May 1982

Hcport No. STAN-CS-82-915

Verification of Concurrent Programs: Proving Eventualities by Well-Founded Ranking

by

Zohar Manna and A. Pnueli

Department of Computer Science

Stanford University
Stanford, CA 94305

VERIFICATION OF CONCURRENT PROGRAMS: PROVING EVENTUALITIES BY WELL-FOUNDED RANKING

by

ZOHAR MANNA
Computer Science Department
Stanford University
Stanford, CA
and
Applied Mathematics Department
The Wcizmann Institute
Rchovot, Israel

AMIR PNUELI
Applied Mathematics Department
The Wcizmann Institute
Rehovot, Israel

ABSTRACT

In this paper, one of a series on verification of concurrent programs, we present proof methods for establishing eventuality and until properties. The methods are based on *well-founded ranking* and arc applicable to both "just" and "fair" computations. Thcsc methods do not assume a decrease of the rank at each computation step. It is sufficient that there exists one process which decreases the rank when activated. Fairness then ensures that the program will eventually attain its goal.

In the finite state case the proofs can be represented by diagrams. Several examples are given.

This research was supported in part by the National Science Foundation under Grants MCS79-09495 and MCS80-06930, by DARPA under Contract N00039-82-C-0250, by the United States Air Force Office of Scientific Research under Grant AFOSR-81-0014, and by the Basic Research Foundation of the Israeli Academy of Sciences.

INTRODUCTION

In a previous report [MP1] we introduced the temporal framework for reasoning about programs. We described a model of concurrent programs which is based on interaction via shared variables and defined the concept of fair execution of such programs. Wc then demonstrated the application of temporal logic formalism for *expressing* properties of concurrent programs. Program properties can be classified according to the syntactic form of the temporal formula expressing them; we studied three classes of properties: invariance properties, eventuality properties and precedence ("until") properties. Most program properties that have been previously considered or studied for sequential and concurrent programs fall into one of these three categories.

In a second report [MP2], we developed proof principles based on temporal logic for *establishing* that concurrent programs possess properties of these classes. We presented a proof method for each class of properties.

- A single invariance principle is adequate for establishing invariance properties.
- For proving eventuality properties, we recommended a chain reasoning approach, in which we follow the possible chains of events until the desired goal is realized. Several proof principles were introduced for establishing the basic steps in the chain. A similar approach is presented in [OL].
- Simple precedence properties may be proved by a combination of invariance proofs **and** eventuality proofs. A forthcoming report ([MP3]) will discuss proof methods for general precedence properties.

In this paper, we present an alternative method for proving eventuality and "until" properties based on convergence functions *(well-founded rankings)*.

In our exposition, we assume that the reader is familiar with the basic concepts and definitions introduced in [MP1] and [MP2].

THE CONVERGENCE FUNCTION APPROACH

Unlike the chain reasoning approach, which displays a variety of strategies and rules, the convergence function approach provides a single uniform principle for proving eventualities of the form:

$$\models \quad \varphi \supset \Diamond \psi,$$

(i.e., if φ ever arises it must be followed by ψ), as well as "until" properties of the form

$$\models \quad \varphi \supset (\chi \, \mathcal{U} \, \psi),$$

(*i.e.,* if φ ever arises it must bc followed by an instant at which ψ is realized and between the occurrences of φ and ψ, χ must hold continuously).

With respect to uniformity, the convcrgcncc function approach resembles the invariance principle for proving invariance properties. Another common feature is that establishing the premises to the proof rule requires only static (non-temporal) reasoning.

Convergence functions have been used successfully in proofs of termination of sequential programs and of rewriting systems (e.g., [M], [DM]). Their use is based on a mapping from the execution states of a program into a well-founded set, such that states which appear later in a computation correspond to lower values in the set. Consequently, a complete computation will correspond to a descending sequence, and an infinite computation would correspond to an infinitely descending sequence of well-founded elements, which is impossible. Such a mapping is called a *convergence function* or a *ranking function*.

A *well-founded structure* $(W, \succ)$ consists of a set W and a partial order $\succ$ on W such that any decreasing sequence $w_0 \succ w_1 \succ w_2 \succ \ldots$, where $w_i \in W$ is finite. A typical and frequently used well-founded structure is $(N, >)$, where N is the set of all nonnegative integers, and $>$ is the usual "greater than" ordering. Obviously we cannot have an infinitely decreasing sequence of nonnegative integers, and therefore $(N, >)$ is indeed a well-founded structure.

A general method for deriving composite well-founded structures from simpler ones is the formation of lexicographical orderings. Let $(W_1, \succ_1)$ and $(W_2, \succ_2)$ be two well-founded structures. Then the structure given by $(W_1 \times W_2, \succ_{lex})$ where the lexicographic ordering $\succ_{lex}$ is defined by

$$(m_1, m_2) \succ_{lex} (n_1, n_2) \quad \Leftrightarrow \quad (m_1 \succ_1 n_1) \text{ or } (m_1 = n_1 \text{ and } m_2 \succ_2 n_2)$$

is also well-founded.

Let us consider the application of the classical convergence function approach to the following concurrent program:

Example A (Program *DGCD* — distributed *gcd* computation)

$$(y_1, y_2) := (x_1, x_2)$$

ℓ_0 : *while* $y_1 \neq y_2$ *do*	m_0 : *while* $y_1 \neq y_2$ *do*
if $y_1 > y_2$ *then* $y_1 := y_1 - y_2$	*if* $y_1 < y_2$ *then* $y_2 := y_2 - y_1$
ℓ_1 : *halt*	m_1 : *halt*
— P_1 —	— P_2 —

This program performs the distributed computation of the gcd (greatest common divisor) of two positive integers inputs x_1, x_2. In the execution of this program, we assume each of the labelled instructions to be atomic in the sense that testing and modification of the variables by one process, say P_1 at ℓ_0, are completed before the other process may access them. Note that when P_1 is activated in a state in which $y_1 < y_2$ it does not modify any of the variables and returns to ℓ_0, thus replicating exactly the original state. Consequently, the termination, and hence the correctness of this program, depends very strongly on the basic assumption of *fairness* that we assume throughout this work. Only under fairness would each of P_1 and P_2 be activated as often as needed until convergence is achieved.

Trying to prove the convergence of this program by well-founded ranking immediately runs into difficulties when we fail to find a mapping into a well-founded set that will decrease at every step of the computation. No such function can exist for the above program since, as observed earlier, some steps may preserve the state and leave the value of a state-dependent convergence function constant. This points out cmphatically that any well-founded argument may succeed only if it takes fairness into 'account.

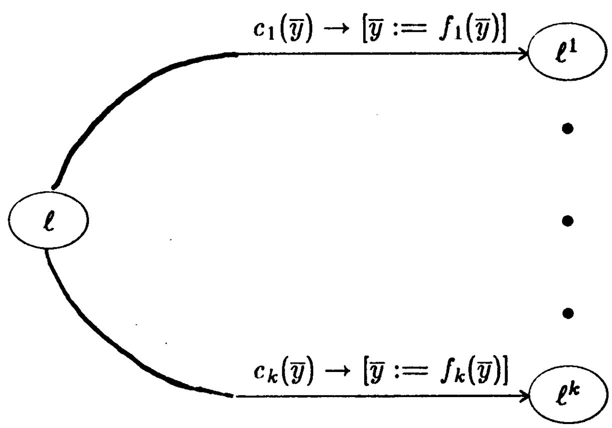
PROGRAMS AND COMPUTATIONS

For complcteness we repeat some of the dclinitions of [MP1] and introduce some additional notation required here. Let P be a program consisting of m parallel processes:

$$P: \quad \overline{y} := f_0(\overline{x}); [P_1 || \ldots || P_m].$$

Each process P_i may be represented as a transition graph with locations (nodes) labclled by clements of $\mathcal{L}_i = \{\ell_0^i, \ldots, \ell_e^i\}$. The edges in the graph are labclled by guarded commands of the form $c(\overline{y}) \rightarrow [\overline{y} := f(\overline{y})]$ whose meaning is that if $c(\overline{y})$ is true the edge may bc traversed while replacing $\overline{y}$ by $f(\overline{y})$.

Let $\ell, \ell^1, \ldots, \ell^k \in \mathcal{L}_j$ bc locations in process P_j:

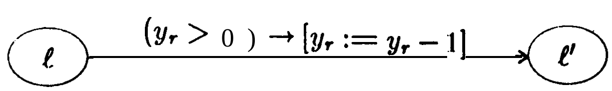


We define $E_\ell(\overline{y}) = c_1(\overline{y}) \vee \ldots \vee c_k(\overline{y})$ to be the *exit condition* at nodc ℓ. Locations in the program can bc classified according to their exit conditions.

- A location is regular if $E_\ell \equiv$ *true*. This is the case with locations such that the sct of conditions labeling their outgoing transitions is exhaustive in the sensc that for every possible value of $\overline{y}$ at lcast one transition is enabled. The only irregular locations are terminal locations and semaphore locations discussed next.

- A location is *terminal* if $E_\ell \equiv$ false. This is the casc with locations labeling ***halt*** instructions which have no outgoing transitions. In our model wc usually label these locations by ℓ_e.

- Any location ℓ such that the exit condition $E_\ell(\overline{y})$ is nontrivial is called a *semaphore* location. Examplcs of such locations arc those corresponding to the instruction ***request***(y_r) whose transition diagram is:



Note that $E_\ell(\overline{y}) = (y_r > 0)$. The *request* instruction is used in order to reserve a resource, where y_r may be considered as counting the number of units of this resource currently available. Its symmetric counterpart, the *release*(y_r) instruction, is used to release a reserved resource. Its transition diagram is:

$$\ell \xrightarrow{\ true \ \rightarrow [y_r := y_r + 1]\ } \ell'$$

The release instruction has as its exit condition $E_\ell \equiv$ *true*. Consequently its location is a regular location.

A state of the program P is a tuple of the form $s = (\overline{\ell}; \overline{\eta})$ with $\overline{\ell} \in \mathcal{L}_1 \times \ldots \times \mathcal{L}_m$ -and $\overline{\eta} \in D^n$, where D is the domain over which the program variables $y_1, \ldots, y_n$ range. The vector $\overline{\ell}$ is the set of current locations which are next to be executed in each of the processes. The vector $\overline{\eta}$ is the set of current values assumed by the program variables $\overline{y}$ at state s.

With each process P_i we associate a state transition function g_i that represents the possible outcomes of the activation of the process P_i on the state s. If we denote by S the set of all possible program states, g_i is a function $g_i : S \rightarrow 2^S$.

Nolc that this definition allows for the possibility that P_i is nondeterministic, since it is possible that $|g_i(s)| > 1$, i.e., there is more than one successor to s. Let $s = (\overline{\ell}; \overline{\eta})$. If ℓ_i is a terminal location, or a semaphore location with $E_{\ell_i}(\overline{\eta}) = false$, then P_i cannot bc activated on s. In such a case g;(s) = ϕ and we say that P_i is *disabled* on s. If ℓ_i is a regular location, or a semaphore location with $E_{\ell_i}(\overline{\eta})$ = *true* then g;(s) $\neq \phi$ and wc say that P_i is *enabled* on s.

A state $s \in$ S such that all processes are disabled on s is called *terminal*. A terminal state corresponds either to a situation in which all processes have terrninated or to a deadlock in which all the nonterminated processes wait in a semaphore location with a false exit condition.

- An *admissible computation* is a labelled (possibly infinite) sequence:

  $$\sigma: \quad s_0 \xrightarrow{P_{i_1}} s_1 \xrightarrow{P_{i_2}} s_2 \xrightarrow{P_{i_3}} \text{a3} \quad . \quad . \quad .$$

  such that every $s_j \in$ S and for every j $\geq$ 0, wc have $s_{j+1} \in g_{i_{j+1}}(s_j)$. Thus, such a computation could arise by an execution of the program starling from the initial state s_0. The computation will be finite only if it terminates in a terminal state s,. We can think of such a computation as generated under the guidance of an imaginary scheduler which at each step selects one of the processes (called the *activated* or scheduled process) and lets it execute a single instruction.

- A -*initialized computation* is an admissible computation in **which** $s_0 = (\ell_0^1, \ldots,$ CT;“; $f_0(\overline{\xi}))$. Here ℓ_0^i is the initial location in process P_i and f_0 is the initial assignment to lhc program variables.

- *A 'j-computation* is a $\overline{\xi}$-initialized computation or a suffix of a -initialized computation. Allowing suffixes of initialized computations enables us to study program behavior which may become observable only later in the computation.

- A ***φ-computation*** is a 'j-computation for any input values $\overline{\xi}$ satisfying a precondition φ.

The next definition embodies the basic assumption of fairness:

An admissible computation σ is *fair* if there is no process P_i such that P_i is enabled an infinite number of times in σ, and P_i is activated only finitely many times. Thus, fairness requires the imaginary scheduler to monitor the number of times a process becomes enabled, and to ensure that repeatedly enabled ones are not neglected forever. Any finite computation is necessarily fair.

In the absence of semaphore instructions, each process P_i is initially enabled and can become disabled only by terminating. Hence we can define the weaker notion of *just computation,* which replaces the requirement of being enabled an infinite number of times by the rcquircmcnt of being continuously enabled.

A computation σ is *just* if there is no process P_i such that P_i is continuously enabled beyond a certain state s in σ, and P_i is activated only finitely many times. Any finite computation is by definition just.

We denote the classes of all fair and just computations of a program P with precondition φ by $\mathcal{F}(\varphi, P)$, $\mathcal{J}(\varphi, P)$ respectively, or $\mathcal{F}(P)$, $\mathcal{J}(P)$ when the precondition φ is implicitly understood.

For an arbitrary program P we have in general

$$\mathcal{F}(P) \subset \mathcal{J}(P),$$

i.e., every fair computation is also just, but there may exist just computations which are unfair.

To see that the first claim holds, let σ be a fair computation. Let P_i be any process that is continuously enabled beyond a certain state in σ. Thus, P_i is certainly enabled an infinite number of times, and by fairness must be activated an infinite number of times. Hence σ is just.

To show that the inclusion between the sets *3(P)* and J(P) may be strict consider the following program which is the simplest program modclling mutual exclusion:

$$y := 1$$

ℓ_0:	*request(y)*	m_0:	*request(y)*
ℓ_1:	*release(y)*	m_1:	*release(y)*
ℓ_2:	*go to* ℓ_0	m_2:	*go to* m_0
$-P_1-$		$-P_2-$	

The following computation:

$$\sigma: \quad (\ell_0, m_0; 1) \xrightarrow{P_1} (\ell_1, m_0; 0) \xrightarrow{P_1} (\ell_2, m_0; 1) \xrightarrow{P_1}$$
$$(\ell_0, m_0; 1) \xrightarrow{P_1} (\ell_1, m_0; 0) \xrightarrow{P_1} (\ell_2, m_0; 1) \longrightarrow \ldots$$

is just. The process P_1 is activated infinitely many times. On the other hand P_2 is never continuously enabled since it is disabled in the infinitely recurring state $(\ell_1, m_0; 0)$, therefore justice does not require it to be activated at all. Obviously σ is unfair since P_2 is also enabled infinitely many times on all recurrences of $(\ell_0, m_0; 1)$, but is never activated.

However when P contains no semaphore instructions wc may use the above observation that a process is continuously enabled if and only if it is enabled infinitely many times, to conclude:

For a program without semaphores: *3(P) = J(P).*

Thus, in order to study programs without semaphores, we need only consider properties that hold for the class of all just computations;

PROGRAMS WITHOUT SEMAPHORES – JUST COMPUTATIONS

In this section we present a proof principle enabling us to prove eventuality properties that hold for the class of just computations $J(P)$.

The basic idea of the proof principle is to assign a convergence function $u : S \rightarrow W$ mapping the program states into a well-founded structure W. However, as shown in examples such as the *DGCD* program above, we should not require the function to decrease at every step. Instead we require that the function never increases and that for each slate there is always a process P_i, called the *helpful* process for this state, such that the activation of this process guarantees a decrease in the value of the function. By justice this *helpful* process will eventually be scheduled, so that any infinite just computation will necessarily generate an infinitely decreasing subsequence of well-founded elements – a contradiction. In the general case, the identity of the helpful process may vary from state to state. We therefore introduce *a helpfulness function* $h : S \rightarrow \{1, \ldots, m\}$ that identifies one helpful process $P_{h(s)}$ for each state $s \in S$.

We suggest the following proof method for proving *precedence* and *eventuality* properties of just computations.

Proof Method J:

For proving eventualities of the form $\varphi \supset \Diamond \psi$, under all just computations of a program P, find a state predicate $Q = Q(s)$, a well-founded structure $(W, \succ)$, a convergence function $u : S \rightarrow W$ and a helpfulness function $h : S \rightarrow \{1, \ldots, m\}$ such that:

$J1.\quad \models \varphi \supset (\psi \vee Q)$

$J2.\quad \models Q(s) \supset (g_{h(s)}(s) \neq \phi)$

$J3.\quad \models [Q(s) \wedge s' \in g_i(s)] \supset [\psi(s') \vee (Q(s') \wedge (u(s) \succeq u(s')))]$ for $i = 1, \ldots, m$

$J4.\quad \models [Q(s) \wedge s' \in g_{h(s)}(s)] \supset [\psi(s') \vee (u(s) \succ u(s'))]$

$J5.\quad \models [Q(s) \wedge s' \in g_i(s) \wedge (u(s) = u(s'))] \supset [\psi(s') \vee (h(s) = h(s'))]$ for $i = 1, \ldots, m$.

Then we may conclude that:

$$J(P) \models \varphi \supset \Diamond \psi.$$

Here $\models w$ means that w is true for all computations of P. The statement $J(P) \models w$ means that w is true for all just computations of P.

In these, $Q(s)$ is an invariant which is expected to remain true from the time φ becomes true until ψ is realized. Requirement $J1$ states that if φ holds for a state then either ψ or Q must hold in this state. $J2$ requires that the process that is helpful for a state s be enabled on s. $J3$ states that each step in the computation either realizes ψ or preserves Q and produces a value of u that is not higher than the value before the step. $J4$ states that taking a helpful step actually decreases the value of u. $J5$ states that a step which does not decrease the value of u must preserve the identity of the helpful process. The last condition is necessary in order to avoid an infinite sequence with constant value of u and continuously changing h. Such a sequence may be just but yet avoid realizing ψ.

Proof:

Let us justify this proof method by showing that if we succeed in finding Q, W, u and h as described above then indeed every just computation must satisfy $\varphi \supset \Diamond \psi$.

Let us consider a *just* computation:

$$\sigma: \quad s_0 \xrightarrow{P_{i_1}} s_1 \xrightarrow{P_{i_2}} s_2 \longrightarrow \ldots,$$

such that $\varphi(s_0)$ is true and ψ is nowhere realized. By J1 and $J3$, $Q(s_i)$ must be true for every s_i in the sequence. By $J2$ the sequence must be infinite since, for every s_i, $P_{h(s_i)}$ is enabled. Again by $J3$ the sequence of u values $u(s_0) \succeq u(s_1) \succeq \ldots$ must be a non-increasing sequence. By the well-foundedness of W there must be a k such that

$$u(s_k) = u(s_{k+1}) = \ldots$$

By $J5$, h also remains constant from s_k on, that is

$$h(s_k) = h(s_{k+1}) = \ldots.$$

Let its constant value be $r = h(s_k)$. In view of $J4$, P_r was never activated beyond s_k because its activation would have caused u to decrease. In view of $J2$, P_r is continuously enabled beyond s_k since everywhere $h(s_i) = r$ for $i \geq k$. This is obviously a blatant case of injustice $- P_r$ being continuously enabled and never activated. Thus, just sequences failing to realize ψ cannot exist, and any just sequence initialized with φ must eventually realize ψ. ∎

By looking at the proof for eventualities we observe that it guarantees the eventual realization of ψ and, by $J1$ and $J3$, as long as ψ is not realized, Q holds. This is exactly the definition of the until expression Q $\mathcal{U}\,\psi$. We therefore have:

Corollary: The proof method J also proves

$$J(P) \models \varphi \supset (Q \;\mathcal{U}\; \psi).$$

The treatment in [LPS] implies that this method is also complete, namely that if $\varphi \supset \Diamond \psi$ is true for all just computations of P then there always exist some Q, W, u, and h satisfying $J1 - J5$.

Related work dealing with similar methods for establishing fair termination, which is a special case of eventuality, is contained in [GFMR], [AO] and [Pa]. Earlier work on the termination of concurrent programs is described in [K], [Pn].

We will now proceed to illustrate the application of this method to proofs of eventuality properties of programs without semaphores.

Example A (Program *DGCD* --- distributed gcd computation):

Consider again the *DGCD* program. Let

$$\varphi: \quad at\,\ell_0 \wedge at\,m_0 \wedge (y_1, y_2) = (x_1, x_2) \wedge x_1 > 0 \wedge x_2 > 0$$

and

$$\psi: \quad at\,\ell_0 \wedge at\,m_0 \wedge y_1 = y_2 = gcd(x_1, x_2).$$

We wish to prove

$$\models \varphi \supset \Diamond\psi,$$

i.e.,

$$\begin{aligned} \mathcal{J}(P) \models [at\,\ell_0 \wedge at\,m_0 \wedge (y_1, y_2) = (x_1, x_2) \wedge x_1 > 0 \wedge x_2 > 0] \\ \supset \Diamond[at\,\ell_0 \wedge at\,m_0 \wedge y_1 = y_2 = gcd(x_1, x_2)]. \end{aligned}$$

That is, being at the starting point of the program with $(y_1, y_2) = (x_1, x_2)$ and positive inputs $x_1 > 0$, $x_2 > 0$, we are guaranteed to eventually get back to that point with y_1 being the greatest common divisor of x_1, x_2.

We choose Q, W, u, and h as follows:

$$Q(s): \quad at\,\ell_0 \wedge at\,m_0 \wedge y_1 > 0 \wedge y_2 > 0 \wedge gcd(y_1, y_2) = gcd(x_1, x_2) \wedge y_1 \neq y_2$$

W : $(N, >)$ – the nonnegative integers with the "greater than" relation

$u(y_1, y_2)$: $\quad y_1 + y_2$

$h(y_1, y_2)$: $\quad$ *if* $y_1 > y_2$ *then* P_1 *else* P_2

We have intentionally displayed h as a function into $\{P_1, P_2\}$ rather than $\{1, 2\}$ to stress the fact that it selects processes. It is not difficult to verify that requirements $J1$ to $J5$ hold for this choice of Q, W, u, and h. In particular, we note that Q implies that when $y_1 > y_2, P_1$ is helpful in decreasing $y_1 + y_2$ while for $y_1 \leq y_2$ (by Q : $y_1 < y_2$) P_2 is helpful. Note that once we are at (ℓ_0, m_0) with $y_1 = y_2$ the program will immediately proceed to the termination state at (ℓ_1, m_1).

AN INDEXING METHOD FOR JUST COMPUTATIONS

A variant of the convergence function approach uses elements of well-founded sets as indices to predicates. As we will show below the two variants are essentially equivalent, but certain problems may admit, proofs that are easier to present in the indexed form than in the convergence function form. As before, the method is based on finding a well-founded set $(V, \succ)$. We then consider predicates $R_v(s)$ with $v \in V$, $s \in S$ which are state predicates indexed by elements of V. States appearing later in the computation will satisfy R_v with lower values of v. Convergence is therefore assured by the impossibility of having a sequence of R_{v_i} with an infinitely decreasing values of v_i. However, as before we cannot, guarantee a strict decrease on every step. We therefore specify a *decrease function* $\delta : V \rightarrow \{1, \ldots, m\}$ which, similarly to the helpfulness function h, identifies the helpful process $P_{\delta(v)}$ Chat corresponds Lo any state s satisfying $R_v(s)$. Note that the identity of the helpful or decreasing process depends only on the index v and not on the state.

With this notation we now formulate the indexing method for just computations.

Proof Method IJ:

For proving eventualities of the form $\varphi \supset 0\ \psi$, under all just computations of a program P, find a well-founded structure $(V, \succ)$, an indexed family of predicates $R_v = R_v(Y)$, $v \in V$, and a *decrease* *function* $\delta : V \rightarrow \{1, \ldots, m\}$ such that:

$IJ1.$ $\models \varphi \supset [\psi \vee (\exists v \in V.R_v)]$

$IJ2.$ $\models R_v(s) \supset (g_{\delta(v)}(s) \neq \phi)$

$IJ3.$ $\models [R_v(s) \wedge s' \in g_i(s)] \supset [\psi(s') \vee \exists u(u \preceq v).R_u(s')]$ for $i = 1, \ldots, m$

$IJ4.$ $\models [R_v(s) \wedge s' \in g_{\delta(v)}(s)] \supset [\psi(s') \vee \exists u(u \prec v).R_u(s')]$

Then we may conclude that

$$\mathcal{J}(P) \models \varphi \supset \Diamond \psi.$$

A stronger conclusion is:

$$\mathcal{J}(P) \models \varphi \supset (\exists v.R_v)\ \mathcal{U}\ \psi.$$

Requirements $IJ1$–$IJ4$ resemble very closely $J1$-$J4$ and fulfill similar roles. There is no need for a counterpart to $J5$ since if s satisfies $R_v(s)$, s' $\in g_i($ s) and also $R_v(s')$ then the decreasing process for s, being determined by v alone, is also the decreasing process for s'. The proof method IJ appeared first in a structured form, applied to nondeterministic programs ([GFMR]).

The similarity between the methods suggest that they are in fact equivalent. Indeed we make the following claim:

Method J is applicable if and only if method IJ is applicable.

Proof:

Assume first that method J is applicable. This means that we have found Q, $(W, \succ)$, u and h satisfying requirements $J1$ to $J5$. To show that this implies the applicability of IJ we choose as follows:

The well-founded structure $(V, \succ_V)$ is given by $V = W \times [1, \ldots, m]$, where

$(w_1, i) \succ_V (w_2, j) \Leftrightarrow w_1 \succ_W w_2$ *or* $(w_1 = w_2$ *and* $i > j)$.

Thus, an element of V is a pair (w, i) with $w \in W$ and $1 \leq i \leq m$, and the ordering $\succ_V$ is the lexicographic ordering induced by the ordering on W and on the natural numbers.

$R_{(w,i)}(s)$ is defined by $Q(s) \wedge [u(s) = w] \wedge [h(s) = i]$

and

$$\delta(w, i) = i.$$

It is an easy matter to verify the fulfilment of requirements $IJ1$ to $IJ4$. Consider for example the verification of condition $IJ3$.

Let s, s' be two states such that $R_{(w,j)}(s)$ holds and s' $\in g_i(s)$. By the definition of R we know that Q(s) is true and $u(s)$ = w, $h(s)$ = j. By J3 either $\psi(s')$ is true which immediately satisfies $IJ3$, or $Q(s')$ holds and $w = u(s) \succeq u(s') = w'$. Thus, by the definition of R, $R_{(w',h(s'))}(s')$ is true. It remains to show that $(w, j) = (w, h(s)) \succeq (w', h(s'))$. If $w \succ w'$ then this is certainly the case. Consider therefore the possibility that w = w'. But then by $J5$ also h(s) = $h(s')$ leading to $(w, h(s)) = (w', h(s'))$ as required.

To go in the other direction assume that $(V, \succ), R_v$ and δ as required for method IJ have been found. We will show how to select Q, (W, $\succ$), u and h that will satisfy the requirements of method J.

For simplicity we assume that the order $\succ$ is a total (linear) order. We may then take the well-founded structure $(V, \succ)$ to be $(W, \succ)$. $Q(s)$ is defined by $\exists v.R_v(s)$ and $u(s)$ is given by $min\{v|R_v(s)\}$ for an s which satisfies Q and arbitrarily otherwise. If W is a total well-founded order every non empty subset of W has a minimal element which is smaller than any other element of the set. The helpful function $h(s)$ is defined as $\delta(u(s))$.

It is an easy matter to verify that Q, u, and h satisfy requirements J1 to $J5$. ∎

DIAGRAM REPRESENTATION OF THE INDEXING METHOD

In the case that the indexing set V is finite there is a convenient graph representation of the indexing method. This is certainly the case when the program P has only finitely many possible states.

In the graph or diagram representation there is a node n_v for each $R_v, v \in V$. Without loss of generality we may assume V to be an initial segment of the natural numbers $V = \{1, 2, \ldots, k\}$. Thus we have nodes $n_i, i = 1, \ldots, k$. A special node n_0, represents ψ. For every s $\in R_i$, s' $\in R_j$ (i.e. $R_i(s) = R_j(s') = true$) such that $s' \in g_\ell(s)$, we draw an edge e from n_i to n_j. The edge e is labelled by P_ℓ, the process effecting the transition. Similarly, for every s $\in R_i$, s' $\in \psi$ such that s' $\in g_\ell(s)$ we draw an edge from n_i to n_0 and label it by P_ℓ.

In order for a diagram to represent a valid proof by method IJ the following conditions must hold:

A. For every edge connecting n_i to n_j we must have $i \geq j$.

B. For every n_i, $i > 0$, there must exist some P_ℓ (the helpful process) such that all edges labelled by P_ℓ lead from n_i to some n_j with $i > j$ and such that P_ℓ is enabled on all states s $\in R_i$.

In the diagram we represent edges corresponding to the helpful process by double arrows $\Rightarrow$.

We illustrate diagram proofs by two additional examples.

Example B (The Peterson-Fischer Algorithm (*PF*) -- a distributed solution of the mutual exclusion problem):

$$y_1 := t_1 := y_2 := t_2 := \perp$$

ℓ_0: noncritical section 1	m_0: noncritical section 2
ℓ_1 : $t_1 := $ *if* $y_2 = F$ *then* F *else* T	m_1: $t_2 :=$ *if* $y_1 = T$ *then* F *else* T
ℓ_2 : $y_1 := t_1$	m_2: $y_2 := t_2$
ℓ_3: *if* $y_2 \neq \perp$ *then* $t_1 := y_2$	m_3: *if* $y_1 \neq \perp$ *then* $t_2 := \neg y_1$
ℓ_4 : $y_1 := t_1$	m_4 : $y_2 := t_2$
ℓ_5: *loop while* $y_1 = y_2$	m_5: *loop while* $\neg y_2 = y_1$
critical section 1 $(y_1, t_1) := (\perp, \perp)$	critical section 2 $(y_2, t_2) := (\perp, \perp)$
ℓ_7: *go to* ℓ_0	m_7: *go to* m_0
$-P_1-$	$-P_2-$

This program provides a distributed solution for achieving mutual exclusion without semaphores; the boxed segments are the critical sections to which we wish to provide exclusive access. It is assumed that both critical and noncritical sections do not modify the variables y_1 and y_2. Also, it is mandatory that the critical section itself must terminate. The program is *distributed* in the sense that each process P_i has its-own memory y_i which is readable by the other but writable only by itself.

The basic idea of the protection mechanism of this program is that when competing for the access rights to their critical sections, P_1 attempts to make $y_1 = y_2$ by the statements ℓ_1 to ℓ_4 while P_2 attempts to make $y_2 = \neg y_1$ in statements m_1 to m_4. The synchronization variables y_1 and y_2 range over the set {I, *F*, *T*}, where $\perp$ signifies no interest in entering the critical section. The partial operator $\neg$ is defined by

$$\neg T = F, \neg F = T, \quad \neg\perp \text{ is undefined.}$$

Hence in writing $\neg y_2 = y_1$ we also imply that $y_1 \neq \perp$ and $y_2 \neq \perp$. Protection is assured essentially by the exclusion of the entry conditions $y_1 \neq y_2$ and $\neg y_2 \neq y_1$ when both y_1 and y_2 are different from $\perp$, since $y_i \neq \perp$ when P_i is waiting to enter its critical section.

A point unique to this algorithm is that although P_1 attempts to establish the condition $y_1 = y_2$ in ℓ_1 to ℓ_4, the condition for P_1 actually entering the critical section is the complementary condition $y_1 \neq y_2$. Thus, if both processes actively compete for entry, P_1 sets y_1 equal to y_2 and then waits for the other process to set y_2 to a value different from y_1. If P_2 is not currently interested in gaining access to the critical section, then $y_2 = \perp$ which will cause the statements in ℓ_1 to ℓ_4 to set y_1 to T; testing at ℓ_5, P_1 will find that indeed $y_1 = T \neq y_2 = \perp$ and enter immediately.

By simple application of the invariance principle it is possible to derive the following invariants:

$$\models (t_1 \neq \perp) \equiv at\,\ell_{2..6}$$

$$\models (y_1 \neq \perp) \equiv at\,\ell_{3..6}$$

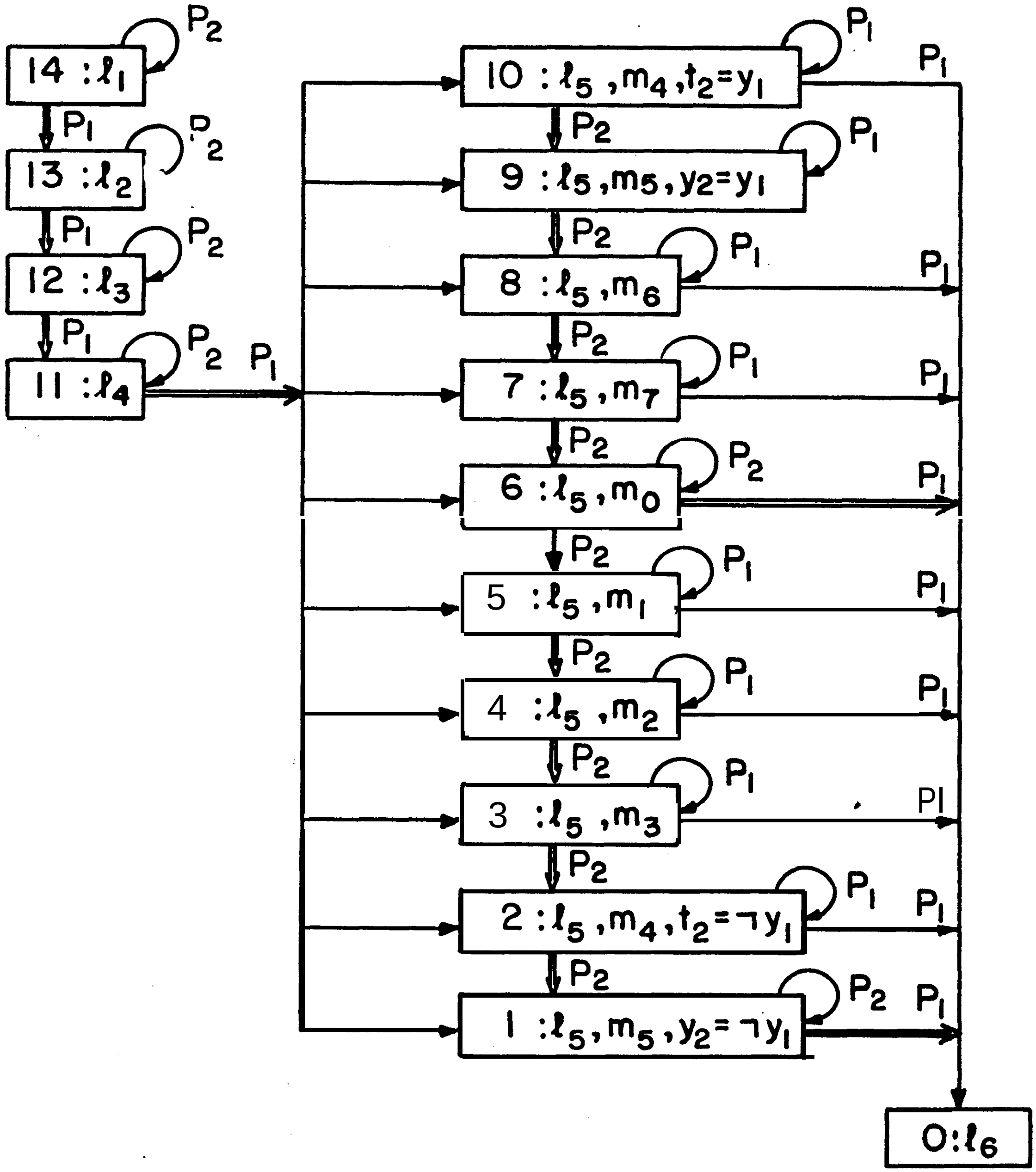


Figure I.

Diogrom Proof for PF

$$\models \ (t_2 \neq \perp) \ \equiv \ at\, m_{2..6}$$

$$\models \ (y_2 \neq \perp) \ \equiv \ at\, m_{3..6},$$

where $at\,\ell_{2..6}$ stands for $at\,\ell_2 \vee at\,\ell_3 \vee \ldots \vee at\,\ell_6$, etc.

The eventuality property we wish to show for this program is

$$\models \ at\,\ell_1 \ \supset \ \Diamond\, at\,\ell_6.$$

In Figure 1 we present a diagram proof for this property. In constructing the diagram we have freely used the four invariants derived above. Observe in particular node number 6

$$6: \ \ \ell_5, m_0$$

in which the helpful process (indicated by a double arrow $\Rightarrow$) is P_1 since we know that $y_2 = \perp$. In this diagram we abbreviate $at\,\ell_5 \wedge at\ m_0$ to ℓ_5, m_0.

To illustrate the application of method *IJ* to the proof of *until* properties, consider the following precedence property:

$$\models [at\,\ell_5 \ \wedge \ \sim \ at\,m_{4..6}] \supset [(\sim \ at\,m_6) \ \mathcal{U} \ (at\,\ell_6)].$$

It states that if P_1 arrived at ℓ_5 before P_2 arrived at any location in $\{m_4, m_5, m_6\}$ then P_1 will be admitted first to its critical section. To prove this we only have to consider the subdiagram consisting of nodes 0 to 7. Certainly,

$$[at\,\ell_5 \ \wedge \ \sim \ at\,m_{4..6}] \supset \ \ [R_7 \vee R_6 \vee R_5 \vee R_4 \vee R_3].$$

Therefore this is an admissible diagram in the sense that condition $IJ1$ is satisfied. It establishes that $at\,\ell_6$ will eventually be realized and all the intermediate states are covered by $\bigvee_{i=1}^{7} R_i$ which implies $\sim at\,m_6$. ∎

Example C (The Dekker program (DK) – a shared variable solution of the mutual exclusion problem):

$$t := 1, \ \ y_1 := y_2 := F$$

ℓ_0 : noncritical section 1	m_0 : noncritical section 2
ℓ_1 : $y_1 := T$	m_1 : $y_2 := T$
ℓ_2 : *if* $y_2 = F$ *then go to* ℓ_7	m_2 : *if* $y_1 = F$ *then go to* m_7
ℓ_3 : *if* $t = 1$ *then go to* ℓ_2	m_3 : *if* $t = 2$ *then go to* m_2
ℓ_4 : $y_1 := F$	m_4 : $y_2 := F$
ℓ_5 : *loop until* $t = 1$	m_5 : *loop until* $t = 2$
ℓ_6 : *go to* ℓ_1	m_6 : *go to* m_1
ℓ_7 : critical section 1	m_7 : critical section 2
$t := 2$	$t := 1$
ℓ_8 : $y_1 := F$	m_8 : $y_2 := F$
ℓ_9 : *go to* ℓ_0	m_9 : *go to* m_0
$-P_1-$	$-P_2-$

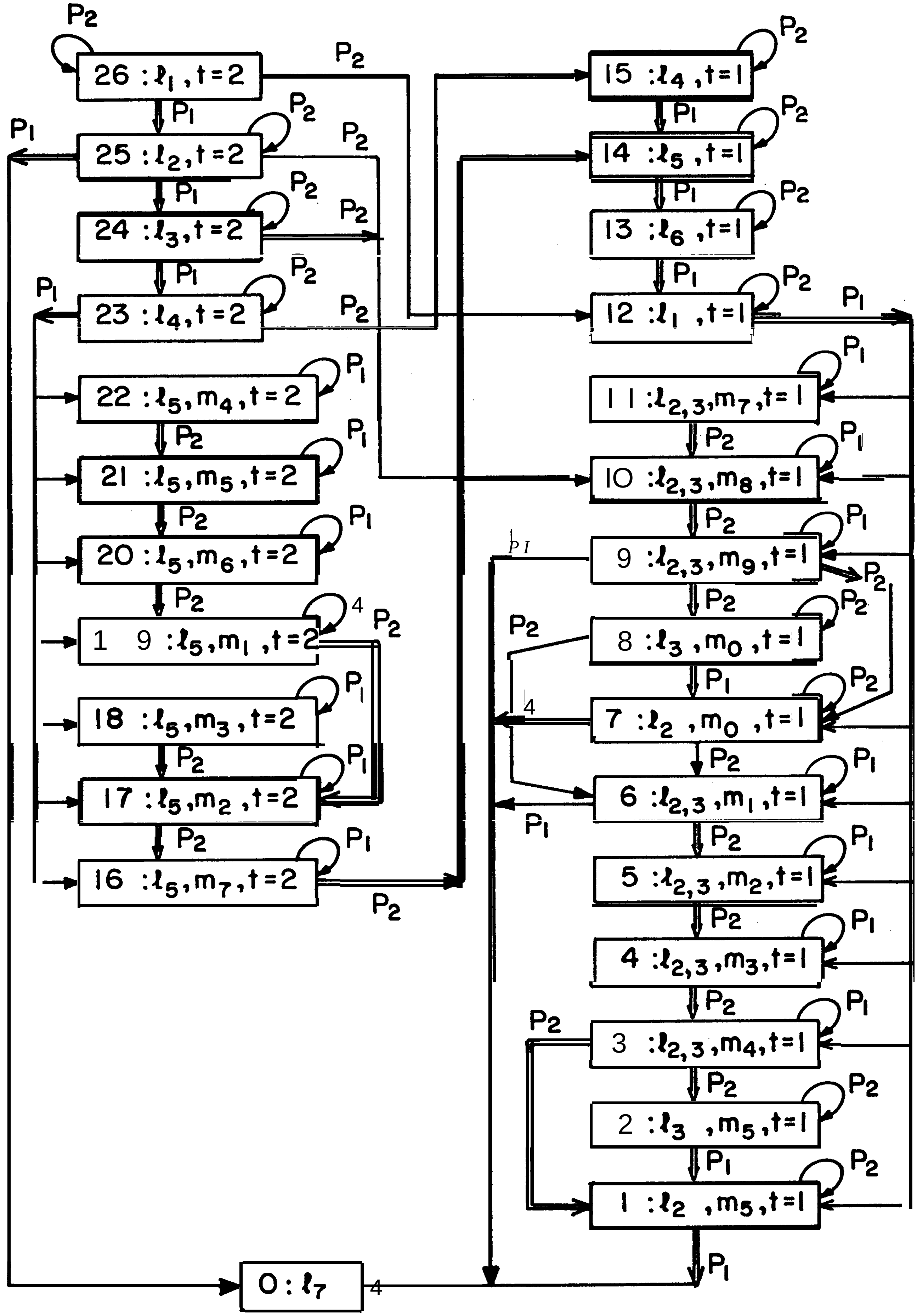


Figure 2.
Diagram Proof of the Program DK

The variable y_1 in process P_1 (and y_2 in P_2 respectively) is set to T at ℓ_1 to signal the intention of P_1 to enter its critical section at ℓ_7. Next P_1 tests at ℓ_2 whether P_2 has any interest in entering its own critical section. This is tested by checking if $y_2 = T$. If $y_2 = F$, P_1 proceeds immediately to its critical section. If $y_2 = T$ we have a competition between the two processes on the access right to their critical sections. This competition is resolved by using the variable t (turn) that has the value 1 if in case of conflict P_1 has the higher priority and the value 2 if P_2 has the higher priority. If P_1 finds that $t = 1$ it knows it is its turn to insist and it leaves y_1 on and just loops between ℓ_2 and ℓ_3 waiting for y_2 to drop to F. If it finds that $t = 2$ it realizes it should yield to P_2 and consequently it turns y_1 off and enters a waiting loop at ℓ_5, waiting for t to change to 1. As soon as P_2 exits its critical section it will reset t to 1 so P_1 will not be waiting forever. Once t has been detected to be 1, P_1 sets y_1 to T and returns to the active competition at ℓ_2.

For the DK program we wish to show:

$$\models \; at\,\ell_1 \;\supset\; \Diamond\, at\,\ell_7.$$

In Figure 2 we present a diagram proof of this property. In constructing the proof we made use of some invariants that are easily derivable, namely:

$$\models \quad (y_1 = T) \equiv (at\,\ell_{2..4} \lor at\,\ell_{7,8})$$

$$\models \quad (y_2 = T) \equiv (at\,m_{2..4} \lor at\,m_{7,8})$$

$$\models \quad (at\,\ell_{3..6} \land t = 2) \supset at\,m_{1\ldots7}.$$

For example, we used the last invariant in order to decide that at node 23 the P_1 successors to states in which $at\,\ell_4 \land (t = 2)$ may be anywhere but at m_0, m_8 or m_9.

Again we may use the extension of the method in order to prove some precedence properties of this program. First we can show:

$$\models [at\,\ell_{2,3} \land (t = 1) \land \sim at\,m_7] \supset [(\sim at\,m_7)\,\mathcal{U}\,(at\,\ell_7)].$$

This is established by considering the subdiagram formed out of nodes n_0 to n_{10}. It ensures that once P_1 is in $\ell_{2,3}$ with $t = 1$, it will precede P_2 in getting to the critical section. An almost trivial observation is that

$$\models \; at\,m_8 \supset [(t = 1)\,\mathcal{U}\,(at\,\ell_7)].$$

In **analyzing** the amount of overtaking by which P_2 can precede P_1 in entering the critical section we find the following:

Once P_1 is in ℓ_1 it will eventually get to ℓ_2. If currently $t = 1$, then the next process to enter its critical section is P_1. Otherwise, in the worst case P_1 proceeds from ℓ_2 to ℓ_5. P_2 cannot enter its critical section more than once without setting t to 1. Once $t = 1$, P_1 returns to ℓ_2 ensuring its priority on the entrance rights to the critical section. A certain amount of overtaking, i.e., P_2 entering its critical section several times before P_1, may take place during the transition of P_1 from ℓ_5 to ℓ_2. ∎

PROGRAMS WITH SEMAPHORES -- FAIR COMPUTATIONS

Next we will consider programs with semaphore instructions. For such programs the classes of just and fair computations do not coincide and we have to go back to consider the more general concept of fair computations. Since always $\mathcal{F}(P) \subseteq \mathcal{J}(P)$, any property that has been proved correct by method J certainly holds for all fair computations. However, the completeness of method J breaks down in the case of programs with semaphores; we are not always guaranteed that method J is applicable.

Hence, we propose a more general method for establishing eventuality properties under *fair computations:*

***Proof** Method **F** :*

For proving eventualities of the form $\varphi \supset \Diamond\psi$, under all fair computations of a program P, find a state predicate Q, a well-founded structure $(W, \succ)$, a convergence function $u : S \rightarrow W$ and a helpfulness function $h : S \rightarrow \{1, \ldots, m\}$ such that:
$F1.\quad \models \varphi \supset (\psi \vee Q)$
$F2.\quad \mathcal{F}(P - \{P_k\}) \models [Q(s) \wedge h(s) = k] \supset \Diamond[\psi \vee (g_k(s) \neq \phi)]$ for $k = 1, \ldots, m$
$F3.\quad \models [Q(s) \wedge s' \in g_i(s)] \supset [\psi(s') \vee (Q(s') \wedge (u(s) \succeq u(s')))]$ for $i = 1, \ldots, m$
$F4.\quad \models [Q(s) \wedge s' \in g_{h(s)}(s)] \supset [\psi(s') \vee (u(s) \succ u(s'))]$
$F5.\quad \models [Q(s) \wedge s' \in g_i(s) \wedge (u(s) = u(s'))] \supset [\psi(s') \vee (h(s) = h(s'))]$ for $i = 1, \ldots, m$.
Then we may conclude that $$\mathcal{F}(P) \models \varphi \supset \Diamond\psi.$$
A stronger conclusion is: $$\mathcal{F}(P) \models \varphi \supset (Q \,\mathcal{U}\, \psi).$$

The requirement imposed by $F2$ is that under all fair computations of $P - \{P_k\}$, i.e., the program consisting of all processes excluding P_k, if Q(s) holds and the helpful process is k then eventually either ψ will be realized or g_k becomes enabled.

The difference between method F and method J is in the second requirement $F2$. While ***J2*** requires that the helpful process is enabled *now,* $F2$ only assures that it will be *eventually* enabled. The apparent disadvantage of $F2$ in comparison with ***J2*** is that while ***J2*** (and all the other requirements) are static, requiring only classical reasoning for their establishment, $F2$ is a temporal requirement, having the same form as the conclusion we set out to prove: $\varphi \supset \Diamond \psi$. Two obvious questions arise: how do we prove $F2$, and is there a danger of circular reasoning?

The answer to both questions lies in the prefix to the $\models$ sign. Since our goal predicate in $F2$ is $g_k(s) \neq \phi$ which expresses the fact that P_k is enabled, we may omit from our considerations any action of P_k, because such an action may bc taken only when P_k is enabled, i.e., from a goal state. Thus we can consider fair computations in which all the processes but P_k participate and show that they eventually get to a state in which P_k is enabled. Consequently, *we* can study a *simpler* program with one process less. The answer to the question of how to verify clause $F2$ is therefore recursively by method F, but applied to a simpler program in which P_k is omitted.

To justify method F consider a fair computation:

$$\sigma : \quad s_0 \xrightarrow{P_{i_1}} s_1 \xrightarrow{P_{i_2}} s_2 \ . \ . \ . \ ,$$

such that $\varphi(s_0)$ is true and ψ is never realized. By $F1$ and F3, $Q(s_i)$ must be true for every s_i in the sequence. By $F2$ the sequence must be infinite, since it implies that either already $g_k(s_i) \neq \phi$ and the sequence cannot stop there, or that there exists a future state s_j for which $\psi \vee (g_k(s_j) \neq \phi)$. Consequently s_i cannot be terminal. By $F3$ the sequence of values $u(s_1), u(s_2), \ . \ . \ .$ satisfies $u(s_1) \succeq u(s_2) \succeq \ . \ . \ .$ and by being well-founded it must eventually stabilize, let us say at s_r, i.e.,

$$u(s_r) = u(s_{r+1}) = \ . \ . \ . \ .$$

From *F5* this implies a constant value of the *h* function as well, i.e.,

$$h(s_r) = h(s_{r+1}) = \ . \ . \ . = k.$$

Since the u value is constant beyond s_r, P_k by $F4$ could not have been activated. Thus the suffix sequence

$$s_r, \ s_{r+1}, \ \ldots$$

is a fair computation of $P - \{P_k\}$. By $F2, P_k$ must be enabled somewhere in it. By considering higher suffixes wc can establish that g_k is enabled an infinite number of times but never activated. Thus σ must be unfair. ∎

In [LPS] it is proved that method F is complete for proving eventuality properties for the class of all fair computations of a program.

AN INDEXING METHOD FOR FAIR COMPUTATIONS

Similarly to the case of **just** computations we can present a well-founded indexing variation of the pri nciplc proposed above.

Proof Method IF:

For proving eventualities of the form $\varphi \supset \Diamond\psi$, under all fair computations of a prograrn P, find a well-founded structure $(V, \succ)$, an indexed family of predicates $R_v = R_v(s)$, $v \in V$, and a *decrease function* $\delta : V \rightarrow \{1, \ldots, m\}$ such that

$IF1.\ \models \varphi \supset [\psi \vee \exists v(v \in V).R_v]$

$IF2.\ \mathcal{F}(P - \{P_{\delta(v)}\}) \models R_v(s) \supset \Diamond[\psi \vee (g_{\delta(v)}(s) \neq \phi)]$

$IF3.\ [R_v(s) \wedge s' \in g_i(s)] \supset [\psi(s') \vee \exists u(u \preceq v).R_u(s')]$ for $i = 1, \ldots, m$

$IF4.\ [R_v(s) \wedge s' \in g_{\delta(v)}(s)] \supset [\psi(s') \vee \exists u(u \prec v).R_u(s')]$.

Then we may conclude that

$$\mathcal{F}(P) \models \varphi \supset \Diamond\psi,$$

A stronger conclusion is:

$$\mathcal{F}(P) \models \varphi \supset (\exists v.R_v)\,\mathcal{U}\,\psi.$$

Sirnilarly to the previous case we can establish the equivalence between this method and the one based on convergence functions. This variation lends itself easily to a diagram representation in the finite state case.

We will proceed to illustrate the application of method F to proofs of eventuality properties of programs with sernaphores.

Example D (Program CP — consumer-producer):

$$b := \Lambda,\quad s := 1,\ cf := 0,\quad ce := N$$

$-P_1$: *Producer* $-$	$-P_2$: *Consumer* $-$
ℓ_0 : *compute* y_1	m_0 : $request(cf)$
ℓ_1 : $request(ce)$	m_1 : $request(s)$
ℓ_2 : $request(s)$	m_2 : $y_2 := head(b)$
ℓ_3 : $t_1 := b \cdot y_1$	m_3 : $t_2 := tail(b)$
ℓ_4 : $b := t_1$	m_4 : $b := t_2$
ℓ_5 : $release(s)$	m_5 : $release(s)$
ℓ_6 : $release(cf)$	m_6 : $release(ce)$
ℓ_7 : **go to** ℓ_0	m_7 : *compute using* y_2
	m_8 : **go to** m_0

The producer P_1 computes at ℓ_0 a value into y_1 without modifying any other shared program variables. It then adds y_1 to the end of the buffer b. The consumer P_2 removes the first clement of the buffer into y_2 and then uses this value for its own purposes (at m_7) without modifying any other shared program variable The maximal capacity of the buffer b is $N > 0$.

In order to ensure the correct synchronization between the processes we use three semaphore variables: The variable s ensures that the accesses to the buffer are protected and provides exclusion between the critical sections $\ell_{3..5}$ and $m_{2..5}$. The variable ce ("count of empties") counts the number of free available slots in the buffer b. It protects b from overflowing. The variable cf ("count of fulls") counts how many items the buffer currently holds. It ensures that the consumer does not attempt to remove an item from an empty buffer.

Here we wish to show that

$$\models at\,\ell_1 \supset \Diamond\; at\,\ell_3.$$

We start by presenting a top-level diagram proof:

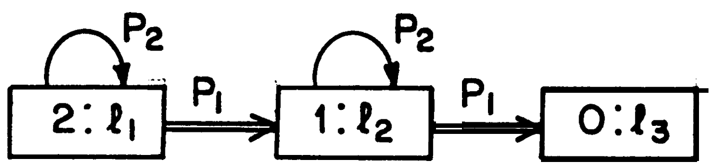


Figure 3.

This diagram proof is certainly trivial. Everywhere, P_1 is the helpful process and leads immediately to the next step. However, we now have to establish clause *IF2* in method *IF*. This calls for the consideration of fair computations of $P - \{P_1\} = P_2$. We thus have to conduct two subproofs:

$$\mathcal{F}(P_2) \models at\,\ell_1 \supset \Diamond(ce > 0)$$

$$\mathcal{F}(P_2) \models at\,\ell_2 \supset \Diamond(s > 0).$$

The first statement ensures that if P_1 is at ℓ_1, P_2 will eventually cause ce to become positive which is the enabling condition for P_1 to be activated at Cr. Similarly, in the second statement P_2 will eventually cause s to become positive, making P_1 enabled at ℓ_2. For both statements we will present diagram proofs.

Consider first the diagram proof for the *at* ℓ_1 case:

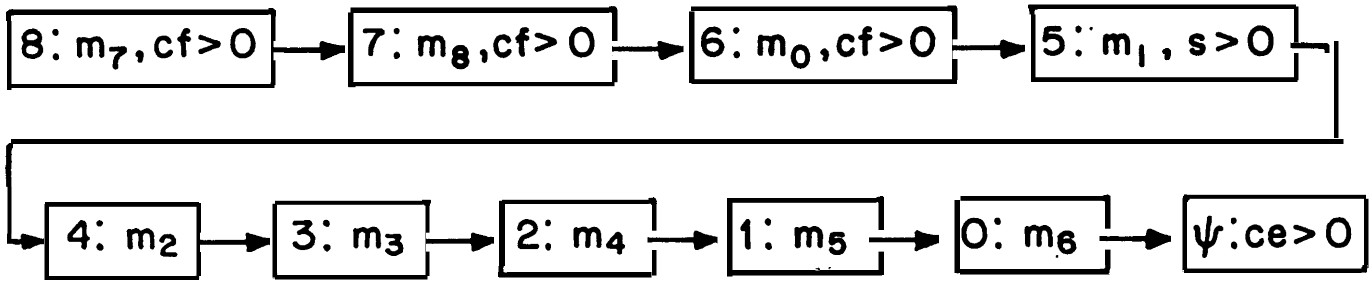


Figure 4.

In the construction of this diagram we use some invariants which are easy to derive. For example, we used

$$at\,\ell_{3..5} + at\,m_{2..6} + s = 1$$

in order to derive that being at ℓ_1 and at m_1 implies $s > 0$. In an expression such as the above we arithmetize propositions by interpreting *false* as 0 and *true* as 1. As another invariant we use

$$cf + ce + at\,\ell_{2..6} + at\,m_{1..6} = N$$

in order to deduce that being at ℓ_1 and at $m_{7,8,0}$ implies that either $ce > 0$ or $cf > 0$.

The diagram proof for ℓ_2 is even simpler:

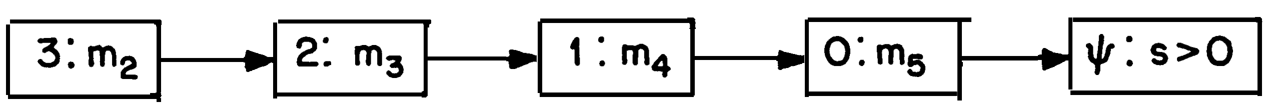


Figure 5.

Example E (Program BC - a distributed computation of the binomial coefficient):

$$y_1 := n, \quad y_2 := 0, \quad y_3 := 1, \quad y_4 := 1$$

ℓ_0 : *if* $y_1 = (n-k)$ *then go to* ℓ_e	m_0 : *if* $y_2 = k$ *then go to* m_e
ℓ_1 : *request*(y_4)	m_1 : $y_2 := y_2 + 1$
$\boxed{\ell_2}$: $t_1 := y_3 \cdot y_1$	m_2 : *loop until* $y_1 + y_2 \leq n$
ℓ_3 : $y_3 := t_1$	m_3 : *request*(y_4)
ℓ_4 : *release*(y_4)	m_4 : $t_2 := y_3 / y_2$
ℓ_5 : $y_1 := y_1 - 1$	m_5 : $y_3 := t_2$
ℓ_6 : *go to* ℓ_0	m_6 : *release*(y_4)
ℓ_e : *halt*	m_7 : *go to* m_0
	m_e : *halt*
$-P_1-$	$-P_2-$

This program computes the binomial coefficient $\binom{n}{k}$ for integers n and k such that $0 \leq k \leq n$. Based on the formula

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{1 \cdot 2 \cdot \ldots \cdot k}$$

process P_1 successively multiplies y_3 by $n, (n-1), \ldots$, while P_2 successively divides y_3 by $1, 2, \ldots$. In order for the division at m_4 to come out evenly, we divide y_3 by y_2 only when at least y_2 factors have been multiplied into y_3 by P_1. The waiting loop at m_2 ensures this.

Without loss of generality we can relabel the instructions in the program, as follows:

Program BC^* - *A relabelled version of the Binomial Coefficient Program;*

$$y_1 := n, \quad y_2 := 0, \quad y_3 := 1, \quad y_4 := 1$$

ℓ_7 : *if* $y_1 = (n-k)$ *then go to* ℓ_1	m_3 : *if* $y_2 = k$ *then go to* m_1
ℓ_6 : *request*(y_4)	m_2 : $y_2 := y_2 + 1$
ℓ_5 : $t_1 := y_3 \cdot y_1$	m_9 : *loop until* $y_1 + y_2 \leq n$
ℓ_4 : $y_3 := t_1$	m_8 : *request*(y_4)
ℓ_3 : *release*(y_4)	m_7 : $t_2 := y_3 / y_2$
ℓ_2 : $y_1 := y_1 - 1$	m_6 : $y_3 := t_2$
ℓ_8 : *go to* ℓ_7	m_5 : *release*(y_4)
ℓ_1 : *halt*	m_4 : *go to* m_3
	m_1 : *halt*
$-P_1-$	$-P_2-$

Here we wish to prove:

$$\models [at\{\ell_7, m_3\} \wedge (y_1, y_2, y_3, y_4) = (n, 0, 1, 1)] \supset \Diamond\, at\{\ell_1, m_1\}.$$

We apply method F with the following:

$$Q : [\, at\,\ell_{3..5} + at\,m_{5..7} + y_4 = 1]$$
$$\wedge\ [((n-k) + at\,\ell_{2..6}) \leq y_1 \leq n]$$
$$\wedge\ [0 \leq y_2 \leq (k - at\,m_2)]$$
$$\wedge\ [at\,\ell_1 \supset (y_1 = n-k)]$$

$(W, \succ)$: $\langle N \times N, >_{lex}\rangle$
the lexicographically ordered domain of pairs of nonnegative integers

$u(\ell_i, m_j; y_1, y_2)$: $(y_1 + k - y_2, i + j)$

$h(\overline{\pi}, \overline{y})$: *if* $at\,\ell_1$ *then* P_2 *else* P_1

Obviously the label sequence was designed in such a way that every step that moves to the next instruction will necessarily decrement u. This is so because the label sequence is always decreasing except for the instructions which decrement y_1 and increment y_2. Changes in the y's have been given the highest priority in the lexicographical ordering.

There are only two situations to be checked. First, when P_1 is at ℓ_1 and P_2 is at m_9 *we* have to show that the next step indeed decrements u. This is so because in such a situation we are assured by Q that both $y_2 \leq k$ and $y_1 = n - k$ hold, leading to $y_1 + y_2 \leq n$, which means that the next step leads to m_8. Another point is to show that being at ℓ_6 guarantees that eventually y_4 will become positive, by the actions of P_2 alone. This is easily established by the following diagram, supported by Q.

$$\boxed{2: m_7} \longrightarrow \boxed{1: m_6} \longrightarrow \boxed{0: m_5} \longrightarrow \boxed{\psi: y_4 > 0}$$

Figure 6.

CONCLUDING REMARKS

When compared with the chain reasoning approach, the convergence function approach appears to provide a more concise representation of a finished proof of an eventuality property. However it may at times reveal less intuitive insight into the reasons the program is correct and certainly offers very little guidance for the design of correct programs. According to whether one is interested in a post analysis or a proof-guided synthesis of programs, one approach should be preferred to the other.

The methods described here extend and elaborate the methods for proving convergence suggested in [LPS]. It is possible to prove completeness of the methods proposed here by an appropriate extension of the completeness proof presented in [LPS].

Closely related approaches but concentrating on nondeterministic rather than concurrent programs are described in [RO] and [GFMR].

ACKNOWLEDGEMENT

We wish to thank Ed Ashcroft, Andrei Broder, Chris Goad, Gabi Kuper, Yoni Malachi, Yoram Moses, Ben Moszkowski, Tmima Olshanski-Koren, ltivi Sherman, Pierre Wolper, and Frank Yellin for careful and critical reading of the manuscript.

REFERENCES

[AO] Apt, K. R., and E. R. Olderog, "Proof rules dealing with fairness," in *Logics of Programs* (D. Kozen, ed.), Lecture Notes in Computer Science 131, Springer Verlag, 1982, pp. 1-8.

[DM] Dershowitz, N., and Z. Manna, "Proving termination with multiset orderings," CACM, Vol. 22, No. 8 (August 1979), pp. 465-476.

[GFMR] Grumberg, O., N. Francez, J. A. Makowsky, and W. P. deRoever, "A proof rule for fair termination of guarded commands," Computer Science Report, Technion, Haifa, 1981.

[K] Keller, R. M., "Formal verification of parallel programs," CACM, Vol. 19, No. 7 (July 1976), pp. 371-384.

[LPS] Lehmann, D., A. Pnueli, and J. Stavi, "Impartiality, justice and fairness: the ethics of concurrent termination," in *Automata Languages and Programming*, Lecture Notes in Computer Science 115, Springer Verlag, 1981, pp. 264-277.

[M] Manna, Z., *Mathematical Theory of Computation*, McGraw Hill, 1974.

[MP1] Manna, Z. and A. Pnueli, "Verification of concurrent programs: The temporal framework," in *The Correctness Problem in Computer Science* (R. S. Boyer and J S. Moore, eds.), International Lecture Series in Computer Science, Academic Press, London, 1982, pp. 215-273. Also, Computer Science Report, Stanford University, Stanford, CA (June 1981).

[MP2] Manna, Z. and A. Pnueli, "Verification of concurrent programs: Temporal proof principles," in *Logic of Programs*, (D. Kozen, ed.), Lecture Notes in Computer Science 131, Springer Verlag, 1982, pp. 200-252. Also, Computer Science Report, Stanford University, Stanford, CA (September 1981).

[MP3] Manna, Z. and A. Pnueli, "Verification of concurren t programs: Precedence proper ties," Computer Science Report, Stanford University, Stanford, CA (forthcoming).

[OG] Owicki, S. and D. Gries, "An axiomatic proof technique for parallel programs," Acta Informatica, Vol. 6, No. 4 (1976), pp. 319-340.

[OL] Owicki, S. and L. Lamport, "Proving liveness properties of concurrent programs," SRI International, unpublished report (October 1980).

[Pa] Park, D., "On the scrnnntics of fair parallclism," in *Abstract Software Specifications* (D. Bjorner, cd.), Lecture Notes in Computer Science 86, Springer Vcrlag, 1980, pp. 504–526.

[Pn] Pnueli, A., "The temporal logic of programs," Proc. 18th Syrnposium on Foundations of Computer Science, I'rovidencc, RI (November 1977), pp. 46-57.