Science of Computer Programming 4 (1984) 257-289
North-Holland

ADEQUATE PROOF PRINCIPLES FOR INVARIANCE AND LIVENESS PROPERTIES OF CONCURRENT PROGRAMS*

Zohar MANNA
Computer Science Department, Stanford University, Stanford, CA 94305, U.S.A.

Amir PNUELI
Applied Mathematics Department, The Weizmann Institute, Rehovot, Israel

Communicated by L. Lamport
Received April 1982
Revised May 1984

Abstract. This paper presents proof principles for establishing invariance and liveness properties of concurrent programs. Invariance properties are established by systematically checking that they are preserved by every atomic instruction in the program. The methods for establishing liveness properties are based on *well-founded assertions* 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 such proofs can be represented by diagrams. Several examples are given.

1. Introduction

Most of the temporal properties of programs can be partitioned in a natural way into two classes. Properties in each of the classes can be characterized by the form of the temporal formulas expressing them.

The first set in this partition is the class of *invariance* properties (*safety* in the terminology of [13]). These are the properties that can be expressed by a temporal formula of the form:

$$\Box\psi \quad \text{or} \quad \varphi \supset \Box\psi.$$

Such a formula, stated for a program P, says that every computation of P continuously satisfies ψ. In the case of the second form it states that whenever φ becomes true, ψ is immediately realized and will hold continuously throughout the rest of

* This research was supported in part by the National Science Foundation under Grants MCS82-14523 and MCS81-11586, 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.

the computation. Some properties falling into this class are: partial correctness, clean (error-free) behavior, mutual exclusion, and absence of deadlocks.

The second set in the partition is the class of *liveness* properties (*eventualities* in the terminology of [7, 14]). These are properties which are expressible by temporal formulas of the form:

$$\Diamond\psi \quad \text{or} \quad \varphi \supset \Diamond\psi.$$

In both cases these formulas guarantee the occurrence of some event ψ; in the first case unconditionally and in the second case conditional on an earlier occurrence of the event φ. Some properties falling into this class are: total correctness, termination, accessibility, lack of individual starvation, and responsiveness.

An extension of the class of liveness properties is the class of *until* properties, whose temporal formulation is of the form:

$$\chi\mathcal{U}\psi \quad \text{or} \quad \varphi \supset \chi\mathcal{U}\psi.$$

In both cases the formulas again guarantee the occurrence of the event ψ, but they also ensure that from now until that occurrence, χ will continuously hold. Some properties falling into the *until* class are: strict (*FIFO*) responsiveness, and bounded overtaking.

A fuller discussion of these classes and the general expression of program properties in temporal logic is provided in [7].

In [10] (an earlier version is presented in [8]) a comprehensive proof system for proving the temporal properties of programs is described. It provides a basis for proving the validity of an arbitrary temporal formula over a given program. However, being so general, it gives very few guidelines for the construction of proofs of properties that belong to special classes.

In this paper we specialize the general approach presented in [10] to the particular classes of invariance and liveness (including *until*) properties. For each of these classes, we recommend a single proof principle that may be uniformly applied to establish properties of this class.

The first proof principle we present is for establishing invariances. This principle is not new, and in one form or another has been suggested by almost every work on the subject of concurrent verification, e.g. [3, 4, 12]. It is a natural extension of the invariant-assertion method for sequential programs (see [6]).

The second proof principle, which establishes liveness (and *until*) properties, is more interesting. It is an extension of the classical method of well-founded assertions for proving termination of sequential programs (see [6]). Similar suggestions emphasizing the role of well-founded induction in proofs of termination are included in many of the works on concurrent verification (e.g., [3] and [4]).

The work in [13] presents an approach which is close to ours. It gives comprehensive coverage of both invariance and liveness properties with an emphasis on the liveness. There is similarity between the proof lattice presented in [13] and our diagram proofs. One direction in which the current paper obviously extends the methods of [13] is the presentation of the well-founded principles, enabling proofs of liveness properties for programs with an infinite number of states.

2. Motivation

A *well-founded structure* $(W, >)$ consists of a set W and a partial order $>$ over W such that any decreasing sequence $w_0 > w_1 > w_2 > \cdots$, where $w_i \in W$, is necessarily finite. A typical and frequently used well-founded structure is $(N, >)$, where N is the set of nonnegative integers, and $>$ is the usual 'greater than' ordering: there is no infinitely decreasing sequence of nonnegative integers.

A general method for deriving composite well-founded structures from simpler ones is the formation of lexicographical orderings. Let $(W_1, >_1)$ and $(W_2, >_2)$ be two well-founded structures. Then the structure given by $(W_1 \times W_2, >_{lex})$, where the lexicographic ordering $>_{lex}$ over $W_1 \times W_2$ is defined by

$$(m_1, m_2) >_{lex} (n_1, n_2) \overset{\text{def}}{\equiv} (m_1 >_1 n_1) \textit{ or } (m_1 = n_1 \textit{ and } m_2 >_2 n_2)$$

is also well-founded.

The basic idea for proving liveness by well-founded assertions is to find an assertion $Q(s; w)$ relating the program state s to a well-founded parameter $w \in W$. Let us assume that we wish to prove $\Diamond\psi$, i.e., that eventually ψ will occur in any computation. In the simple approach we require a descent of the parameter w on every computation step until ψ is attained. This means that whenever $Q(s; w)$ holds and s' is a possible successor of s under one computation step, then either s' satisfies ψ or there exists a w' such that $w > w'$ and $Q(s'; w')$ holds. Thus, any computation that fails to achieve ψ generates an infinitely descending sequence of W-elements, which is impossible.

Consider for example the following program, which computes the *gcd* (greatest common divisor) of two positive integers x_1 and x_2.

Program GCD. Sequential *gcd* computation:

```
$(y_1, y_2) := (x_1, x_2)$;
while $y_1 \neq y_2$ do
    if $y_1 > y_2$ then $y_1 := y_1 - y_2$
                  else $y_2 := y_2 - y_1$
```

Suppose we wanted to prove that any computation of this program will eventually reach a state in which $y_1 = y_2$, i.e., $\Diamond(y_1 = y_2)$. We may choose the well-founded structure to be the set N of nonnegative integers with the usual greater-than $>$ ordering, and the parameterized assertion

$$Q(y_1, y_2; n):\ (y_1 > 0) \wedge (y_2 > 0) \wedge (y_1 + y_2 = n).$$

Here the state is specified by the values of the variables y_1, y_2, and the well-founded parameter is $n \geq 0$. We consider the execution of the body of the loop as one computation step. Clearly whenever ψ has not been achieved yet and consequently $y_1 \neq y_2$, the execution of this statement leads to new values of y_1 and y_2, say y_1' and y_2' such that $y_1 + y_2 > y_1' + y_2'$. Taking the parameter value to be $n' = y_1' + y_2'$, the assertion $Q(y_1', y_2'; n')$ holds for the new values of y_1, y_2, but with a parameter value $n' = y_1' + y_2' < y_1 + y_2 = n$. This establishes that any computation is bound to achieve $y_1 = y_2$, i.e. $\Diamond(y_1 = y_2)$.

This example demonstrates that the simple approach, requiring descent on *each* step, works successfully for sequential programs.

However it may easily fail for concurrent programs. Let us consider the following concurrent program that performs the distributed computation of the *gcd* of two positive integers x_1, x_2.

Program DGCD. Distributed *gcd* computation:

```
                         (y1, y2) := (x1, x2)

  l0: while y1 ≠ y2 do                  m0: while y1 ≠ y2 do
        if y1 > y2 then y1 := y1 − y2         if y1 < y2 then y2 := y2 − y1
  l1: halt                              m1: halt
        − P1 −                                − P2 −
```

In the execution of this program, we assume each of the labelled instructions is atomic in the sense that test 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 strongly on the basic assumption of *fairness* that we assume throughout this work. For this program, the assumption of fairness requires that if a process has not terminated it must eventually be activated. Only under fairness would each of P_1 and P_2 be activated as often as needed until termination is achieved.

Trying to prove the termination of this program by the simple approach of well-founded assertions immediately runs into difficulties when we fail to find an assertion $Q(s; w)$ with a well-founded parameter w that will decrease at *every* step of the computation. No such assertion can exist for the above program since, as observed, some steps may preserve the state and leave the value of a state-dependent parameter constant. This points out emphatically that a well-founded argument may succeed for this program only if it takes fairness into account and does not insist on a decrease of the parameter at every step.

The basic observation made in [5] and implied in [8] is that it is sufficient that, at any stage of the computation, we can identify one of the processes such that any computation step of this process will guarantee a decrease. A slightly different but essentially equivalent formulation of the same principle was independently developed in [2].

The liveness principles that we present here can be developed as part of the formal temporal system of [10]. But once the principles are justified they can be used without any additional temporal reasoning. Since these rules can be shown to be complete, it follows that they are the only rules which are needed in order to prove liveness properties.

3. Programs and computations

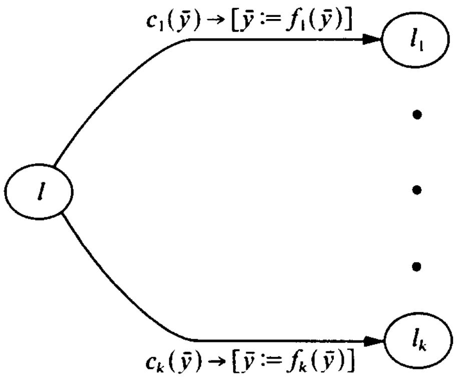
The computation model used in our presentation is based on the shared-variables model of concurrent programs. For a fuller discussion of the model we refer the reader to [7]. As implied by [10] the same techniques are easily adaptable to deal with other models based on synchronous as well as asynchronous communication.

Let P be a program consisting of m parallel processes:

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

Each process P_i is represented as a transition graph with locations (nodes) labelled by elements of $\mathscr{L}_i = \{l_0^i, \ldots, l_t^i\}$. The edges in the graph are labelled by guarded commands of the form $c(\bar{y}) \rightarrow [\bar{y} := f(\bar{y})]$, whose meaning is that if $c(\bar{y})$ is true the edge may be traversed while replacing $\bar{y}$ by $f(\bar{y})$.

Let $l, l_1, \ldots, l_k \in \mathscr{L}_j$ be locations in process P_j;


$c_1(\bar{y}) \rightarrow [\bar{y} := f_1(\bar{y})]$
l_1
l
l_k
$c_k(\bar{y}) \rightarrow [\bar{y} := f_k(\bar{y})]$

We define $E_l(\bar{y}) = c_1(\bar{y}) \vee \cdots \vee c_k(\bar{y})$ to be the *exit condition* at node l. Locations in the program can be classified according to their exit conditions:

– A location is *regular if* $E_l \equiv$ *true*. This is the case of locations such that the set of conditions labeling their outgoing transitions is exhaustive in the sense that for every possible value of $\bar{y}$ at least one transition is enabled.

– A location is *terminal* if $E_l \equiv \mathit{false}$. This is the case of locations labeling *halt* instructions which have no outgoing transitions. In our model there is usually only one such location per process. This location for process P_i will be labelled l_t^i.
– Any location l such that the exit condition $E_l(\bar{y})$ is nontrivial, i.e., it is neither identically *true* nor *false*, is called a *semaphore* location. Examples of such locations are those corresponding to the instruction *request*(y_r) whose transition diagram is:

$$\begin{tikzcd} l \arrow[r, "(y_r > 0) \to {[y_r := y_r - 1]}"] & l' \end{tikzcd}$$

Note that $E_l(\bar{y}) = (y_r > 0)$. The *request* instruction is used in order to reserve a resource, where y_r counts the number of units of this resource currently available. In this paper, the only semaphore locations we consider are the locations having a *request*(y) transition departing from them.

The symmetric counterpart to the *request* instruction is the *release*(y_r) instruction. It is used to release one unit of a reserved resource. Its transition diagram is:

$$\begin{tikzcd} l \arrow[r, "\mathit{true} \to {[y_r := y_r + 1]}"] & l' \end{tikzcd}$$

The location of the release instruction is regular.

A *state* of the program P is a tuple of the form $s = \langle \bar{l}; \bar{\eta} \rangle$ with $\bar{l} \in \mathscr{L}_1 \times \cdots \times \mathscr{L}_m$ and $\bar{\eta} \in D^n$, where D is the domain over which the program variables $y_1, \ldots, y_n$ range. The vector $\bar{l}$ is the list of current locations which are next to be executed in each of the processes. The vector $\bar{\eta}$ is the list of current values assumed by the program variables $\bar{y}$ at state s.

Let $s = \langle l^1, \ldots, l^i, \ldots, l^m; \bar{\eta} \rangle$ be a state. We say that process P_i is *enabled* on s if $E_{l^i}(\bar{\eta})$ is true. This implies that if we let P_i run at this point, there is at least one condition c_j among the edges departing from l^i that is true. Otherwise, we say that P_i is *disabled* on s. An example of a disabled process P_i is the case that l^i labels an instruction *request*(y) and $y = 0$; another example is that l^i labels a *halt* statement. A state is defined to be *terminal* if no P_i is enabled on it.

Given a program P we define the notion of a *computation step* of P.

Let $s = \langle l^1, \ldots, l^m; \bar{\eta} \rangle$ and $\tilde{s} = \langle \tilde{l}^1, \ldots, \tilde{l}^m; \tilde{\bar{\eta}} \rangle$ be two states of P. Let τ be a transition in P_i of the form:

$$\begin{tikzcd} l^i \arrow[r, "c(\bar{y}) \to {[\bar{y} := f(\bar{y})]}", "\tau"'] & \tilde{l}^i \end{tikzcd}$$

such that $c(\bar{\eta}) = \mathit{true}$, $\tilde{\bar{\eta}} = f(\bar{\eta})$, and for every $j \neq i$, $\tilde{l}^j = l^j$. Then we say that $\tilde{s}$ is a *successor* of s under the transition τ (a τ-*successor* for short), and write:

$$s \xrightarrow{\tau} \tilde{s}.$$

If $\bar{s}$ is a τ-successor of s under some transition $\tau \in P_i$, then we may also describe $\bar{s}$ as being obtainable from s by a P_i-step (a single computation step of P_i), and write:

$$s \xrightarrow{P_i} \bar{s}.$$

An *initialized admissible computation* of a program P for an input $\bar{x} = \bar{\xi}$ is a labelled sequence of states of P

$$\sigma: \quad s_0 \xrightarrow{P_{i_0}} s_1 \xrightarrow{P_{i_1}} s_2 \xrightarrow{P_{i_2}} s_3 \longrightarrow \cdots$$

satisfying the following conditions:

(A) *Initialization.* The first state s_0 has the form:

$$s_0 = \langle \bar{l}_0; g(\bar{\xi}) \rangle,$$

where $\bar{l}_0 = (l_0^1, \ldots, l_0^m)$ is the vector of initial locations in all the processes. The values $g(\bar{\xi})$ are the initial values assigned to the $\bar{y}$ variables for the input $\bar{\xi}$.

(B) *State to state sequencing.* Every step in the computation $s \rightarrow^{P_i} \bar{s}$, is justified by $\bar{s}$ being obtainable from s by a single P_i-step.

(C) *Maximality.* The sequence is maximal, i.e., it is either infinite or ends in a state s_k which is terminal.

(D_F) *Fairness.* Every P_i which is enabled in infinitely many states of σ must be activated infinitely many times in σ, i.e., there must be an infinite number of P_i-steps in σ.

We define an *admissible computation* of P for input $\bar{\xi}$ to be either an initialized admissible computation or a suffix of an initialized admissible computation. The class of all *admissible* computations of program P is the set of all sequences which are admissible computations for some input $\bar{\xi}$. We denote the class of all admissible computations of the program P by $\mathscr{A}(P)$. A state s is defined to be *accessible* by the program P if it appears in an admissible computation.

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

To emphasize the fact that in our standard definition the computations are required to be fair we sometimes refer to the class of admissible computations also as the class of *fair computations* of P, and denote it by $\mathscr{A}_F(P)$.

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.

(D_J) *Justice.* Every P_i which is continuously enabled beyond a certain state s in σ, must be activated infinitely many times in σ.

We refer to the class of all suffixes of computations that satisfy conditions A, B, C and D_J for some input $\bar{\xi}$, as the class of all *just computations*, and denote it by $\mathcal{A}_J(P)$.

For an arbitrary program P we have in general:

$$\mathcal{A}_F(P) \subseteq \mathcal{A}_J(P),$$

i.e., every fair computation is also just. For programs that contain semaphore instructions, there may exist just computations which are unfair.

To see that the first claim holds, let σ be an infinite fair computation. Let P_i be any process that is continuously enabled beyond a certain state in σ. Then, 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 $\mathcal{A}_F(P)$ and $\mathcal{A}_J(P)$ may be strict, consider the following program which is the simplest program modelling mutual exclusion:

```
          y := 1

   l0: request(y)      m0: request(y)

   l1: release(y)      m1: release(y)

   l2: goto l0         m2: goto m0

       - P1 -              - P2 -
```

Note that here and in the following examples we prefer to present the programs as lists of labelled instructions. The corresponding representation in transition diagram form is obvious.

The following computation:

$$\sigma\colon\ (l_0, m_0; 1) \xrightarrow{P_1} (l_1, m_0; 0) \xrightarrow{P_1} (l_2, m_0; 1) \xrightarrow{P_1}$$

$$(l_0, m_0; 1) \xrightarrow{P_1} (l_1, m_0; 0) \xrightarrow{P_1} (l_2, m_0; 1) \xrightarrow{P_1} \cdots$$

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 $(l_1, m_0; 0)$. Consequently 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 $(l_0, m_0; 1)$ and $(l_2, m_0; 1)$, but is never activated.

However, when a program 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:

$$\mathcal{A}_F(P) = \mathcal{A}_J(P).$$

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

An admissible computation is said to be *convergent* if it is finite:

$$\sigma\colon\ s_0 \xrightarrow{P_{i_0}} s_1 \xrightarrow{P_{i_1}} \cdots \xrightarrow{P_{i_{f-1}}} s_f.$$

If the final state s_f of a convergent computation is of the form $s_f = \langle l_t^1, \ldots, l_t^m; \bar{\eta}\rangle$, where each l_t^i labels a halt instruction, we say that the *computation has terminated.* Otherwise, we say that the computation is *blocked* (*deadlocked*).

In order to describe properties of states we introduce a vector of *location variables* $\bar{\pi} = (\pi_1, \ldots, \pi_m)$. Each π_i ranges over $\mathscr{L}_i$, and assumes the value l^i in a state

$$s = \langle l^1, \ldots, l^i, \ldots, l^m; \bar{\eta}\rangle.$$

Thus we may describe a state $s = \langle \bar{l}; \bar{\eta}\rangle$ by saying that in this state $\bar{\pi} = \bar{l}$ and $\bar{y} = \bar{\eta}$.

A *state formula* $Q = Q(\bar{\pi}; \bar{y})$ is any first-order formula. It is built from terms and predicates over the location and program variables $(\bar{\pi}; \bar{y})$ and may also refer to additional variables. We will also refer to state formulas as *assertions.*

A state formula may refer also to the input variables $\bar{x}$. Our computational model explicitly assumes that no statement may modify the values of the input variables. Consequently in any states belonging to a $\bar{\xi}$-computation, the values of $\bar{x}$ in s are necessarily $\bar{\xi}$.

A state s that satisfies a state formula φ is referred to as a *φ-state.*

We frequently abbreviate the statement $\pi_i = l$ to *at* l or simply l. Since the $\mathscr{L}_i$'s are pairwise disjoint, there is no difficulty in identifying the particular π_i which assumes the value l. A similar notation *at* $\bar{l}$ is used to make a statement about all the locations in the state, namely $\bar{\pi} = \bar{l}$.

Let the following be a transition τ in process P_i:

$$\begin{tikzcd} l^i \arrow[rrr, "c(\bar{y}) \to {[\bar{y} := f(\bar{y})]}", "\tau"'] & & & \bar{l}^i \end{tikzcd}$$

Let $\varphi(\bar{\pi}; \bar{y})$ and $\psi(\bar{\pi}; \bar{y})$ be two state formulas. We say:

– *The transition τ leads from φ to ψ,* if every τ-successor of an accessible φ-state is a ψ-state. Thus, if $\tilde{s} = \langle l^1, \ldots, \bar{l}^i, \ldots, l^m; \bar{\bar{\eta}}\rangle$ is a τ-successor of the accessible state $\langle l^1, \ldots, l^i, \ldots, l^m; \bar{\eta}\rangle$, which of course implies that $c(\bar{\eta}) = \mathit{true}$ and $\bar{\bar{\eta}} = f(\bar{\eta})$, then the following implication must be true:

$$\varphi(l^1, \ldots, l^i, \ldots, l^m; \bar{\eta}) \supset \psi(l^1, \ldots, \bar{l}^i, \ldots, l^m; \bar{\bar{\eta}}).$$

One way of establishing that τ leads from φ to ψ is to show the general validity of the following implication:

$$[\varphi(l^1, \ldots, l^i, \ldots, l^m; \bar{y}) \wedge c(\bar{y})] \supset \psi(l^1, \ldots, \bar{l}^i, \ldots, l^m; f(\bar{y}))$$

for every choice of $(l^1, \ldots, l^{i-1}, l^{i+1}, \ldots, l^m) \in \mathscr{L}_1 \times \cdots \times \mathscr{L}_{i-1} \times \mathscr{L}_{i+1} \times \cdots \times \mathscr{L}_m$. This is a stronger statement, since it does not utilize the fact that s is accessible.

This notion is extended to processes and then to the complete program as follows:
- *A process P_i leads from φ to ψ*, if every transition τ in P_i leads from φ to ψ.
- *A program P leads from φ to ψ*, if every process P_i leads from φ to ψ.

In the following, when we present a formal or an informal argument that establishes the fact that a process P_i leads from φ to ψ, we say that this fact is *provable*, and denote it by writing:

$$\vdash P_i \text{ leads from } \varphi \text{ to } \psi.$$

An analogous notation is used for the full program P leading from φ to ψ.

4. The language of temporal logic

Temporal logic is a language that enables a natural expression of properties of time sequences. Since our main interest is in stating and proving properties of computations of some program, we will consider temporal formulas to be interpreted over the sequences of states arising in computations.

With the computation

$$\tilde{\sigma}:\ s_0 \xrightarrow{P_{i_0}} s_1 \xrightarrow{P_{i_1}} \cdots$$

we associate the sequence of states

$$\sigma:\ s_0, s_1, \cdots$$

Note that infinite computations are associated with infinite state sequences and finite computations are associated with finite sequences. This is an improvement on the version presented in [7-11] that required all considered state sequences to be infinite. In order to achieve this, finite computations were artificially extended by an infinite duplication of the final state.

The basic formulas of the language are the state formulas (assertions). As already mentioned, these are formulas written in some first-order language that describe a property of a program state. For example, for a program with variables y_1, y_2 and location l, the formula $at\ l \supset (y_1 = y_2)$ is a state formula (assertion) which is true for all states such that either the program is not currently at the location l, or such that currently $y_1 = y_2$.

The basic state formulas may now be extended by combining the boolean operators ($\neg, \wedge, \vee, \supset, \equiv$) and quantifiers ($\forall, \exists$) of first-order logic with four temporal operators, called respectively:

$\Box$: *always* (henceforth)

$\Diamond$: *sometimes* (eventually)

$\bigcirc$: *nexttime*

$\mathcal{U}$: *until*

The interpretation of general temporal formulas over (computation) sequences is defined as follows:

Let σ: $s_0, s_1, \ldots$ be a nonempty sequence of states.

We define the length of σ, denoted by $l(\sigma)$, as follows. For a finite sequence σ: $s_0, \ldots, s_k$ we let $l(\sigma) = k$. For an infinite sequence we define $l(\sigma) = \omega$, the first infinite ordinal.

For a state formula φ,

$$\sigma \models \varphi,$$

i.e., σ satisfies φ, if and only if s_0 (the first state of σ) satisfies φ.

The boolean connectives and first-order logic quantifiers are interpreted in the natural way, for example

$$\sigma \models (\varphi_1 \vee \varphi_2) \text{ if and only if } \sigma \models \varphi_1 \textit{ or } \sigma \models \varphi_2.$$

To interpret the temporal operators we introduce the notation $\sigma^{(k)}, 0 \leq k \leq l(\sigma)$, standing for the sequence obtained from σ by removing the first k elements, i.e.

$$\sigma^{(k)} = s_k, s_{k+1}, \ldots .$$

Then:

$$\sigma \models \Box\varphi \quad \text{if and only if } \forall k(0 \leq k \leq l(\sigma)), \sigma^{(k)} \models \varphi$$

$$\sigma \models \Diamond\varphi \quad \text{if and only if } \exists k(0 \leq k \leq l(\sigma)), \sigma^{(k)} \models \varphi$$

$$\sigma \models \bigcirc\varphi \quad \text{if and only if } l(\sigma) > 0 \text{ and } \sigma^{(1)} \models \varphi$$

$$\sigma \models \varphi \mathcal{U} \psi \text{ if and only if } \exists k(0 \leq k \leq l(\sigma)) \text{ such that } \sigma^{(k)} \models \psi \text{ and } \forall i, 0 \leq i < k, \sigma^{(i)} \models \varphi.$$

For the simple cases that φ and ψ are state formulas the general definitions above can be given the following intuitive interpretation:

σ satisfies $\Box\varphi$	if and only if *all* states in σ satisfy φ
σ satisfies $\Diamond\varphi$	if and only if *some* state in σ satisfies φ
σ satisfies $\bigcirc\varphi$	if and only if the *second* state in σ satisfies φ
σ satisfies $\varphi \mathcal{U} \psi$	if and only if some state s' in σ satisfies ψ and all the states *until* s' (excluding s') satisfy φ.

Note that in the definition above, σ can satisfy $\bigcirc\varphi$ only if a second state s_1 exists in σ.

Some more complicated combinations are very useful. For example,

– The formula

$$\Box\bigcirc\Diamond\varphi$$

means that φ must be true on infinitely many states of σ. Note that when all sequences are assumed to be infinite, $\Box\Diamond\varphi$ is the natural expression for the fact that σ contains infinitely many φ-states. However, once we allow finite sequences, a *finite* sequence σ may satisfy $\Box\Diamond\varphi$ by having its last state satisfying φ. Yet, of course, such a finite sequence cannot contain infinitely many φ-states. The more complicated expression $\Box\bigcirc\Diamond\varphi$ forces any sequence σ, satisfying it, to be infinite and contain infinitely many φ-states.

– The general *nested until* formula

$$p_1\mathcal{U}(p_2\mathcal{U}\ldots(p_k\mathcal{U}q)),$$

means that σ starts with a sequence of states satisfying p_1, followed by a sequence of states satisfying $p_2,\ldots$ followed by a sequence of states satisfying p_k, followed by a state satisfying q. Any of these sequences can be empty.

A temporal formula φ is defined to be *valid* for the program P, *P-valid* for short, if every admissible computation $\sigma\in\mathcal{A}_F(P)$ satisfies φ. We denote this fact by

$$\mathcal{A}_F(P)\models\varphi.$$

When the identity of the program P is clearly determined by the context, we omit the prefix $\mathcal{A}_F(P)$ and write simply $\models\varphi$.

In the following we present some proof principles or rules that establish P-validity for some formulas. Whenever we want to state that the P-validity of a formula φ has been established by a rule, we write

$$\mathcal{A}_F(P)\vdash\varphi,$$

or simply $\vdash\varphi$ when the identity of P is determined by the context.

5. The invariance principle

A very simple and intuitive principle suffices in order to establish invariance properties

Invariance Rule: INV
Let $\varphi(\bar{\pi};\bar{y})$ be a state formula

$$\frac{\vdash P \text{ leads from } \varphi \text{ to } \varphi}{\vdash \varphi\supset\Box\varphi}$$

The form of this rule, which will be used throughout the paper, states that if the premise, "P leads from φ to φ", has been established, then the consequence, "$\varphi\supset\Box\varphi$", logically follows as a P-valid formula. The way to establish the premise is to check all the atomic instructions in each of the processes and verify that each of them always leads from φ to φ.

It is very easy to convince ourselves of the validity of this rule. Consider an admissible computation of P whose initial state s_0 satisfies φ. Since all subsequent states are derived from previous states by atomic actions of the program P, all of which have been shown to preserve φ, φ must be an invariant of the computation.

With the addition of an extra premise, guaranteeing that all initial states satisfy φ, we can conclude the *unconditional* invariance of φ over all admissible computations.

Initialized Invariance Rule: I-INV
Let $\varphi(\bar{\pi}; \bar{y})$ be a state formula

$$\frac{\begin{array}{c}\vdash [at\ \bar{l}_0 \wedge \bar{y} = g(\bar{x})] \supset \varphi \\ \vdash P \text{ leads from } \varphi \text{ to } \varphi\end{array}}{\vdash \Box \varphi}$$

The first premise in the rule assures that any legal initial state, defined by having all processes reset to their initial locations $\bar{l}_0 = (l_0^1, \ldots, l_0^m)$, and the program variables $\bar{y}$ initialized to $g(\bar{x})$, must satisfy φ. The second premise ensures, as before, that once φ is established, it is preserved forever after. Hence any accessible state must satisfy φ.

As an application of the I-INV rule let us prove a property of semaphore variables.

Example (semaphore variables). A *semaphore varible* is a variable y such that the initial value it receives is a nonnegative integer, and the only instructions that may alter its value are *request*(y) and *release*(y) instructions.

Let y be some semaphore variable. Let

$$\phi(y): \ y \geq 0.$$

By definition, any proper initialization $\bar{y} := g(\bar{x})$ must assign to y a nonnegative value, establishing φ initially. Consider next the instructions that can modify y. Since y is a semaphore variable, the only such instructions are *request*(y) and *release*(y).

A *request* instruction is equivalent to $(y > 0) \rightarrow [y := y - 1]$. Therefore the condition that it leads from φ to φ is

$$[(y \geq 0) \wedge (y > 0)] \supset (y-1) \geq 0,$$

which is always true.

A *release* instruction is equivalent to $y := y + 1$. It certainly leads from φ to φ since

$$(y \geq 0) \supset (y + 1 \geq 0).$$

Obviously, all the other transitions do not modify y and hence lead from $y \geq 0$ to $y \geq 0$. Thus all the premises to the I-INV rule are established, and it follows that in any accessible state $y \geq 0$. □

Earlier we indicated that one way to establish "τ leads from φ to ψ" was by proving a *verification condition* appropriate for τ. However the verification condition did not utilize the fact that it is supposed to hold only for *accessible* states. The fact of accessibility may be introduced by the following rule that uses invariance properties rather than infers them:

> *Accessibility Rule*: ACC
> Let φ, χ and ψ be state formulas
>
> $\vdash \Box\varphi$
>
> $\vdash \tau$ leads from $(\varphi \wedge \chi)$ to $(\varphi \supset \psi)$
>
> ---
>
> $\vdash \tau$ leads from χ to ψ

To justify the rule, consider an accessible χ-state s. Since it is accessible it must satisfy the invariant φ and is therefore also a $(\varphi \wedge \chi)$-state. Let $\tilde{s}$ be any τ-successor of s. By the second premise it is a $(\varphi \supset \psi)$-state, and since obviously it is also accessible, it must also satisfy the invariant φ. Consequently it is also a ψ-state.

The validity of an invariance property does not depend on whether we consider fair or just computations. Liveness properties, on the other hand, may behave differently on just or fair computations. Consequently we need different sets of rules for just and fair liveness.

6. Rules for just computations

In this section we present a proof principle enabling us to prove liveness properties that hold for the class of just computations $\mathscr{A}_J(P)$. This will suffice for proving liveness properties of programs without semaphore instructions.

The basic liveness proof rule for just computations is given by:

> *Just Liveness Rule*: J-LIVE
> *Let* $\varphi(\bar{\pi}; \bar{y})$ and $\psi(\bar{\pi}; \bar{y})$ be two state formulas and P_k be one of the processes
>
> A. $\vdash P$ leads from φ to $\varphi \vee \psi$
>
> B. $\vdash P_k$ leads from φ to ψ
>
> C. $\vdash \varphi \supset [\psi \vee \mathit{Enabled}(P_k)]$
>
> ---
>
> $\vdash \varphi \supset (\varphi \mathcal{U} \psi)$

Note that the conolusion is somewhat stronger than simple liveness and guarantees not only the eventual occurrence of ψ but that φ will continuously hold until then. It implies $\varphi \supset \Diamond\psi$.

To establish the validity of the rule, suppose that conditions A to C hold. Let σ be a just computation such that initially φ holds but, contrary to our conclusion, ψ is never realized. By condition A, the only way out of φ is to achieve ψ. Hence φ must be continuously true all along σ. By condition C, the 'helpful' process P_k is continuously enabled throughout σ. Consequently σ cannot contain a terminal state and must therefore be infinite. By justice, eventually P_k will be activated from a φ-state. By condition B, this would lead to ψ, contradicting our assumption that ψ is never realized.

In applying this basic principle to prove a liveness property of a program we often observe the following pattern: There is a sequence of state formulas (assertions) $\varphi_0, \varphi_1, \ldots, \varphi_r$ such that the initial state satisfies φ_r and the desired goal is $\psi = \varphi_0$. We then repeatedly apply the J-LIVE principle to show that being at φ_i, $0 < i \leq r$, we eventually get to φ_{i-1}, i.e. $\varphi_i \supset \Diamond\varphi_{i-1}$. This of course establishes that being at φ_r, or as a matter of fact at any φ_i, $0 \leq i \leq r$, we are guaranteed to eventually achieve φ_0. If we summarize the premises for each application of the J-LIVE rule, we obtain the following useful rule:

Just Chain Rule—J-CHAIN

Let $\varphi_0, \varphi_1, \ldots, \varphi_r$ be a sequence of assertions satisfying the following requirements:

A. For $i = 1, \ldots, r$

$$\vdash P \text{ leads from } \varphi_i \text{ to } \left(\bigvee_{\leq i} \varphi_j\right)$$

B. For $i = 1, \ldots, r$ there exists a k_i, such that:

$$\vdash P_{k_i} \text{ leads from } \varphi_i \text{ to } \left(\bigvee_{j<i} \varphi_j\right)$$

C. For $i = 1, \ldots, r$ and k_i as above:

$$\frac{\vdash \varphi_i \supset \left[\left(\bigvee_{j<i} \varphi_j\right) \vee \mathit{Enabled}(P_{k_i})\right]}{\vdash \left(\bigvee_{i=0}^{r} \varphi_i\right) \supset \left(\bigvee_{i=1}^{r} \varphi_i\right) \mathcal{U} \varphi_0}$$

Diagram representation of the CHAIN rule

In presenting a proof according to the CHAIN rule it is usually sufficient to identify $\varphi_0, \varphi_1, \ldots, \varphi_r$ and for each $i = 1, \ldots, r$ to point out the 'helpful' process P_{k_i}. It can be left to the reader to verify that premises A to C are satisfied for each i.

We prefer to present such proofs in the form of a diagram. Consider a diagram consisting of nodes that correspond to the assertions $\varphi_0, \varphi_1, \ldots, \varphi_r$.

For each two accessible states s_i (satisfying φ_i) and s_j (satisfying φ_j) and a process P_l such that $s_i \rightarrow^{P_l} s_j$, we draw an edge $\rightarrow$ from the node φ_i to the node φ_j and label it by P_l, the process responsible for the transition.

$$\begin{tikzcd}\boxed{\varphi_i} \arrow[r, "P_l"] & \boxed{\varphi_j}\end{tikzcd}$$

All edges corresponding to the helpful process P_{k_i}, are drawn as double arrows $\Rightarrow$

$$\begin{tikzcd}\boxed{\varphi_i} \arrow[r, Rightarrow, "P_{k_i}"] & \boxed{\varphi_j}\end{tikzcd}$$

In order for a diagram to represent a valid proof by the J-CHAIN rule the following conditions must hold:
- Every successor of an accessible φ_i-state, for $i > 0$, satisfies some $\varphi_j, j \geq 0$.
- For every edge connecting φ_i to φ_j we must have $i \geq j$.
- For every edge connecting φ_i to φ_j and labelled by P_{k_i} we must have $i > j$
- For every accessible state s, if $i > 0$ is the lowest index such that s satisfies φ_i, then P_{k_i} must be enabled on s.

We illustrate diagram proofs by two examples. The first demonstrates a complex invariance proof accompanied by a relatively simple liveness proof. The other example demonstrates a more involved liveness proof with a relatively easy invariance proof.

Example 1. The following program provides a distributed solution for achieving mutual exclusion without semaphores.

Program PF. The Peterson-Fischer algorithm [16]:

$$(y_1, t_1, y_2, t_2) := (\bot, \bot, \bot, \bot)$$

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* $y_2 \neq \bot$ *then* $t_1 := y_2$	m_3: *if* $y_1 \neq \bot$ *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* $\neg y_2 = y_1$

```
┌──────────────────────────┐        ┌──────────────────────────┐
│ l6: critical section 1   │        │ m6: critical section 2   │
│    (y1, t1) := (⊥, ⊥)    │        │    (y2, t2) := (⊥, ⊥)    │
└──────────────────────────┘        └──────────────────────────┘
   l7: goto l0                         m7: goto m0
      – P1 –                              – P2 –
```

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, t_1, y_2 and t_2. Also the critical sections must terminate. The program is *distributed* in the sense that each process P_i has a private variable y_i which is readable by the other process but can be written only by P_i.

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

$$\neg T = F, \qquad \neg F = T, \qquad \neg\bot \text{ is undefined.}$$

(Hence in writing $\neg y_2 = y_1$ we also imply that $y_1 \neq \bot$ and $y_2 \neq \bot$.) Protection is essentially assured by the fact that when both processes compete on the entry to the critical section, both $y_1 \neq \bot$ and $y_2 \neq \bot$. Under these assumptions, the entry conditions to the critical sections, $y_1 \neq y_2$ and $\neg y_2 \neq y_1$ respectively, cannot both be true at the same time.

When P_1 gets to l_5 it waits until $y_1 \neq y_2$ and then enters the critical section. This condition is satisfied either if $y_2 = \bot$ (since $y_1 \neq \bot$ at l_5), implying that P_2 is not currently interested in entering the critical section, or if $y_1 = \neg y_2$ (and $y_1 \neq \bot$) which implies that P_2 reached m_5 *after* P_1 got to l_5. This is because in l_1 to l_4, P_1 attempts to set $y_1 = y_2$; if now P_1 finds $y_1 = \neg y_2$ *at* l_5, it knows that P_2 changed the value of y_2 *after* P_1 last read this value. This argument is only intuitive since P_2 may have changed y_2 after P_1 last read it and yet arrive at m_5 before P_1 arrived at l_5. This is why we need a formal proof of both protection and liveness.

Symmetrically, when P_2 arrives at m_5 it waits until $\neg y_2 \neq y_1$. This can occur only if $y_1 = \bot$, implying that P_1 is not currently interested in entering the critical section, or if $y_2 = y_1$ (and $y_1 \neq \bot$) which now implies that P_1 modified the value of y_1 after P_2 last read it. This is because in m_1 to m_4, P_2 attempts to make $\neg y_2 = y_1$.

An interesting fact about the algorithm is that two groups of instructions, one consisting of $\{l_1, l_2\}$ and the other consisting of $\{l_3, l_4\}$, seem to be redundantly trying to achieve the same goal. Both groups try to make $y_1 = y_2$ if $y_2 \neq \bot$, and $y_1 \neq y_2$ otherwise. Why should we have this redundancy? The answer is that if we could perform the assignment

$$y_1 := \textit{if } y_2 = F \textit{ then } F \textit{ else } T$$

as one atomic instruction, then only one such instruction would have been necessary. Since we use an interleaving model for concurrency we have had to break this monolithic instruction into two atomic instructions such as given in l_1 and l_2. This faithfully models the possibility that y_2 could change its value before y_1 is assigned the intended value.

Such breaking is required whenever an instruction contains more than a single critical reference to a shared variable, if the interleaving model is to represent all the possible behaviors of real concurrent executions of such instructions. Consequently we break the instruction into two simpler instructions, the first fetching the value of y_2 and computing in t_1 the intended value, and the second moves t_1 into y_1.

However, now that the other process may change y_2 between these two instructions the algorithm with a single pair of such instructions is no longer correct. That is, there exists a computation that violates mutual exclusion. The critical interference point is between l_1 and l_2. By duplicating the sequence of l_1, l_2 at l_3, l_4 and similarly in P_2, we make it impossible for the other process to repeat its damaging interaction both when P_1 is at l_2 and when it is at l_4. By essentially duplicating the broken instruction twice, computations that violate mutual exclusion will be shown to be impossible.

By simple application of the initialized invariance rule I-INV, it is possible to derive the following invariants:

$$I_1:\ (t_1 \neq \bot) \equiv at\ l_{2\ldots6}$$

$$I_2:\ (y_1 \neq \bot) \equiv at\ l_{3\ldots6}$$

$$I_3:\ (t_2 \neq \bot) \equiv at\ m_{2\ldots6}$$

$$I_4:\ (y_2 \neq \bot) \equiv at\ m_{3\ldots6},$$

where $at\ l_{2\ldots6}$ stands for $at\ l_2 \vee at\ l_3 \vee \cdots \vee at\ l_6$, etc. Note that stating that I_1 is an invariant is the same as stating that $\vdash \Box[(t_1 \neq \bot) \equiv at\ l_{2\ldots6}]$.

In order to derive safety we prove the following sequence of invariants:

$$I_5:\ (y_1 = t_1) \vee at\ l_2 \vee at\ l_4$$

$$I_6:\ (y_2 = t_2) \vee at\ m_2 \vee at\ m_4$$

$$I_7:\ at\ l_{4,5} \supset [(t_2 = \bot) \vee (t_1 = t_2) \vee (t_1 = y_1)]$$

$$I_8:\ at\ m_{4,5} \supset [(t_1 = \bot) \vee (t_2 = \neg t_1) \vee (t_2 = y_2)]$$

$$I_9:\ [at\ l_{4\ldots6} \wedge at\ m_6] \supset (y_2 = t_1)$$

$$I_{10}:\ [at\ m_{4\ldots6} \wedge at\ l_6] \supset (y_1 = \neg t_2).$$

– *Invariants I_5 and I_6*. The invariants I_5 and I_6 are easy to verify since the only transitions that may cause y_1 and t_1 to differ are $l_1 \to l_2$ and $l_3 \to l_4$ and the only transitions that may cause y_2 and t_2 to differ are $m_1 \to m_2$ and $m_3 \to m_4$.

– *Invariants I_7 and I_8.* In order to verify I_7 and I_8 we observe that they hold initially since both *at* $l_{4,5}$ and *at* $m_{4,5}$ are initially false. Next, we assume that they hold at a certain instant and show that both I_7 and I_8 are preserved by each individual transition.

We show first that I_7 is preserved. Let us denote by t'_1, y'_1, t'_2, y'_2 the values of the respective variables *after* a transition. We only consider transitions that affect variables on which I_7 depends. Consider first such transitions that can be made by P_1.

$l_3 \to l_4$: If $y_2 = \bot$ then t_1 is not changed and hence by I_5, $t'_1 = t_1 = y_1$. Therefore let us consider the case that $y_2 \neq \bot$ and hence by I_3 and I_4, $t_2 \neq \bot$. We also have $t'_1 = y_2$. The following two cases are considered:

Case 1: $y_2 = t_2$. Then $t'_1 = y_2 = t_2$ satisfying the second disjunct of I_7.

Case 2: $y_2 = \neg t_2$. In view of I_6, the assumption $y_2 \neq \bot$ and I_4, P_2 can only be at m_4. From I_8, the fact that P_1 is at l_3 (hence $t_1 \neq \bot$), and the assumption $y_2 = \neg t_2$, it follows that $t_2 = \neg t_1$. We thus obtain $t'_1 = y_2 = \neg t_2 = \neg(\neg t_1) = t_1$. Since $t_1 = y_1$ while P_1 is at l_3, we obtain $t'_1 = y_1$ satisfying the third disjunct of I_7.

$l_4 \to l_5$: $y'_1 = t_1$ satisfying the third disjunct of I_7.

Next, we consider transitions of P_2 made while P_1 is at $l_{4,5}$ that affect variables appearing in I_7.

$m_1 \to m_2$: $t'_2 = \neg y_1$ since $y_1 \neq \bot$. If $y_1 = t_1$ then I_7 continues to hold. We may therefore assume that $y_1 = \neg t_1$ which leads to $t'_2 = \neg(\neg t_1) = t_1$, satisfying the second disjunct of I_7.

$m_3 \to m_4$: Similarly to the case above, since $y_1 \neq \bot$ while P_1 is at $l_{4,5}$, this transition assigns $t'_2 = \neg y_1$. By the same argument as above I_7 must still hold after this transition.

$m_6 \to m_7$: Sets t_2 to $\bot$ satisfying the first disjunct of I_7.

In a similar way we establish that I_8 is preserved under any transition initiated from a state that satisfies $I_7 \wedge I_8$. Consequently, both I_7 and I_8 are invariants.

– *Invariants I_9 and I_{10}.* Next, let us consider I_9 (and symmetrically I_{10}).

The only transition of P_1 that could affect I_9 is $l_3 \to l_4$ while P_2 is at m_6. But then $t'_1 = y_2$.

The only transition of P_2 that could affect I_9 is $m_5 \to m_6$ while P_1 is at $l_{4\ldots6}$. The fact that $m_5 \to m_6$ is possible implies that $\sim(\neg y_2 = y_1)$, i.e. $y_1 = y_2$. By I_7 either $t_1 = y_1$ or $t_1 = t_2$. In the first case we have $t_1 = y_1 = y_2$ and in the second case $t_1 = t_2 = y_2$ is ensured directly. Note that when P_2 is at m_5, $t_2 = y_2$. Thus in any case $t_1 = y_2$.

– *Safety.* The safety of this algorithm is expressed by the statement of mutual exclusion. This means that it is never the case that while P_1 is at l_6, P_2 is at m_6, i.e.,

$$\sim(at\, l_6 \wedge at\, m_6).$$

To derive safety assume a state in which both *at* l_6 and *at* m_6 is true. By I_9 and I_{10} we have that $y_2 = t_1$ and $y_1 = \neg t_2$ at the same time. By I_5 and I_6 we also have $y_1 = t_1$ and $y_2 = t_2$. This leads to both $y_1 = y_2$ and $y_1 = \neg y_2$ which is contradictory. Hence, mutual exclusion is guaranteed.

– *Liveness.* The liveness property we wish to show for this program is

$$at\, l_1 \supset \Diamond at\, l_6.$$

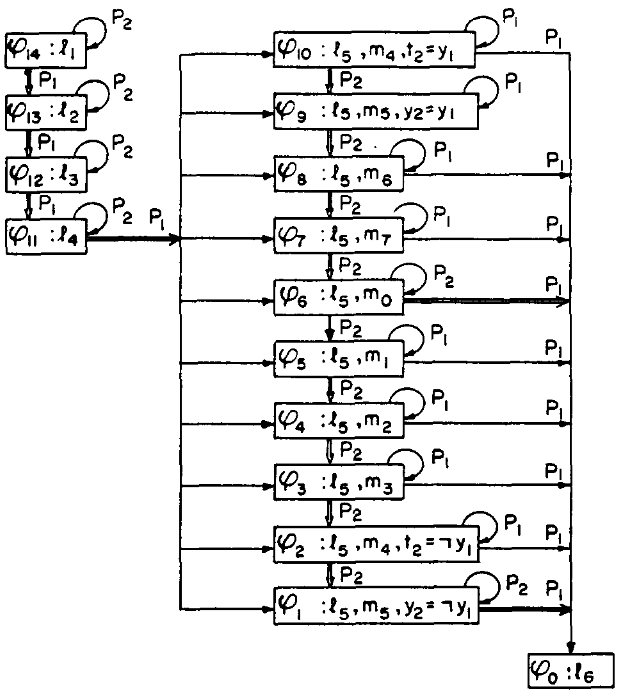


Fig. 1. Diagram proof of the Program *PF*.

In Fig. 1 we present a diagram proof for this property. In constructing the diagram we have freely used some of the invariants derived above. Observe for example the node corresponding to the assertion:

φ_6: $at\ l_5 \wedge at\ m_0$.

Here the helpful process (indicated by a double arrow $\Rightarrow$) is P_1 since we know (by I_4) that while P_2 is at $m_0, y_2 = \perp$ and while P_1 is at l_5 (by I_2) that $y_1 \neq \perp$, hence whenever P_1 is activated at l_5 it proceeds immediately to l_6, i.e., arrives at a state satisfying φ_0. In this diagram we abbreviate $at\ l_5 \wedge at\ m_0$ to l_5, m_0.

– *Precedence.* To illustrate the application of the CHAIN rule to the proof of *until* properties, consider the following precedence property:

$$[at\ l_5 \wedge \sim at\ m_{4\ldots6}] \supset [(\sim at\ m_6)\ \mathcal{U}\ (at\ l_6)].$$

It states that if P_1 arrives at l_5 *before* P_2 arrives at any location in $\{m_4, m_5, m_6\}$ then P_1 will be the first process to enter its critical section. To prove this fact we only have to consider the subdiagram of Fig. 1 consisting of nodes φ_0 to φ_7.

It is a general property of proof diagrams that if a diagram establishes conditions A to C of the J-CHAIN rule for assertions $\varphi_0, \ldots, \varphi_r$ then it also establishes these conditions for each prefix chain $\varphi_0, \ldots, \varphi_k$, $k \leq r$. Thus, conditions A to C are fulfilled for the particular prefix chain $\varphi_0, \ldots, \varphi_7$.

We may therefore conclude:

$$\left(\bigvee_{i=0}^{7} \varphi_i\right) \supset \left(\left(\bigvee_{i=1}^{7} \varphi_i\right) \mathcal{U} \varphi_0\right).$$

By examination of the relevant assertions it is easy to derive the following two implications:

$$[at\, l_5 \wedge {\sim} at\, m_{4\ldots6}] \supset \left(\bigvee_{i=0}^{7} \varphi_i\right) \quad \text{and} \quad \left(\bigvee_{i=1}^{7} \varphi_i\right) \supset {\sim} at\, m_6.$$

The three implications together yield the desired precedence property.

Example 2. The following program provides another shared-variable solution for achieving mutual exclusion without semaphores. Historically it was one of the earliest such solutions.

Program DK. The Dekker algorithm [15]:

```
                         $(t, y_1, y_2) := (1, F, F)$

$l_0$: 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 goto* $l_7$              $m_2$: *if* $y_1 = F$ *then goto* $m_7$
$l_3$: *if* $t = 1$ *then goto* $l_2$              $m_3$: *if* $t = 2$ *then goto* $m_2$
$l_4$: $y_1 := F$                              $m_4$: $y_2 := F$
$l_5$: *loop until* $t = 1$                    $m_5$: *loop until* $t = 2$
$l_6$: *goto* $l_1$                            $m_6$: *goto* $m_1$
 ┌──────────────────────────┐               ┌──────────────────────────┐
 │ $l_7$: critical section 1 │               │ $m_7$: critical section 2 │
 │        $t := 2$          │               │        $t := 1$          │
 │ $l_8$: $y_1 := F$          │               │ $m_8$: $y_2 := F$          │
 └──────────────────────────┘               └──────────────────────────┘
$l_9$: *goto* $l_0$                            $m_9$: *goto* $m_0$
     – $P_1$ –                                       – $P_2$ –
```

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 $y_2 = F$. If $y_2 = F$, P_1 proceeds immediately to its critical section. If $u_2 = T$ we have a competition between the two processes on access rights to the critical sections. This competition is resolved by using the variable t (turn) that has the value 1 if 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 can insist and so it leaves y_1 on and 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.

In order to prove safety, i.e. mutual exclusion for the DK program it is sufficient to establish the following invariants:

$$I_1:\ (y_1=T)\equiv(at\ l_{2\ldots4}\vee at\ l_{7,8})$$

$$I_2:\ (y_2=T)\equiv(at\ m_{2\ldots4}\vee at\ m_{7,8}).$$

They can be justified by considering the local transitions in P_1 and P_2 independently.

– *Safety.* Safety now follows from I_1 and I_2 as an invariant:

$$I_3:\ {\sim}at\ l_{7,8}\vee{\sim}at\ m_{7,8}.$$

The only two transitions that could falsify I_3 are:

$l_2\to l_7$ while P_2 is at $m_{7,8}$. But then by $I_2, y_2=T$ and the transition $l_2\to l_7$ is impossible.

$m_2\to m_7$ while P_1 is at $l_{7,8}$. Similarly impossible by I_1.

– *Liveness.* The liveness property of Program *DK* is given by:

$$at\ l_1\supset\Diamond at\ l_7.$$

In Fig. 2 we present a diagram proof of this property. In constructing the diagram we are aided by the previously derived invariants I_1, I_2, I_3 and the following two additional invariants:

$$I_4:\ at\ m_8\supset(t=1)$$

$$I_5:\ [at\ l_{3\ldots6}\wedge(t=2)]\supset at\ m_{1\ldots7}.$$

In particular we use I_5 when constructing the P_1-successors to node φ_{23}. In all of these successors P_1 is at l_4 and $t=2$ holds, hence by I_5, P_2 is restricted to the range of locations $m_{1\ldots7}$ which is represented by the nodes $\varphi_{16},\ldots,\varphi_{22}$.

To justify the above invariants, consider first I_4. There are two potentially falsifying transitions that have to be checked:

$m_7\to m_8$: sets t to 1.

$l_7\to l_8$ while P_2 is at m_8: This transition is impossible since by I_3 while P_2 is at m_8, P_1 cannot be at l_7.

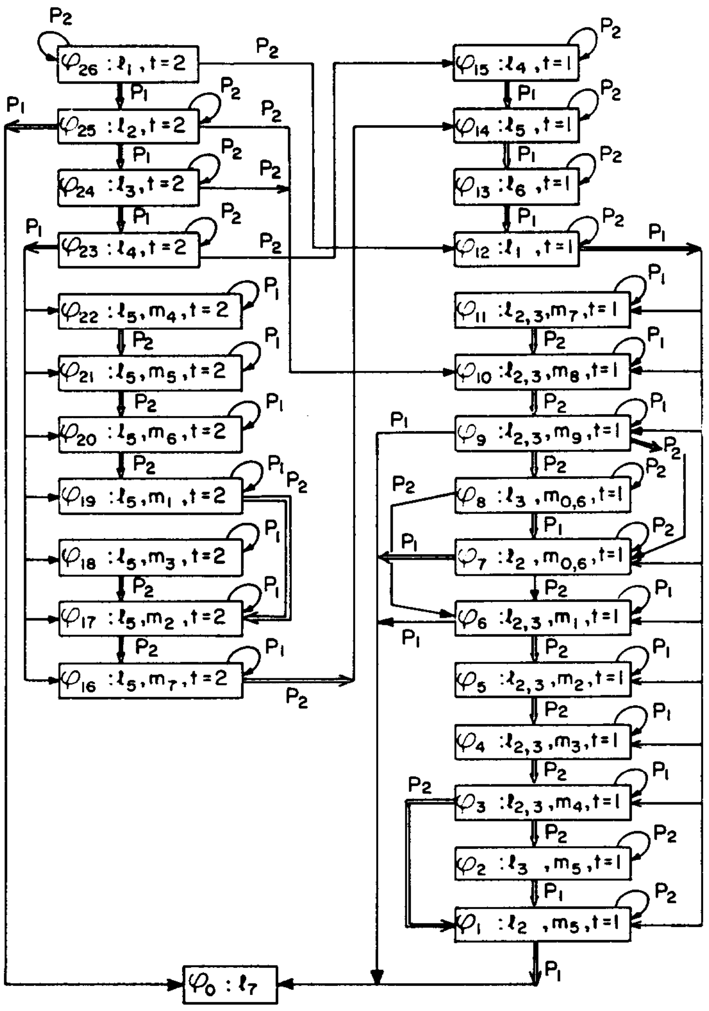
Consider next I_5. Here the potentially falsifying transitions are:

$l_2 \to l_3$ while $t=2$: This transition is possible only when $y_2 = T$ which, due to I_2, implies that P_2 is either in $m_{2...4}$ or in $m_{7,8}$. In view of I_4, P_2 cannot be at m_8 while $t=2$. Hence P_2 is restricted to $m_{2...4}$ or m_7, which is a subset of $m_{1...7}$.

$m_7 \to m_8$: Sets t to 1 and hence makes the antecedent of I_5 false.

– *Precedence.* Again we may use the full (*until*) version of the rule in order to prove some precedence properties of this program. First we can show:

$$[at\, l_{2,3} \wedge (t=1) \wedge \sim at\, m_7] \supset [(\sim at\, m_7)\mathcal{U}(at\, l_7:)].$$



$\varphi_{26}: l_1, t=2$
$\varphi_{25}: l_2, t=2$
$\varphi_{24}: l_3, t=2$
$\varphi_{23}: l_4, t=2$
$\varphi_{22}: l_5, m_4, t=2$
$\varphi_{21}: l_5, m_5, t=2$
$\varphi_{20}: l_5, m_6, t=2$
$\varphi_{19}: l_5, m_1, t=2$
$\varphi_{18}: l_5, m_3, t=2$
$\varphi_{17}: l_5, m_2, t=2$
$\varphi_{16}: l_5, m_7, t=2$
$\varphi_{15}: l_4, t=1$
$\varphi_{14}: l_5, t=1$
$\varphi_{13}: l_6, t=1$
$\varphi_{12}: l_1, t=1$
$\varphi_{11}: l_{2,3}, m_7, t=1$
$\varphi_{10}: l_{2,3}, m_8, t=1$
$\varphi_9: l_{2,3}, m_9, t=1$
$\varphi_8: l_3, m_{0,6}, t=1$
$\varphi_7: l_2, m_{0,6}, t=1$
$\varphi_6: l_{2,3}, m_1, t=1$
$\varphi_5: l_{2,3}, m_2, t=1$
$\varphi_4: l_{2,3}, m_3, t=1$
$\varphi_3: l_{2,3}, m_4, t=1$
$\varphi_2: l_3, m_5, t=1$
$\varphi_1: l_2, m_5, t=1$
$\varphi_0: l_7$
P1
P2

Fig. 2. Diagram proof of the Program *DK*.

This is established by considering the subdiagram of Fig. 2 formed out of nodes φ_0 to φ_{10}. It ensures that once P_1 is in $l_{2,3}$ with $t=1$, it will precede P_2 in getting to the critical section.

A full analysis of the number of times that P_2 may enter the critical section before P_1 does, from the time that P_1 is at l_1, leads to the following conclusions:

Once P_1 is at l_1 it will eventually get to l_2. If currently $t=1$, then, by the until property derived above, the next process to enter the critical section is P_1. If $t=2$, then in the worst case P_1 proceeds from l_2 to l_5. Meanwhile, P_2 can enter its critical section at most once before resetting t to 1. Once $t=1$, P_1 returns to l_2 and has again, by the established until property, the priority on the entry rights to the critical section. Additional overtaking, i.e., additional entries of P_2 to its critical section, may occur while P_1 is moving from l_1 to l_2 or through the sequence $l_5 \to l_6 \to l_1 \to l_2$.

It is interesting to compare our diagram proofs with the proof lattices suggested in [13] as a compact representation of proofs of liveness properties. One difference between the two representations is that an edge in our diagram corresponds to a transition that occurs in one atomic step. In the proof lattice, the fact that the node φ_i is connected by edges to $\varphi_{j_1}, \ldots, \varphi_{j_k}$ states that

$$\varphi_i \supset \Diamond(\varphi_{j_1} \vee \cdots \vee \varphi_{j_k})$$

has been established. Viewed in our framework, we may consider the proof lattice to be a proof diagram for a CHAIN rule in which premises A, B and C have been replaced by the single premise:

$$\text{D. For } i=1,\ldots,r\text{:} \quad \vdash \varphi_i \supset \Diamond\Big(\bigvee_{j<i} \varphi_j\Big).$$

The establishment of condition D for each $i>0$ is then based on the J-LIVE rule. Consequently, our representation describes the proof to a greater detail, specifying, for example, the identity of the helpful process for each assertion.

Parameterized assertions

The J-CHAIN rule assumes a finite number of links in the chain. It is quite adequate for finite state programs, i.e., programs where the variables range over finite domains. However, once we consider programs over the integers it is no longer sufficient to consider only finitely many assertions. In fact, sets of assertions of high cardinality may be needed. The obvious generalization of a finite set of assertions $\{\varphi_i \mid i=0,\ldots,r\}$ is to consider a single assertion $\varphi(\alpha)$, parameterized by a parameter α taken from a well-founded structure $(\mathscr{A}, >)$. Obviously, the most important property of our chain of assertions is that program transitions eventually lead from φ_i to φ_j with $j<i$. This property can also be stated for an arbitrary well-founded ordering. Thus a natural generalization of the chain reasoning rule is the following:

Just Well-Founded Liveness Rule: J-WELL
Let $(\mathcal{A}, >)$ be a well-founded structure.
Let $\varphi(\alpha) = \varphi(\alpha; \bar{\pi}; \bar{y})$, $\alpha \in \mathcal{A}$, be a parameterized state formula.
Let $h: \mathcal{A} \to [1 \ldots m]$ be a function identifying for each $\alpha \in \mathcal{A}$ the helpful process $P_{h(\alpha)}$ for states satisfying $\varphi(\alpha)$.

A. $\vdash P$ leads from $\varphi(\alpha)$ to $\psi \vee (\exists \beta \leqslant \alpha . \varphi(\beta))$

B. $\vdash P_{h(\alpha)}$ leads from $\varphi(\alpha)$ to $\psi \vee (\exists \beta < a . \varphi(\beta))$

C. $\vdash \varphi(\alpha) \supset [\psi \vee (\exists \beta < \alpha . \varphi(\beta)) \vee \mathit{Enabled}(P_{h(\alpha)})]$

$\vdash (\exists \alpha . \varphi(\alpha)) \supset (\exists \alpha . \varphi(\alpha)) \mathcal{U} \psi.$

We refer to h as the *helpfulness* function.

This rule can be justified by induction over arbitrary well-founded ordered sets.

Example (distributed gcd). As an illustration of the J-WELL principle we reconsider Program *DGCD* for the distributed computation of the *gcd* function.

Program DGCD. Distributed *gcd* computation:

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

l_0: while y_1 ≠ y_2 do                 m_0: while y_1 ≠ 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 -
```

In Fig. 3 we present a proof diagram of the liveness property:

$$\Diamond[\mathit{at}\ l_1 \wedge \mathit{at}\ m_1 \wedge (y_1 = y_2)]$$

for this program.

In this diagram we mix applications of the J-CHAIN rule with an application of the J-WELL rule. The J-WELL rule ensures that from $\varphi = \varphi_3$ we will eventually exit to φ_2 or to φ_1, i.e., $\psi = \varphi_1 \vee \varphi_2$. The well-founded structure that we use is that of lexicographic pairs (n, k) of which $n \in N$ is a positive integer and $k \in \{1, 2\}$. The second parameter k is determined according to whether $y_1 > y_2$ or $y_1 < y_2$. In turn it determines the helpful process. If $k = 1$, then $y_1 > y_2$, and any transition of P_1 (namely $l_0 \to l_0$) will decrement $n = y_1 + y_2$, thus decrementing the pair (n, k). On that same state, any transition of P_2 leaves y_1, y_2, and hence n and k invariant. For $k = 2$ the situation is reversed, P_2 being the helpful process.

Once we are in φ_2 or φ_1 the arrival at φ_0 is ensured by the usual application of the J-CHAIN rule. □

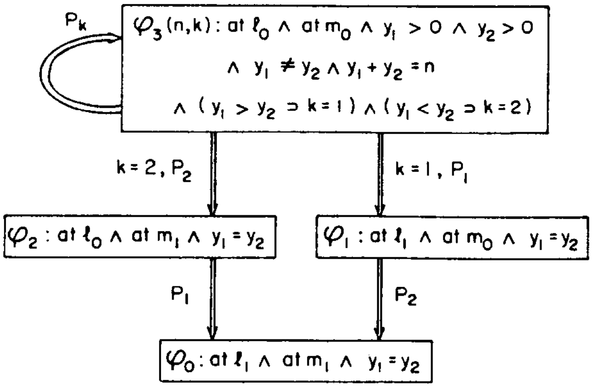

P_k
$\varphi_3(n,k)$: at $\ell_0 \wedge$ at $m_0 \wedge y_1 > 0 \wedge y_2 > 0$ $\wedge\ y_1 \neq y_2 \wedge y_1 + y_2 = n$ $\wedge\ (y_1 > y_2 \supset k = 1) \wedge (y_1 < y_2 \supset k = 2)$
$k = 2, P_2$
$k = 1, P_1$
φ_2 : at $\ell_0 \wedge$ at $m_1 \wedge y_1 = y_2$
φ_1 : at $\ell_1 \wedge$ at $m_0 \wedge y_1 = y_2$
P_1
P_2
φ_0: at $\ell_1 \wedge$ at $m_1 \wedge y_1 = y_2$

Fig. 3. Diagram proof of the Program *DGCD*.

Note that in proof diagrams containing parameterized assertions, we allow edges of the helpful process to lead back to the same node, provided that they always lead to a lower value of the well-founded parameter.

7. Rules for fair computations

Next we consider programs with semaphore instructions. For such programs the classes of just and fair computations do not necessarily coincide and we have to consider the more general concept of fair computations. Since $\mathcal{A}_F(P) \subseteq \mathcal{A}_J(P)$, any property that has been proved correct by the J-WELL rule certainly holds for all fair computations. However, the completeness of the J-WELL rule breaks down in the case of programs with semaphores; we are not always guaranteed that this rule is applicable. Hence, we propose a more general method for establishing eventuality properties under fair computations:

The basic liveness principle under the assumption of *fair computations* is given by:

Liveness Rule: LIVE
Let $\varphi(\bar{\pi}; \bar{y})$ and $\psi(\bar{\pi}; \bar{y})$ be two state formulas and P_k one of the processes

A. $\vdash P$ leads from φ to $\varphi \vee \psi$
B. $\vdash P_k$ leads from φ to ψ
C. $\vdash \varphi \supset \Diamond[\psi \vee \mathit{Enabled}(P_k)]$

$$\overline{}$$
$\vdash \varphi \supset (\varphi \mathcal{U} \psi)$

To justify the liveness rule LIVE, let σ be a fair computation such that φ is initially true. By condition A, φ will hold until ψ is realized. Assume therefore that ψ is never realized and hence φ holds in all states of σ. By condition B, P_k was never activated, since any activation of P_k from a φ-state would have realized ψ immediately. By condition C, each state, being a φ-state, is eventually followed by a state in which either ψ holds or P_k is enabled. By our assumption that ψ never occurs, the latter must be the case, i.e. P_k is enabled. Consequently, σ must be infinite, since otherwise its last state must be such that P_k is enabled on it, contradicting the definition of execution sequences being maximal. We may now repeat the argument above for every φ-state. This shows the existence of an infinite sequence of states on which P_k is enabled. Thus P_k is enabled infinitely many times on σ but never activated, contradicting our initial assumption that σ is a fair computation. Consequently, any fair computation beginning in a φ-state must contain a ψ-state.

The difference between the LIVE and the J-LIVE rule is in condition C. While the J-LIVE rule requires that the helpful process is enabled *now*, the LIVE rule only assures that it will be *eventually* enabled. An apparent advantage of the J-LIVE version of condition C is that it is *static*, i.e. contains no temporal operators. The LIVE version of condition C, in comparison is *dynamic*, i.e. is a temporal statement, having the same form as the conclusion we set out to prove: $\varphi \supset \Diamond\psi$. Two obvious questions arise: How do we prove condition C of the LIVE rule? Is there a danger of circular reasoning?

The answer to both questions lies in the observation that in establishing condition C we may ignore the process P_k. This is because as soon as it is enabled we have already arrived at a goal state (i.e., one satisfying $\psi \vee \mathit{Enabled}(P_k)$). Thus, if currently P_k is disabled, only the other processes may cause it to become enabled again; P_k can never help itself become enabled.

To emphasize this point we may rewrite condition C as:

$$\mathscr{A}_F(P-\{P_k\}) \vdash \varphi \supset \Diamond[\psi \vee \mathit{Enabled}(P_k)].$$

This means that if we consider all fair computations of the program obtained from P by omitting the process P_k, then φ guarantees the eventual realization of $\psi \vee \mathit{Enabled}(P_k)$. In the modified program we should consider as initial states all the states accessible by P. Thus, circular reasoning is avoided since we consider as a premise to our rule a simpler program with one process less than the original program.

Note that the static version of condition C always implies the dynamic version.

We may now develop the CHAIN and WELL rules in a similar way by appropriately generalizing condition C. Thus to obtain the CHAIN rule we replace condition C of the J-CHAIN rule by:

$$\vdash \varphi_i \supset \Diamond\left[\left(\bigvee_{j<i} \varphi_j\right) \vee \mathit{Enabled}(P_{k_i})\right].$$

The full WELL rule is given by:

Well-Founded Liveness Rule: WELL
Let $(\mathscr{A}, >)$ be a well-founded structure.
Let $\varphi(\alpha) = \varphi(\alpha : \bar{\pi}; \bar{y})$, $\alpha \in \mathscr{A}$, be a parameterized state formula.
Let $h : \mathscr{A} \to [1 \ldots m]$ be a function identifying for each $\alpha \in \mathscr{A}$ the helpful process $P_{h(\alpha)}$ for states satisfying $\varphi(\alpha)$.

A. $\vdash P$ leads from $\varphi(\alpha)$ to $\psi \vee (\exists \beta \leqslant \alpha . \varphi(\beta))$
B. $\vdash P_{h(\alpha)}$ leads from $\varphi(\alpha)$ to $\psi \vee (\exists \beta < \alpha . \varphi(\beta))$
C′. $\vdash \varphi(\alpha) \supset \Diamond[\psi \vee (\exists \beta < \alpha . \varphi(\beta)) \vee \mathit{Enabled}(P_{h(\alpha)})]$

$\vdash (\exists \alpha . \varphi(\alpha)) \supset (\exists \alpha . \varphi(\alpha)) \mathscr{U} \psi$

We refer to h as the *helpfulness* function.

We will proceed to illustrate first the application of the CHAIN rule and then the application of the WELL rule to proofs of liveness properties of programs with semaphores.

Example 3. This example demonstrates the application of the *chain* rule for programs with semaphores.

Program PC. Producer-consumer:

```
                    (b, s, cf, ce) := (Λ, 1, 0, n)

  l0: compute y1                          m0: request(cf)
  l1: request(ce)                         m1: request(s)
  l2: request(s)
                                        ┌──────────────────┐
┌────────────────┐                      │ m2: y2 := head(b)│
│ l3: t1 := b·y1 │                      │ m3: t2 := tail(b)│
│ l4: b := t1    │                      │ m4: b := t2      │
│ l5: release(s) │                      │ m5: release(s)   │
└────────────────┘                      └──────────────────┘
  l6: release(cf)                         m6: release(ce)
  l7: goto l0                             m7: compute using y2
                                          m8: goto m0

  – P1: Producer –                        – P2: Consumer –
```

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 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 $n > 0$.

In order to ensure the correct synchronization between the processes we use three semaphore variables: The variable s ensures that accesses to the buffer are protected and provides exclusion between the critical sections $l_{3\ldots5}$ and $m_{2\ldots5}$. 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.

– *Liveness.* Here we wish to show that

$$at\, l_1 \supset \Diamond at\, l_3.$$

We start by presenting the top-level diagram proof (Fig. 4). 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 C in the CHAIN rule. This calls for the consideration of fair computations of $P-\{P_1\}=\{P_2\}$. We thus have to construct two subproofs:

$$\mathscr{A}_F(P_2) \vdash at\, l_1 \supset \Diamond(ce>0),$$

$$\mathscr{A}_F(P_2) \vdash at\, l_2 \supset \Diamond(s>0).$$

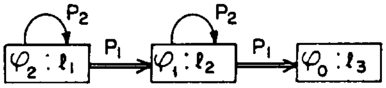


Fig. 4.

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 l_1. 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 present diagram proofs.

Consider first the diagram proof for the $at\, l_1$ case (Fig. 5). In the construction of this diagram we use some invariants which are easy to derive. The first invariant is:

$$I_1\colon\ at\, l_{3\ldots5}+at\, m_{2\ldots5}+s=1$$

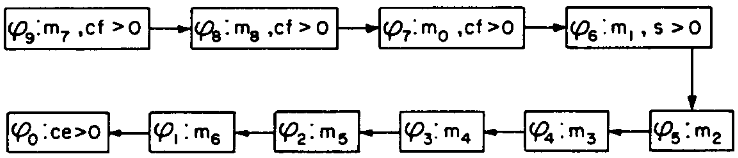


Fig. 5.

It has been used 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. The second invariant we use is

$$I_2\colon\ cf+ce+at\, l_{2\ldots6}+at\, m_{1\ldots6}=n.$$

It is used 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 the *at* l_2 case is even simpler (Fig. 6).

$$\begin{tikzcd}
\boxed{\varphi_4 : m_2} \arrow[r] & \boxed{\varphi_3 : m_3} \arrow[r] & \boxed{\varphi_2 : m_4} \arrow[r] & \boxed{\varphi_1 : m_5} \arrow[r] & \boxed{\varphi_0 : s>0}
\end{tikzcd}$$

Fig. 6.

Example 4. The following program demonstrates the application of the WELL rule for programs with semaphores.

Program BC. Binomial coefficient.

$$(y_1, y_2, y_3, y_4) := (n, 0, 1, 1)$$

```
l_0: if $y_1 = (n-k)$ then goto $l_7$
l_1: request($y_4$)
    ┌──────────────────────┐
    │ l_2: $t_1 := y_3 \cdot y_1$ │
    │ l_3: $y_3 := t_1$           │
    │ l_4: release($y_4$)         │
    └──────────────────────┘
l_5: $y_1 := y_1 - 1$
l_6: goto $l_0$
l_7: halt

        – $P_1$ –
```

```
m_0: if $y_2 = k$ then goto $m_8$
m_1: $y_2 := y_2 + 1$
m_2: loop until $y_1 + y_2 \leq n$
m_3: request($y_4$)
    ┌──────────────────────┐
    │ m_4: $t_2 := y_3/y_2$ │
    │ m_5: $y_3 := t_2$     │
    │ m_6: release($y_4$)   │
    └──────────────────────┘
m_7: goto $m_0$
m_8: halt

        – $P_2$ –
```

This is a distributed computation of 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:

$$(y_1, y_2, y_3, y_4) := (n, 0, 1, 1)$$

```
l7: if y1 = (n − k) then goto l1
l6: request(y4)
  ┌─────────────────────┐
  │ l5: t1 := y3 · y1   │
  │ l4: y3 := t1        │
  │ l3: release(y4)     │
  └─────────────────────┘
l2: y1 := y1 − 1
l8: goto l7
l1: halt

              – P1 –
```

```
m3: if y2 = k then goto m1
m2: y2 := y2 + 1
m9: loop until y1 + y2 ≤ n
m8: request(y4)
  ┌─────────────────────┐
  │ m7: t2 := y3/y2     │
  │ m6: y3 := t2        │
  │ m5: release(y4)     │
  └─────────────────────┘
m4: goto m3
m1: halt

              – P2 –
```

The liveness property we wish to prove is:

$$[at\{l_7, m_3\} \wedge (y_1, y_2, y_3, y_4) = (n, 0, 1, 1)] \supset \Diamond at\{l_1, m_1\}.$$

We derive first several invariants needed for the liveness proof:

I_1: $(at\, l_{3\ldots5} + at\, m_{5\ldots7} + y_4) = 1$

I_2: $((n-k) + at\, l_{2\ldots6}) \leqslant y_1 \leqslant n$

I_3: $0 \leqslant y_2 \leqslant (k - at\, m_2)$

I_4: $at\, l_1 \supset (y_1 = n - k)$.

For our well-founded domain we choose:

$$W = \langle N \times \{0, \ldots, 17\} \times \{0, 1\}, >_{lex} \rangle.$$

That is, the domain of triples of integers (r, s, t) such that $r \geqslant 0$, $0 \leqslant s \leqslant 17$ and $0 \leqslant t \leqslant 1$. The ordering defined on them is the lexicographic ordering on triples.

The parameterized assertion is:

$$\varphi(w; l_i, m_j; y_1, y_2) = \varphi((r, s, t); l_i, m_j; y_1, y_2):$$
$$(r = y_1 + k - y_2) \wedge (s = i + j) \wedge (t = at\, l_1).$$

Thus s is the sum of the indices of the locations of the two processes: also $t = 1$ if and only if P_1 is at l_1; otherwise $t = 0$.

The helpfulness function is:

$$h(r, s, t) = \begin{cases} P_2 & \text{if } t = 1, \\ P_1 & \text{otherwise.} \end{cases}$$

The sequence of labels was designed in such a way that moving to the next instruction will necessarily lead to a lower value of (r, s, t). 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. The parameter t has been added in order to make h dependent on $w=(r, s, t)$.

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 (κ, s, t). This is so because in such a situation we are assured by I_3, I_4 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 diagram in Fig. 7, supported by invariants I_1 to I_4.

$$\begin{tikzcd}
\boxed{\varphi_3 : m_7} \arrow[r] & \boxed{\varphi_2 : m_6} \arrow[r] & \boxed{\varphi_1 : m_5} \arrow[r] & \boxed{\varphi_0 : y_4 > 0}
\end{tikzcd}$$

Fig. 7.

8. Concluding remarks

We have presented two basic proof principles, the I-INV rule for establishing invariance properties and the WELL rule for establishing liveness properties. While we have not discussed the issues of completeness here, both are complete. We refer the reader to [11] for proof of completeness of the I-INV rule, and to [5] for the completeness of the WELL rule.

We believe that the level of detail (and formality) at which these rules are formalized leads to an optimal presentation of proofs for human readers. It summarizes the dependence on the program structure under the general "leads from φ to ψ" notion. Usually, for a particular φ and ψ, no detailed proof of this statement is needed. In more subtle cases, as presented in some of our examples, we need to consider some critical transitions in detail. The diagram representation of the proof offers even a more succinct presentation, where only the key ideas are pointed out.

Closely related approaches to well-founded methods for liveness which concentrate on nondeterministic rather than concurrent programs are described in [1] and [2].

Acknowledgment

We wish to thank Bengt Jonsson and Frank Yellin for careful and critical reading of the manuscript.

References

[1] K.R. Apt and E.R. Olderog, Proof rules dealing with fairness, in: D. Kozen, Ed., *Logics of Programs*, Lecture Notes in Computer Science, **131** (Springer, Berlin, 1982) 1–8.

[2] O. Grumberg, N. Francez, J.A. Makowsky and W.P. deRoever, A proof rule for fair termination of guarded commands, in: J.W. DeBakker and J.C. Van Vliet, Eds., *Algorithmic Languages* (North-Holland, Amsterdam, 1981) 399–416.

[3] R.M. Keller, Formal verification of parallel programs, *Comm. ACM* **19** (7) (1976) 371–384.

[4] L. Lamport, Proving the correctness of multiprocess programs, *IEEE Trans. Software Engrg.* 3(2) (1977) 125–143.

[5] D. Lehmann, 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, Berlin, 1981) 264–277.

[6] Z. Manna, *Mathematical Theory of Computation* (McGraw-Hill, New York, 1974).

[7] Z. Manna and A. Pnueli, Verification of concurrent programs: The temporal framework, in: R.S. Boyer and J.S. Moore, Eds., *The Correctness Problem in Computer Science*, International Lecture Series in Computer Science (Academic Press, London, 1982) 215-273.

[8] Z. Manna and A. Pnueli, Verification of concurrent programs: Temporal proof principles, in: D. Kozen, Ed., *Logic of Programs*, Lecture Notes in Computer Science **131** (Springer, Berlin, 1982) 200–252.

[9] Z. Manna and A. Pnueli, Verification of concurrent programs: A temporal proof system, *Proc. 4th School on Advanced Programming*, Amsterdam, Holland (1982) 163–255.

[10] Z. Manna and A. Pnueli, How to cook a temporal proof system for your pet language, *Proc. Symposium on Principles of Programming Languages*, Austin, Texas (1983) 141–154.

[11] Z. Manna and A. Pnueli, Proving precedence properties—the temporal way, in: *Automata Languages and Programming*, Lecture Notes in Computer Science **154** (Springer, Berlin, 1983) 491-512.

[12] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, *Acta Informat.* **6**(4) (1976) 319-340.

[13] S. Owicki and L. Lamport, Proving liveness properties of concurrent programs, *ACM Trans. Programming Languages and Systems* 4(3) (1982) 455–495.

[14] A. Pnueli, The temporal logic of programs, *Proc. 18th Symposium on Foundations of Computer Science*, Providence, RI (1977) 46–57.

[15] E.W. Dijkstra, Co-operating sequential processes, in: F. Genuys, Ed., *Programming Languages* (Academic Press, New York, 1968) 43-112.

[16] G.L. Peterson and M.J. Fischer, Economical solutions for the critical section problem in a distributed system, *Proc. 9th ACM Symposium on Theory of Computing*, Boulder, CO (1977) 91-97.