An Exercise in the Verification of Multi-Process Programs*

Zohar Manna
Stanford University †
and
Weizmann Institute of Science‡

Amir Pnueli
Weizmann Institute of Science: ‡

October, 1989

Abstract

We present an approach to the verification of a multi-process program consisting of a fixed but unbounded number of processes executing an identical program. The approach is illustrated on an algorithm for mutual exclusion that contains tests that refer to many shared variables at the same time. We analyze the algorithm first under the assumption that these tests are *atomic*. We then consider the more realistic assumption that they are *molecular*, i.e. performed by several steps, each reading a single shared variable. We show that the algorithm is correct only for the limited implementation in which the variables are checked in ascending order of indices.

*This research was supported in part by the National Science Foundation under grant CCR-8812595, by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211, by the United States Air Force Office of Scientific Research under contracts AFOSR 87-0149 and 88-0281, and by the European Community ESPRIT Basic Research Action project 3096 (SPEC).

†Department of Computer Science, Stanford University, Stanford, CA 94305

‡Department of Applied Mathematics, Weizmann Institute, Rehovot, Israel

1 Introduction

Many concurrent programs consist of a set of processes that execute an identical program (which may refer to the process identifier as a parameter). A challenging problem is to provide methods for the *uniform* verification of such a system, i.e. proving correctness for *any* number of processes.

We present an approach for such uniform verification and illustrate it on a new algorithm for mutual exclusion ([Szy88]). We analyze the algorithm first under the assumption that tests that refer to many shared variables are *atomic*, i.e. all variables are checked in a single step. We then consider the more realistic assumption that these tests are *molecular* (see [PZ86]), i.e. performed by several steps, each reading a single shared variable. We show that the algorithm is correct only for the limited implementation in which the variables are checked in ascending order of indices and present a counter-example for more general implementations of molecular tests.

We examine only the *safety* property of *mutual exclusion*. The more complete paper [MP89]) considers other non-safety properties, such as *accessibility* and *precedence*.

The paper contains no new theoretical results. Rather, it recommends the adoption of a set notation for expressing the control state of a system with an unbounded, and even dynamic, set of processes, within the framework of old and tried proof methods, such as [Lam77, MP84] (see also [PZ86] where this set notation has been introduced for the analysis of probabilistic algorithms).

The algorithm we have chosen to verify is an ideal example for demonstrating the acute need for formal verification of concurrent programs, as well as the style and level of verification that is currently possible. We refer the reader to [Szy88] for some of its important features, such as using single-writer bounded shared variables and enjoying the property of *linear* delay (which we verify in [MP89] as a precedence property). These features make this algorithm a significant improvement over most of its predecessors.

Although the algorithm appears to be quite simple and innocuous, the only way we could convince ourselves of its correctness was to construct the formal proof outlined in this paper. Szymanski presented an informal proof, which is as convincing as informal proofs can be. In fact, our formal proof derives its main ideas from a formalization of his informal arguments. However, if the question of correctness is crucial, such as having to decide whether to include this algorithm as a contention-resolving component in a hardware chip, we see no way but to carry out a formal verification.

We have learned two lessons from carrying out this verification exercise. The less encouraging lesson is that it requires a non-negligible deal of creativity and dexterity in manipulating logical formulae to come up with the appropriate set of auxiliary assertions (and other constructs needed for the proof). This is so even if the correct intuition is given and all that is required is to formalize that intuition. The more encouraging lesson is that, once the appropriate constructs have been found, the rest of the verification process, which requires the construction of the verification conditions (proof obligations) and proving their validity, can to a large extent be automated. It is not that we have come up with a surprisingly new automatic theorem prover. But inspection of the kinds of assertions generated for a proof of an algorithm like the one we study here convinced us that for a large and interesting class of algorithms all these assertions belong to a decidable class.

2 Programs and Computations

Our computational model is a *transition system*. In it, a program P consists of the following components.

- $V = \{u_0, ..., u_{n-1}\}$ – A finite set of *state variables*. Some variables represent *data* variables, which are explicitly manipulated by the program text. Others are *control* variables, which represent, for example, the location of control in each of the processes in a concurrent program. Each variable is associated with a domain over which it ranges.

- Σ – A set of *states*. Each state $s \in \Sigma$ is an interpretation of V, assigning to each variable $y \in V$ a value over its domain, which we denote by $s[y]$.

- $\mathcal{T}$ – A finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is associated with an assertion $\rho_\tau(V, V')$, called the *transition relation*, which refers to both an unprimed and a primed version of the state variables. The purpose of ρ_τ is to express the relation between a state s and its successor s' under τ. The unprimed version refers to values in s; the primed version to values in s'. For example, the assertion $x' = x + 1$ states that $s'[x]$ is greater by 1 than $s[x]$.

- Θ – The *precondition.* This is an assertion characterizing all the initial states. A state is defined to be *initial* if it satisfies Θ.

We define state s' to be a τ*-successor* of state s if

$$\langle s, s' \rangle \models \rho_\tau(V, V'),$$

where $\langle s, s' \rangle$ is the joint interpretation that interprets $x \in V$ as $s[x]$ and x' as $s'[x]$.

A state s is *terminal* if it has no τ-successor for any $\tau \in \mathcal{T}$.

We define a *computation* of a program P to be any finite or infinite sequence of states

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

satisfying:

- *Initiality* s_0 is initial, i.e. $s_0 \models \Theta$.
- *Consecution* For each $j = 0, 1, ...,$ state s_{j+1} is a τ-successor of state s_j, for some $\tau \in \mathcal{T}$.
- *Termination* Either σ is infinite or it ends in a terminal state s_k.

We denote by $\mathcal{C}omp(P)$ the set of all computations of program P. We say that a state s is P*-accessible* if it appears in some computation of P. Clearly, any τ-successor of a P-accessible state is also P-accessible.

We assume an underlying assertional language, which contains the predicate calculus, and interpreted symbols for expressing the standard operations and relations over some concrete domains. We refer to a formula in the assertional language as an *assertion.* For an assertion p and a state s such that p holds on s, we say that s is a p*-state.*

3 The Program as a Transition System

The program we wish to study can be given as

MUTEX :: $flag$: **array**$[0 \,..\, n-1]$ **of** $0..4$ **where** $flag := 0$;
$P[0] \;||\; P[1] \;||\; ... \;||\; P[n-1]$

Each process $P[i]$ is given by:

l_0: **loop forever do**
begin
l_1 : Non Critical
l_2 : $flag[i] := 1$
l_3 : **wait until** $\forall j : 0 \leq j < n : (flag[j] < 3)$
l_4 : $flag[i] := 3$
l_5 : **if** $\exists j : 0 \leq j < n : (flag[j] = 1)$ **then**
begin
l_6: $flag[i] := 2$
l_7: **wait until** $\exists j : 0 \leq j < n : (flag[j] = 4)$
end
l_8 : $flag[i] := 4$
l_9 : **wait until** $\forall j : 0 \leq j < i : (flag[j] < 2)$
l_{10}: Critical
l_{11}: **wait until** $\forall j : i < j < n : (flag[j] < 2 \ \vee \ flag[j] > 3)$
l_{12}: $flag[i] := 0$
end

Viewing the program as a transition system requires the identification of the following four components:

- V – The state variables, are

$$L_0, ...L_{12}, \quad flag[0], ..., flag[n-1].$$

  $L_0, ..., L_{12}$, are control variables that range over subsets of $\{0, ..., n-1\}$. At any state of the computation, L_k contains the indices of the processes that currently are ready to execute the statement labeled l_k. Variables $flag[0], ..., flag[n-1]$ naturally represent the current values of the corresponding program variables.

- Σ – The states, consist of all the possible assignments to the state variables of values in their respective domains.

- Θ – The precondition is given by the assertion

$$\Theta \quad : \quad (L_0 = \{0, ..., n-1\}) \wedge (L_{1..12} = \phi) \wedge \bigwedge_{i=0} : n-1(flag[i] = 0)$$

Thus, at the initial state of the program, all processes reside at the location l_0, and $flag[0], ..., flag[n-1]$ are zero.

We introduce the following abbreviations and notations:

$$\begin{array}{lcl} move(i,k,m) & : & (L'_k = L_k - \{i\}) \wedge (L'_m = L_m \cup \{i\}) \\ stay & : & \bigwedge_{k=0} : 12(L'_k = L_k) \end{array}$$

Clearly, $move(i,k,m)$ describes the movement of control within process $P[i]$ from l_k to l_m, while $stay$ describes the case that control does not move in any of the processes.

$$\begin{array}{lcl} L_{i_1,i_2,...,i_m} & = & L_{i_1} \cup L_{i_2} \cup ... \cup L_{i_m} \\ F_{i_1,i_2,...,i_m} & = & F_{i_1} \cup F_{i_2} \cup ... \cup F_{i_m} \end{array}$$

$$\begin{array}{lcll} F_k & = & \{i : 0 \leq i < n : flag[i] = k\} & \\ N_i & = & |L_i| & \\ L_{i..k} & = & L_i \cup L_{i+1} \cup ... \cup L_k & \text{for } i < k \\ F_{i..k} & = & F_i \cup F_{i+1} \cup ... \cup F_k & \text{for } i < k \end{array}$$

The Transitions

Our first analysis of algorithm MUTEX treats all tests (e.g. the one in statement l_3) as *atomic*: each is performed by a single transition. Consequently, we have for each process $P[i]$ and each location l_k a transition $\tau_k[i]$ and a corresponding transition relation $\rho_k[i]$.

We present below the two transitions corresponding to locations l_1 and l_5; for the full set of transitions, see [MP89].

The transition relation for l_1 is given by

$$\rho_1[i] : \quad (i \in L_1) \wedge \Big(stay \vee move(i,1,2)\Big)$$

According to this formula, process $P[i]$ may choose non-deterministically to remain at l_1 or to move from l_1 to l_2.

The transition relation for l_5 is given by

$$\rho_5[i] : \quad (i \in L_5) \wedge \Big([(F_1 \neq \phi) \wedge move(i,5,6)] \vee [(F_1 = \phi) \wedge move(i,5,8)]\Big)$$

When at l_5, $P[i]$ may proceed to l_6 if some process $P[j]$ has $flag[j] = 1$ and therefore $F_1 \neq \phi$. It may proceed to l_8 if no process has its $flag$ equal to 1, and hence $F_1 = \phi$.

4 Invariance Properties

Assertion p is *valid over program* P (also described as being P-*valid*,) written as $P \models p$, if p holds over all the P-accessible states. Clearly, if p is P-valid it is an *invariant* property of P: it holds over all the states that arise in any computation of the program P.

In this section we present a single proof rule that is adequate for proving the invariance of an assertion p over a program P, i.e. proving $P \models p$.

We will illustrate this rule by proving the main safety property of program **MUTEX**, mutual exclusion. This property can be expressed by the assertion $N_{10} \leq 1$, which limits the number of processes that can be concurrently executing at l_{10}. Thus, we have to prove

$$\textbf{MUTEX} \models (N_{10} \leq 1)$$

Since most of our reasoning is done within the P-validity framework, we omit the prefix $P \models$ and simply write p to mean $P \models p$.

The following rule **INV** is the main working tool for establishing invariance properties. It uses the notation p' to refer to the *primed version* of assertion p, which is obtained from p by replacing each variable $x \in V$ occurring in p by its primed version x'.

$$\begin{array}{|lll|}
\hline
\textbf{INV} & \text{I1.} & \Theta \rightarrow p \\
& \text{I2.} & (p \wedge \rho_\tau) \rightarrow p' \quad \text{for every } \tau \in \mathcal{T} \\
\cline{2-3}
& & P \models p \\
\hline
\end{array}$$

By premise I1 of rule **INV**, p holds initially, and by premise I2 it is propagated from each state to its successor. Hence, p is an invariant of the program.

Example 4.1 Consider the trivial transition system P defined as follows. There is one state variable, x. There is one transition τ, whose transition relation is $\rho_\tau : (x' = x + 2)$. The precondition is $\Theta : (x = 0)$. Transition system P has the single (infinite) computation $\langle x : 0\rangle, \langle x : 2\rangle, \langle x : 4\rangle, \ldots$.

We wish to prove for this program the invariant

$$even(x),$$

claiming that all the values of x are even. To prove this property, we use rule **INV** with $p : even(x)$. The rule requires showing the validity of the following two premises

I1. $(x = 0) \rightarrow (even(x))$

I3. $\Big((even(x)) \wedge (x' = x + 2)\Big) \rightarrow (even(x'))$

Clearly, these premises are valid, which establishes the invariance of $even(x)$.

Simple Invariants for mutex

We now establish invariants for MUTEX, which together will yield our desired result. First, we establish invariants that connect for each i the location of $P[i]$ with $flag[i]$. These invariants can be expressed as relations between F_k and L_r for various values of k and r.

IF0.	F_0	$=$	$L_{0..2}$	IF3.	F_3	$\subseteq$	$L_{5,6,8}$
IF1.	F_1	$=$	$L_{3,4}$	IF4.	F_4	$=$	$L_{9..12}$
IF2.	F_2	$\subseteq$	$L_{7,8}$	IL8.	$F_{2,3}$	$\supseteq$	L_8

Invariants IF0,...,IF4 restrict the locations at which $P[i]$ can reside when $flag[i]$ is $0, ..., 4$. For example, invariant IF4 claims that $flag[i]$ is 4 *iff* $P[i]$ is at one of $l_9, ..., l_{12}$. Invariant IL8 claims that,, when $P[i]$ is at l_8, $flag[i]$ is 2 or 3. This is the only location for which $flag[i]$ is not uniquely determined.

Proving Mutual Exclusion

Having prepared the machinery for proving invariance properties, we proceed to establish the main invariance property of MUTEX, mutual exclusion.

We refer the reader to [Szy88] for a detailed explanation of the basic ideas of MUTEX. Here we extract just the main observation. The tortuous path a process has to follow on its way from the non-critical section at l_1 to the critical section at l_{10} can be broken into different segments. We refer to location l_4 as the *doorway*, to segment $l_{5..7}$ as the *waiting room*, and to segment $l_{8..12}$, which contains the critical section, as the *inner sanctum*.

The basic claims on which mutual exclusion is based are the following:

C1. Whenever a process enters an empty inner sanctum, the doorway is locked, i.e. $L_4 = \phi$. The doorway remains locked until the last process leaves the inner sanctum. While this property is expressed as evolution in time, it certainly implies the invariant

$$A_0 \quad : \quad L_{8..12} \neq \phi \rightarrow L_4 = \phi,$$

which claims that if $L_{8..12}$ is non-empty then L_4 is empty. If this is indeed an invariant, then the non-emptiness of $L_{8..12}$ should prevent any new processes coming to l_3 to cross over into l_4. The only thing that can prevent them from crossing over is if $flag[j]$ of some process equals 3 or 4. Thus, we must also have

$$A_1 \quad : \quad L_{8..12} \neq \phi \rightarrow L_{8..12} \cap F_{3,4} \neq \phi.$$

Note that we require that one of the processes in $l_{8..12}$ has a flag value of 3 or 4. This is because a flag value of 3 held by a process at $l_{5,6}$ is unstable in the sense that it may very soon change to 2 again, by the statement at l_6.

C2. If a process i is at $l_{10..12}$, then it has the least index of all the processes in $l_{5..12}$. This is expressed by the invariant

$$A_2 \quad : \quad (k < i \wedge i \in L_{10..12}) \rightarrow k \notin L_{5..12}.$$

C3. If some process is at l_{12}, then all the processes in $l_{5..12}$ must have a flag value of 4. This is expressed by the invariant

$$A_3 \quad : \quad (i \in L_{12} \wedge k \in L_{5..12}) \rightarrow k \in F_4.$$

Thus, as soon as a process enters the inner sanctum the doorway gets locked. This leaves the processes in the waiting room and the inner sanctum isolated from the rest of the processes and lets them compete on the entry to the critical section. By claim C2, only one process at a time can reside in region $l_{10..12}$, which includes the critical section, the process whose index is minimal among all the processes in $l_{5..12}$. It follows from C1–C3 that mutual exclusion is maintained.

Taking the four assertions together, we obtain the conjunction

$$p_0 \quad : \quad A_0 \wedge A_1 \wedge A_2 \wedge A_3$$

which can be shown to be an invariant, from which by A_2 mutual exclusion follows immediately.

5 The Molecular Case

The assumption that compound tests are atomic is not realistic, and we now consider such tests to be *molecular*. To model the most general form of a molecular test, we introduce additional state variables $C_0, ..., C_{n-1}$. In each state, variable C_i contains the set of all indices j that have already been checked by process $P[i]$. We illustrate this by considering two transition relations.

$$\rho_2[i]: \quad i \in L_2 \wedge move(i,2,3) \wedge flag'[i] = 1 \wedge C_i' = \phi$$

This relation shows that one of the actions performed by transition $\tau_3[i]$ is to reset the *already checked* variable to ϕ, preparing for the test at l_3.

$$\rho_3[i,j]: \quad (i \in L_3) \wedge \begin{pmatrix} [|C_i| = n \wedge move(i,3,4)] \\ \vee \\ [j \notin C_i \wedge flag(j) < 3 \wedge stay \wedge C_i' = C_i \cup \{j\}] \\ \vee \\ [j \notin C_i \wedge flag(j) \geq 3 \wedge stay \wedge C_i' \subseteq C_i] \end{pmatrix}$$

The transition relation for the molecular test at l_3 consists of three options, represented by three clauses. The first clause covers the case that $|C_i| = n$, which implies that all indices have been checked and we may proceed from l_3 to l_4. The second clause identifies an index j that satisfies $flag(j) < 3$ and that is not yet in C_i. This index is added to C_i. The last clause detects an unchecked index j, which does not satisfy $flag(j) < 3$. Here we allow an arbitrary change of C_i as long as it does not increase. At the two extremes, this allows both for the implementation in which C_i is emptied whenever we find a bad index j, and the implementation that ignores bad indices and retains C_i at its current value.

A Counter-Example

It would have been nice to extend our proof for the atomic case to a proof for the molecular case, allowing the most general implementations of molecular tests, as presented above. Unfortunately, this is not possible. The following counter example considers a program with three processes and traces their general progress until they reach a state in which $P[0]$ and $P[2]$ execute their respective critical sections at the same time, violating mutual exclusion.

The offending computation is presented in the following table, which lists some intermediate states that occur in the computation until the violating state is reached. For each presented state we list, from left to right, the location of $P[0]$, $flag[0]$, C_0, the location of $P[1]$, $flag[1]$, the location of $P[2]$, and $flag[2]$. Note that, while at l_3, process $P[0]$ checks the indices $2, 1, 0$ in *descending* order.

$0 \in$	$flag[0]$	C_0	$1 \in$	$flag[1]$	$2 \in$	$flag[2]$
L_0	0	$\{\}$	L_0	0	L_0	0
L_2	·	·	L_3	1	L_3	1
·	·	·	L_4	·	L_4	·
·	·	·	·	·	L_5	3
·	·	·	·	·	L_7	2
·	·	·	L_5	3	·	·
·	·	·	L_9	4	·	·
·	·	·	L_{11}	·	·	·
L_3	1	·	·	·	·	·
·	·	$\{2\}$	·	·	·	·
·	·	·	·	·	L_9	4
·	·	·	L_{12}	·	·	·
·	·	·	L_0	0	·	·
·	·	·	·	·	L_{10}	·
·	·	$\{2,1\}$	·	·	·	·
·	·	$\{2,1,0\}$	·	·	·	·
L_4	·	·	·	·	·	·
L_9	4	·	·	·	·	·
L_{10}	·	·	·	·	L_{10}	·

6 Restricted Implementation of Molecular Tests

The remedy to the problem encountered above is to restrict the implementation of molecular tests. In particular, we should not allow the selection of the next index to be checked to be completely arbitrary.

Consider a typical situation, in which some process $P[k]$ is currently at l_3 attempting to proceed to l_4. Assume also that it is blocked at l_3 by process $P[r]$, which is currently in the inner sanctum with $flag[r] = 4$. This means that that index $j = r$ has not been checked yet by $P[k]$, and will not be successfully checked as long as $flag[r] = 4$. Eventually, $P[r]$ will want to depart from the inner sanctum at l_{12}. Assume that $P[r]$ is not the last process in the inner sanctum. It is essential to ensure that when $P[r]$ departs from the inner sanctum, another process in the inner sanctum will continue to block $P[k]$. According to claim C2, processes depart from the inner sanctum in order of increasing indices. According to C3, if $P[r]$ is not the last process to depart, then all the processes remaining in the inner sanctum must have their $flag$ equal to 4, and as observed before, must have an index higher than r. It follows that, if we take care that $P[k]$ at l_3 examines the indices in *ascending order*, then if $P[k]$ is blocked by r it has not successfully checked any index higher than r. Consequently, when r departs and leaves behind some other process, with index $t > r$, t can serve as a replacement for r, continuing to block k.

Motivated by this argument, we restrict the implementation of molecular tests to examines $flag[j]$ in order of increasing j. To do so, we replace state variables $C_0, \ldots, C_{n-1}$, by the simpler integer variables $j_0, \ldots, j_{n-1}$. In each state, j_i contains the least value of j that has not yet been successfully tested by $P[i]$. The two transition relations below illustrate how the variables are manipulated by the molecular tests.

$$\rho_2[i]: \quad i \in L_2 \wedge move(i,2,3) \wedge flag'[i] = 1 \wedge j'_i = 0$$

This relation shows that on moving to l_3, the transition $\tau_3[i]$ resets j_i to 0.

$$\rho_3[i]: \quad i \in L_3 \wedge \left(\begin{array}{l} [j_i = n \wedge move(i,3,4)] \\ \quad \vee \\ [j_i < n \wedge flag(j_i) < 3 \wedge stay \wedge j'_i = j_i + 1] \\ \quad \vee \\ [j_i < n \wedge flag(j_i \geq 3 \wedge stay \wedge j'_i \leq j_i] \end{array} \right)$$

As before, this relation consists of three clauses, corresponding to: completion of the test, advancement when detecting a good next index, and retreat when detecting a bad next index. Again, the retreat is general enough to allow both restarting from $j_i = 0$ and keeping j_i at its current value.

Reproving Mutual Exclusion

It is helpful to derive the assertions for the molecular case as refinments of the assertions for the atomic case. Consider any region of consecutive locations that is mentioned in one of the assertions $A_0, \ldots, A_3$ and that is preceded by a compound test. For example, $l_{10..12}$ is such a region. Assertion A_2 states that if $k < i$ and i belongs to $L_{10..12}$, then k cannot be in $L_{5..12}$. In the atomic case, one of the considerations used in proving this assertions is that $P[i]$ cannot pass the atomic test at l_9 if $k < i$ is anywhere at $l_{5..12}$. This is because, by the simple invariants connecting flag values to locations, that would have implied $flag[k] \geq 2$.

In the molecular case, the test at l_9 is not passed in one step. Process $P[i]$ may reside at l_9 for several steps, checking the values of $flag[j_i]$ for many values of j_i. The important question concerning k is whether $P[i]$ has already tested the value of $flag[k]$. This can be observed by checking whether $j_i > k$. If so, then $flag[k]$ has already been tested and found satisfactory, i.e. less than 2.

Consequently, to adapt assertion A_2 to the molecular case, we should replace the simple region reference $i \in L_{10..12}$ by the extended reference $i \in L_{10..12} \vee (i \in L_9 \wedge j_i > k)$. By applying such range extensions to assertions A_0, A_2, and A_3, we obtain the following assertions:

$$B_0 : \quad (i \in L_5 \wedge j_i > k) \quad \rightarrow \quad \neg\Big[k \in L_4 \vee (k \in L_3 \wedge j_k > i)\Big]$$

$$B_1 : \quad i \in L_{8..12} \quad \rightarrow \quad \exists r : (r \in L_{8..12} \cap F_{3,4}) : \neg\Big[k \in L_4 \vee (k \in L_3 \wedge j_k > r)\Big]$$

$$B_2 : \quad \Big[(k < i) \wedge (i \in L_{10..12} \vee i \in L_9 \wedge j_i > k)\Big] \quad \rightarrow \quad k \notin L_{5..12}$$

$$B_3 : \quad \Big[(i \in L_{12} \vee (i \in L_{11} \wedge j_i > k)) \wedge (k \in L_{5..12})\Big] \quad \rightarrow \quad k \in F_4$$

Assertions B_0 and B_1 refine assertions A_0 and A_1 to the molecular case. The basic idea is to show for any k that if $P[i]$ is at $l_{8..12}$ or l_5 with $j_i > k$, i.e. having already checked $flag[k]$, then $P[k]$ cannot be at l_4, and if it is at

l_3, then it's j_k value is below some r that blocks it from proceeding into l_4, by having $flag[r] > 2$. If $P[i]$ is at l_5, we can take r to be i itself. If $P[i]$ is at $l_{8..12}$, we can only claim the existence of some blocking r, such that $P[r]$ is also at $l_{8..12}$ and $flag[r] > 2$.

We form now the conjunction

$$\varphi \quad : \quad B_0 \wedge B_1 \wedge B_2 \wedge B_3$$

and claim that it is an invariant of **MUTEX**. It is beyond the scope of this paper to consider all the transitions and show that each preserves φ. We will, however, consider some of the more interesting cases.

Consider what transitions may affect the assertion B_1. A critical transition of $P[i]$ is one that moves from l_5 to l_8. However, due to B_0, the right hand side of the implication of B_1 will hold after the transition with $r = i$ and $flag[i] = 3$. A critical transition of k would have been one that increases j_k beyond r. However, due to $flag[r] > 2$, such a transition is disabled.

Lastly, we consider the transition of $P[r]$ from l_{12} to l_0, while resetting its flag to 0. There are two possibilities. If r is the last process in $l_{8..12}$, then after the transition $L_{8..12}$ will become empty, causing B_1 to hold trivially. If r is not the last, there is another process, say $P[t]$, in $l_{8..12}$. Then, due to B_2, which states that r is the minimal process in $l_{8..12}$, r is smaller than t. Therefore, if $j_k \leq r$ it is also $\leq t$. Due to B_3, $flag[t]$ equals 4. Consequently, after the transition, B_1 still holds if we use t as a substitute for r. For this argument to hold it is essential that the indices j in l_3 are scanned in increasing order.

Acknowledgement

We gratefully acknowledge the encouragement and editorial help extended by David Gries, which greatly improved the style and readability of the paper.

References

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Engin.*, 3:125–143, 1977.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comp. Prog.*, 32:257–289, 1984.

[MP89] Z. Manna and A. Pnueli. Tools for the practicing verifier. Technical report, Dept. of Computer Science, Stanford University, 1989.

[PZ86] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.

[Szy88] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. 1988 International Conference on Supercomputing Systems*, pages 621–626, St. Malo, France, July 1988.