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:

$$\Box \psi \text{ or } \varphi \Rightarrow \Box \psi.$$  

Such a formula, stated for a program $P$, says that every computation of $P$ continuously satisfies $\psi$. In the case of the second form it states that whenever $\varphi$ becomes true, $\psi$ 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:

$$\Diamond \psi \lor \varphi \Rightarrow \Diamond \psi.$$ 

In both cases these formulas guarantee the occurrence of some event $\psi$; in the first case unconditionally and in the second case conditional on an earlier occurrence of the event $\varphi$. 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:

$$\chi \mathcal{U} \psi \lor \varphi \Rightarrow \chi \mathcal{U} \psi.$$ 

In both cases the formulas again guarantee the occurrence of the event $\psi$, but they also ensure that from now until that occurrence, $\chi$ 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

\[
(m_1, m_2) >_{lex} (n_1, n_2) \overset{\text{def}}{=} (m_1 >_1 n_1) \text{ or } (m_1 = n_1 \text{ and } m_2 >_2 n_2)
\]
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 \(\psi\) will occur in any computation. In the simple approach we require a descent of the parameter \(w\) on every computation step until \(\psi\) 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 \(\psi\) or there exists a \(w'\) such that \(w > w'\) and \(Q(s'; w')\) holds. Thus, any computation that fails to achieve \(\psi\) 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

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

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

\[
Q(y_1, y_2; n): (y_1 > 0) \land (y_2 > 0) \land (y_1 + y_2 = n).
\]
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 $\psi$ 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:

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

\[l_0: \text{while } y_1 \neq y_2 \text{ do} \quad m_0: \text{while } y_1 \neq y_2 \text{ do}\]

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

\[l_1: \text{halt} \quad m_1: \text{halt}\]

\[- P_1 - \quad - P_2 -\]

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:

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

Each process \( P_i \) is represented as a transition graph with locations (nodes) labelled by elements of \( \mathcal{L}_i = \{l_i^0, \ldots, l_i^r\} \). The edges in the graph are labelled by guarded commands of the form \( c(y) \Rightarrow [\bar{y} := f(\bar{y})] \), whose meaning is that if \( c(\bar{y}) \) is true the edge may be traversed while replacing \( \bar{y} \) by \( f(\bar{y}) \).

Let \( l, l_1, \ldots, l_k \in \mathcal{L}_j \) be locations in process \( P_j \);

We define \( E_i(\bar{y}) = c_1(\bar{y}) \vee \cdots \vee c_k(\bar{y}) \) 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 \( E_i = \text{true} \). 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 $E_i = \text{false}$. 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 $l_i^t$.

Any location $l$ such that the exit condition $E_l(y)$ 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($y$) whose transition diagram is:

![Transition Diagram for Request Instruction]

Note that $E_l(y) = (y > 0)$. The request instruction is used in order to reserve a resource, where $y$ 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($y$) instruction. It is used to release one unit of a reserved resource. Its transition diagram is:

![Transition Diagram for Release Instruction]

The location of the release instruction is regular.

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

Let $s = (l_1', \ldots, l_m'; \eta)$ be a state. We say that process $P_i$ is enabled on $s$ if $E_i(\eta)$ 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 $s = (l_1', \ldots, l_m'; \eta)$ and $\tilde{s} = (\tilde{l}_1', \ldots, \tilde{l}_m'; \tilde{\eta})$ be two states of $P$. Let $\tau$ be a transition in $P_i$ of the form:

![Transition Diagram for Computation Step]

such that $c(\tilde{\eta}) = \text{true}$, $\tilde{\eta} = f(\tilde{\eta})$, 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 $\tau$ (a $\tau$-successor for short), and write:

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

\[
    s \xrightarrow{P_i} s'.
\]

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

\[
    \sigma: s_0 \xrightarrow{P_{i_0}} s_1 \xrightarrow{P_{i_1}} s_2 \xrightarrow{P_{i_2}} s_3 \xrightarrow{} \cdots
\]

satisfying the following conditions:

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

\[
    s_0 = (\bar{I}_0; g(\bar{\xi})),
\]

where \( \bar{I}_0 = (I_{0_1}, \ldots, I_{0_n}) \) is the vector of initial locations in all the processes. The values \( g(\bar{\xi}) \) are the initial values assigned to the \( \xi \) variables for the input \( \bar{\xi} \).

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

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

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

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 \( \mathcal{A}(P) \). A state \( s \) is defined to be accessible by the program \( P \) if it appears in an admissible computation.

By \( D_r \), a computation \( \sigma \) is fair if there is no process \( P_i \) such that \( P_i \) is enabled an infinite number of times in \( \sigma \), 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 \( \mathcal{A}_F(P) \).

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) Justice. Every \( P \), which is continuously enabled beyond a certain state \( s \) in \( \sigma \), must be activated infinitely many times in \( \sigma \).
We refer to the class of all suffixes of computations that satisfy conditions $A$, $B$, $C$ and $D_i$ for some input $\xi$, 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 $\sigma$ be an infinite fair computation. Let $P_i$ be any process that is continuously enabled beyond a certain state in $\sigma$. Then, $P_i$ 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 $\mathcal{A}_F(P)$ and $\mathcal{A}_J(P)$ may be strict, consider the following program which is the simplest program modelling mutual exclusion:

\begin{align*}
   y &:= 1 \\
   l_0: \text{request}(y) &\quad m_0: \text{request}(y) \\
   l_1: \text{release}(y) &\quad m_1: \text{release}(y) \\
   l_2: \text{goto } l_0 &\quad m_2: \text{goto } m_0 \\
   - P_1 - &\quad - P_2 -
\end{align*}

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:

\begin{align*}
   \sigma: (l_0, m_0; 1) &\xrightarrow{p_1} (l_1, m_0; 0) &\xrightarrow{p_1} &\xrightarrow{p_1} (l_2, m_0; 1) \\
   (l_0, m_0; 1) &\xrightarrow{p_1} (l_1, m_0; 0) &\xrightarrow{p_1} &\xrightarrow{p_1} (l_2, m_0; 1) \xrightarrow{p_1} \ldots
\end{align*}

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 $\sigma$ 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:

\[ \sigma: s_0 \xrightarrow{P_0} s_1 \xrightarrow{P_1} \cdots \xrightarrow{P_{\gamma-1}} s_\gamma. \]

If the final state \( s_\gamma \) of a convergent computation is of the form \( s_\gamma = (l_1', \ldots, l_m'; \tilde{\eta}) \), where each \( l_i' \) 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** \( \tilde{\pi} = (\pi_1, \ldots, \pi_m) \). Each \( \pi_i \) ranges over \( \mathcal{L}_i \), and assumes the value \( l'_i \) in a state \( s = (l_1', \ldots, l_i', \ldots, l_m'; \tilde{\eta}) \).

Thus we may describe a state \( s = (\tilde{l}; \tilde{\eta}) \) by saying that in this state \( \tilde{\pi} = \tilde{l} \) and \( \tilde{\eta} = \tilde{\eta} \).

A **state formula** \( Q = Q(\tilde{\pi}; \tilde{\eta}) \) is any first-order formula. It is built from terms and predicates over the location and program variables \( (\tilde{\pi}; \tilde{\eta}) \) 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 \( \tilde{\xi} \)-computation, the values of \( \bar{x} \) in \( s \) are necessarily \( \tilde{\xi} \).

A state \( s \) that satisfies a state formula \( \varphi \) is referred to as a **\( \varphi \)-state**.

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

Let the following be a transition \( \tau \) in process \( P_i \):

\[ l' \xrightarrow{c(\tilde{\eta}) \rightarrow [\tilde{\eta} := f(\tilde{\eta})]} \tilde{l}' \]

Let \( \varphi(\tilde{\pi}; \tilde{\eta}) \) and \( \psi(\tilde{\pi}; \tilde{\eta}) \) be two state formulas. We say:

- **The transition** \( \tau \) **leads from** \( \varphi \) **to** \( \psi \), if every \( \tau \)-successor of an accessible \( \varphi \)-state is a \( \psi \)-state. Thus, if \( \delta = (l_1', \ldots, \tilde{l}, \ldots, l_m'; \tilde{\eta}) \) is a \( \tau \)-successor of the accessible state \( (l_1', \ldots, l_i', \ldots, l_m'; \tilde{\eta}) \), which of course implies that \( c(\tilde{\eta}) = \text{true} \) and \( \tilde{\eta} = f(\tilde{\eta}) \), then the following implication must be true:

\[ \varphi(l_1', \ldots, l_i', \ldots, l_m'; \tilde{\eta}) \Rightarrow \psi(l_1', \ldots, \tilde{l}', \ldots, l_m'; \tilde{\eta}). \]

One way of establishing that \( \tau \) leads from \( \varphi \) to \( \psi \) is to show the general validity of the following implication:

\[ [\varphi(l_1', \ldots, l_i', \ldots, l_m'; \tilde{\eta}) \land c(\tilde{\eta})] \Rightarrow \psi(l_1', \ldots, \tilde{l}', \ldots, l_m'; f(\tilde{\eta})) \]

for every choice of \((l_1', \ldots, l_{i-1}', l_{i+1}', \ldots, l_m') \in \mathcal{L}_i \times \cdots \times \mathcal{L}_{i-1} \times \mathcal{L}_{i+1} \times \cdots \times \mathcal{L}_m\). 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 $\varphi$ to $\psi$, if every transition $\tau$ in $P_i$ leads from $\varphi$ to $\psi$.
- A program $P$ leads from $\varphi$ to $\psi$, if every process $P_i$ leads from $\varphi$ to $\psi$.

In the following, when we present a formal or an informal argument that establishes the fact that a process $P_i$ leads from $\varphi$ to $\psi$, 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 $\varphi$ to $\psi$.

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

$$\sigma: \quad s_0 \xrightarrow{P_0} s_1 \xrightarrow{P_1} \cdots$$

we associate the sequence of states

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

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 $\text{at } l \implies (y_1 = y_2)$ 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 ($\neg, \land, \lor, \Rightarrow, \equiv$) and quantifiers ($\forall, \exists$) of first-order logic with four temporal operators, called respectively:

- $\Box$: always (henceforth)
- $\Diamond$: sometimes (eventually)
- $\bigcirc$: nexttime
- $\forall$: until
The interpretation of general temporal formulas over (computation) sequences is defined as follows:

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

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

For a state formula $\varphi$,

$$\sigma \models \varphi,$$

i.e., $\sigma$ satisfies $\varphi$, if and only if $s_0$ (the first state of $\sigma$) satisfies $\varphi$.

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

$$\sigma \models (\varphi_1 \lor \varphi_2)$$

if and only if $\sigma \models \varphi_1$ 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 $\sigma$ by removing the first $k$ elements, i.e.

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

Then:

$$\sigma \models \Box \varphi$$

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

$$\sigma \models \Diamond \varphi$$

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

$$\sigma \models \lozenge \varphi$$

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

$$\sigma \models \varphi \mathcal{U} \psi$$

if and only if $\exists k (0 \leq k \leq l(\sigma))$ such that $\sigma^{(k)} \models \psi$

and $\forall i, 0 \leq i < k, \sigma^{(i)} \models \varphi$.

For the simple cases that $\varphi$ and $\psi$ are state formulas the general definitions above can be given the following intuitive interpretation:

$\sigma$ satisfies $\Box \varphi$ if and only if all states in $\sigma$ satisfy $\varphi$

$\sigma$ satisfies $\Diamond \varphi$ if and only if some state in $\sigma$ satisfies $\varphi$

$\sigma$ satisfies $\lozenge \varphi$ if and only if the second state in $\sigma$ satisfies $\varphi$

$\sigma$ satisfies $\varphi \mathcal{U} \psi$ if and only if some state $s'$ in $\sigma$ satisfies $\psi$ and all the states until $s'$ (excluding $s'$) satisfy $\varphi$.

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

Some more complicated combinations are very useful. For example,

- The formula

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

5. The invariance principle

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

$$\text{Invariance Rule: INV}$$

Let $\varphi(\bar{\pi}; \bar{\gamma})$ be a state formula

$$\vdash P \text{ leads from } \varphi \text{ to } \varphi$$

$$\vdash \varphi \Rightarrow \Box \varphi$$

The form of this rule, which will be used throughout the paper, states that if the premise, "$P$ leads from $\varphi$ to $\varphi$", has been established, then the consequence, "$\varphi \Rightarrow \Box \varphi$", 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 $\varphi$ to $\varphi$. 
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 \( \varphi \). Since all subsequent states are derived from previous states by atomic actions of the program \( P \), all of which have been shown to preserve \( \varphi \), \( \varphi \) must be an invariant of the computation.

With the addition of an extra premise, guaranteeing that all initial states satisfy \( \varphi \), we can conclude the unconditional invariance of \( \varphi \) over all admissible computations.

---

**Example (semaphore variables).** A *semaphore variable* 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 \( \text{request}(y) \) and \( \text{release}(y) \) 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 \( \varphi \) initially. Consider next the instructions that can modify \( y \). Since \( y \) is a semaphore variable, the only such instructions are \( \text{request}(y) \) and \( \text{release}(y) \).

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

\[
[(y \geq 0) \land (y > 0)] \Rightarrow (y - 1) \geq 0,
\]

which is always true.

A release instruction is equivalent to \( y := y + 1 \). It certainly leads from \( \varphi \) to \( \varphi \) since

\[
(y \geq 0) \Rightarrow (y + 1 \geq 0).
\]
Obviously, all the other transitions do not modify \( y \) and hence lead from \( y \geq 0 \) to \( y > 0 \). Thus all the premises to the L-INV rule are established, and it follows that in any accessible state \( y \geq 0 \).

Earlier we indicated that one way to establish "\( \tau \) leads from \( \varphi \) to \( \psi \)" was by proving a verification condition appropriate for \( \tau \). 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:

\[
\text{Accessibility Rule: ACC}
\]

Let \( \varphi, \chi \) and \( \psi \) be state formulas

\[
\vdash \neg \Diamond \varphi
\]

\[
\vdash \tau \text{ leads from } (\varphi \land \chi) \text{ to } (\varphi \supset \psi)
\]

\[
\vdash \tau \text{ leads from } \chi \text{ to } \psi
\]

To justify the rule, consider an accessible \( \chi \)-state \( s \). Since it is accessible it must satisfy the invariant \( \varphi \) and is therefore also a \( (\varphi \land \chi) \)-state. Let \( \bar{s} \) be any \( \tau \)-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 \( \varphi \). Consequently it is also a \( \psi \)-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 \( \mathcal{A}_j(P) \). This will suffice for proving liveness properties of programs without semaphore instructions.

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

\[
\text{Just Liveness Rule: J-LIVE}
\]

Let \( \varphi(\pi; \tilde{y}) \) and \( \psi(\pi; \tilde{y}) \) be two state formulas and \( P_k \) be one of the processes

A. \( \vdash P \text{ leads from } \varphi \text{ to } \varphi \lor \psi \)

B. \( \vdash P_k \text{ leads from } \varphi \text{ to } \psi \)

C. \( \vdash \varphi \supset [\psi \lor \text{Enabled}(P_k)] \)

\( \vdash \varphi \supset (\varphi \supset \psi) \)
Note that the conclusion is somewhat stronger than simple liveness and guarantees not only the eventual occurrence of $\psi$ but that $\varphi$ will continuously hold until then. It implies $\varphi \Rightarrow \Diamond \psi$.

To establish the validity of the rule, suppose that conditions A to C hold. Let $\sigma$ be a just computation such that initially $\varphi$ holds but, contrary to our conclusion, $\psi$ is never realized. By condition A, the only way out of $\varphi$ is to achieve $\psi$. Hence $\varphi$ must be continuously true all along $\sigma$. By condition C, the 'helpful' process $P_k$ is continuously enabled throughout $\sigma$. Consequently $\sigma$ cannot contain a terminal state and must therefore be infinite. By justice, eventually $P_k$ will be activated from a $\varphi$-state. By condition B, this would lead to $\psi$, contradicting our assumption that $\psi$ 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 $\varphi_0$, and the desired goal is $\psi = \varphi_0$. We then repeatedly apply the $J$-LIVE principle to show that being at $\varphi_i$, $0 < i \leq r$, we eventually get to $\varphi_{i-1}$, i.e. $\varphi_i \Rightarrow \Diamond \varphi_{i-1}$. This of course establishes that being at $\varphi_n$ or as a matter of fact at any $\varphi_i$, $0 \leq i \leq r$, we are guaranteed to eventually achieve $\varphi_0$. 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 $i = 1, \ldots, r$

$$\vdash P \text{ leads from } \varphi_i \text{ to } \left( \bigvee_{j=i}^{r} \varphi_j \right)$$

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:

$$\vdash \varphi_i \Rightarrow \left[ \left( \bigvee_{j<i} \varphi_j \right) \lor \text{Enabled}(P_{k_i}) \right]$$

$$\vdash \left( \bigvee_{i=0}^{r} \varphi_i \right) \Rightarrow \left( \bigvee_{i=1}^{r} \varphi_i \right) \cup \varphi_0$$

*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 $\varphi_i$) and $s_j$ (satisfying $\varphi_j$) and a process $P_i$ such that $s_i \rightarrow P_i s_j$, we draw an edge $\rightarrow$ from the node $\varphi_i$ to the node $\varphi_j$ and label it by $P_i$, the process responsible for the transition.

$\varphi_i \rightarrow P_i \rightarrow \varphi_j$

All edges corresponding to the helpful process $P_k$, are drawn as double arrows $\Rightarrow$

$\varphi_i \Rightarrow P_k \Rightarrow \varphi_j$

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 $\varphi_i$-state, for $i > 0$, satisfies some $\varphi_j$, $j \geq 0$.
- For every edge connecting $\varphi_i$ to $\varphi_j$ we must have $i \geq j$.
- For every edge connecting $\varphi_i$ to $\varphi_j$ and labelled by $P_k$, we must have $i > j$.
- For every accessible state $s$, if $i > 0$ is the lowest index such that $s$ satisfies $\varphi_i$, then $P_k$ 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
  
  $l_1$: $t_1 := \text{if } y_2 = F \text{ then } F \text{ else } T$

- $l_2$: $y_1 := t_1$

- $l_3$: $\text{if } y_2 \neq \bot \text{ then } t_1 := y_2$

- $l_4$: $y_1 := t_1$

- $l_5$: loop while $y_1 = y_2$

- $m_0$: noncritical section 2

- $m_1$: $t_2 := \text{if } y_1 = T \text{ then } F \text{ else } T$

- $m_2$: $y_2 := t_2$

- $m_3$: $\text{if } y_1 \neq \bot \text{ then } t_2 := \neg y_1$

- $m_4$: $y_2 := t_2$

- $m_5$: loop while $\neg y_2 = y_1$
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 y₁, t₁, y₂ and t₂. Also the critical sections must terminate. The program is distributed in the sense that each process Pᵢ has a private variable yᵢ which is readable by the other process but can be written only by Pᵢ.

The basic idea of the protection mechanism of this program is that when competing for the access rights to the critical sections, P₁ attempts to make y₁ = y₂ in statements l₁ to l₄ while P₂ attempts to make ¬y₂ = y₁ in statements m₁ to m₄. The synchronization variables y₁ and y₂ range over the set {⊥, F, T}, where ⊥ signifies no interest in entering the critical section. The partial operator ¬ is defined by

¬T = F, ¬F = T, ¬⊥ is undefined.

(Hence in writing ¬y₂ = y₁ we also imply that y₁ ≠ ⊥ and y₂ ≠ ⊥.) Protection is essentially assured by the fact that when both processes compete on the entry to the critical section, both y₁ ≠ ⊥ and y₂ ≠ ⊥. Under these assumptions, the entry conditions to the critical sections, y₁ ≠ y₂ and ¬y₂ ≠ y₁ respectively, cannot both be true at the same time.

When P₁ gets to l₁ it waits until y₁ ≠ y₂ and then enters the critical section. This condition is satisfied either if y₂ = ⊥ (since y₁ ≠ ⊥ at l₂), implying that P₂ is not currently interested in entering the critical section, or if y₁ = ¬y₂ (and y₁ ≠ ⊥) which implies that P₂ reached m₅ after P₁ got to l₅. This is because in l₁ to l₄, P₁ attempts to set y₁ = y₂; if now P₁ finds y₁ = ¬y₂ at l₅, it knows that P₂ changed the value of y₂ after P₁ last read this value. This argument is only intuitive since P₂ may have changed y₂ after P₁ last read it and yet arrive at m₅ before P₁ arrived at l₅. This is why we need a formal proof of both protection and liveness.

Symmetrically, when P₂ arrives at m₅ it waits until ¬y₂ ≠ y₁. This can occur only if y₁ = ⊥, implying that P₁ is not currently interested in entering the critical section, or if y₂ = y₁ (and y₁ ≠ ⊥) which now implies that P₁ modified the value of y₁ after P₂ last read it. This is because in m₁ to m₄, P₂ attempts to make ¬y₂ = y₁.

An interesting fact about the algorithm is that two groups of instructions, one consisting of \{l₁, l₂\} and the other consisting of \{m₅, l₅\}, seem to be redundantly trying to achieve the same goal. Both groups try to make y₁ = y₂ if y₂ ≠ ⊥, and y₁ ≠ y₂ otherwise. Why should we have this redundancy? The answer is that if we could perform the assignment

\[ y₁ := \text{if } y₂ = F \text{ then } F \text{ else } T \]
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 \( \text{I-INV} \), it is possible to derive the following invariants:

\[
\begin{align*}
I_1: \quad & (t_1 \neq \bot) \Rightarrow (a \wedge t_1) \\
I_2: \quad & (y_1 \neq \bot) \Rightarrow (a \wedge t_1) \\
I_3: \quad & (t_2 \neq \bot) \Rightarrow (a \wedge t_1) \\
I_4: \quad & (y_2 \neq \bot) \Rightarrow (a \wedge t_1) 
\end{align*}
\]

where \( a \wedge t \) stands for \( a \wedge t \wedge a \wedge t \wedge \cdots \wedge a \wedge t \), etc. Note that stating that \( I_1 \) is an invariant is the same as stating that \( \vdash \square[(t_1 \neq \bot) \Rightarrow (a \wedge t_1)] \).

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

\[
\begin{align*}
I_5: \quad & (y_1 = t_1) \vee (a \wedge t_1) \\
I_6: \quad & (y_2 = t_2) \vee (a \wedge t_4) \\
I_7: \quad & (a \wedge t_4) \Rightarrow [(t_2 = \bot) \vee (t_1 = t_2) \vee (t_1 = y_1)] \\
I_8: \quad & (a \wedge t_4) \Rightarrow [(t_2 = \bot) \vee (t_2 = \neg t_1) \vee (t_2 = y_2)] \\
I_9: \quad & [(a \wedge t_4) \wedge (a \wedge t_6)] \Rightarrow (y_2 = t_1) \\
I_{10}: \quad & [(a \wedge t_4) \wedge (a \wedge t_6)] \Rightarrow (y_1 = \neg t_2).
\end{align*}
\]

- **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 \( l_1 \rightarrow l_2 \) and \( l_3 \rightarrow l_4 \) and the only transitions that may cause \( y_2 \) and \( t_2 \) to differ are \( m_1 \rightarrow m_2 \) and \( m_3 \rightarrow m_4 \).
- **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 $t', y', t', y'$ 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 $P_1$.

$l_3 \rightarrow l_4$: If $y_3 = \perp$ then $t_3$ is not changed and hence by $I_5$, $t'_3 = t_3 = y_1$. Therefore let us consider the case that $y_3 \neq \perp$ and hence by $I_3$ and $I_4$, $t_3 \neq \perp$. We also have $t'_3 = y_2$. The following two cases are considered:

Case 1: $y_2 = y_3$. Then $t'_3 = y_2 = t_3$ satisfying the second disjunct of $I_7$.

Case 2: $y_2 = \neg t_2$. In view of $I_6$, the assumption $y_3 \neq \perp$ and $I_4$, $P_2$ can only be at $m_4$. From $I_8$, the fact that $P_1$ is at $l_1$ (hence $t_1 \neq \perp$), and the assumption $y_3 = \neg t_3$, it follows that $t_3 = \neg t_4$. We thus obtain $t'_3 = y_2 = \neg t_2 = \neg(\neg t_3) = t_1$. Since $t_1 = y_1$ while $P_1$ is at $l_3$, we obtain $t'_3 = y_1$ satisfying the third disjunct of $I_7$.

$l_4 \rightarrow l_5$: $y'_5 = t_5$ 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 $I_7$.

$m_1 \rightarrow m_2$: $t'_2 = \neg y_1$ since $y_1 \neq \perp$. 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 $I_7$.

$m_3 \rightarrow m_4$: Similarly to the case above, since $y_1 \neq \perp$ 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.

$m_6 \rightarrow m_7$: Sets $t_2$ to $\perp$ 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 $l_3 \rightarrow l_4$ while $P_2$ is at $m_6$. But then $t'_3 = y_2$.

The only transition of $P_2$ that could affect $I_9$ is $m_5 \rightarrow m_6$ while $P_1$ is at $l_{4,6}$. The fact that $m_5 \rightarrow m_6$ 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 $y_2 = t_4$ 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

$$at l_1 \Rightarrow \Diamond at l_6.$$
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:

$$\varphi_6: \text{at } l_5 \land \text{at } m_0.$$  

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 $m_0$, $y_2 = \bot$ and while $P_1$ is at $l_5$ (by $I_2$) that $y_1 \neq \bot$, hence whenever $P_1$ is activated at $l_5$ it proceeds immediately to $l_6$, i.e., arrives at a state satisfying $\varphi_0$. In this diagram we abbreviate $\text{at } l_5 \land \text{at } m_0$ 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:

$$[\text{at } l_5 \land \neg \text{at } m_{4..6}] \supset [\neg (\text{at } m_6) \cup (\text{at } l_6)].$$

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 $\varphi_0$ to $\varphi_7$.

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:

\[ (\bigvee_{i=0}^7 \varphi_i) \vdash \left( \bigvee_{i=1}^7 \varphi_i \right) \cup \varphi_0. \]

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

\[ [at l_5 \land \neg at m_{6\ldots6}] \vdash \left( \bigvee_{i=0}^7 \varphi_i \right) \quad \text{and} \quad \left( \bigvee_{i=1}^7 \varphi_i \right) \vdash \neg at m_6. \]

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]:

\( (t, y_1, y_2) := (1, F, F) \)

\( l_0: \text{noncritical section 1} \)

\( l_1: \quad y_1 := T \)

\( l_2: \quad \text{if } y_2 = F \text{ then goto } l_7 \)

\( l_3: \quad \text{if } t = 1 \text{ then goto } l_2 \)

\( l_4: \quad y_1 := F \)

\( l_5: \quad \text{loop until } t = 1 \)

\( l_6: \quad \text{goto } l_1 \)

\( m_6: \text{noncritical section 2} \)

\( m_1: \quad y_2 := T \)

\( m_2: \quad \text{if } y_1 = F \text{ then goto } m_7 \)

\( m_3: \quad \text{if } t = 2 \text{ then goto } m_2 \)

\( m_4: \quad y_2 := F \)

\( m_5: \quad \text{loop until } t = 2 \)

\( m_6: \quad \text{goto } m_1 \)

\( m_7: \text{critical section 1} \)

\( t := 2 \)

\( l_5: \quad y_1 := F \)

\( l_6: \quad \text{goto } l_0 \)

\( - P_1 - \)

\( l_7: \text{critical section 1} \)

\( t := 2 \)

\( l_5: \quad y_1 := F \)

\( l_6: \quad \text{goto } l_0 \)

\( - P_1 - \)

\( m_7: \text{critical section 2} \)

\( t := 1 \)

\( m_8: \quad y_2 := F \)

\( m_9: \quad \text{goto } m_0 \)

\( - P_2 - \)

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 \( y_2 = T \) 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:

\[
I_1: \ (y_1 = T) \equiv (at \ l_{2..4} \lor \ at \ l_{7,8})
\]

\[
I_2: \ (y_2 = T) \equiv (at \ m_{2..4} \lor \ at \ m_{7,8}).
\]

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:

\[
I_3: \ 
eg at \ l_{7,8} \lor \neg at \ m_{7,8}.
\]

The only two transitions that could falsify \( I_3 \) are:

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

\( m_2 \rightarrow m_7 \) while \( P_1 \) is at \( l_{7,8} \). Similarly impossible by \( I_1 \).

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

\[
at \ l_1 \Rightarrow \Box at \ l_7.
\]

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:

\[
I_4: \ at \ m_8 \Rightarrow (t = 1)
\]

\[
I_5: \ [at \ l_{3..6} \land (t = 2)] \Rightarrow at \ m_{1..7}.
\]

In particular we use \( I_5 \) when constructing the \( P_1 \)-successors to node \( \varphi_{23} \). 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 \( m_{1..7} \) 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:

- \( m_7 \rightarrow m_8 \): sets \( t \) to 1.
- \( l_7 \rightarrow l_6 \) while \( P_2 \) is at \( m_6 \): This transition is impossible since by \( I_5 \) while \( P_2 \) is at \( m_8, P_1 \) cannot be at \( l_7 \).
Consider next $I_5$. Here the potentially falsifying transitions are:

$\ell_2 \rightarrow \ell_1$ 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 $m_{1..7}$.

$m_7 \rightarrow m_8$: 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:

$$[\text{at } l_{2,3} \land (t = 1) \land \neg \text{at } m_7] \Rightarrow [\neg (\text{at } m_7) \cup (\text{at } l_7)].$$
This is established by considering the subdiagram of Fig. 2 formed out of nodes \( \varphi_0 \) to \( \varphi_{10} \). 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 \( l_t \), leads to the following conclusions:

Once \( P_1 \) is at \( l_t \), it will eventually get to \( l_{t+1} \). 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 \( l_3 \). 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 \( l_1 \rightarrow l_2 \rightarrow l_1 \rightarrow l_2 \).

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 \( \varphi_i \) is connected by edges to \( \varphi_{j_1}, \ldots, \varphi_{j_k} \) states that

\[
\varphi_i \models \Diamond (\varphi_{j_1} \lor \cdots \lor \varphi_{j_k})
\]

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

\[
D. \text{ For } i = 1, \ldots, r: \quad \neg \varphi_i \models \Diamond \left( \bigvee_{j<i} \varphi_j \right).
\]

The establishment of condition \( D \) for each \( i > 0 \) is then based on the \( \jmath\)-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 \( \jmath\)-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 \( \{ \varphi_i | i = 0, \ldots, r \} \) is to consider a single assertion \( \varphi(\alpha) \), parameterized by a parameter \( \alpha \) taken from a well-founded structure \( (\mathcal{A}, \succ) \). Obviously, the most important property of our chain of assertions is that program transitions eventually lead from \( \varphi_i \) to \( \varphi_j \) 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 \((\mathcal{A}, \succ)\) be a well-founded structure.
Let \(\varphi(\alpha) = \varphi(\alpha; \vec{y}; \vec{z})\), \(\alpha \in \mathcal{A}\), be a parameterized state formula.
Let \(h: \mathcal{A} \rightarrow [1 \ldots m]\) be a function identifying for each \(\alpha \in \mathcal{A}\) the helpful process \(P_{h(\alpha)}\) for states satisfying \(\varphi(\alpha)\).

A. \(\neg P\) leads from \(\varphi(\alpha)\) to \(\psi \lor (\exists \beta \leq \alpha . \varphi(\beta))\)
B. \(\neg P_{h(\alpha)}\) leads from \(\varphi(\alpha)\) to \(\psi \lor (\exists \beta < \alpha . \varphi(\beta))\)
C. \(\neg \varphi(\alpha) \supset \left[ \psi \lor (\exists \beta < \alpha . \varphi(\beta)) \lor \text{Enabled}(P_{h(\alpha)}) \right]\)

\[\neg (\exists \alpha . \varphi(\alpha)) \supset (\exists \alpha . \varphi(\alpha)) \cup \psi.\]

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:

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

\[
\begin{align*}
&l_0: \text{while } y_1 \neq y_2 \text{ do } \quad \text{m}_0: \text{while } y_1 \neq y_2 \text{ do } \\
&\quad \text{if } y_1 > y_2 \text{ then } y_1 := y_1 - y_2 \quad \text{if } y_1 < y_2 \text{ then } y_2 := y_2 - y_1 \\
&l_1: \text{halt} \quad \text{m}_1: \text{halt} \\
&\quad - P_1 - \quad - P_2 -
\end{align*}
\]

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

\[
\Diamond [\text{at } l_1 \land \text{at } m_1 \land (y_1 = y_2)]
\]

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 \(\varphi = \varphi_0\), we will eventually exit to \(\varphi_2\) or to \(\varphi_1\), i.e., \(\psi = \varphi_1 \lor \varphi_2\). The well-founded structure that we use is that of lexicographic pairs \((n, k)\) of which \(n \in \mathbb{N}\) 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 \(l_0 \rightarrow l_0\)) 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 \(\varphi_2\) or \(\varphi_1\), the arrival at \(\varphi_0\) is ensured by the usual application of the J-CHAIN rule. \(\Box\)
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 \textsc{j-well} rule certainly holds for all fair computations. However, the completeness of the \textsc{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 \textit{fair computations} is given by:

\textbf{Liveness Rule:} \textsc{live}

Let $\varphi(\vec{x}; \vec{y})$ and $\psi(\vec{x}; \vec{y})$ be two state formulas and $P_k$ one of the processes

A. $\vdash P$ leads from $\varphi$ to $\varphi \lor \psi$

B. $\vdash P_k$ leads from $\varphi$ to $\psi$

C. $\vdash \varphi \Rightarrow \Diamond[\psi \lor Enabled(P_k)]$

$\vdash \varphi \Rightarrow (\varphi \cup \psi)$
To justify the liveness rule \textsc{live}, let $\sigma$ be a fair computation such that $\varphi$ is initially true. By condition A, $\varphi$ will hold until $\psi$ is realized. Assume therefore that $\psi$ is never realized and hence $\varphi$ holds in all states of $\sigma$. By condition B, $P_k$ was never activated, since any activation of $P_k$ from a $\varphi$-state would have realized $\psi$ immediately. By condition C, each state, being a $\varphi$-state, is eventually followed by a state in which either $\psi$ holds or $P_k$ is enabled. By our assumption that $\psi$ never occurs, the latter must be the case, i.e. $P_k$ is enabled. Consequently, $\sigma$ 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 $\varphi$-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 $\sigma$ but never activated, contradicting our initial assumption that $\sigma$ is a fair computation. Consequently, any fair computation beginning in a $\varphi$-state must contain a $\psi$-state.

The difference between the \textsc{live} and the \textsc{j-live} rule is in condition C. While the \textsc{j-live} rule requires that the helpful process is enabled now, the \textsc{live} rule only assures that it will be eventually enabled. An apparent advantage of the \textsc{j-live} version of condition C is that it is static, i.e. contains no temporal operators. The \textsc{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: $\varphi \Rightarrow \Diamond \psi$. Two obvious questions arise: How do we prove condition C of the \textsc{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 $\psi \vee \text{Enabled}(P_k)$). 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:

$$A_{\varphi}(P \setminus \{P_k\}) \vdash \varphi \Rightarrow \Diamond [\psi \vee \text{Enabled}(P_k)].$$

This means that if we consider all fair computations of the program obtained from $P$ by omitting the process $P_k$, then $\varphi$ guarantees the eventual realization of $\psi \vee \text{Enabled}(P_k)$. 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 \textsc{chain} and \textsc{well} rules in a similar way by appropriately generalizing condition C. Thus to obtain the \textsc{chain} rule we replace condition C of the \textsc{j-chain} rule by:

$$\vdash \varphi \Rightarrow \Diamond \left[ \bigvee_{\varphi_j \leq \varphi} \psi_j \vee \text{Enabled}(P_k) \right].$$
The full Well rule is given by:

\[
\text{Well-Founded Liveness Rule: WELL}
\]

Let \((\mathcal{A}, >)\) be a well-founded structure.

Let \(\varphi(\alpha) = \varphi(\alpha: \pi; \delta), \alpha \in \mathcal{A}\) be a parameterized state formula.

Let \(h:\mathcal{A} \rightarrow [1 \ldots m]\) be a function identifying for each \(\alpha \in \mathcal{A}\) the helpful process \(P_{h(\alpha)}\) for states satisfying \(\varphi(\alpha)\).

\[
A. \quad \vdash P \text{ leads from } \varphi(\alpha) \text{ to } \psi \lor (\exists \beta \leq \alpha \cdot \varphi(\beta))
\]

\[
B. \quad \vdash P_{h(\alpha)} \text{ leads from } \varphi(\alpha) \text{ to } \psi \lor (\exists \beta < \alpha \cdot \varphi(\beta))
\]

\[
C'. \quad \varphi(\alpha) \Rightarrow \bigcirc[\psi \lor (\exists \beta < \alpha \cdot \varphi(\beta)) \lor \text{Enabled}(P_{h(\alpha)})]
\]

\[\vdash (\exists \alpha \cdot \varphi(\alpha)) \Rightarrow (\exists \alpha \cdot \varphi(\alpha)) \forall \psi\]

We refer to \(h\) as the \textit{helpfulness} function.

We will proceed to illustrate first the application of the \textit{Chain} rule and then the application of the \textit{Well} rule to proofs of liveness properties of programs with semaphores.

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

**Program PC.** Producer-consumer:

\[(b, s, cf, ce) := (\lambda, 1, 0, n)\]

\begin{align*}
&l_0: \text{ compute } y_1 \\
&l_1: \text{ request}(ce) \\
&l_2: \text{ request}(s) \\
&l_3: t_1 := b \cdot y_1 \\
&l_4: b := t_1 \\
&l_5: \text{ release}(s) \\
&m_0: \text{ request(cf)} \\
&m_1: \text{ request(s)} \\
&m_2: y_2 := \text{head}(b) \\
&m_3: t_2 := \text{tail}(b) \\
&m_4: b := t_2 \\
&m_5: \text{ release}(s) \\
&m_6: \text{ release}(ce) \\
&m_7: \text{ compute using } y_2 \\
&m_8: \text{ goto } m_0
\end{align*}

\[\quad - \text{ \textit{P}_1: Producer} - \]

\[\quad - \text{ \textit{P}_2: Consumer} - \]

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 $l_{1,5}$ 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.

- **Liveness.** Here we wish to show that

$$at\ l_i \Rightarrow \Diamond at\ l_i.$$  

We start by presenting the top-level diagram proof (Fig. 4). This diagram proof is certainly trivial. Everywhere, $P_i$ 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 $P - \{P_i\} = \{P_2\}$. We thus have to construct two subproofs:

$$\mathcal{A}_F(P_2) \vdash at\ l_i \Rightarrow \Diamond (ce > 0),$$

$$\mathcal{A}_F(P_2) \vdash at\ l_2 \Rightarrow \Diamond (s > 0).$$

![Diagram](image)

The first statement ensures that if $P_i$ is at $l_i$, $P_2$ will eventually cause $ce$ to become positive which is the enabling condition for $P_i$ to be activated at $l_i$. Similarly, in the second statement $P_3$ will eventually cause $s$ to become positive, making $P_i$ enabled at $l_2$. For both statements we present diagram proofs.

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

$$I_1: at\ l_{1,5} + at\ m_{2,5} + s = 1$$

![Diagram](image)

It has been used in order to derive that being at $l_i$ and at $m_i$ 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

$$I_2: cf + ce + at\ l_{2,6} + at\ m_{1,6} = n.$$
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).

\[
\begin{array}{c}
\varphi_4 : m_2 \\
\varphi_3 : m_3 \\
\varphi_2 : m_4 \\
\varphi_1 : m_5 \\
\varphi_0 : s > 0
\end{array}
\]

Fig. 6.

Example 4. The following program demonstrates the application of the \textit{well} rule for programs with semaphores.

**Program BC.** Binomial coefficient.

\[
(y_1, y_2, y_3, y_4) := (n, 0, 1, 1)
\]

\[
\begin{align*}
  l_0: & \text{ if } y_1 = (n-k) \text{ then goto } l_1 \\
  l_1: & \text{ request}(y_4) \\
  l_2: & \text{ } y_1 := y_3 \cdot y_1 \\
  l_3: & \text{ } y_3 := t_1 \\
  l_4: & \text{ release}(y_4) \\
  l_5: & \text{ } y_1 := y_1 - 1 \\
  l_6: & \text{ goto } l_0 \\
  l_7: & \text{ halt}
\end{align*}
\]

\[
\begin{align*}
  m_0: & \text{ if } y_2 = k \text{ then goto } m_6 \\
  m_1: & \text{ } y_2 := y_2 + 1 \\
  m_2: & \text{ loop until } y_1 + y_2 \leq n \\
  m_3: & \text{ request}(y_4) \\
  m_4: & \text{ } y_2 := y_3 / y_2 \\
  m_5: & \text{ } y_3 := t_2 \\
  m_6: & \text{ release}(y_4) \\
  m_7: & \text{ goto } m_0 \\
  m_8: & \text{ halt}
\end{align*}
\]

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 \( y_1 \) 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:
\[(y_1, y_2, y_3, y_4) := (n, 0, 1, 1)\]

\[
\begin{align*}
  l_7: & \quad \text{if } y_1 = (n - k) \text{ then goto } l_1 \\
  l_8: & \quad \text{request}(y_a) \\
  l_5: & \quad t_1 := y_3 \cdot y_1 \\
  l_4: & \quad y_3 := t_1 \\
  l_3: & \quad \text{release}(y_4) \\
  l_2: & \quad y_1 := y_1 - 1 \\
  l_6: & \quad \text{goto } l_7 \\
  l_1: & \quad \text{halt}
\end{align*}
\]

\[
\begin{align*}
  m_3: & \quad \text{if } y_2 = k \text{ then goto } m_1 \\
  m_2: & \quad y_2 := y_2 + 1 \\
  m_6: & \quad \text{loop until } y_1 + y_2 \leq n \\
  m_8: & \quad \text{request}(y_4) \\
  m_7: & \quad t_2 := y_3 / y_2 \\
  m_5: & \quad y_1 := t_2 \\
  m_4: & \quad \text{release}(y_4) \\
  m_1: & \quad \text{halt}
\end{align*}
\]

- \(P_1\) -

- \(P_2\) -

The liveness property we wish to prove is:

\[\left[\mathrm{at}\{l_7, m_3\} \land (y_1, y_2, y_3, y_4) = (n, 0, 1, 1)\right] \Rightarrow \diamond \mathrm{at}\{l_1, m_1\}.\]

We derive first several invariants needed for the liveness proof:

\[
\begin{align*}
  I_1: & \quad (at l_{3...3} + at m_{5...7} + y_4) = 1 \\
  I_2: & \quad ((n - k) + at l_{2...n}) \leq y_1 \leq n \\
  I_3: & \quad 0 \leq y_2 \leq (k - at m_2) \\
  I_4: & \quad at l_1 \Rightarrow (y_1 = n - k).
\end{align*}
\]

For our well-founded domain we choose:

\[W = \langle \mathbb{N} \times \{0, \ldots, 17\} \times \{0, 1\}, \succ_{\text{lex}} \rangle.\]

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

The parameterized assertion is:

\[
\varphi(w; l_n, m_j; y_1, y_2) = \varphi((r, s, t); l_n, m_j; y_1, y_2):
\]

\[
(r = y_1 + k - y_2) \land (s = i + j) \land (t = at l_1).
\]

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 \(I_1\) and \(P_2\) is at \(m_6\) we have to show that the next step indeed decrements \((r, s, t)\). 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 \(I_6\) 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 \(I_4\).

![Fig. 7.](image)

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 \(\varphi\) to \(\psi\)" notion. Usually, for a particular \(\varphi\) and \(\psi\), 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

Proof principles for invariance and liveness of concurrent programs


