is that they measure the “distance” either to the next R-state or to the stability point of a run  $r$  over a computation  $\sigma$ . The stability point of an accepting run  $r$ , if it is defined, is the point beyond which  $r$  visits only S-states.

Premise (11) ensures that if, in response to seeing the initial program state  $s_0$ , the automaton chooses to start a run at the automaton state  $q \in Q$ , then  $\alpha_q$  holds at  $s_0$ .

• Premise (12) ensures that if the run  $r$  has already progressed up to the automaton state  $q$ , and, seeing the next program state  $s'$ , the automaton has chosen to proceed to the automaton state  $q'$  (which is possible only if  $s'$  satisfies  $c(q, q')$ ), then  $\alpha_{q'}$  holds at  $s'$ .

Together, (11) and (12) guarantee that any run  $r$  over a computation  $\sigma$  that enters  $q$  on seeing  $s$ , is such that  $s \models \alpha_q$ .

Premise (I3) ensures that no finite run over a computation  $\sigma$  can terminate in a non accepting state. This is done by requiring that if the automaton enters  $q \in B$  on reading  $s$ , and hence  $s \models \alpha_q$ , then  $s$  cannot be the last state in  $\sigma$ .

Premise (R1) requires that the invariant  $\alpha_q$ , associated with the automaton state  $q$ , implies that the ranking function  $\rho_q$  is defined.

Premise (R3) requires that if the automaton can move from the state  $q$  to the stable state  $q' \in S$  in response to the progress of the computation from  $s$  (at  $q$ ) to  $s'$  (at  $q'$ ), and  $s$  satisfies  $\alpha_q$ , then  $\rho_{q'}(s') \preceq \rho_q(s)$ . This shows that in any stable automaton-transition the rank does not increase.

Premise (R3) is similar to (R2). However, it considers a *bad* automaton transition, i.e., from the state  $q$  into a bad state  $q' \in B$ . It requires that such a transition causes a strict decrease in the rank, i.e.,  $\rho_{q'}(s') \prec \rho_q(s)$ .

Note that on performing a *recurrent* automaton transition, i.e.,  $q \rightarrow q'$  where  $q' \in R$ , the rank is allowed to change *arbitrarily*.

### Soundness

It is easy to argue that if we succeed in finding invariants  $\alpha_q$ , a well-founded relation  $(W, \prec)$ , and ranking functions  $\rho_q$ , such that all the premises are satisfied, then this establishes the validity of A over  $P$ .

Assume that all the premises are satisfied. Consider a computation

$$\sigma : s_0, s_1, s_2, \dots$$

and a run  $r$  over it.

$$r : q_0, q_1, q_2, \dots$$

We show that  $r$  is accepting for A.

If  $r$  is finite then, by (I3), its last automaton state must be in  $R \cup S$  and hence  $r$  is accepting.

If  $r$  is infinite, consider the sequence of ranks generated by applying  $\rho_{q_i}$  to  $s_i$ ,  $i = 0, 1, \dots$ ,

$$k : \rho_{q_0}(s_0), \rho_{q_1}(s_1), \rho_{q_2}(s_2), \dots$$

The premises (11) and (12) ensure that  $s_i \models \alpha_{q_i}$  for each  $i = 0, 1, \dots$ . The premise (R1) then implies that all the expressions in  $k$  are defined and yield values taken from  $W$ .

We consider now two cases. If  $r$  contains infinitely many occurrences of R-states, it is obviously accepting. Otherwise, there is a position  $j$  such that for all  $m \geq j$ ,  $q_m \in B \cup S$ .

Combining (R2) and (R3) together, we obtain that the sequence  $k$ , from position  $j$  on, forms a non-increasing sequence:

$$\rho_{q_j}(s_j) \succeq \rho_{q_{j+1}}(s_{j+1}) \succeq \rho_{q_{j+2}}(s_{j+2}) \succeq \dots$$

In addition, (R3) ensures that for each  $q_{m+1} \in B$  there is a strict decrease in rank, i.e.,  $\rho_{q_m}(s_m) \succ \rho_{q_{m+1}}(s_{m+1})$ . It follows that  $r$  can contain only finitely many occurrences of B-states, because otherwise

it would have contained an infinitely decreasing subsequence of elements of  $W$ , which is impossible due to the well-foundedness of  $(W, <)$ . Hence all states that appear infinitely many times in  $r$  are from  $S$ , and  $r$  is accepting also in this case.

We conclude that the V-rule is sound.  $\square$

For simplicity, and with no loss of generality, we presented the rule for the restricted case of complete automata. When we consider arbitrary V-automata, the following can be observed:

In any case, the premises of the V-rule ensure that all *complete* runs over computations of  $P$  are accepting. Thus, for an arbitrary automaton, the V-rule establishes *weak* validity.

To establish validity of an arbitrary V-automaton over the program  $P$  we add another premise:

(14) *P-Completeness*

$$\{\alpha_q\} \mathcal{P} \left\{ \bigvee_{q' \in Q} c(q, q') \right\} \quad \text{for each } q \in Q.$$

Thus, while the automaton  $A$  may be incomplete, premise (14) ensures that it is complete over all program states that can be generated by the program  $P$ , and hence guarantees that  $A$  has no incomplete runs over computations of  $P$ .

## 7. EXAMPLE

As an example of the use of the V-rule, consider the following program  $\mathcal{P}_1$ :

```

initially  $x = 0, y = 1$ 
loop forever do
  when  $x = 0$  do  $y := y + 1$ 
  or
  when  $x = 0$  do  $x := 1$ 
  or
  when  $x = 1$  do  $y := y - 1$ 

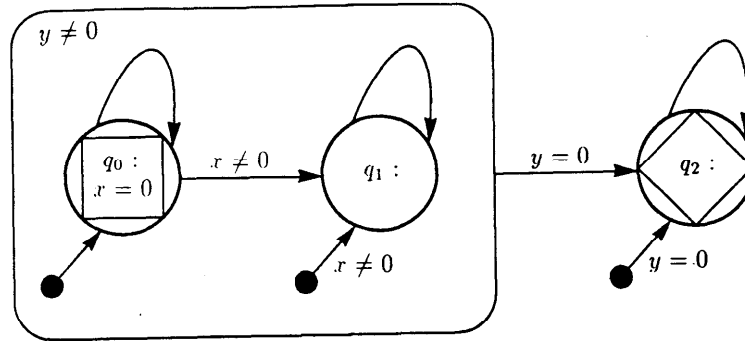
```

The property of this program that we wish to verify can be expressed by the formula

$$\square(x = 0) \vee \diamond(y = 0).$$

It states that in any computation, either continuously  $x = 0$  or eventually  $y = 0$ .

A (complete) automaton specifying this property is given by  $\mathcal{A}_1$ :



This automaton switches to  $q_2$  and accepts (since  $q_2$  is a trap R-state) as soon as it detects a program state in which  $y = 0$ . This covers the disjunct  $\Diamond(y = 0)$ . If however, no such occurrence is detected, the automaton stays forever in  $q_0$  or  $q_1$ . In this case, the automaton moves to  $q_1$  and rejects, as soon as it detects an  $x \neq 0$ . Otherwise, i.e., if continuously  $y \neq 0$  but also  $x = 0$ , the automaton stays at  $q_0$  and accepts, due to stability. This covers the disjunct  $\Box(x = 0)$  of the temporal formula.

To apply the V-rule we choose as follows:

- Invariants. We associate the assertions  $\alpha_0, \alpha_1, \alpha_2$  with the automaton states  $q_0, q_1, q_2$ , respectively. They are given by:

$$\begin{aligned}\alpha_0 &: (x = 0) \wedge (y > 0), \\ \alpha_1 &: (x = 1) \wedge (y > 0), \\ \alpha_2 &: \text{T}.\end{aligned}$$

- Well-founded relation. We use the set of ordinals  $\omega + 1$ , that is, all the natural numbers (including 0) plus the ordinal  $\omega$  (the first infinite ordinal). An isomorphic domain can be represented by the set of pairs  $\{(1, 0)\} \cup \{(0, m) \mid 0 \leq m\}$  ordered lexicographically, i.e..

$$(0, m) \prec (1, 0) \quad \text{for any } m \geq 0.$$

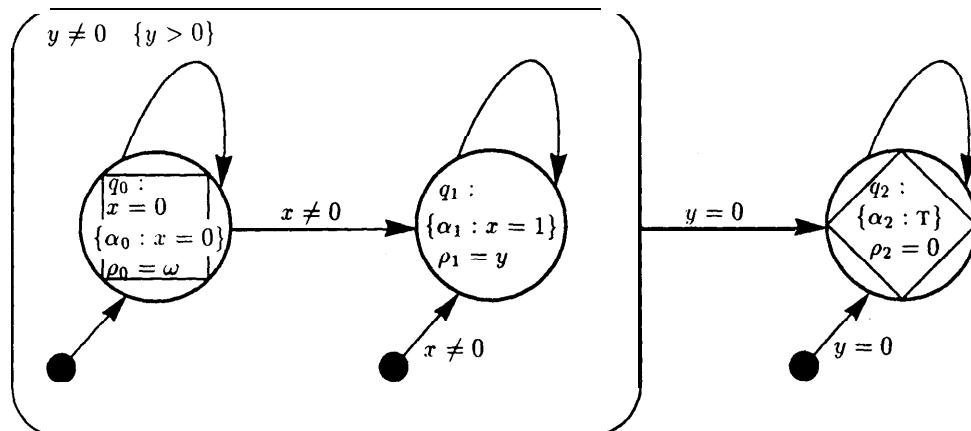
$$(0, 772) \prec (0, 772') \quad \text{iff } 777 < 777'.$$

- Ranking functions. We associate the ranking functions  $\rho_0, \rho_1, \rho_2$  with the automaton states  $q_0, q_1, q_2$ , respectively. They are given by:

$$\rho_0 = \omega, \quad \rho_1 = y, \quad \rho_2 = 0.$$

To provide a graphical representation of the selected elements, we show an annotated version of the automaton. In this version, the invariant  $\alpha_q$  (enclosed within braces) and the ranking function (appearing in the form  $\rho_q = e$  for some expression  $e$ ) annotate the node  $q$ . Consistently with our previous conventions, any assertion annotating a super-state is interpreted as an additional conjunct in

the assertions annotating the contained states.



Let us consider the different premises of the v-rule under this choice.

(11) Here we have to verify

$$[\Theta \wedge c(q)] \rightarrow \alpha_q \quad \text{for each } q \in Q$$

In the present case  $\Theta : (x = 0) \wedge (y = 1)$ . Therefore we have to verify the following:

$$(0) \quad [[(x = 0) \wedge (y = 1)] \wedge [(x = 0) \wedge (y \neq 0)]] \rightarrow [(x = 0) \wedge (y > 0)],$$

$$(1) \quad [[(x = 0) \wedge (y = 1)] \wedge [(x \neq 0) \wedge (y \neq 0)]] \rightarrow [(x = 1) \wedge (y > 0)],$$

$$(2) \quad [[(x = 0) \wedge (y = 1)] \wedge (y = 0)] \rightarrow \text{T}.$$

All these formulas are obviously valid.

(12) Here we have to verify, for each  $q, q' \in Q$ , the validity of the requirement

$$\{\alpha_q\} \mathcal{P}_1 \{c(q, q') \rightarrow \alpha_{q'}\}.$$

For each program transition  $t$ , we can identify its enabling condition  $p_t$  and the values it reassigns to the variables  $x$  and  $y$ , which we denote by  $x'$  and  $y'$ , respectively. The following table summarizes these elements for the three transitions of the program  $\mathcal{P}_1$ :

transition $t$	$p_t$	$x'$	$y'$
$t_1$	$x = 0$	$x$	$y + 1$
$t_2$	$x = 0$	$1$	$y$
$t_3$	$x = 1$	$x$	$y - 1$

We observe that for all of these transitions  $y' \geq y - 1$ .

Let us denote by  $c'(q, q')$  and  $\alpha'_{q'}$  the expressions for  $c(q, q')$  and  $\alpha_{q'}$  in which we substitute  $x'$  and  $y'$  for  $x$  and  $y$ , respectively.

Therefore we have to show that for each transition  $t \in \{t_1, t_2, t_3\}$ , the following formula is valid:

$$[\alpha_q \wedge p_t \wedge c'(q, q')] \rightarrow \alpha_{q'}.$$

We show this by considering all automaton transitions  $q \rightarrow q'$ , that is, all pairs  $q, q' \in Q$  for which  $c(q, q')$  is different from  $F$ .

$$q_0 \rightarrow q_0 : [(x = 0) \wedge (y > 0) \wedge p_t \wedge (x' = 0) \wedge (y' \neq 0)] \rightarrow [(x' = 0) \wedge (y' > 0)].$$

The first conjunct of the consequent obviously holds. In view of  $y' \geq y - 1$ ,  $y > 0$  implies  $y' \geq 0$ . Together with  $y' \neq 0$  this yields  $y' > 0$ , establishing the second conjunct.

$$q_0 \rightarrow q_1 : [(x = 0) \wedge (y > 0) \wedge p_t \wedge (x' \neq 0) \wedge (y' \neq 0)] \rightarrow [(x' = 1) \wedge (y' > 0)].$$

The only program transition  $t$  allowing  $x = 0$  and  $x' \neq 0$  is  $t_2$  which leads to  $x' = 1$ . The second conjunct  $y' > 0$  is established as before.

$$q_1 \rightarrow q_1 : [(x = 1) \wedge (y > 0) \wedge p_t \wedge (y' \neq 0)] \rightarrow [(x' = 1) \wedge (y' > 0)].$$

The possible values of  $x'$  are  $x' = x = 1$  or  $x' = 1$ . Consequently, in both cases  $x' = 1$ . The conjunct  $y' > 0$  is established as before.

$$q \rightarrow q_2 \text{ for } q \in \{q_0, q_1, q_2\} : [\alpha_q \wedge p_t \wedge c'(q, q_2)] \rightarrow T.$$

These formulas are trivially valid.

(13) Here we have to verify, for each  $q \in B$ , the validity of the formula

$$\alpha_q \rightarrow \left( \bigvee_{t \in T} p_t \right).$$

The only bad state in this automaton is  $q_1$ , and the disjunction of the enabling conditions for all three program transitions is  $(x = 0) \vee (x = 1)$ . The resulting formula is

$$[(x = 1) \wedge (y > 0)] \rightarrow [(x = 0) \vee (x = 1)],$$

which is trivially valid.

(R1) Here we have to verify, for each  $q \in Q$ , the validity of

$$\alpha_q \rightarrow (\rho_q \in W).$$

The only  $q \in Q$  for which  $\rho_q$  is not a constant from  $W$  is  $q_1$ . For  $q_1$ , we show that

$$[(x = 1) \wedge (y > 0)] \rightarrow (u > 0).$$

which obviously holds.

(R2) Here we have to verify, for each  $q \in Q$  and  $q' \in S$ , the validity of the formula

$$\{\alpha_q \wedge (\rho_q = w)\} \mathcal{P}_1 \{c(q, q') \rightarrow (\rho_{q'} \leq w)\}.$$

This amounts to showing, for each program-transition  $t$ ,

$$[\alpha_q \wedge p_t \wedge c'(q, q')] \rightarrow (\rho'_{q'} \leq \rho_q).$$

As before, we denote by  $c'(q, q')$  and  $\rho'_{q'}$  the expressions for  $c(q, q')$  and  $\rho_{q'}$  in which we have substituted  $x'$  and  $y'$  for  $x$  and  $y$ , respectively.

Since  $S = \{q_0\}$ , there is only one relevant automaton transition to be considered.

$$q_0 \rightarrow q_0 : [(x = 0) \wedge (y > 0) \wedge p_t \wedge (x' = 0) \wedge (y' \neq 0)] \rightarrow (\omega \leq \omega),$$

which is obviously valid.

(R3) Here we have to verify, for each  $q \in Q$  and  $q' \in B$ , the validity of the requirement:

$$\{\alpha_q \wedge (\rho_q = w)\} \mathcal{P}_1 \{c(q, q') \rightarrow (\rho_{q'} < w)\},$$

which is equivalent to showing the validity of:

$$[\alpha_q \wedge p_t \wedge c'(q, q')] \rightarrow (\rho'_{q'} < \rho_q) \text{ for every } t \in T.$$

The only state  $q' \in B$  is  $q_1$ . We therefore consider:

$$q_0 \rightarrow q_1 : [(x = 0) \wedge (y > 0) \wedge p_t \wedge (x' \neq 0) \wedge (y' \neq 0)] \rightarrow (y' < w).$$

This is valid since  $y'$  is a finite integer, which is always smaller than  $w$ .

$$q_1 \rightarrow q_1 : [(x = 1) \wedge (y > 0) \wedge p_t \wedge (y' \neq 0)] \rightarrow (y' < y).$$

The only program transition enabled under  $x = 1$  is  $t_3$ , for which  $y' = y - 1 < y$ .

### Temporal Proof

For comparison, let us consider the proof of the same property using a temporal proof system similar to the one presented in [MP2]. Such proof systems are usually based on three basic rules.

The first rule states that any temporal formula  $\varphi$  implied by the initial condition  $\Theta$  is valid over all computations of  $P$ . This rule can be stated as

<div style="display: flex; justify-content: space-between;"> <div style="width: 20%;">INIT rule</div> <div style="width: 60%; text-align: center;"> <math display="block">\frac{\Theta \rightarrow \varphi}{\varphi}</math> </div> </div>
---

The second rule establishes *safety* properties expressible by a formula of the form  $\Box(\alpha \rightarrow \alpha \mathcal{U} \beta)$  for assertions  $\alpha, \beta$ .

<div style="display: flex; justify-content: space-between;"> <div style="width: 20%;">INV rule</div> <div style="width: 60%; text-align: center;"> <math display="block">\frac{\{\alpha\} P \{\alpha \vee \beta\}}{\Box(\alpha \rightarrow \alpha \mathcal{U} \beta)}</math> </div> </div>
--

The third rule is somewhat more involved, and uses well-founded induction to establish conditional liveness properties expressible by a formula of the form  $\Box (CI \rightarrow \Diamond \beta)$  for assertions  $\alpha, \beta$ . We will refer to this rule as the PROGRESS rule.

The property we wish to verify for the program  $\mathcal{P}_1$  is given by the formula

$$\Box(x = 0) \vee \Diamond(y = 0).$$

This formula is neither a safety formula nor a liveness formula. Hence no single rule can be used to establish it, and some combination of the rules is called for.

Indeed, the major steps in the verification of this formula are the following:

*Step 1:* Verify

$$\Box \left[ [(x = 0) \wedge (y > 0)] \rightarrow [(x = 0) \wedge (y > 0)] \wedge \Box [(x = 1) \wedge (y > 0)] \right].$$

This can be established by the INV rule, checking that the premise

$$\{(x = 0) \wedge (y > 0)\} \mathcal{P}_1 \{((x = 0) \vee (x = 1)) \wedge (y > 0)\}$$

holds. This is equivalent to showing, for all transitions  $t \in \{t_1, t_2, t_3\}$ , the validity of the formula

$$[(x = 0) \wedge (y > 0) \wedge p_t] \rightarrow [((x' = 0) \vee (x' = 1)) \wedge (y' > 0)].$$

By the precondition  $x = 0$  it is clear that only  $t_1$  and  $t_2$  are enabled:

- For  $t_1$ , we have  $x' = x = 0$ ,  $y' = y + 1 > 0$ .
- For  $t_2$ , we have  $x' = 1$ ,  $y' = y > 0$ .

*Step 2:* Verify

$$\Box \left[ [(x = 1) \wedge (y > 0)] \rightarrow \Diamond (y > 0) \right].$$

This is established by the PROGRESS rule using the ordering over the natural numbers as the well-founded relation. The ranking function used is  $\rho : y$ . Since, under  $x = 1$ , the only enabled transition is  $t_3$ , it is clear that each step of the program strictly decreases  $\rho$ .

*step 3:* Use temporal reasoning on the results of the two preceding steps to establish that

$$[(x = 0) \wedge (y = 1)] \rightarrow [\Box (x = 0) \vee \Diamond (y = 0)].$$

Now apply the INIT rule, using the fact that  $\Theta$  is  $(x = 0) \wedge (y = 1)$  for the program  $\mathcal{P}_1$ , and obtain

$$\Box \left[ \Box \left[ [(x = 0) \wedge (y > 0)] \rightarrow \Box \left[ \Box (x = 0) \vee \Diamond (y = 0) \right] \right] \right].$$

Even though the temporal-logic proof presented above is obviously more concise than the automaton proof, it calls for more creativity and heuristic planning than the automaton proof. This planning involves the decomposition of the proof into three steps and deciding which rules (including temporal reasoning) to use in each of the steps. After making these decisions, the subsequent efforts are devoted to the construction of appropriate invariants and ranking functions. Obviously, the invariants and the ranking functions used in the two proof methods are closely related.

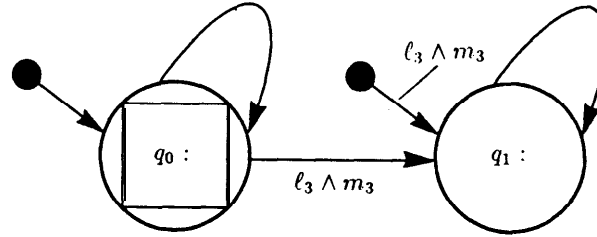
In contrast, the automaton proof is almost mechanical, in the sense that one did not have to decompose the proof into steps, and could proceed directly to the construction of invariants and ranking functions.

The price we paid for this uniformity is that more verification conditions had to be checked than in the case of the temporal logic proof. This can be explained by the fact that the decomposition of the proof into separate steps is equivalent to a decomposition of the automaton into several sub-automata.





is at location  $\ell_3$ . This safety property is specified by the automaton  $\mathcal{A}_2$  below.



This automaton has two states, the good state  $q_0$  associated with the condition  $\neg(\ell_3 \wedge m_3)$ , and the bad state  $q_1$ , to which we switch on detection of a program state satisfying  $\ell_3 \wedge m_3$ . Once we are in  $q_1$  we can never get out.  $\blacksquare$

It is easier to verify the validity of a safety automaton over a program than that of a general automaton.

S-rule (Validity of a complete safety automaton)

Find for each state  $q \in Q$  an assertion  $\alpha_q$ , such that:

- (11) *Initiality* — as before
- (12) *Consecution* — as before
- (13') *Impossibility*

$$\alpha_q = \text{F for each } q \in B.$$

Note that (13') implies (I3), but also ensures that no run over a computation of the program can visit a bad state.

Returning to the example of the mutual exclusion algorithm, we can pick

$$\begin{aligned} \alpha_0 &: (y_1 \equiv \ell_{1..3}) \wedge (y_2 \equiv m_{1..3}) \wedge \\ &\quad [\ell_{0,1} \vee m_{0,1} \vee (\ell_{2,3} \wedge m_2 \wedge \neg t) \vee (\ell_2 \wedge m_{2,3} \wedge t)], \\ \alpha_1 &: \text{F.} \end{aligned}$$

In the assertion  $\alpha_0$  we used, among others, the abbreviations

$$\ell_{0,1} = \ell_0 \vee \ell_1 \quad \text{and} \quad \ell_{1..3} = \ell_1 \vee \ell_2 \vee \ell_3.$$

To check that this choice, which obviously satisfies (I3'), also satisfies (11) and (I2), we consider each premise in turn. Both of them can be simplified, due to the fact that  $\alpha_1 = \text{F}$ . Thus, it only remains to check:

$$(II) \quad \Theta \rightarrow \alpha_0, \quad \Theta \rightarrow \neg(\ell_3 \wedge m_3).$$

This is obvious due to the fact that

$$\Theta : \ell_0 \wedge m_0 \wedge (\neg y_1) \wedge (\neg y_2) \wedge t,$$

and to the implications  $\ell_0 \rightarrow \neg \ell_3$ , etc.

(I2) This premise requires to verify the following three clauses:

- (a)  $\{\alpha_0\} \mathcal{P}_2 \{\top \rightarrow \alpha_0\}$
- (b)  $\{\alpha_0\} \mathcal{P}_2 \{(\ell_3 \wedge m_3) \rightarrow \mathbf{F}\}$
- (c)  $\{\mathbf{F}\} \mathcal{P}_2 \{\mathbf{T} \rightarrow \mathbf{F}\}$ .

Clause (a) can be simplified to

$$\{\alpha_0\} \mathcal{P}_2 \{\alpha_0\},$$

which claims that  $\alpha_0$  is preserved over all transitions. This has to be checked for each transition separately, and is the major part of the verification effort.

Clause (b) simplifies to

$$\{\alpha_0\} \mathcal{P}_2 \{\neg(\ell_3 \wedge m_3)\},$$

which, due to the fact that  $\alpha_0$  implies  $\neg(\ell_3 \wedge m_3)$ , follows from clause (a).

Clause (c) is trivially valid since it has the precondition  $\mathbf{F}$ .

Similarly to the general case, to extend the S-rule to possibly incomplete automata, we require the additional premise (14).

## SIMULATION BETWEEN AUTOMATA

We observe that for any automaton  $A$ , there is a set of invariants to which we refer as the *native* invariants, and which automatically satisfy the premises (11) and (12). These are the invariants defined by

$$N_q = e(q) \vee \left( \bigvee_{q' \in Q} c(q', q) \right).$$

It is trivial to check that the native invariants  $N_q$  satisfy (11) and (12). In some cases they also satisfy (13) and it is possible to choose ranking functions  $\rho_q$  that together with the  $N_q$ 's, also satisfy (R1)-(R3).

This, unfortunately, has not been the case with the automaton  $\mathcal{A}_1$  considered for the program  $\mathcal{P}_1$  in the example of Section 7. The native invariants for this automaton are given by

$$N_0 : (x = 0) \wedge (y \neq 0)$$

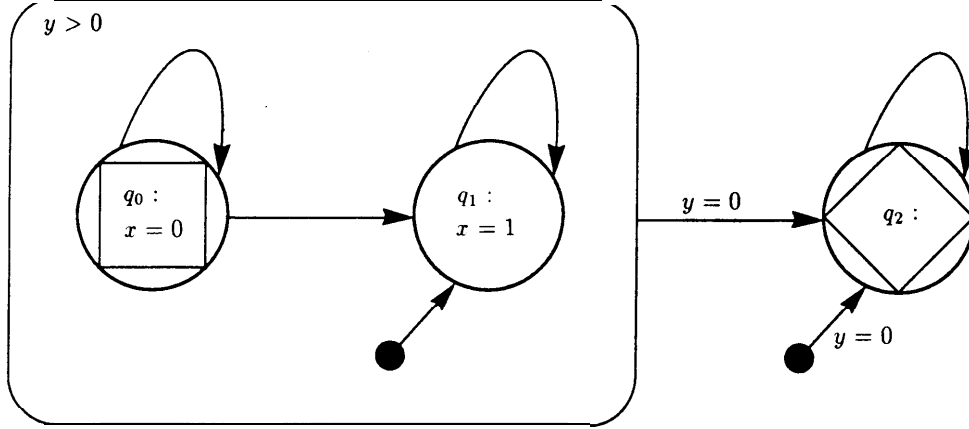
$$N_1 : y \neq 0$$

$$N_2 : \top.$$

Note, for example, that even though two of the edges entering  $q_2$  are labeled by  $y = 0$ , the third edge, corresponding to  $c(q_2, q_2)$ , is labeled by  $\mathbf{T}$ , which makes  $N_2 = \mathbf{T}$ .

It is easy to see that the invariants  $N_0 - N_2$  do not satisfy (13), nor do they support the premises (R1)-(R3) with the ranking functions we wish to use. Consequently, we could not use the native invariants for proving the validity of  $A_1$  over the program  $\mathcal{P}_1$ , and had to come up with stronger assertions  $\alpha_0 - \alpha_2$ .

On the other hand, let us consider the automaton  $A'$ , presented below.



This automaton is obtained from  $A_1$  by deleting the original labels from the edges and the nodes and labeling each node  $q'_i$  by the assertion  $\alpha_i$ . By our encapsulation conventions, we factored out the conjunct,  $y > 0$  from  $\alpha_0$  and  $\alpha_1$  and used it to label the super-state containing both  $q_0$  and  $q_1$ . What is the relation between the automata  $\mathcal{A}_1$  and  $\mathcal{A}'_1$ ? First, we observe that the two automata are structurally similar. This means that there is a one-to-one correspondence between the states of  $\mathcal{A}_1$  and the states of  $A'$ , which respects membership in the acceptance sets  $R$  and  $S$ . Thus, for  $i = 0, 1, 2$ ,  $q_i \in R$  iff  $q'_i \in R'$ , and  $q_i \in S$  iff  $q'_i \in S'$ .

Secondly, we can show that any run  $r$  of  $A_1$  over a PI-computation  $\sigma$  corresponds to a run  $r'$  of  $A'_1$  over  $\sigma$ , such that  $r$  visits  $q_j$  at step  $i$  iff  $r'$  visits  $q'_j$  at that step. This is an immediate consequence of the fact that  $\alpha_0 - \alpha_2$  satisfy the premises (I1), (I2) and hence, whenever we visit  $q_j$ ,  $\alpha_j$  is known to hold.

We describe the relation, holding between  $A_1$  and  $A'$ , by saying that  $A'$ , *simulates*  $\mathcal{A}_1$ , relative to  $\mathcal{P}_1$ . A precise and more general definition of this relation will be presented later.

Obviously if  $A'_1$  simulates  $\mathcal{A}_1$ , and  $A'$ , is valid over the program  $\mathcal{P}_1$ , then so is  $\mathcal{A}_1$ . This is because any run  $r$  of  $A_1$  induces a run  $r'$  of  $\mathcal{A}'_1$ , which passes through R-states or S-states exactly at the same steps that  $r$  does. Since  $\mathcal{A}'_1$  is valid,  $r'$  is accepted, and therefore, so is  $r$ .

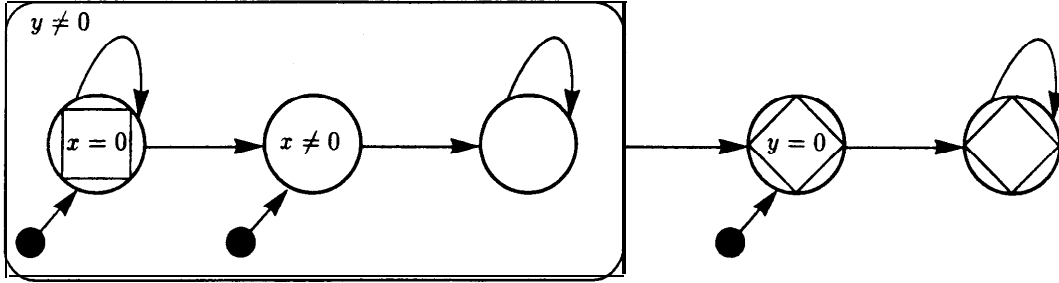
A similar conclusion holds for weak validity. Thus, if  $A_1$  and  $A'$ , are not necessarily complete,  $\mathcal{A}'_1$  simulates  $A_1$ , and  $A'$ , is weakly valid over  $\mathcal{P}$ , then also  $\mathcal{A}_1$  is weakly valid over  $\mathcal{P}$ . This shows that if  $\mathcal{A}_1$  (but not necessarily  $A'_1$ ) is complete, and  $A'_1$ , simulating  $A_1$ , is weakly valid over  $\mathcal{P}$ , then  $\mathcal{A}_1$  is valid over  $\mathcal{P}$ . This is because for a complete automaton validity and weak validity coincide.

The graphical representation of the automaton  $\mathcal{A}'_1$  has a special form, which we call a *node-labeled* automaton. This means that only nodes are labeled by assertions, but never any edges. In terms of

our original (non-graphical) definition of an automaton, a node-labeled automaton has an assertion  $\varphi_q$  associated with each state  $q$ , such that

- For every  $q \in Q$ ,  $e(q)$  is either **F** or  $\varphi_q$ .
- For every  $q, q' \in Q$ ,  $c(q, q')$  is either **F** or  $\varphi_{q'}$ .

It can be shown that every automaton is equivalent to a node-labeled automaton. For example, the following automaton is the node-labeled equivalent of  $A_1$ :



For node-labeled automata, it is very easy to compute the native invariants. They are simply given by  $N_q = \varphi_q$ . Thus, for the automaton  $A'_1$ , the native invariants are given by

$$N'_0 = \alpha_0 = (x = 0) \wedge (y > 0),$$

$$N'_1 = \alpha_1 = (x = 1) \wedge (y > 0),$$

$$N'_2 = \alpha_2 = \tau.$$

It is now easy to show that the native invariants of  $A'$ , satisfy (I3), and together with the ranking functions  $\rho_i$ , as chosen above, satisfy (R1)–(R3).

We may therefore describe the verification process of  $A$  over  $P$  as being a two-stage process. The first stage consists of finding invariants  $\alpha_q$  that satisfy the premises (11) and (12). This stage can be described as finding an automaton  $A'$  which simulates the original automaton  $A$ , relative to  $P$ , such that the native invariants of  $A'$  are the  $\alpha_q$ 's.

The second stage consists of finding ranking functions  $\rho_q$  that, together with the invariants  $\alpha_q$ , satisfy (13) and (R1)–(R3). This stage can be described as showing that  $\rho_q$ , together with the native invariants of  $A'$ ,  $N'_q$ , satisfy (13) and (R1)–(R3).

The extension from  $A$  to  $A'$ , considered above, only involved strengthening the entry and transition conditions. It is useful to consider more general extensions, in which single states of  $A$  are refined into sets of states in  $A'$ . The general definition is given by:

Let  $A' = (Q', R', S', E', C')$  and  $A = (Q, R, S, E, C)$  be two V-automata and  $P$  a program. We say that  $A'$  *simulates*  $A$  relative to  $P$ , if there exists a function  $f : Q' \rightarrow Q$  such that:

- $q' \in R', S' \Leftrightarrow f(q') \in R, S$ , respectively.

- For any run  $r$  of  $A$  over a computation of  $\mathcal{P}$ , there exists a run  $r'$  of  $A'$  over the same computation such that  $r = f(r')$ .

In this definition, if  $r' = q'_0, q'_1, \dots$  then  $f(r) = f(q'_0), f(q'_1), \dots$ . For a mapping  $f : Q' \rightarrow Q$  as above, we denote:

$$f^{-1}(q) = \{q' \in Q' \mid f(q') = q\}.$$

Our previous observation also extends to this general definition: Obviously, if  $A'$  simulates  $A$  and  $A'$  is valid over  $P$ , then so is  $A$ .

The following proof rule provides a method for proving that  $A'$  simulates  $A$ , relative to  $P$ .

E-rule ( $A'$  simulates  $A$ , relative to  $P$ )

To show that  $A'$  simulates  $A$ , relative to  $P$ :

Find a mapping  $f : Q' \rightarrow Q$ , such that

$$q' \in R', S' \Leftrightarrow f(q') \in R, S, \text{ respectively.}$$

Then, verify the following requirements:

(E1) For each  $q \in Q$ ,

$$[\Theta \wedge e(q)] \rightarrow \left( \bigvee_{q' \in f^{-1}(q)} e'(q') \right).$$

(E2) For each  $q_1, q_2 \in Q$  and  $q'_1 \in f^{-1}(q_1)$ ,

$$\left\{ N'_{q'_1} \right\} \mathcal{P} \left\{ c(q_1, q_2) \rightarrow \left( \bigvee_{q'_2 \in f^{-1}(q_2)} c'(q'_1, q'_2) \right) \right\},$$

where  $N'_{q'_1}$  is the native invariant of  $A'$  at  $q'_1$ .

The purpose of the two premises is to ensure that each step in a run  $r$  of  $A$  over  $\sigma$ , a computation of  $\mathcal{P}$ , can be emulated by  $A'$ .

Premise (E1) ensures that if  $q \in Q$  is the first automaton state in  $r$ , corresponding to the first program state  $s_0$  in  $a$ , then a corresponding initial state  $q' \in f^{-1}(q)$  can be found in  $A'$ .

Premise (E2) ensures that if the run  $r$  of  $A$  has proceeded up to the automaton state  $q_1$ , while the simulating run  $r'$  of  $A'$  has proceeded up to  $q'_1 \in f^{-1}(q_1)$ , then the next step of  $A$  can also be simulated by  $A'$ . If the next step of  $A$  is from  $q_1$  to  $q_2$  in response to the program state  $s$ , which must therefore satisfy  $c(q_1, q_2)$ , then the premise guarantees some  $q'_2 \in f^{-1}(q_2)$  such that  $s \models c'(q'_1, q'_2)$ .

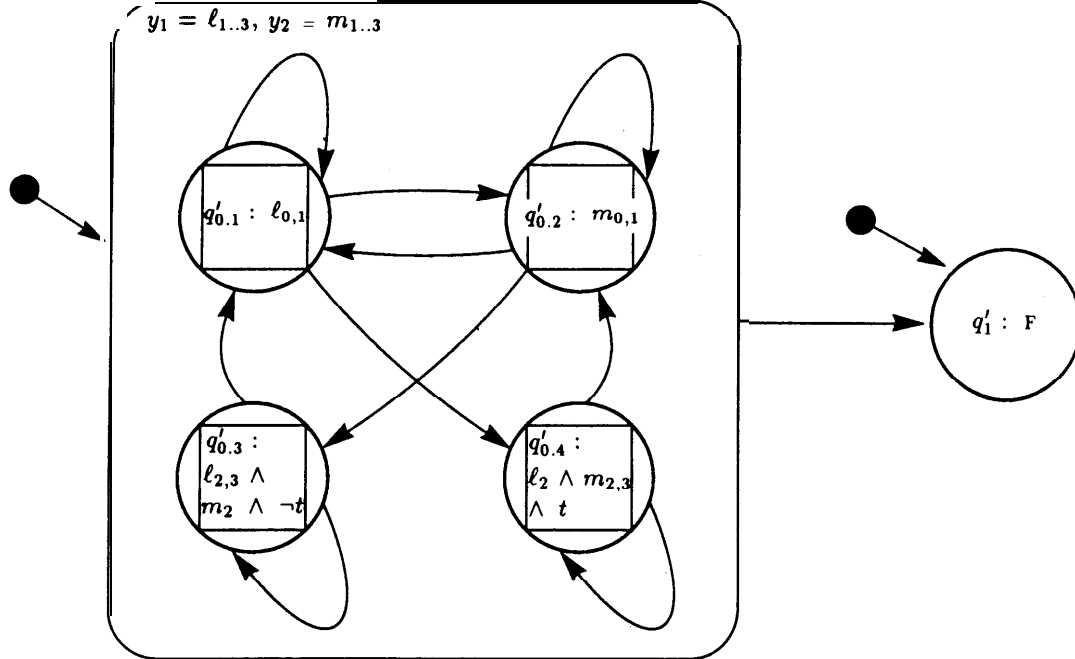
The definition of simulation is a *global* concept, requiring the existence of a simulating run  $r'$  for each original run  $r$ . The E-rule translates this into a *local* property, requiring that it will always be possible to extend a finite segment of a simulating run one step further.

### Example

Let us reconsider the safety property expressed by the automaton  $\mathcal{A}_2$ , and stating that mutual

exclusion is maintained for the program  $\mathcal{P}_2$ . Since this is a safety automaton, its validation requires finding invariants that satisfy (I1), (I2), and also (I3').

Following the discussion above, we suggest the automaton  $\mathcal{A}'_2$  presented below.



To identify the function  $f$  mapping the states of  $\mathcal{A}'_2$  onto states of  $\mathcal{A}_2$ , we use the naming convention, by which the names of states in  $\mathcal{A}'_2$  are of the form  $i,j$ , such that  $f(q'_{i,j}) = q_i$ . In the special case that there is only one state in  $\mathcal{A}'_2$  corresponding to  $q_i$  in  $\mathcal{A}_2$ , we use the simpler name  $q'_i$ .

We claim that  $\mathcal{A}'_2$  simulates  $\mathcal{A}_2$ . This case illustrates the more general situation, that the mapping between  $Q'$  and  $Q$  is not one-to-one. The native invariants of this automaton are:

$$N'_{0,1} : (y_1 \equiv l_{1..3}) \wedge (y_2 \equiv m_{1..3}) \wedge l_{0,1}$$

$$N'_{0,2} : (y_1 \equiv l_{1..3}) \wedge (y_2 \equiv m_{1..3}) \wedge m_{0,1}$$

$$N'_{0,3} : (y_1 \equiv l_{1..3}) \wedge (y_2 \equiv m_{1..3}) \wedge l_{2,3} \wedge m_2 \wedge (\neg t)$$

$$N'_{0,4} : (y_1 \equiv l_{1..3}) \wedge (y_2 \equiv m_{1..3}) \wedge l_2 \wedge m_{2,3} \wedge t.$$

It is trivial to see that (the incomplete automaton)  $\mathcal{A}'_2$  is weakly valid over  $\mathcal{P}_2$ , since no run of it can ever move to  $q'_1$ , whose entry condition is F. Thus, the main verification effort is to show that  $\mathcal{A}'_2$  indeed simulates  $\mathcal{A}_2$ . This requires showing:

$$(E1) \quad [\Theta \wedge \neg(l_3 \wedge m_3)] \rightarrow [N'_{0,1} \vee N'_{0,2} \vee N'_{0,3} \vee N'_{0,4}]$$

$$[\Theta \wedge (l_3 \wedge m_3)] \rightarrow F,$$

which are obvious, and

$$\begin{aligned}
\text{(E2)} \quad & \{N'_{0.1}\} \mathcal{P}_2 \{N'_{0.1} \vee N'_{0.2} \vee N'_{0.4}\} \\
& \{N'_{0.2}\} \mathcal{P}_2 \{N'_{0.2} \vee N'_{0.1} \vee N'_{0.3}\} \\
& \{N'_{0.3}\} \mathcal{P}_2 \{N'_{0.3} \vee N'_{0.1}\} \\
& \{N'_{0.4}\} \mathcal{P}_2 \{N'_{0.4} \vee N'_{0.2}\}.
\end{aligned}$$

In principle we also have to show that

$$\{N'_{0.i}\} \mathcal{P}_2 \{(\ell_3 \wedge m_3) \rightarrow F\}, \text{ for } i = 1, \dots, 4, \quad \text{and} \quad \{F\} \mathcal{P}_2 \{F\}$$

But this follows from the four clauses above, since each  $N'_{0,j}$  implies  $\neg(\ell_3 \wedge m_3)$ .

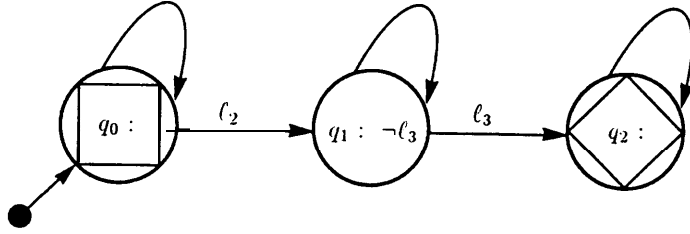
We claim that these four clauses are easier to understand than the single verification condition

$$\{\alpha_0\} \mathcal{P}_2 \{\alpha_0\},$$

where  $\alpha_0$ , is the invariant we used before to verify  $\mathcal{A}_2$ , and is equivalent, to  $N'_{0.1} \vee N'_{0.2} \vee N'_{0.3} \vee N'_{0.4}$ . The reason is that the four sub-assertions partition the state space covered by  $\alpha_0$ , in a way that makes it easier to follow each transition in the program and convince ourselves that, indeed, it can only lead from one  $N'_{0,j}$  to another, as shown in the diagram.

### A Liveness Property

The second property of this program is a liveness property, specified by  $\mathcal{A}_3$  below. This automaton states that whenever process  $\mathcal{P}_2$  is at  $\ell_2$ , it will eventually get to  $\ell_3$ , that is,  $\square (2 \rightarrow \diamond \ell_3)$ :

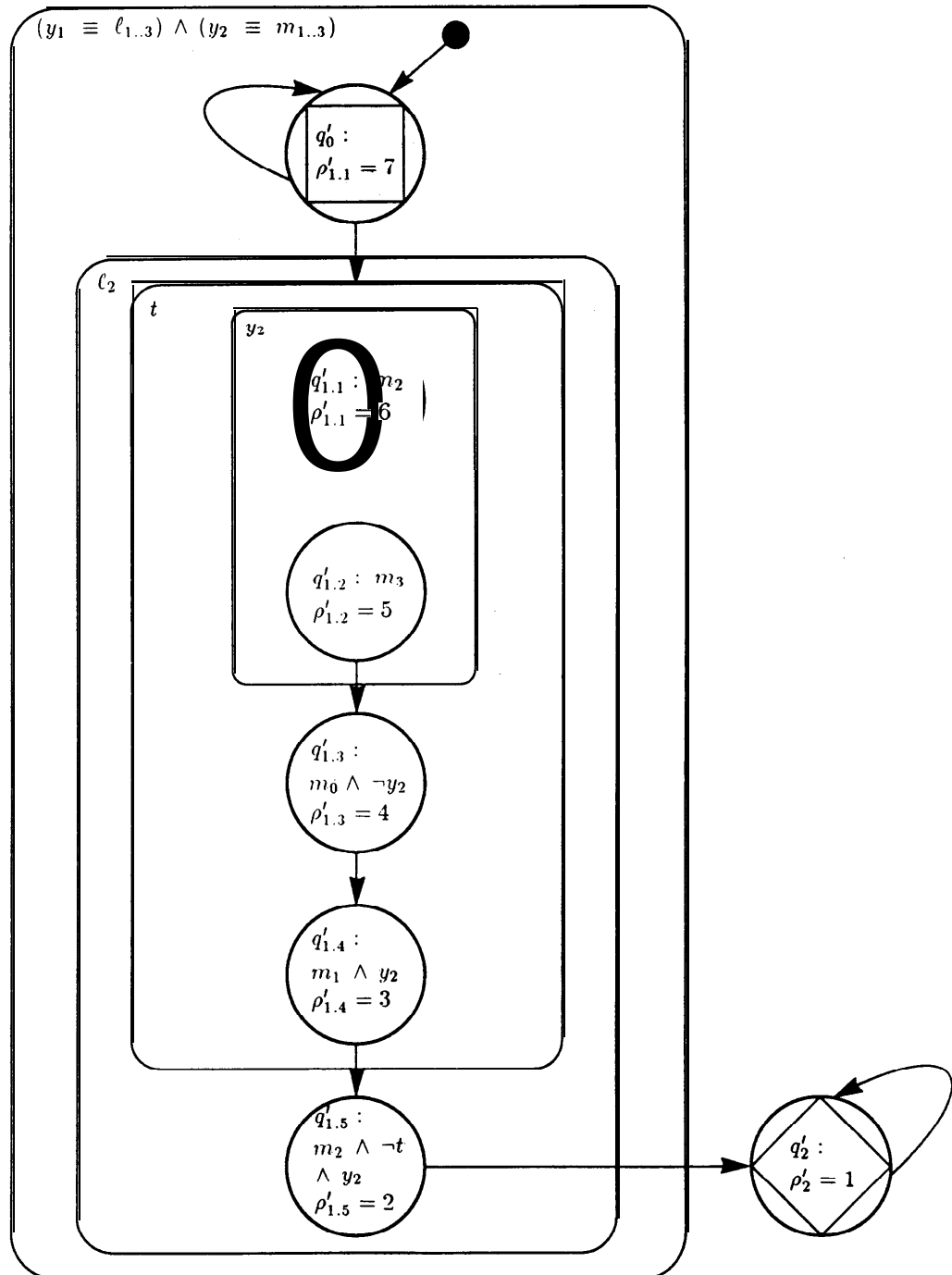


A symmetric property holds for the process  $\mathcal{P}_2$ .

We present in the next page an automaton  $\mathcal{A}'_3$  simulating the automaton  $\mathcal{A}_3$ . We annotate the states by their common entry and transition conditions which are also identical to their native invariants, and by their ranking functions.

It is not difficult to see that the simulating automata, presented by  $\mathcal{A}'_2$  and  $\mathcal{A}'_3$  above, are very similar to the proof diagrams recommended in [OL] and [MP2] for concise presentation of the main elements in the proof of a temporal property. This provides a common framework for both specification automata and proof diagrams.





## 10. EXPRESSIBILITY

In this short section we show that the specification power of  $\forall$ -automata is identical to that of the logic ETL, which is an extended version of temporal logic, studied by Wolper in [W].

For this proof we need the notion of a Büchi automaton. A *Büchi automaton*  $B$  is an automaton as defined in this paper with the restriction that  $S=\emptyset$ . A Büchi automaton  $B$  accepts a computation  $\sigma$  if there exists at least one run of  $B$  over  $\sigma$  that is accepting. Thus, while a  $\forall$ -automaton accepts a computation  $\sigma$  if *all* runs over  $\sigma$  are accepting, a Büchi automaton accepts  $\sigma$  if some run over  $\sigma$  is accepting.

$\forall$ -automata, in particular those with  $R=\emptyset$ , are dual to Büchi automata. To explain the duality relation, we call a  $\forall$ -automaton  $A=(Q, \emptyset, S, E, C)$  and a Büchi automaton  $B=(Q, R, \emptyset, E, C)$  *dual*, if  $R=Q - S$ . We denote this fact by writing  $B = A$ , or  $A = \tilde{B}$ . Dual automata are related by the following proposition.

**Proposition:**  $A$  accepts a computation  $\sigma$  iff  $B$  rejects  $\sigma$

**Proof:** To prove the proposition, we observe that for a given  $\sigma$ ,  $r$  is a run of  $A$  over  $\sigma$  iff it is also a run of  $B$  over  $\sigma$ . Let  $run(\sigma)$  be the set of all runs common to  $A$  and  $B$  over  $\sigma$ .

With the above notations, we can establish the following list of equivalences:

$$\begin{aligned}
 A \text{ accepts } \sigma &\Leftrightarrow \forall r \in run(\sigma) : Inf(r) \subseteq S \\
 &\Leftrightarrow \forall r \in run(\sigma) : Inf(r) \cap R = \emptyset \\
 &\Leftrightarrow \neg \exists r \in run(\sigma) : Inf(r) \cap R \neq \emptyset \\
 &\Leftrightarrow B \text{ rejects } \sigma. \quad \blacksquare
 \end{aligned}$$

We now use the results of [W], stating that Büchi automata have the same expressive power as ETL, to show:

**Proposition:**  $\forall$ -automata have the same expressive power as ETL.

**Proof:** Let  $\varphi$  be an ETL-formula. Clearly, also  $\neg\varphi$  is an ETL-formula, according to [W], there exists a Büchi automaton  $\tilde{B}(\neg\varphi)$  expressing the same property, i.e., accepting all computations that satisfy  $\neg\varphi$ , and rejecting all computations that satisfy  $\varphi$ . Let  $A = \tilde{\tilde{B}}(\neg\varphi)$ . Then  $A$  accepts precisely all the computations that  $\tilde{B}(\neg\varphi)$  rejects, namely, all the computations that satisfy  $\varphi$ .

To show the other direction, denote by  $\varphi(B)$  the ETL-formula (whose existence is ensured by [W]) expressing the same property as  $B$ . Then for the  $\forall$ -automaton  $A = \tilde{\tilde{B}}$ , the ETL-formula  $\neg\varphi(A)$  specifies precisely the same property as  $A$ .  $\blacksquare$

## 11. COMPLETENESS

In this section we sketch a proof of completeness of the v-rule for proving the validity of a complete automaton  $A$  over a program  $P$ . We consider only the case that  $P$  has no fairness requirements. The general case will be discussed in the next section that deals with fairness. We also assume that the considered automaton  $A$  has an empty  $R$ .

The claim of completeness of the v-rule can be expressed by the following theorem.

**Theorem:** If  $A$  is valid over  $P$ , then there exist a well-founded set  $(W, <)$ , invariants  $\alpha_q$ , and ranking functions  $\rho_q$ , expressible in the language  $L$ , that satisfy the premises of the v-rule.

We split the discussion into two parts. In the first part, called *semantic completeness*, we only show the existence of  $(W, <)$ ,  $\alpha_q$ , and  $\rho_q$  that satisfy the premises of the V-rule. In the second part, called *syntactic completeness*, we consider the question of expressing  $\alpha_q$  and  $\rho_q$  in the language  $L$ .

### (A) Semantic Completeness

Let  $A$  be a complete automaton that is valid over  $P$ . We define a *computation segment* of the program  $P$  to be a finite sequence of program states, that satisfies requirements (1) and (2) in the definition of computations, but is not necessarily maximal. A run  $r$  over a computation segment  $\sigma$  is defined exactly in the same way as a run over a computation.

#### Invariants

We give a *verbal* definition for the invariant  $\alpha_q$  for each  $q \in Q$ :

$$s \models \alpha_q \Leftrightarrow \text{There exists a computation segment, whose last program state is } s, \text{ and a corresponding run, whose last automaton state is } q.$$

Thus,  $s$  satisfies  $\alpha_q$  iff there exists a computation segment of the form  $\sigma * s$ , and a corresponding run of the form  $r * q$ , for some sequences  $\sigma$  and  $r$ .

In this definition we use  $*$  to denote concatenation of an element to a sequence. The sequences  $\sigma$  and  $r$ , can also be  $\Lambda$  (the empty sequence).

We show that these invariants satisfy the premises (I1)-(I3) of the v-rule.

- To show (I1), let  $q \in Q$  be an automaton state and  $s \in \Sigma$  be a program state such that  $s \models \Theta \wedge e(q)$ . We claim that, in this case,  $\Lambda * s$  is a computation-segment, and  $\Lambda * q$  is a run over it. Hence, by the definition of  $\alpha_q$ ,  $s \models \alpha_q$ . It follows that  $\Theta \wedge e(q)$  implies  $\alpha_q$ .
- To show (I2), let  $s \models \alpha_q$  and  $s' = t(s)$  for some transition  $t \in T$  of  $P$ . We have to show that  $s' \models (c(q, q') \rightarrow \alpha_{q'})$ . Assume that  $s' \models c(q, q')$ . By the definition of  $\alpha_q$ ,  $s \models \alpha_q$  implies the existence of a computation segment  $\sigma * s$  and a run  $r * q$  over it. Since  $s' = t(s)$ ,  $\sigma' = \sigma * s * s'$  is a computation segment of  $P$ , and since  $s' \models c(q, q')$ ,  $r * q * q'$  is a run over  $u'$ . It follows that  $s' \models \alpha_{q'}$ .
- To show (I3), let  $s \models \alpha_q$  and  $q \in B$  (i.e.,  $q \notin R \cup S$ ). By the definition of  $\alpha_q$ , there exists a computation segment  $\sigma * s$  and a run  $r * q$  over it. If  $s$  is terminal in  $P$ , i.e.,  $t(s)$  is undefined for all  $t \in T$ , then  $\sigma * s$  is a computation of  $P$  and  $r * q$  a run over it, which is rejecting, due to

$q \in B$ . This contradicts the assumption that  $A$  is valid over  $\mathcal{P}$ . Hence  $s$  cannot be terminal, and must, therefore satisfy  $\bigvee_{t \in T} p_t$ .

### Ranking Functions

We follow the techniques of [LPS], [GFMR], and [AP] in assigning ordinals to program states in a computation. This assignment defines (for each automaton state  $q$ ) a function  $\rho_q$ , mapping program states into the ordinals, which are taken as the domain of the well-founded relation  $(W, \prec)$  required in the rule.

We start by constructing the run tree  $\mathcal{R}$  of the automaton  $A$  over the program  $P$ . The root of the run tree is a special node called  $r_0$ . The immediate descendants of  $r_0$  are nodes  $n_1, n_2, \dots$  labeled by pairs of the form  $\langle s_i, q_i \rangle$ , where  $s_i$  is a program state satisfying  $\Theta$ , and  $q_i \in Q$  is an initial automaton state such that  $s_i \models c(q_i)$ . We assume that the program states are enumerable, and hence a sequence of all the initial program states, i.e., those that satisfy  $\Theta$ , can be constructed.

Except for the root node  $r_0$ , which we may consider to be labeled by  $r_0$ , all the nodes in  $\mathcal{R}$  are labeled by pairs  $\langle s, q \rangle$ . Let  $n \neq r_0$  be a node labeled by  $\langle s, q \rangle$ . The direct descendants of  $n$  are defined as follows: For each  $s' = t(s)$  and  $q'$  such that  $s' \models c(q, q')$ ,  $n$  has a direct descendant  $n'$  labeled by  $\langle s', q' \rangle$ . Obviously, each node, except possibly  $r_0$ , has only finitely many direct descendants, corresponding to the finitely many transitions in  $T$  and states in  $Q$ . For a node  $n \neq r_0$  we denote its immediate parent by  $par_{\mathcal{R}}(n)$ .

It is not difficult to see that any rooted maximal path  $\pi$  in  $\mathcal{R}$  (i.e., a path starting at  $r_0$ ) defines a computation  $\sigma$  and a run  $r$  over it. This also shows that for all pairs  $\langle s, q \rangle$  labeling nodes in the tree,  $s \models \alpha_q$  holds for the invariants  $\alpha_q$  defined above, since  $\langle s, q \rangle$  are reachable by a computation segment and a run over it.

Conversely, for any computation  $\sigma = s_0, s_1, \dots$  and any run  $r = q_0, q_1, \dots$  over  $\sigma$ , there exists a maximal path in  $\mathcal{R}$  whose sequence of labels is  $r_0, \langle s_0, q_0 \rangle, \langle s_1, q_1 \rangle, \dots$ .

Nodes in the tree that are labeled by  $\langle s, q \rangle$ , for  $q \in S$ , are called *stable nodes*. All other nodes, including  $r_0$ , are called *unstable*. Since  $A$  accepts all computations of  $P$  (and  $R = \emptyset$ ), all paths in  $\mathcal{R}$  contain only finitely many unstable nodes.

For any node  $n$  there exists a path  $\pi$  leading from  $r_0$  to  $n$ . We define the *lowest unstable ancestor* of  $n$ , denoted by  $lua(n)$ , to be the last unstable node on the path  $\pi$ . Since  $r_0$  itself is unstable,  $lua(n)$  is always defined. If  $n$  is unstable then  $lua(n) = n$ .

### Compressed Tree

From the tree  $\mathcal{R}$  we construct another tree  $\mathcal{C}$ , called the *compressed tree*, with a mapping from the nodes in  $\mathcal{R}$  onto the nodes of  $\mathcal{C}$ . The nodes in  $\mathcal{C}$  consist of a node  $\bar{n}$  for each *unstable* node  $n$  in  $\mathcal{R}$ . The root of  $\mathcal{C}$  is  $\bar{r}_0$ . For all other nodes  $\bar{n}$ , we define the parent of  $\bar{n}$ ,  $par_{\mathcal{C}}(\bar{n})$ , by the expression:

$$par_{\mathcal{C}}(\bar{n}) = \overline{lua(par_{\mathcal{R}}(n))}.$$

By this definition, to find the parent of a node  $\bar{n}$  in  $\mathcal{C}$ , go back to  $\mathcal{R}$ . Locate  $n'$ , the parent of  $n$  in  $\mathcal{R}$ . Find  $n''$ , the *lua* of  $n'$ , which must be unstable. Then take  $\bar{n}''$  to be the parent of  $\bar{n}$  in  $\mathcal{C}$ .

The mapping between nodes in  $\mathcal{R}$  and nodes in  $\mathcal{C}$  is given by  $f(n) = \overline{lua(n)}$ . It has the property that, if  $n$  is the parent of  $n'$  in  $\mathcal{R}$ , then either  $f(n) = f(n')$ , or  $f(n)$  is the parent of  $f(n')$  in  $\mathcal{C}$ . This shows that a path  $\pi$  in  $\mathcal{R}$  can be mapped to a path in  $\mathcal{C}$ , which we denote by  $f(\pi)$ . Conversely, for every path  $\bar{\pi}$  in  $\mathcal{C}$ , there exists a path  $\pi$  in  $\mathcal{R}$  such that  $f(\pi) = \bar{\pi}$ . Since all paths in  $\mathcal{R}$  have only finitely many unstable nodes, it follows that all paths in  $\mathcal{C}$  are *finite*.

However, the price paid for achieving the finite-path property is that, in general, many nodes in  $\mathcal{C}$  may have infinite degree, while in  $\mathcal{R}$  all nodes, except possibly  $r_0$ , have a uniformly bounded finite degree.

A generalization of this construction is presented in [Ha2], where it is described as a transformation from “thin” infinite-path trees, into “fat” but finite-path trees.

A tree with the finite-path property is also called a well-founded tree. If all the degrees are countable, which is the case here, we can assign countable ordinal ranks  $\delta(\bar{n})$  to all the nodes  $\bar{n}$  in such a tree. The assignment is the following:

- If  $\bar{n}$  is a leaf, then  $\delta(\bar{n}) = 0$ .
- If  $\bar{n}$  is not a leaf, then  $\delta(\bar{n}) = \text{lub}\{\delta(\bar{n}') + 1 \mid \bar{n} = \text{par}_{\mathcal{C}}(\bar{n}')\}$ ,

where *lub* is the least upper bound.

#### Definition of Ranking Functions

We are now ready to define the ranking function  $\rho_q(s)$  for every automaton state  $q$  and program state  $s$  such that  $s \models \alpha_q$ . As premise (R1) requires, these are the only program states for which  $\rho_q(s)$  has to be defined.

Let  $\langle s, q \rangle$  be a pair such that  $s \models \alpha_q$ . There exist one or more nodes  $n$  in  $\mathcal{R}$  which are labeled by  $\langle s, q \rangle$ . Consider the tree  $R(n)$  which is the sub-tree of  $\mathcal{R}$  rooted at  $n$ , i.e.,  $R(n)$  consists of  $n$  and all of its descendants. Note that in the case that  $\mathcal{R}$  has more than one node  $n$  labeled by  $\langle s, q \rangle$ , all the trees  $R(n)$  are isomorphic (which means that there is a one-to-one mapping between their nodes which preserves the labels), so it does not matter which one we pick.

Construct  $C(n)$ , the compressed version of  $R(n)$ , in a way similar to the construction of  $\mathcal{C}$  from  $\mathcal{R}$ . The only difference is that for the computation of the *lua* in  $R(n)$ , the root  $n$  is considered unstable, regardless of whether  $n$  is unstable in  $\mathcal{R}$ . Assign ordinals to the nodes of  $C(n)$ . Let  $\delta$  be the ordinal assigned to the root  $\bar{n}$  of  $C(n)$ . Then we define  $\rho_q(s) = \delta$ .

This ranking has two important properties. Let  $n$  be the parent of  $n'$  in  $\mathcal{R}$ , where  $n, n'$  are labeled by  $\langle s, q \rangle$  and  $\langle s', q' \rangle$ , respectively. Then it can be shown that:

- Always  $\rho_q(s) \geq \rho_{q'}(s')$ .
- If  $q' \notin S$ , i.e.,  $n'$  is unstable in  $\mathcal{R}$ , then  $\rho_q(s) > \rho_{q'}(s')$ .

These two properties lead to the fact that premises (R2) and (R3) are satisfied by the  $\alpha_q$  and  $\rho_q$  chosen above.

To see this, consider a state  $s$  satisfying  $\alpha_q$ , and a successor state  $s'$  satisfying  $c(q, q')$ . Premise (R2) requires showing that  $\rho_{q'}(s') \preceq \rho_q(s)$ , while (R3) requires that if  $q' \notin S$ , then  $\rho_{q'}(s') \prec \rho_q(s)$ . By

our construction both  $\langle s, q \rangle$  and  $\langle s', q' \rangle$  are nodes in the tree  $R$ , with  $\langle s, q \rangle$  being the parent of  $\langle s', q' \rangle$ . Hence, by the two observations above, the two premises are satisfied.

### (B) Syntactic Completeness

Next, we have to show that the  $\alpha_q$  and  $\rho_q$  chosen above can be expressed in the assertion language  $L$ .

A small technical problem is that, formally,  $\rho_q$  is not an assertion but a function. This can be resolved by replacing the ranking function by a *ranking* assertion  $\varphi_q(w)$ , parametrized by an element  $w$  of the well-founded set  $W$ . The meaning of the assertion  $\varphi_q(w)$  is given by:

$$s \models \varphi_q(w) \Leftrightarrow \rho_q(s) = w.$$

After this modification, it remains to show that  $\alpha_q$  and  $\varphi_q(w)$  are expressible in  $L$ . We refer the reader to [AP], [SRG], and [Fr] for arguments showing that if  $L$  contains the fixpoint operators of the  $\mu$ -calculus, then the assertions  $\alpha_q$  and  $\varphi_q(w)$ , verbally defined above, can be expressed in  $L$ .

### Discussion

One of the points emerging when considering the V-rule is that the need for infinite ordinals is essential in this process. In previous papers, the need for infinite ordinals was usually attributed to concurrency, fairness, or unbounded non-determinism in the programming language or computational model (see for example [LPS], [Fr], [AP]). As the present paper shows, the need for infinite ordinals arises even in a simple non-deterministic program without concurrency, fairness, or unbounded non-deterministic constructs, such as program  $\mathcal{P}_1$  in the example of Section 7. We advance the thesis that the element responsible for the need for infinite ordinals is the *specification language*. As soon as we deal with non-terminating programs, and have a specification language strong enough to express liveness properties more complex than unconditional eventualities, infinite ordinals are necessary.

## 12. FAIRNESS

In this section we sketch the extensions needed to the approach in order to accommodate *fairness*. While fairness was included in the basic model of programs, represented by a set of fairness requirements  $\langle \varphi, \psi \rangle$ , all the proof rules, and claims of completeness, were discussed up to this point with the restrictive assumption of an empty set of fairness requirements. Suppose we are presented with a program  $P$  that contains fairness requirements  $\langle \varphi_i, \psi_i \rangle$  for  $i = 1, \dots, n$ , and we are asked to verify a property  $\chi$ .

There are two basic approaches to solve this problem. The first is by extending the property to be proven. The second is by extending the proof rules.

According to the first approach we use the V-rule, as presented above, to prove the extended property

$$\chi' : \bigvee_{i=1}^n [\Box \Diamond \varphi_i \wedge \Box \Box \quad (+Z)] \vee \chi$$

over the program  $P'$ , obtained from  $P$  by omitting all the fairness requirements.

The property  $\chi'$  is valid over the program  $P'$  iff every computation of  $P'$  either violates one of the fairness requirements, or satisfies  $\chi$ . In other words,  $\chi$  is satisfied by all the computations of  $P$ . i.e., those computations of  $P'$  that also satisfy the fairness requirements of  $P$ .

Since the extension, expressed above by temporal logic, is readily expressed by automata, this provides one way of dealing with fairness. Note that if  $\chi$  is one of the properties, such as  $\Box \varphi$  or  $\Diamond \varphi$ , for which the temporal proof system provides direct rules, the extended property  $\chi'$  does not have this form any longer. Consequently the direct rules are no longer applicable without further changes, that explicitly reflect the fairness requirements. As a result, temporal logic proof systems (see, e.g., [MP2]), provide several versions of rules for establishing liveness properties such as  $\Diamond \varphi$ . A basic version (similar to our  $v$ -rule) does not use any fairness assumption. Another version relies on the weak fairness properties of the program, and still another one, utilizes also the strong fairness properties.

The second approach, considered here, follows precisely that route, and suggests a modified V-rule, in which the modifications directly reflect the fairness requirements. We present here only the simpler version of weak fairness. A fairness requirement of the form  $\langle \varphi, \psi \rangle$  is called a weak-fairness requirement if  $\varphi = \top$ .

This requirement demands that if the computation is infinite, it should contain infinitely many program states satisfying  $\psi$ . Furthermore, we illustrate in detail only the case of a single weak fairness requirement.

The rule, modified for a single weak-fairness requirement, calls for two invariants,  $\alpha_q^1$  and  $\alpha_q^2$ , to be associated with each  $q \in Q$ . A well-founded relation and ranking functions are needed, as before. Denoting  $\alpha_q^1 \vee \alpha_q^2$  by  $\alpha_q$ , premises (I1)–(I3) and (R1) remain unchanged.

The premise (R2) is replaced by:

$$(R2.1) \left\{ \alpha_q^1 \wedge (\rho_q = w) \right\} \mathcal{P} \left\{ c(q, q') \rightarrow \left[ \left[ \alpha_{q'}^i \wedge (\rho_{q'} \prec w) \right] \vee \left[ \alpha_{q'}^i \wedge (\rho_{q'} = w) \right] \right] \right\} \\ \text{for each } q \in Q, q' \in S \cup B, i = 1, 2$$

$$(R2.2) \left[ \alpha_q^1 \wedge \psi \right] \rightarrow \alpha_q^2 \text{ for each } q \in Q,$$

$$(R2.3) \left\{ \alpha_q^2 \wedge (\rho_q = w) \right\} \mathcal{P} \left\{ c(q, q') \rightarrow (\rho_{q'} \prec w) \right\} \text{ for each } q \in Q, q' \in B.$$

We can explain this modification by observing that the invariant  $\alpha_q$  has been split into the two invariants  $\alpha_q^1$  and  $\alpha_q^2$ . The case of  $\alpha_q^2$  records the fact that a state satisfying  $\psi$  has been detected since the last decrease in  $\rho$ . In the simpler case, where no fairness was assumed, we relied on the well-founded ranking to ensure that we either visit infinitely many R-states, or that we visit only finitely many bad states (states in B). Concentrating on a run that avoids R-states from a certain point on, the original premise (R3) required that the rank decreases on any visit to a had state. Since the ranking is well-founded, we can only visit finitely many bad states.

Here, we do not insist on finitely many bad states in general. Only runs that have infinitely many S-states are required to have this property. Thus, we do not require a decrease on every visit to a bad state. Instead, we require a decrease on each first visit to a bad state, following a visit to a  $\psi$ -state. Consequently, infinitely many bad states contradict the well-foundedness of the ranking only if they are coupled with infinitely many  $\psi$ -states.

To see how this requirement is expressed by the premises (R2.1)–(R2.3), observe the following:

By (I2), each successor of a state satisfying  $\alpha_q^1 \vee \alpha_q^2$  must also satisfy  $\alpha_q^1 \vee \alpha_q^2$ .

Premise (R2.1) requires that as long as we do not visit an R-state (and the interesting case is when we visit only finitely many R-states), the rank cannot increase, which means that it either decreases or remains the same. The premise also requires that if the rank has not decreased, then the identity of the sub-assertion, i.e.,  $\alpha_q^1$  or  $\alpha_q^2$ , is preserved over the transition. If the rank decreases, then we only know that  $\alpha_{q'}$  (i.e., either  $\alpha_{q'}^1$  or  $\alpha_{q'}^2$ ) holds after the transition.

Premise (R2.2) forces any state satisfying  $\alpha_q^1$  and  $\psi$  to also satisfy  $\alpha_q^2$ . The switch to  $\alpha_q^2$  records a visit to a  $\psi$ -state.

Premise (R2.1) for  $i = 2$  ensures that once we are in the  $\alpha^2$  mode of the assertions, we remain in this mode as long as the rank does not decrease. This is because we wish to preserve the information that, we have recently encountered a  $\psi$ -state until the next visit to a B-state, which will force such a decrease.

Premise (R2.3) requires that a transition starting at  $\alpha_q^2$  and entering a B-state, should decrease  $\rho$ . Observe that this is weaker than the original (R3), which it replaces, since here the decrease is required only if the transition started at a  $\alpha_q^2$ -state, while the original premise requires a decrease from any  $\alpha_q$ -state. Note that on entry to a B-state from a program state satisfying  $\alpha_q^2$ , which by (R2.3) is accompanied by a rank decrease, we are allowed, by (R2.1) to switch back to  $\alpha_q^1$ , thus erasing the record of a recent encounter of  $\psi$ .

Hence, if  $\psi$  occurs on entry to any state, it causes  $\alpha_q^2$  to become true (due to (R2.2)). If the next transition enters a B-state it causes  $\rho$  to decrease and allows  $\alpha_q^2$  to change back to  $\alpha_q^1$ . If the next transition enters an S-state, the  $\alpha^2$ -mode is preserved until the first exit from S. If that exit moves to a B-state, it causes a decrease in  $\rho$ . If the exit is to an R-state, then  $\rho$  may assume an arbitrary value.

This rule is sound and complete for the simple case of a single weak-fairness requirement. It is easy to see how it can be generalized to the case of  $n > 1$  weak-fairness requirements, associated with the assertions  $\psi_1, \dots, \psi_n$ . We split each  $\alpha_q$  into  $\alpha_q^1, \dots, \alpha_q^{n+1}$ , where  $\alpha_q = \alpha_q^1 \vee \dots \vee \alpha_q^{n+1}$ . The three premises assume then the form:

$$(R2.1) \quad \left\{ \alpha_q^i \wedge (\rho_q = w) \right\} \mathcal{P} \left\{ c(q, q') \rightarrow \left[ \left[ \alpha_{q'}^i \wedge (\rho_{q'} < w) \right] \vee \left[ \alpha_{q'}^i \wedge (\rho_{q'} = w) \right] \right] \right\} \\ \text{for each } q \in Q, q' \in S \cup B, i = 1, \dots, n+1,$$

$$(R2.2) \quad [\alpha_q^i \wedge \psi_i] \rightarrow \alpha_q^{i+1} \quad \text{for each } q \in Q, i = 1, \dots, n$$

$$(R2.3) \quad \left\{ \alpha_q^{n+1} \wedge (\rho_q = w) \right\} \mathcal{P} \left\{ c(q, q') \rightarrow (\rho_{q'} < w) \right\} \quad \text{for each } q \in Q, q' \in B.$$

In this representation  $\alpha_q^i$  for  $i = 2, \dots, n+1$  records the fact that we have already encountered  $\psi_1, \dots, \psi_{i-1}$  since the last decrease in rank.

It can be shown that the two suggested extensions, extending the property and extending the rule, are closely related. There exists a direct translation between a proof using  $\alpha_q, \rho_q$  on the extended property  $\chi'$  and a proof using  $\alpha_q^1, \dots, \alpha_q^{n+1}, \rho_q$  in the extended rule on the property  $\chi$ .



### 13. MODEL CHECKING

In this section we sketch an algorithm that checks the validity of a given V-automaton A over a given finite-state program P. The finiteness restriction is obtained by restricting the set of variables, to which the program and the automaton refer, to a finite subset  $V' \subseteq V$ , all of whose variables range over finite domains. This restriction leads to a finite number of program states  $\Sigma$  and to the fact that all assertions, even the ones containing quantifiers or fixpoint operators, are decidable, i.e., their validity or inconsistency can be checked algorithmically. The latter is due to the fact that any formula in the assertion language L is equivalent, under the restriction of finiteness, to a boolean combination of atomic formulas of the form  $u = a$ , for some variable  $u \in V$  and a constant a in the domain over which u ranges.

For simplicity, we assume that P has no fairness requirements. In case it has, they could be added to A in the manner discussed in the section dealing with fairness. We also assume that the automaton A is complete and has an empty R. As we have seen, these assumptions lead to no loss of generality.

A finite-state program P can be represented as a finite graph  $G_P$ , whose nodes  $N_P$  are the states  $\Sigma$ , and whose edges  $E_P$  connect  $s$  to  $s'$  iff  $s' = t(s)$  for some transition  $t \in T$ . A subset of the states  $\Sigma_0 \subseteq \Sigma$  is designated as the subset of the *initial* nodes in the graph. These are all the states  $s$  that satisfy  $\Theta$ , the precondition of P.

The algorithm starts by constructing a graph  $G = (N, E)$  that can be viewed as the *cross-product* of the program graph  $G_P$  with the automaton A (viewed as a flat transition graph).

The nodes  $N$  in the graph G are pairs of the form  $\langle s, q \rangle$  where  $s \in \Sigma$  is a program state and  $q \in Q$  is an automaton state. Some nodes,  $N_0 \subseteq N$ , are designated as initial in G. They are all the nodes  $\langle s, q \rangle$  such that

$$s \in \Sigma_0 \quad \text{and} \quad s \models e(q).$$

Edges in  $E$  are drawn between  $\langle s, q \rangle$  and  $\langle s', q' \rangle$  iff

$$s' = t(s) \text{ for some } t \in T \quad \text{and} \quad s' \models c(q, q').$$

Note that in the construction of  $N_0$  and  $E$ , we used the fact that questions such as  $s \models \varphi$  are decidable for each  $s \in \Sigma$  and  $\varphi$  in L.

Consider a maximal path  $\pi$  in G, starting at some initial node  $n \in N_0$ . Clearly, such a path defines a computation  $\sigma$  of P and a run  $r$  of A over it. The algorithm presented here searches for a *rejecting* run. If it fails to find such a run, we conclude that A is valid over P.

**Algorithm:** As a preliminary transformation, we eliminate from G all the nodes which are unreachable from initial nodes. These can never participate in a rejecting run.

We now repeat the following sequence of steps until either success or failure are announced. Some of the steps remove nodes and edges from the graph, so when we refer to G and its elements, we mean the elements in the current version of G.

- (1) If the graph G is empty, stop and announce failure (to find a rejecting run). This means that A is valid over P.

- (2) Decompose the graph into maximal strongly connected components, and let  $C \subseteq G$  be such a component which is terminal, i.e., there are no edges in the current version of  $G$  connecting nodes in  $C$  to nodes in  $G-C$ .
- (3) If  $C$  is a singleton node  $\langle s, q \rangle$ , with no edges leaving it, consider the following subcases:
  - If  $s$  is not terminal in  $P$  (i.e.,  $t(s)$  is defined for some  $t \in T$ ) or  $q \in S$ , the node  $\langle s, q \rangle$  cannot participate in a rejecting run. Proceed to step 5.
  - Otherwise, we have identified a reachable node  $\langle s, q \rangle$  such that  $s$  is terminal in  $P$  and  $q \notin S$ . This means that  $A$  is not valid over  $P$ . Stop and announce success.
- (4) Otherwise,  $C$  is a component that has a cycle  $K$  going through all the nodes in  $C$ . If  $C$  has a node  $\langle s, q \rangle$  with  $q \notin S$ , stop and report success. The infinite path starting at some initial node, reaching  $C$ , and then infinitely repeating  $K$ , passes infinitely many times through the automaton state  $q$ , and hence generates a rejecting run.
- (5) The component  $C$  cannot participate in a rejecting run. Delete all nodes in  $C$  and all edges leading to these nodes from  $G - C$ . Return to (1).  $\blacksquare$

The above algorithm is very similar to several previous model checking algorithms for linear and branching temporal logics as well as automata-based algorithms, such as in [CES], [LP], [EL] and [VW]. The complexity of the algorithm is linear in the size of the initial graph  $G$ , i.e., in  $(A \mid x \mid G_P \mid)$ .

## ACKNOWLEDGEMENT

The decision to introduce non-determinism into our automata was inspired, to some extent, by a suggestion of Moshe Vardi to examine the Büchi automaton corresponding to the negation of a formula, and show that it accepts no computations of the program. This suggestion and the related discussion are gratefully acknowledged. His approach to verification by automata is presented in [V].

We thank Fred Schneider for helpful discussions concerning his work on the subject of automata as specification devices. We also thank Martín Abadi, Marianne Baudinet, Tom Henzinger and Alur Rajeev for critical reading of the manuscript.

## REFERENCES

- [AP] K. Apt, G.D. Plotkin — Countable Nondeterminism and Random Assignment, *JACM* 33,4 (1986), 724–767.
- [AS] B. Alpern, F.B. Schneider — Verifying Temporal Properties without using Temporal Logic, to appear in *TOPLAS*.
- [CES] E.M. Clarke, E.A. Emerson, A.P. Sistla — Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications, *TOPLAS* 8,2 (April 1986). 244-263.

- [D] E.W. Dijkstra — *A Discipline of Programming*. Prentice Hall (1976).
- [EL] E.A. Emerson, C.L. Lei -- Modalities for Model Checking: Branching Time Strikes Back. *12th Symp. on Principles of Programming Languages* (1985), 84-96.
- [Fl] R.W. Floyd — Assigning Meanings to Programs, in *Mathematical Aspects of Computer Science*, 19th Symp. of Appl. Math., American Mathematical Society, Providence (1967), 19-32.
- [Fr] N. Francez — *Fairness*, Springer-Verlag (1986).
- [GFMR] O. Grumberg, N. Francez, J.A. Makowsky, W.P. deRoever — 4 Proof Rule for Fair Termination of Guarded Commands, *Information and Control* 66 (1985), 83-102.
- [Hal] D. Harel — Statecharts: A Visual Formalism for Complex Systems, *Technical Report*. Weizmann Institute (1984).
- [Ha2] D. Harel — Effective Transformations on Infinite Trees, with Applications to High Undecidability, Dominoes, and Fairness, *JA CM* 33.1 (1986), 224-248.
- [Ho] C.A.R. Hoare — An Axiomatic Approach to Computer Programming, *CACM* 12 (1969), 576-583.
- [LP] O. Lichtenstein, A. Pnueli — Checking that Finite-State Concurrent Programs Satisfy their Linear Specifications. *12th Symp. on Principles of Programming Languages* (1985). 97-107.
- [LPS] D. Lehmann, A. Pnueli, J. Stavi — Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. *LNCS* 115. Springer-Verlag (1981).
- [MP1] Z. Manna, A. Pnueli — Verification of Concurrent Programs: The Temporal Framework, in *The Correctness Problem in Computer Science* (R.S. Boyer, J.S. Moore, eds.), Academic Press (1981), 215-274.
- [MP2] Z. Manna, A. Pnueli — Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs, *Science of Computer Programming* 4 (1984), 257-289.
- [OL] S. Owicki, L. Lamport — Proving Liveness Properties of Concurrent Programs, *TOPLAS* 4.3 (1982), 455-495.
- [P] G.L. Peterson — Myths about the Mutual-Exclusion Problem, *Information Processing Letters* 12,3 (1981), 115-116.
- [SRG] F.A. Stomp, W.P. deRoever, R.T. Gerth — The P-Calculus as an Assertion Language for Fairness Arguments. *Technical Report* 84-12. Utrecht (1984).
- [V] M.Y. Vardi — Verification of Concurrent Programs: The Automata-Theoretic Framework, *2nd Symp. on Logic in Computer Science*. Ithaca (1987), 167-176.
- [VW] M.Y. Vardi, P. Wolper — An Automata-Theoretic Approach to Automatic Program Verification, *IEEE Symp. on Logic in Computer Science*, Cambridge (1986), 332-344.
- [W] P. Wolper — Temporal Logic can be More Expressive, *22nd Symp. on Foundations of Computer Science* (1981), 340-348.
- [WVS] P. Wolper, M.Y. Vardi, A.P. Sistla — Reasoning about Infinite Computation Paths. *24th Symp. on Foundations of Computer Science*. Tucson (1983), 185-194.

