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
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 are applicable to both "just" and "fair" computations. These 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. We 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:

\[ \varphi \supset \Diamond \psi, \]

(i.e., if \( \varphi \) ever arises it must be followed by \( \psi \)), as well as “until” properties of the form

\[ \varphi \supset (\chi \mathbf{U} \psi), \]

(i.e., if \( \varphi \) ever arises it must be followed by an instant at which \( \psi \) is realized and between the occurrences of \( \varphi \) and \( \psi, \chi \) must hold continuously).

With respect to uniformity, the convergence 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 \((\mathbb{N}, \succ)\), where \(\mathbb{N}\) is the set of all nonnegative integers, and \(\succ\) is the usual “greater than” ordering. Obviously we cannot have an infinitely decreasing sequence of nonnegative integers, and therefore \((\mathbb{N}, \succ)\) 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 \((W_1 \times W_2, \succ_{\text{lex}})\) where the lexicographic ordering \(\succ_{\text{lex}}\) is defined by

\[
(m_1, m_2) \succ_{\text{lex}} (n_1, n_2) \iff (m_1 \succ_1 n_1) \text{ or } (m_1 = n_1 \text{ and } m_2 \succ_2 n_2)
\]

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)

\[
(y_1, y_2) := (x_1, x_2)
\]

\[
\ell_0: \quad \text{while } y_1 \neq y_2 \text{ do}
\]

\[
\text{if } y_1 > y_2 \text{ then } y_1 := y_1 - y_2
\]

\[
\ell_1: \quad \text{halt}
\]

\[
- \quad P_1
\]

\[
\ell_0: \quad \text{while } y_1 \neq y_2 \text{ do}
\]

\[
\text{if } y_1 < y_2 \text{ then } y_2 := y_2 - y_1
\]

\[
\ell_1: \quad \text{halt}
\]

\[
- \quad P_2
\]

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 \(\ell_0\), 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 \(\ell_0\), 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 emphatically that any well-founded argument may succeed only if it takes fairness into account.

PROGRAMS AND COMPUTATIONS

For completeness we repeat some of the definitions of [MP1] and introduce some additional notation required here. Let \( P \) be a program consisting of \( m \) parallel processes:

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

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

Let \( \ell, \ell^1, \ldots, \ell^k \in \mathcal{L}_i \) be locations in process \( P_i \):

We define \( E_\ell(\vec{y}) = c_1(\vec{y}) \lor \ldots \lor c_k(\vec{y}) \) to be the exit condition at node \( \ell \). Locations in the program can be classified according to their exit conditions.

- A location is regular if \( E_\ell \equiv \text{true} \). This is the case with locations such that the set of conditions labeling their outgoing transitions is exhaustive in the sense that for every possible value of \( \vec{y} \) at least one transition is enabled. The only irregular locations are terminal locations and semaphore locations discussed next.

- A location is terminal if \( E_\ell \equiv \text{false} \). This is the case with locations labeling halt instructions which have no outgoing transitions. In our model we usually label these locations by \( \ell_e \).

- Any location \( \ell \) such that the exit condition \( E_\ell(\vec{y}) \) is nontrivial is called a semaphore location. Examples of such locations are those corresponding to the instruction \( \text{request}(y_r) \) whose transition diagram is:

\[ \ell \quad (y_r > 0) \rightarrow [y_r := y_r - 1] \rightarrow \ell' \]
Note that $E_\ell(\bar{y}) = (y_r > 0)$. 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:

$$
\text{true} \rightarrow [y_r := y_r + 1] \rightarrow p'
$$

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

A state of the program $P$ is a tuple of the form $s = (\bar{l}; \bar{y})$ with $\bar{l} \in L_1 \times \ldots \times L_m$ and $\bar{y} \in D^n$, where $D$ is the domain over which the program variables $y_1, \ldots, y_n$ range. The vector $\bar{l}$ is the set of current locations which are next to be executed in each of the processes. The vector $\bar{y}$ is the set of current values assumed by the program variables 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 $g_i : S \rightarrow 2^S$.

Note 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 $s = (\bar{l}; \bar{y})$. If $\ell_i$ is a terminal location, or a semaphore location with $E_{\ell_i}(\bar{y}) = \text{false}$, then $P_i$ cannot be activated on $s$. In such a case $g_i(s) = \phi$ and we say that $P_i$ is disabled on $s$. If $\ell_i$ is a regular location, or a semaphore location with $E_{\ell_i}(\bar{y}) = \text{true}$ then $g_i(s) \neq \phi$ and we 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 terminated 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:

$$
\sigma : \quad s_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots
$$

such that every $s_j \in S$ and for every $j \geq 0$, we have $s_{j+1} \in g_i(s_j)$. Thus, such a computation could arise by an execution of the program starting 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 $s_0 = (\ell_0; \ldots, CT; f_0(\bar{y}))$. Here $\ell_0$ is the initial location in process $P_i$ and $f_0$ is the initial assignment to the program variables.

- A 'j-computation is a -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 \varphi-computation is a 'j-computation for any input values $\bar{y}$ satisfying a precondition $\varphi$.

The next definition embodies the basic assumption of fairness:
An admissible computation $\sigma$ is fair if there is no process $P_i$ such that $P_i$ is enabled an infinite number of times in $\sigma$, 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 requirement of being continuously enabled.

A computation $\sigma$ is just if there is no process $P_i$ such that $P_i$ is continuously enabled beyond a certain state $s$ in $\sigma$, 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 $\varphi$ by $F(\varphi, P)$, $J(\varphi, P)$ respectively, or $F(P)$, $J(P)$ when the precondition $\varphi$ is implicitly understood.

For an arbitrary program $P$ we have in general

$$F(P) \subset J(P),$$

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 $\sigma$ be a fair computation. Let $P_i$ be any process that is continuously enabled beyond a certain state in $\sigma$. Thus, $P_i$ is certainly enabled an infinite number of times, and by fairness must be activated an infinite number of times. Hence $\sigma$ is just.

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

$$y := 1$$

$$\ell_0: \text{request}(y) \quad m_0: \text{request}(y)$$

$$\ell_1: \text{release}(y) \quad m_1: \text{release}(y)$$

$$\ell_2: \text{go to } \ell_0 \quad m_2: \text{go to } m_0$$

$$\neg P_1 \quad \neg P_2$$

The following computation:

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

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 $\sigma$ 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 we 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: $F(P) = J(P)$.
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 state 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 \( \varphi \supset 0 \psi \), 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:

\( J1. \quad \equiv \varphi \supset (\psi \lor Q) \)

\( J2. \quad \equiv Q(s) \supset (g_{h(s)}(s) \neq \varphi) \)

\( J3. \quad \equiv [Q(s) \land s' \in g_i(s)] \supset (\psi(s') \lor (Q(s') \land (u(s) \succeq u(s')))) \)

for \( i = 1, \ldots, m \)

\( J4. \quad \equiv [Q(s) \land s' \in g_{h(s)}(s)] \supset (\psi(s') \lor (u(s) \succeq u(s')))) \)

\( J5. \quad \equiv [Q(s) \land s' \in g_i(s) \land (u(s) = u(s'))] \supset (\psi(s') \lor (h(s) = h(s')))) \)

for \( i = 1, \ldots, m. \)

Then we may conclude that:

\( J(P) \vdash \varphi \supset \bigcirc \psi. \)

Here \( \vdash \) means that \( w \) is true for all computations of \( P \). The statement \( J(P) \vdash w \) 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 \( \varphi \) becomes true until \( \psi \) is realized. Requirement \( J_1 \) states that if \( \varphi \) holds for a state then either \( \psi \) or \( Q \) must hold in this state. \( J_2 \) requires that the process that is helpful for a state \( s \) be enabled on \( s \). \( J_3 \) states that each step in the computation either realizes \( \psi \) or preserves \( Q \) and produces a value of \( u \) that is not higher than the value before the step. \( J_4 \) states that taking a helpful step actually decreases the value of \( u \). \( J_5 \) 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 \( \psi \).

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:

\[
\sigma : s_0 \rightarrow P_{i_1} s_1 \rightarrow P_{i_2} s_2 \rightarrow \ldots
\]
such that \( \varphi(s_0) \) is true and \( \psi \) is nowhere realized. By \( J_1 \) and \( J_3, Q(s_i) \) must be true for every \( s_i \) in the sequence. By \( J_2 \) the sequence must be infinite since, for every \( s_i, P_{h(s_i)} \) is enabled. Again by \( J_3 \) the sequence of \( u \) values \( u(s_0) \geq u(s_1) \geq \ldots \) 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 \( J_5 \), \( h \) 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 \( J_4 \), \( P_r \) was never activated beyond \( s_k \) because its activation would have caused \( u \) to decrease. In view of \( J_2, P_r \) is continuously enabled beyond \( s_k \) since everywhere \( h(s_i) = r \) for \( i > k \). This is obviously a blatant case of injustice \(- P_r \) being continuously enabled and never activated. Thus, just sequences failing to realize \( \psi \) cannot exist, and any just sequence initialized with \( \varphi \) must eventually realize \( \psi \). □

By looking at the proof for eventualities we observe that it guarantees the eventual realization of \( \psi \) and, by \( J_1 \) and \( J_3 \), as long as \( \psi \) is not realized, \( Q \) holds. This is exactly the definition of the until expression \( Q \cup \psi \). We therefore have:

Corollary: The proof method \( J \) also proves

\[
J(P) \vdash \varphi \supset (Q \cup \psi).
\]

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 \( Q, W, u, \) and \( h \) satisfying \( J_1 \rightarrow J_5 \).

Related work dealing with similar methods for establishing fair termination, which is a special case of eventuality, is contained in [GFM1], [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

\[ \varphi : \text{at} \ell_0 \land \text{at} m_0 \land (y_1, y_2) = (x_1, x_2) \land x_1 > 0 \land x_2 > 0 \]

and

\[ \psi : \text{at} \ell_0 \land \text{at} m_0 \land y_1 = y_2 = \text{gcd}(x_1, x_2). \]

We wish to prove

\[ \varphi \supset \Diamond \psi, \]

i.e.,

\[ J(P) \models [\text{at} \ell_0 \land \text{at} m_0 \land (y_1, y_2) = (x_1, x_2) \land x_1 > 0 \land x_2 > 0] \supset \Diamond [\text{at} \ell_0 \land \text{at} m_0 \land y_1 = y_2 = \text{gcd}(x_1, x_2)]. \]

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 \(Q, W, u, h\) as follows:

\[ Q(s) : \text{at} \ell_0 \land \text{at} m_0 \land y_1 > 0 \land y_2 > 0 \land \text{gcd}(y_1, y_2) = \text{gcd}(x_1, x_2) \land y_1 \neq y_2 \]

\[ W : (N, >) \text{ the nonnegative integers with the "greater than" relation} \]

\[ u(y_1, y_2) : y_1 + y_2 \]

\[ h(y_1, y_2) : \text{if } y_1 > y_2 \text{ then } P_1 \text{ else } P_2 \]

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 \(Q, W, u, h\). 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 \((C, m_0)\) with \(y_1 = y_2\) the program will immediately proceed to the termination state at \((t_1, m_1)\).

A N 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 \((V, >)\). 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 \(R_v\) with an infinitely decreasing values of \(v\). 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 to 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 $R_v = R_v(\forall), v \in V$, and a decrease function $\delta : V \rightarrow \{1, \ldots, m\}$ such that:

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

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

$IJ3. \models [R(s) \land s' \in g_v(s) \supset [\psi(s') \land \exists u(u \leq v). R_u(s')]]$ for $i = 1, \ldots, m$

$IJ4. \models [R_v(s) \land s' \in g_{\delta(v)}(s) \supset [\psi(s') \lor \exists u(u < v). R_u(s')]]$

Then we may conclude that

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

A stronger conclusion is:

$J(P) \models \varphi \supset (\exists v. R_v) \cup \psi$.

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 $R_v(s), s' \in g_v(s)$ 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 $I$ 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 $(V, \succ)\nu$ is given by $V = W \times [1, \ldots, m]$, where

$(w_1, i) >_V (w_2, j) \iff w_1 > w_2 \text{ or } (w_1 = w_2 \text{ and } i > j)$.

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

$R_{(w,i)}(s)$ is defined by $Q(s) \land [u(s) = w] \land [h(s) = i]$.
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 $s' \in g_k(s)$. By the definition of $R$ we know that $Q(s)$ is true and $u(s) = w, h(s) = j$. By $IJ3$ either $\psi(s')$ is true which immediately satisfies $IJ3$, or $Q(s')$ holds and $w = u(s') \geq u(s') = w'$. Thus, by the definition of $R$, $R_{(w', h(s'))}(s')$ is true. It remains to show that $(w, j) = (w, h(s)) \geq (w', h(s'))$. If $w > w'$ then this is certainly the case. Consider therefore the possibility that $w = w'$. But then by $IJ5$ also $h(s) = h(s')$ leading to $(w, h(s)) = (w', h(s'))$ as required.

To go in the other direction assume that $(V, >), R, \delta$ as required for method $IJ$ have been found. We will show how to select $Q, (w, >), u$ and $h$ that will satisfy the requirements of method $I$.

For simplicity we assume that the order $>$ is a total (linear) order. We may then take the well-founded structure $(V, >)$ to be $(W, >)$. $Q(s)$ 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 $Q, u, \delta$ satisfy requirements $IJ1$ to $IJ5$.  

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 $\psi$. For every $s \in R_i, s' \in R_j$ (i.e. $R_i(s) = R_j(s') \neq \bot$) such that $s' \in g_k(s)$, we draw an edge $e$ from $n_i$ to $n_j$. The edge $e$ is labelled by $P_{s'}$, the process effecting the transition. Similarly, for every $s \in R_i, s' \in \psi$ such that $s' \in g_k(s)$ we draw an edge from $n_i$ to $n_0$ and label it by $P_{s'}$.

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 $i \geq j$.

13. For every $n_i, i > 0$, there must exist some $P_{s'}$ (the helpful process) such that all edges labelled by $P_{s'}$ lead from $n_i$ to some $n_j$ with $i > j$ and such that $P_{s'}$ 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):

\[
y_1 := t_1 := y_2 := t_2 := \bot
\]

\(l_0\): noncritical section 1
\(l_1\): \(t_1 := \begin{cases} F \text{ if } y_2 = F \text{ then } F & \text{else } T \end{cases} \)
\(l_2\): \(y_1 := t_1 \)
\(l_3\): \(t_1 = \bot\) then \(t_1 := y_2 \)
\(l_4\): \(y_1 := t_1 \)
\(l_5\): loop while \(y_1 = y_2 \)

[Critical section 1]

\((y_1, t_1) := (\bot, \bot)\)

\(l_7\): go to \(l_0\)

\(-P_1-\)

\(m_0\): noncritical section 2
\(m_1\): \(t_2 := \begin{cases} F \text{ if } y_1 = T \text{ then } F & \text{else } T \end{cases} \)
\(m_2\): \(y_2 := t_2 \)
\(m_3\): \(t_2 := \neg y_1 \)
\(m_4\): \(y_2 := t_2 \)
\(m_5\): loop while \(\neg y_2 = y_1 \)

[Critical section 2]

\((y_2, t_2) := (\bot, \bot)\)

\(m_7\): go to \(m_0\)

\(-P_2-\)

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 \(\bot\) signifies no interest in entering the critical section. The partial operator \(\neg\) is defined by

\[-I = F, \neg F = T, \neg \bot\text{ is undefined.}\]

Hence in writing \(\neg y_2 = y_1\) we also imply that \(y_1 \neq \bot\) and \(y_2 \neq \bot\). 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 \(\bot\), since \(y_i \neq \bot\) 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 \(y_2 = \bot\) 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 \(y_1 = T \neq y_2 = \bot\) and enter immediately.

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

\[\equiv (t_1 \neq \bot) \equiv at l_{2,6}\]

\[\equiv (y_1 \neq \bot) \equiv at l_{3,6}\]
Figure I.
Diogram Proof for PF
\[ (t_2 \neq \perp) \equiv \text{atm}_{2..6} \]
\[ (y_2 \neq \perp) \equiv \text{atm}_{3..6}, \]

where \( \text{at}_2 \text{ at}_3 \text{ at}_4 \text{ at}_5 \text{ at}_6 \) stands for \( \text{at}_2 \text{ at}_3 \text{ at}_4 \text{ at}_5 \text{ at}_6 \), etc.

The eventuality property we wish to show for this program is

\[ \text{at}_1 \supset \Diamond \text{at}_6. \]

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: \( t_5, m_0 \)

in which the helpful process (indicated by a double arrow \( \Rightarrow \)) is \( P_1 \) since we know that \( y_2 = \perp \).

To illustrate the application of method \( IJ \) to the proof of \( \text{until} \) properties, consider the following precedence property:

\[ [\text{at}_5 \land \sim \text{atm}_{4..6}] \supset [(\sim \text{at}_6) \cup (\text{at}_6)]. \]

It states that if \( P_1 \) arrived at \( t_5 \) 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,

\[ [\text{at}_5 \land \sim \text{atm}_{4..6}] \supset [R_7 \lor R_6 \lor R_5 \lor R_4 \lor R_3]. \]

Therefore this is an admissible diagram in the sense that condition \( IJ \) is satisfied. It establishes that \( \text{at}_6 \) will eventually be realized and all the intermediate states are covered by \( \bigvee_{i=1}^{7} R_i \) which implies \( \sim \text{at}_m. \]

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

\[
\begin{align*}
\ell_0 &: \text{noncritical section 1} \\
\ell_1 &: y_1 := T \\
\ell_2 &: \text{if } y_2 = F \text{ then go to } \ell_7 \\
\ell_3 &: \text{if } t = 1 \text{ then go to } \ell_2 \\
\ell_4 &: y_1 := F \\
\ell_5 &: \text{loop until } t = 1 \\
\ell_6 &: \text{go to } \ell_1 \\
\ell_7 &: \text{critical section 1} \\
& \quad t := 2 \\
\ell_8 &: y_1 := F \\
\ell_9 &: \text{go to } \ell_0 \\
-P_1-
\end{align*}
\]

\[
\begin{align*}
m_0 &: \text{noncritical section 2} \\
m_1 &: y_2 := T \\
m_2 &: \text{if } y_1 = F \text{ then go to } m_7 \\
m_3 &: \text{if } t = 2 \text{ then go to } m_2 \\
m_4 &: y_2 := F \\
m_5 &: \text{loop until } t = 2 \\
m_6 &: \text{go to } m_1 \\
m_7 &: \text{critical section 2} \\
& \quad t := 1 \\
m_8 &: y_2 := F \\
m_9 &: \text{go to } m_0 \\
-P_2-
\end{align*}
\]
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 \( \ell_1 \) to signal the intention of \( P_1 \) to enter its critical section at \( \ell_7 \). Next \( P_1 \) tests at \( \ell_2 \) 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 \( \ell_2 \) and \( \ell_3 \) 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 \( \ell_5 \), 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 \( \ell_2 \).

For the DK program we wish to show:

\[ \equiv \text{at} \ell_1 \supset \Diamond \text{at} \ell_7. \]

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:

\[ \equiv (y_1 = T) \equiv (\text{at} \ell_2,4 \lor \text{at} \ell_7,8) \]

\[ \equiv (y_2 = T) \equiv (\text{at} m_2,4 \lor \text{at} m_7,8) \]

\[ \equiv (\text{at} \ell_3,6 \land t = 2) \supset \text{at} m_1 \ldots 7. \]

For example, we used the last invariant in order to decide that at node 23 the \( P_1 \) successors to states in which \( \text{at} \ell_4 \land (t = 2) \) 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:

\[ \equiv [\text{at} \ell_2,3 \land (t = 1) \land \sim \text{at} m_7] \rightarrow [(\sim \text{at} m_7) \cup (\text{at} \ell_7)]. \]

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

\[ \equiv \text{at} m_8 \supset [(t = 1) \cup (\text{at} \ell_7)]. \]

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 \( \ell_1 \) it will eventually get to \( \ell_2 \). 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 \( \ell_2 \) to \( \ell_5 \). \( P_2 \) cannot enter its critical section more than once without setting \( t \) to 1. Once \( t = 1 \), \( P_1 \) returns to \( \ell_2 \) 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 \( \ell_5 \) to \( \ell_2 \).
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 $\mathcal{J}$ certainly holds for all fair computations. However, the completeness of method $\mathcal{J}$ breaks down in the case of programs with semaphores; we are not always guaranteed that method $\mathcal{J}$ is applicable.

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

**Proof** Method $\mathcal{F}$:

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.** $\models \varphi \supset (\psi \lor Q)$
- **F2.** $\mathcal{F}(P - \{P_k\}) \models [Q(s) \land h(s) = k] \supset [\psi \lor (g_k(s) \neq \phi)]$ for $k = 1, \ldots, m$
- **F3.** $\models [Q(s) \land s' \in g_i(s)] \supset [\psi(s') \lor (Q(s') \land (u(s) \succeq u(s')))]$ for $i = 1, \ldots, m$
- **F4.** $\models [Q(s) \land s' \in g_h(s)] \supset [\psi(s') \lor (u(s) \succ u(s'))]$ for $i = 1, \ldots, m$
- **F5.** $\models [Q(s) \land s' \in g_i(s) \land (u(s) = u(s'))] \supset [\psi(s') \lor (h(s) = h(s'))]$ for $i = 1, \ldots, m$.

Then we may conclude that $\mathcal{F}(P) \models \varphi \supset \Diamond \psi$.

A stronger conclusion is: $\mathcal{F}(P) \models \varphi \supset (Q \cup \psi)$.

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 $\psi$ will be realized or $g_k$ becomes enabled.

The difference between method $\mathcal{F}$ and method $\mathcal{J}$ 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: $\varphi \supset 0 \psi$. 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 $g_k(s) \neq \phi$ it expresses the fact that $P_k$ is enabled, we may omit from our considerations any
action of $P_k$, because such an action may be 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:

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

such that $\varphi(s_0)$ is true and $\psi$ 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 $\psi \lor g_k(s_j) \neq \phi$. Consequently $s_i$ cannot be terminal. By $F3$ the sequence of values $u(s_1), u(s_2), \ldots$ satisfies $u(s_1) \supseteq u(s_2) \supseteq \ldots$ and by being well-founded it must eventually stabilize, let us say at $s_r$, i.e.,

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

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

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

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 suffix sequences we can establish that $g_k$ is enabled an infinite number of times but never activated. Thus $\sigma$ must be unfair.

In [1,PS] it is proved that method $F$ is complete for proving eventual 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 principle proposed above.
Proof Method IF:

For proving eventualities of the form $\varphi \supset \diamond \psi$, under all fair computations of a program $P$, find a well-founded structure $(V, >)$, 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.** $\varphi \supset \{\psi \lor \exists v(v \in V).R_v\}$

**IF2.** $\mathcal{F}(P - \{P_{e(v)}\}) \models R_v(s) \supset \{\psi \lor (g_{b(v)}(s) \neq \phi)\}$

**IF3.** $[R_v(s) \land s' \in g_1(s)] \supset [\psi(s') \lor \exists u(u < v).R_u(s')]$ for $i = 1, \ldots, m$

**IF4.** $[R_v(s) \land s' \in g_{b(v)}(s)] \supset [\psi(s') \lor \exists u(u < v).R_u(s')]$.

Then we may conclude that

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

A stronger conclusion is:

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

Similarly 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 semaphores.

Example D (Program CP — consumer-producer):

$b := A, s := 1, cf := 0, ce := N$

| $e_0$ | compute $y_1$ |
| $e_1$ | request($ce$) |
| $e_2$ | request($s$) |
| $e_3$ | $t_1 := b \cdot y_1$ |
| $e_4$ | $b := t_1$ |
| $e_5$ | release($s$) |
| $e_6$ | release($cf$) |
| $e_7$ | go to $e_0$ |

$m_0 : request(cf)$

$m_1 : request(s)$

$m_2 : y_2 := head(b)$

$m_3 : t_2 := tail(b)$

$m_4 : b := t_2$

$m_5 : release(s)$

$m_6 : release(ce)$

$m_7 : compute using \ y_2$

$m_8 : go to m_0$

$-P_1 : \text{Producer} -$  $-P_2 : \text{Consumer} -$  

The producer $P_1$ computes at $e_0$ 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 element 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 $N > 0$.  

19
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 \( l_{3..5} \) 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

\[ \nu \ 	ext{at} \ l_1 \supset 0 \ 	ext{at} \ l_3. \]

We start by presenting a top-level diagram proof:

![Diagram](image)

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:

\[
F(P_2) \models \text{at} \ l_1 \supset \Box (ce > 0)
\]

\[
F(P_2) \models \text{at} \ l_2 \supset \Box (s > 0).
\]

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:
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 $\ell_1$ 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 $\ell_1$ and at $m_{7,8,9}$ implies that either $ce > 0$ or $cf > 0$.

The diagram proof for $\ell_2$ is even simpler:
Example E (Program $BC$ - a distributed computation of the binomial coefficient):

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

\begin{align*}
\ell_0 &: \text{ if } y_1 = (n - k) \text{ then go to } \ell_e \\
\ell_1 &: \text{ request}(y_1) \\
\ell_2 &: \ t_1 := y_3 \cdot y_1 \\
\ell_3 &: \ y_3 := t_1 \\
\ell_4 &: \text{ release}(y_4) \\
\ell_5 &: \ y_1 := y_1 - 1 \\
\ell_6 &: \text{ go to } \ell_0 \\
\ell_e &: \text{ halt}
\end{align*}

$-P_1-$

$-P_2-$

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;

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

\begin{align*}
\ell_1 &: \text{ if } y_1 = (n - k) \text{ then go to } \ell_1 \\
\ell_6 &: \text{ request}(y_4) \\
\ell_5 &: \ t_1 := y_3 \cdot y_1 \\
\ell_4 &: \ y_3 := t_1 \\
\ell_3 &: \text{ release}(y_4) \\
\ell_2 &: \ y_1 := y_1 - 1 \\
\ell_8 &: \text{ go to } \ell_7 \\
\ell_1 &: \text{ halt}
\end{align*}

$-P_1-$

$-P_2-$

Here we wish to prove:

$$\equiv [\text{at}\{\ell_7, m_3\} \land (y_1, y_2, y_3, y_4) = (n, 0, 1, 1)] \supset \text{at}\{\ell_1, m_1\}.$$
We apply method $F$ with the following:

$$Q : \ell_3 \cdot 5 + \ell m_5 \cdot 7 + y_4 = 1$$

$$\land [((n - k) + \ell_2 \cdot 6) \leq y_1 \leq n]$$

$$\land [0 \leq y_2 \leq (k - \ell m_2)]$$

$$\land [\ell_1 \triangleright (y_1 = n - k)]$$

$$(W, >) : \langle N \times N, >_{lex} \rangle$$

the lexicographically ordered domain of pairs of nonnegative integers


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

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

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 arc only two situations to be checked. First, when $P_1$ is at $\ell_1$ 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 arc 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 $\ell_6$ 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 $Q$.

![Diagram](image)

**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 Mosc, Ben Moszkowski, Tmima Olshanski-Koren, Itivi Sherman, Pierre Wolper, and Frank Yellin for careful and critical reading of the manuscript.

REFERENCES


