Tools and Rules for the Practicing Verifier*

Zohar Manna
Stanford University †
and
Weizmann Institute of Science‡

Amir Pnueli
Weizmann Institute of Science‡

June 28, 1991

Abstract

The paper presents a minimal proof theory which is adequate for proving the main important temporal properties of reactive programs. The properties we consider consist of the classes of *invariance*, *response*, and *precedence* properties. For each of these classes we present a small set of rules that is complete for verifying properties belonging to this class. We illustrate the application of these rules by analyzing and verifying the properties of a new algorithm for mutual exclusion.

1 Introduction

In this paper we present a minimal proof theory that is adequate for proving interesting properties of concurrent programs. The simple theory is illustrated on a single example, which is a new and interesting algorithm for mutual exclusion [Szy88].

There are several points we would like to demonstrate in this paper. The first and main point is that a very little general (temporal) theory is required to handle the most important properties of concurrent programs. The types of properties, on which a practicing verifier (hoping that such a position will eventually become a standard in any quality assurance team) typically spends most of his time, usually fall into two or three simple classes. By presenting a simple but complete set of rules for verifying

*This research was supported in part by the National Science Foundation under grants CCR-89-11512, and CCR-89-13641; 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-90-0057, 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

properties belonging to each of these classes, we provide the practicing verifier with precisely the tools he needs. This pragmatic approach can be nicely complemented by a more theoretical presentation of a comprehensive theory of a language of specification (temporal logic would have been our choice), its power to express a wide spectrum of program properties, and a comprehensive proof theory and investigation of its completeness (see for example [MP89a]). However, it may be an educational mistake to require the study of such a comprehensive approach as an essential requisite for the pragmatic application of the verification tools that result from the general theory.

Consequently, the approach we take in this paper is to circumvent the general theory of temporal logic and proceed as directly as possible to the introduction of the classes of properties that are most frequently verified, and to the proof rules that are appropriate for their verification.

There are three classes of properties we consider in this paper, and believe to cover the majority of properties one would ever wish to verify.

- *Invariance* – An invariance property refers to an assertion p, and requires that p is an invariant over all the computations of a program P, i.e., all the states arising in a computation of P satisfy p. In temporal logic notation, such properties are expressed by $\Box p$, for a state formula p.

- *Response* – A response property refers to two assertions p and q, and requires that every p-state (a state satisfying p) arising in a computation is eventually followed by a q-state. In temporal logic notation this is written as $p \Rightarrow \Diamond q$. In the Unity notation (see [CM88]), this property is called a *leads-to* property, and written as $p \mapsto q$.

- *Precedence* – A simple precedence property refers to three assertions p, q, and r. It requires that any p-state initiates a q-interval (i.e., an interval all of whose states satisfy q) which, either runs to the end of the computation, or is terminated by an r-state. Such a property is useful in order to express the restriction that, following a certain condition, one future event will always be preceded by another future event. For example, it may express the property that, from the time a certain input has arrived, there will be an output before the next input. Note that this does not guarantee that output will actually be produced. It only guarantees that the next input (if any) will be preceded by an output. In temporal logic, this property is expressed by $p \Rightarrow (q \mathsf{U} r)$, using the *unless* operator (weak *until*) U. More complex precedence properties refer to a sequence of assertions $q_0, \ldots q_{m-1}$, and replace the requirement of a single q-interval, by a requirement of a succession of a q_0-interval, followed by a q_1-interval, ..., followed by a q_{m-1}-interval.

According to the classification of properties in [AS85], the invariance and precedence properties are *safety* properties, while the response properties are *liveness* properties. Referring to the classification of properties in [MP89a], the response properties defined here are a special case of the *responsiveness* class defined there (which allows p and q to be *past* formulae rather than assertions). The class of precedence properties and proof rules associated with it have been introduced first in [MP83].

We refer the reader to [MP89b] for a top down approach, which attempts to present the most general proof rules that cover as many properties as possible. Here, however, we take the opposite approach of presenting rules that are closely tailored for the restricted classes that are most frequently needed. This reduction in generality is justified only if we can demonstrate a gain in the convenience and efficacy of using those rules for verifying properties that fall in these classes. This brings us to the second point we wish to make in this paper.

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. 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

The basic computational model we use to represent programs is that of a *fair transition system*. In this model, a program P consists of the following components.

- $V = \{u_0, ..., u_{n-1}\}$ – A finite set of *state variables*. Some of these variables represent *data* variables, which are explicitly manipulated by the program text. Other variables are *control* variables, which represent, for example, the location of control in each of the processes in a concurrent program. We assume each variable to be 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 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 the transition relation ρ_τ is to express a relation between a state s and its successor s'. We use the unprimed version to refer to values in s, and the primed version to refer to values in s'. For example, the assertion $x' = x + 1$ states that the value of x in s' is greater by 1 than its value in s.

- Θ – The *precondition*. This is an assertion characterizing all the initial states, i.e., states at which the computation of the program can start. A state is defined to be *initial* if it satisfies Θ.

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

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

where $\langle s, s' \rangle$ is the joint interpretation which interprets $x \in V$ as $s[x]$, and interprets x' as $s'[x]$. Following this definition, we can view the transition τ as a function $\tau : \Sigma \mapsto 2^\Sigma$, defined by:

$$\tau(s) = \{s' \mid s' \text{ is a } \tau\text{-successor of } s\}.$$

We say that the transition τ is *enabled* on the state s, if $\tau(s) \neq \phi$. Otherwise, we say that τ is *disabled* on s. We say that a state s is *terminal* if all the transitions $\tau \in \mathcal{T}$ are disabled on it. The enabledness of a transition τ can be expressed by the formula

$$En(\tau) : \ (\exists V')\rho_\tau(V, V'),$$

which is true in s iff s has some τ-successor.

Assume a program P for which the above components have been specified. Consider

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

a finite or infinite sequence of states of P.

We say that the transition $\tau \in \mathcal{T}$ is enabled *at position* k of σ if τ is enabled on s_k. We say that the transition τ is *taken* (completed) at position $k+1, k = 0, 1, \ldots,$ if s_{k+1} is a τ-successor of s_k. Note that several different transitions can be considered as taken at the same position.

The sequence σ is defined to be a *computation* of P if it satisfies the following requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $j = 0, 1, \ldots$, the state s_{j+1} is a τ-successor of the state s_j, i.e., $s_{j+1} \in \tau(s_j)$, for some $\tau \in \mathcal{T}$.
- *Termination* — Either σ is infinite, or it ends in a state s_k which is terminal.
- *Justice* — For each transition $\tau \in \mathcal{T}$, it is not the case that τ is continually enabled beyond some position j in σ (i.e., τ is enabled at every position $k \geq j$) while τ is not taken beyond j.

For a program P, we denote by $\mathit{Comp}(P)$ the set of all computations of 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.* For a computation $\sigma : s_0, s_1, \ldots,$ such that s_j is a p-state, we call j a *p-position.*

Set Notation

We introduce the following notation to facilitate a compact representation of sets of natural numbers.

A *set specification* consists of a list of one or more *set specifiers*, where each specifier is either a single natural number, or an *interval specifier* of the form $a..b$, for $a \leq b$, natural numbers. The set defined by the interval specifier $a..b$ consists of all the integers not smaller than a and not larger than b, i.e.,

$$\{a..b\} \quad = \quad \{m \mid a \leq m \leq b\}$$

The set defined by a list of specifiers is the union of the sets defined by the individual specifiers. Thus, the set specified by $\{1, 3..5, 7\}$ consists of the natural numbers

$$\{1\ ,\ 3\ ,\ 4\ ,\ 5\ ,\ 7\}.$$

In the following, we define on several occasions a family of sets A_a indexed by natural numbers. These definitions immediately extend to define sets indexed by general set specifications as follows:

$$A_{sp_1,\ldots,sp_k} \quad = \bigcup_{a \in \{sp_1,\ldots,sp_k\}} A_a.$$

Thus, $A_{1,3..5,7}$ is given by

$$A_1 \cup A_3 \cup A_4 \cup A_5 \cup A_7.$$

3 The Program as a Fair Transition System

The program we wish to study can be given as

$$\textbf{MUTEX} :: \left[\begin{array}{c} \mathit{flag}\colon \textbf{array}[0..n-1] \textbf{ of } 0..4 \textbf{ where } \mathit{flag}[0..n-1] = 0 \\ P[0] \mid\mid P[1] \mid\mid \ldots \mid\mid P[n-1] \end{array} \right]$$

Each process $P[i], i = 0, \ldots, n-1$ of the program is given by:

local j : $[0..n-1]$ **where** $j = 0$

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

Below, we identify the four components of a fair transition system, namely, state variables, states, transitions, and precondition, for the MUTEX program. This identification enables us to view the program as a fair transition system, and apply to it the verification methods that will be later presented for a general fair transition system.

- V – The state variables are given by

$$L_0, ...L_{12}, \quad flag[0], ..., flag[n-1], \quad j_0, ..., j_{n-1}.$$

  The variables $L_0, ..., L_{12}$, are control variables that range over subsets of $\{0, ..., n-1\}$. At any state of the computation, L_k, for $k = 0, ..., 12$, contains the indices of all 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. The variables $j_0, ..., j_{n-1}$ represent the current values of the local variable j of the processes $P[0], ..., P[n-1]$, respectively. As we will see below, we assume that a compound test such as $\forall j : 0 \leq j < n : (flag[j] < 3)$ is performed by several atomic tests, each checking the current value of $flag[j]$ for some j. The variable j_i indicates that the next flag value to be tested by $P[i]$ is $flag[j_i]$.

- 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

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

  Thus, at the initial state of the program, all processes reside at the location ℓ_0, and the values of $flag[0], ..., flag[n-1]$ and of $j_0, ..., j_{n-1}$ are all zero.

To express the movement of control effected by the transitions, we introduce the following abbreviations:

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

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 the control does not move in any of the processes.

Note that the movement of control from ℓ_k to ℓ_m is represented by claiming that the new value of the set L_k, which contains the indices of all the processes that currently reside at ℓ_k, equals its old value minus the process i that has moved away. Similarly, L_m is updated by the addition of i.

The Transitions

Before presenting the actual transitions corresponding to the MUTEX program, we present a general approach to the assignment of transitions to compound tests, such as the tests appearing in statements $\ell_3, \ell_5, \ell_7, \ell_9$, and ℓ_{11} of the program. These tests all perform a check of whether a certain condition $p(j)$ holds for all or some $j = 0, ..., n-1$. We do not consider the interpretation of such tests as *atomic*, assuming them to be fully completed by a single transition, as a realistic representation of what really happen in concurrent systems. Instead, we consider them as *molecular* (see [PZ86]), and assign a separate transition to the check of $p(j)$ for each individual j. We refer the readers to [MP89c] for an analysis of the same program under the assumption of atomic compound tests, as well as a comparison of several versions of molecular compound tests.

There are three types of compound tests that appear in the **MUTEX** program. We discuss each of them separately. To represent an intermediate situation in the performance of a compound test by the process $P[i]$, we use the state variable j_i that points at the next value of j, for which $p(j)$ should be tested. In the representation we consider here, j_i is initiated at 0 and incremented by 1 to get to the new index to be tested. Consequently the value $j_i = n$ indicates the completion of the compound test.

In [MP89c] we also consider other orders in which the range $0..n-1$ can be scanned, and study the effect the different scanning orders may have on the behavior of the program. In fact, we show there that the program is correct if we follow an *ascending* scanning order, which is the one adopted here, and is incorrect for any other scanning order.

In defining the transition relation ρ_τ corresponding to the transition τ, we adopt the following convention. We present a *compact transition relation* R_τ, which contains the conditions under which τ is enabled, and the effect τ has on the variables it may modify. The full transition relation ρ_τ is given by a conjunction of R_τ with a list of clauses $u' = u$ for each variable u whose primed version does not appear in R_τ, i.e., a variable that is obviously preserved by τ.

Assume that the following compound test appears in the program for the process $P[i]$, for some predicate $p(j)$ which depends on j.

ℓ_r: **wait until** $\forall j : (0 \leq j < n) : p(j)$
ℓ_s:

With this statement we associate the transition $\tau_r[i]$, whose compact transition relation is given by

$$R_r[i] : \ (i \in L_r) \wedge \left(\begin{array}{ll} & [(j_i = n) \wedge move(i, r, s)] \\ \vee & [(j_i < n) \wedge p(j_i) \wedge stay \wedge (j_i' = j_i + 1)] \\ \vee & [(j_i < n) \wedge \neg p(j_i) \wedge stay \wedge (j_i' \leq j_i)] \end{array} \right)$$

The first clause of this formula corresponds to the case that the compound test has terminated, as is identified by $j_i = n$. This means that for each $j = 0, ..., n-1$, we have encountered a state in which $p(j)$ was true. By no means is it implied that there ever was a state in which $p(j)$ held for all $j = 0, ..., n-1$ at the same time.

The second clause of this transition corresponds to the case that j_i is still in the range $0, ..., n-1$ and $p(j_i)$ is found to be true. In this case, j_i is stepped up, but control still remains at ℓ_r.

The third clause corresponds to the case that $p(j_i)$ is found to be false. Several strategies are possible at this point. Some implementations may decide to restart the testing cycle from the beginning, and consequently reset j_i to 0 on detecting a false $p(j_i)$. Other implementations leave j_i as it is and will try again to test $p(j_i)$ until it is found to be true. The clause presented above is general enough to cover both these strategies by requiring only that j_i does not increase. Obviously, if we prove the program to be correct under this more general representation, the results will hold, in particular, for the two specific implementations we have described above.

Next, let us consider a statement of the form

ℓ_r: **wait until** $\exists j : (0 \leq j < n) : p(j)$
ℓ_s:

With this statement we associate the transition $\tau_r[i]$, whose compact transition relation is given by

$$R_r[i] : \ (i \in L_r) \wedge \left(\begin{array}{ll} & [p(j_i) \wedge move(i, r, s)] \\ \vee & [\neg p(j_i) \wedge stay \wedge (j_i' = (j_i + 1)\mathbf{mod}\ n)] \end{array} \right)$$

The first clause of this formula corresponds to the case that $p(j_i)$ is found to hold. In this case, the process $P[i]$ moves on to ℓ_s.

The second clause corresponds to the case that $p(j_i)$ does not hold. In this case $P[i]$ remains at ℓ_r and j_i is stepped to its next value. The incrementation of j_i is done modulo n, so that the value following $n-1$ is again 0.

Finally, let us consider the statement

ℓ_r : **if** $\exists j : (0 \leq j < n) : p(j)$
  then [ℓ_s : ...]
  else [ℓ_t : ...]

With this statement we associate the transition $\tau_r[i]$, whose compact transition relation is given by

$$R_r[i] : \ (i \in L_r) \wedge \left(\begin{array}{ll} & [(j_i = n) \wedge move(i, r, t)] \\ \vee & [(j_i < n) \wedge p(j_i) \wedge move(i, r, s)] \\ \vee & [(j_i < n) \wedge \neg p(j_i) \wedge stay \wedge (j_i' = j_i + 1)] \end{array} \right)$$

The first clause of this formula corresponds to the case that the search for a j that satisfies $p(j)$ has been completed, apparently without finding such a j. Consequently, the result of the compound test is *false* and we proceed to the **else** clause.

The second clause of the formula corresponds to the case that the current value of j_i satisfies $p(j_i)$. This means that the test is successful and we proceed to the **then** clause.

The third clause of the formula corresponds to the case that the current value of j_i does not satisfy $p(j_i)$. We therefore step j_i to its next value and stay in place.

Having considered the general form of the transitions associated with the three types of molecular tests we have in our program, we proceed to present the transitions for the program.

We recall that according to our set notations

$$\begin{array}{lcll} L_{i_1,i_2,\ldots,i_m} & = & L_{i_1} \cup L_{i_2} \cup \ldots \cup L_{i_m} & \\ L_{i..k} & = & L_i \cup L_{i+1} \cup \ldots \cup L_k & \quad \text{for } i < k \end{array}$$

Below, we list the transitions associated with the process $P[i]$. For each such process there exist one or more transitions corresponding to each statement. For the statement labeled by ℓ_r we denote the corresponding transition by $\tau_r[i]$ and the associated compact transition relation by $R_r[i]$.

- $R_0[i] : (i \in L_0) \wedge move(i, 0, 1)$
  This transition corresponds to the case that $P[i]$ is at ℓ_0 and moves inside the **loop** statement.
- $R_1[i] : (i \in L_1) \wedge \big(stay \vee move(i, 1, 2)\big)$

  This compact transition relation consists of two clauses representing a non-deterministic choice. The first clause corresponds to the case that the process $P[i]$ decides to remain in its non-critical section for awhile longer. The situation that, from a certain point on, a process remains forever in its non-critical section (which we want to include) is represented by this process consistently choosing this clause of the transition relation from that point on.

  The second clause of the compact transition relation corresponds to the case that $P[i]$ decides to quit its non-critical section and move from ℓ_1 to ℓ_2.
- $R_2[i] : (i \in L_2) \wedge move(i, 2, 3) \wedge (flag'[i] = 1) \wedge (j_i' = 0)$

  This transition corresponds to the case that the process $P[i]$ moves from ℓ_2 to ℓ_3 while setting $flag[i]$ to 1. According to our convention, $flag'[k] = flag[k]$ for all $k \neq i$. Note that since ℓ_3 performs a molecular test, we reset j_i to 0 on entering ℓ_3 as preparation for the compound test to be performed at ℓ_3.

- $R_3[i]$:

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

The first clause of this compact transition relation corresponds to a successful termination of the test, as a result of which, $P[i]$ moves to ℓ_4. The second clause corresponds to the case that the next tested value of j_i satisfies $flag[j_i] < 3$, as a result of which, j_i is incremented to its next value. The last clause corresponds to the case that a tested $flag[j_i]$ is found to be greater or equal to 3. In this case, we allow resetting j_i to any value not exceeding its current value.

- $R_4[i] : (i \in L_4) \wedge move(i,4,5) \wedge (flag'[i] = 3) \wedge (j_i' = 0)$

Process $P[i]$ moves to ℓ_5 while setting $flag[i]$ to 3 and resetting j_i to 0.

- $R_5[i]$:

$$(i \in L_5) \wedge \left(\begin{array}{ll} & [(j_i = n) \wedge move(i,5,8)] \\ \vee & [(j_i < n) \wedge (flag[j_i] = 1) \wedge move(i,5,6)] \\ \vee & [(j_i < n) \wedge (flag[j_i] \neq 1) \wedge stay \wedge (j_i' = j_i + 1)] \end{array} \right)$$

The first clause of the compact transition relation corresponds to the case that the test has terminated unsuccessfully, and consequently $P[i]$ moves to ℓ_8. The second clause represents the case that $flag[j_i] = 1$. Consequently, $P[i]$ moves to ℓ_6. The last clause corresponds to the case that the current value of j_i does not satisfy $flag[j_i] = 1$. Consequently, the process stays in the test and steps j_i to the next value.

- $R_6[i] : (i \in L_6) \wedge move(i,6,7) \wedge (flag'[i] = 2) \wedge (j_i' = 0)$

Set $flag[i]$ to 2 and j_i to 0.

- $R_7[i]$:

$$(i \in L_7) \wedge \left(\begin{array}{ll} & [(flag[j_i] = 4) \wedge move(i,7,8)] \\ \vee & [(flag[j_i] \neq 4) \wedge stay \wedge (j_i' = (j_i + 1) \mathbf{mod}\ n)] \end{array} \right)$$

The first clause of the compact transition relation represents the case that $flag[j_i]$ equals 4. In that case the search has terminated and $P[i]$ moves to ℓ_8. The second clause corresponds to the case that $flag[j_i]$ does not equal 4. In that case the search continues by stepping j_i to its next value.

- $R_8[i] : (i \in L_8) \wedge move(i,8,9) \wedge (flag'[i] = 4) \wedge (j_i' = 0)$

Process $P[i]$ moves from ℓ_8 to ℓ_9 while setting $flag[i]$ to 4 and j_i to 0.

- $R_9[i]$:

$$(i \in L_9) \wedge \left(\begin{array}{ll} & [(j_i = i) \wedge move(i, 9, 10)] \\ \vee & [(j_i < i) \wedge (flag[j_i] < 2) \wedge stay \wedge (j'_i = j_i + 1)] \\ \vee & [(j_i < i) \wedge (flag[j_i] \geq 2) \wedge stay \wedge (j'_i \leq j_i)] \end{array} \right)$$

  The first clause of the compact transition relation represents a successful completion of the test, which runs for j_i ranging from 0 to $i - 1$. $P[i]$ moves to ℓ_{10}. The second clause represents the case that $j_i < i$ and the current j_i satisfies $flag[j_i] < 2$. Consequently, the process increments j_i. The last clause represents the case that the current j_i does not satisfy $flag[j_i] < 2$.

- $R_{10}[i] : (i \in L_{10}) \wedge move(i, 10, 11) \wedge (j'_i = i + 1)$

  The activity of the process inside the critical section is represented by the single transition that moves from ℓ_{10} to ℓ_{11}. This represents the commitment that, differently from the non-critical section, the activity within the critical section must always terminate. Note that on moving to ℓ_{11} we reset j_i to $i+1$ to initialize the search at ℓ_{11} to start from that value.

- $R_{11}[i] : (i \in L_{11}) \wedge$

$$\left(\begin{array}{ll} & [(j_i = n) \wedge move(i, 11, 12)] \\ \vee & [(j_i < n) \wedge (flag[j_i] < 2 \vee flag[j_i] > 3) \wedge stay \wedge (j'_i = j_i + 1)] \\ \vee & [(j_i < n) \wedge (2 \leq flag[j_i] \leq 3) \wedge stay \wedge (j'_i \leq j_i)] \end{array} \right)$$

  The first clause of this compact transition relation corresponds to a successful termination of the test. Consequently, $P[i]$ moves to ℓ_{12}. The second clause corresponds to the case that $j_i < n$ and $flag[j_i] < 2 \vee flag[j_i] > 3$. Consequently, process $P[i]$ moves to the next value of j_i. The third clause corresponds to the case that $2 \leq flag[j_i] \leq 3$, and therefore j_i is reset to any value not exceeding its current value.

- $R_{12}[i] : (i \in L_{12}) \wedge move(i, 12, 0) \wedge (flag'[i] = 0)$

  Process $P[i]$ moves from ℓ_{12} to the location at which the main loop restarts another execution of its body, while resetting $flag[i]$ to 0.

4 Invariance Properties

For an assertion p, we say that p is (generally) *valid*, and write $\models p$, if p is true on all possible states. All the known tautologies and theorems of the predicate calculus are obviously valid.

We say that the assertion p is *valid over the program P* (also described as being *P-valid*), and write $P \models p$, if p holds over all the P-accessible states.

Clearly, if the assertion p is P-valid it is an *invariant* property of the program P. That is, it holds over all the states that can arise in any computation of the program P.

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

We will illustrate these rules by proving the main properties of the program MUTEX. To facilitate the expression of properties for this program, we introduce the following notation:

$$\begin{array}{lclcll} N_i & = & |L_i| & & & \\ N_{i_1,i_2,\ldots,i_m} & = & |L_{i_1,i_2,\ldots,i_m}| & = & N_{i_1} + N_{i_2} + \ldots + N_{i_m} & \\ N_{i..k} & = & |L_{i..k}| & = & N_i + N_{i+1} + \ldots + N_k & \text{for } i < k \end{array}$$

The main invariance property of the program MUTEX can be expressed by the assertion $N_{10} \leq 1$. This assertion limits the number of processes that can be concurrently executing at ℓ_{10}, which corresponds to the critical section, to be at most 1. Thus, we have to prove

$$P \models (N_{10} \leq 1)$$

for the MUTEX program.

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 only exception to this convention are rules that deal at the same time with both general and P-validity, such as the IMP rule presented below.

IMP

(Import) rule: $\quad (\models p) \vdash (P \models p)$.

This rule states that if the assertion p is generally valid, it is in particular P-valid. It is used to import general validities into the P-validity framework.

MP

(Modus Ponens) rule: $\quad \{p \rightarrow q, p\} \ \vdash \ q$.

This rule infers the P-validity of q from the P-validity of $p \rightarrow q$ and q.

The above two auxiliary rules are independent of the particular program analyzed. The following INV rule refers to the elements of the program, and is the main working tool for establishing invariance properties.

The rule uses a special case of a particular formula, to which we refer as the *verification condition* of the transition τ, relative to the assertions p and q. This formula has the form

$$(p \wedge \rho_\tau) \rightarrow q'$$

In this formula, ρ_τ is the transition relation corresponding to τ, and q', the *primed version* of the assertion q, is obtained from q by replacing each variable occurring in q by its primed version. Let s and s' be two states. Since ρ_τ holds over the joint interpretation $\langle s, s' \rangle$ iff s' is a τ-successor of s, and q' states that q holds over s', it is not difficult to see that

> If the verification condition $(p \wedge \rho_\tau) \rightarrow q'$ is P-valid, then every τ-successor of a p-state is a q-state.

The INV rule is given by

$$\begin{array}{lll} \text{INV} & \text{I1.} & \varphi \rightarrow p \\ & \text{I2.} & \Theta \rightarrow \varphi \\ & \text{I3.} & (\varphi \wedge \rho_\tau) \rightarrow \varphi' \quad \text{for every } \tau \in \mathcal{T} \\ \hline & & p \end{array}$$

The INV rule uses an auxiliary assertion φ which, by premise I2, holds initially, and by premise I3 is propagated from each state to its successor. This shows that φ is an invariant of the program, that is, it holds continuously over all computations of P. Since, by I1, the assertion φ implies p, it follows that p is also an invariant of the program.

Example Consider the trivial program with a single state variable x, the pre-condition $x = 0$, and a single transition τ whose transition relation is given by $\rho_\tau : x' = x + 1$. Observe that this program has a single infinite computation, given by

$$\langle x : 0 \rangle, \langle x : 1 \rangle, \langle x : 2 \rangle, \ldots$$

We wish to prove for this program the trivial invariance property

$$x \geq 0.$$

To prove this property, we use the INV rule with $p = \varphi : (x \geq 0)$. The rule requires showing the validity of the following three premises:

I1. $(x \geq 0) \rightarrow (x \geq 0)$

I2. $(x = 0) \rightarrow (x \geq 0)$

I3. $\big((x \geq 0) \wedge (x' = x + 1)\big) \rightarrow (x' \geq 0)$

Clearly all the three premises are generally valid, which establishes the invariance of $x \geq 0$.

We proceed to establish several invariants for the program MUTEX, which together will yield the desired result.

Simple Invariants

First, we establish a list of invariants that connect for each $i = 0, ..., n-1$ the location of $P[i]$ with the value of $flag[i]$. To facilitate the expression of these invariants, we define

$$F_k = \{i \mid 0 \leq i < n \ , \ flag[i] = k\}.$$

Thus, F_k, for $k = 0, ..., 4$, denotes the set of indices i such that $flag[i] = k$. We also recall the abbreviations

$$\begin{array}{lcll} F_{i_1,i_2,...,i_m} & = & F_{i_1} \cup F_{i_2} \cup ... \cup F_{i_m} & \\ F_{i..k} & = & F_i \cup F_{i+1} \cup ... \cup F_k & \quad \text{for } i < k \end{array}$$

Using these notations, the invariants relating the location of processes to their $flag$ values can be expressed as relations between F_k and L_r for various values of r and k.

$$\begin{array}{llcl} \text{IF0.} & F_0 & = & L_{0..2} \\ \text{IF1.} & F_1 & = & L_{3,4} \\ \text{IF2.} & F_2 & \subseteq & L_{7,8} \\ \text{IF3.} & F_3 & \subseteq & L_{5,6,8} \\ \text{IF4.} & F_4 & = & L_{9..12} \end{array} \qquad \begin{array}{llcl} \text{IL5.} & L_5 & \subseteq & F_3 \\ \text{IL6.} & L_6 & \subseteq & F_3 \\ \text{IL7.} & L_7 & \subseteq & F_2 \\ \text{IL8.} & L_8 & \subseteq & F_{2,3} \end{array}$$

The invariants IF0,...,IF4 restrict the locations at which $P[i]$ can reside when the value of $flag[i]$ is $0, ..., 4$, respectively. For example, the invariant IF4 claims that the value of $flag[i]$ is 4 *iff* $P[i]$ is at one of the locations $\ell_9, ..., \ell_{12}$. The invariants IL5,...,IL8 restrict the value of $flag[i]$ while $P[i]$ is at the locations $\ell_5, \ldots, \ell_8$, respectively. For example, the invariant IL8 claims that when $P[i]$ is at ℓ_8 its flag value must be 2 or 3. This is the only location in the program in which the value of the flag is not uniquely determined.

Let us see, for example, how an invariant such as IF1 is established. To prove $F_1 = L_{3,4}$, we actually prove

$$(i \in F_1) \leftrightarrow (i \in L_{3,4}),$$

for every $i = 0, ..., n-1$. We apply the INV rule with $p = \varphi : \ (i \in F_1) \leftrightarrow (i \in L_{3,4})$. There are three premises to verify.

Premise I1 is trivial since $\varphi = p$ for our case. Premise I2 requires showing that Θ implies $F_1 = L_{3,4}$. It is not difficult to see that Θ actually implies $F_1 = L_{3,4} = \phi$, since initially there are no processes whose flag value is 1, and there are no processes residing at either ℓ_3 or ℓ_4.

The premise that requires more attention is premise I3. Here we are called for writing a separate implication of the form $(\varphi \wedge \rho_\tau) \rightarrow \varphi'$, for every transition τ in the program. There are some simple heuristics that let us discard immediately many transitions as automatically guaranteed to preserve φ. The simplest and most effective one is:

All transitions that do not modify any of the variables on which φ depends are guaranteed to preserve φ.

This heuristic leads immediately to the conclusion that, for the assertion $F_1 = L_{3,4}$, we should only be concerned with the following transitions that we consider one by one (we represent the transitions by the unique locations with which they are associated):

$\ell_2[i]$ – The transition relation for this transition implies $(i \in L_3') \wedge (i \in F_1')$, since it causes $P[i]$ to move to ℓ_3 and sets $flag[i]$ to 1. Consequently, it implies φ'.

$\ell_3[i]$ – Even though this transition can potentially modify both L_3 and L_4, it does it in a way that preserves $L_{3,4}$. Consequently, the transition relation implies $(F_1' = F_1) \wedge (L_{3,4}' = L_{3,4})$, which ensures that φ is preserved.

$\ell_4[i]$ – The corresponding transition relation implies $i \notin F_1'$ (the transition sets $flag[i]$ to 3), and $i \notin L_{3,4}'$ (the transition leaves ℓ_4). Consequently, φ' is established, as both sides of the equivalence become false.

It is clear that these are the only transitions that modify any of the variables on which φ depends.

We conclude that $(i \in F_1) \leftrightarrow (i \in L_{3,4})$ is an invariant assertion, and therefore so is $F_1 = L_{3,4}$.

Proving Mutual Exclusion

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

We refer the reader to [Szy88] for a detailed explanation of the basic ideas on which the MUTEX program is based. Here we extract just the main observations. 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 partitioned into several segments. We refer to the location ℓ_4 as the *doorway*, to the section $\ell_{5..7}$ as the *waiting room* and to the section $\ell_{8..12}$, 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, i.e., $L_{8..12}$ changes its value from empty to non-empty, the doorway is locked, i.e., $L_4 = \phi$. The doorway remains locked until the last process leaves the inner sanctum. This 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 must be empty. If we believe this to be a true invariant, then the fact that $L_{8..12}$ is non-empty should prevent any new processes coming to ℓ_3 to cross over into ℓ_4. The only thing that can prevent processes 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 $\ell_{8..12}$ has a flag value of 3 or 4. This is because a flag value of 3 which is held by a process at $\ell_{5,6}$ 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 $\ell_{10..12}$, then it must be the minimal (having the least index) of all the processes in $\ell_{5..12}$. This is expressed by the invariant

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

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

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

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 for the entry to the critical section. By claim C2., only one process at a time can reside in the region $\ell_{10..12}$ which includes the critical section – the process whose index is minimal among all the processes in $\ell_{5..12}$. It follows that mutual exclusion is maintained.

If we were working in a framework such that the compound tests are considered atomic, then the conjunction

$$\varphi_0 \quad : \quad A_0 \wedge A_1 \wedge A_2 \wedge A_3$$

could have been shown to be invariant from which, by A_2, mutual exclusion would have followed.

Unfortunately, we have to deal with molecular tests, which require an extension to the above list of invariants. Consider any region of consecutive locations that is mentioned in one of the previous invariants, and which is preceded by a compound test. For example, $\ell_{10..12}$ is such a region, where the relevant compound test is the one at ℓ_9. The 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 $\ell_{5..12}$. This is because the simple invariants connecting flag values to locations imply that $flag[k] \geq 2$ while $P[k]$ is at $\ell_{5..12}$.

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 various 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 j_i is greater than k, then we know that the value of $flag[k]$ has already been tested and found satisfactory, i.e., smaller than 2.

Consequently, to adapt the assertion A_2 to the molecular case, we should replace the simple region reference $i \in L_{10..12}$, appearing there, by the extended reference $i \in L_{10..12} \vee (i \in L_9 \wedge j_i > k)$. By applying such range extensions to the assertions $A_0, \ldots, A_3$, 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 together assertions A_0 and A_1 to the molecular case. The basic idea is to show for any k that if $P[i]$ is either at $\ell_{8..12}$ or at 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 $\ell_{8..12}$, we can only claim the existence of such a blocking r, such that $P[r]$ is also at $\ell_{8..12}$ 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 the program 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, for example, what transitions may possibly affect the assertion B_1. A critical transition of $P[i]$ is the 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 (due to IL5) $flag[i] = 3$. Another potentially critical transition of k is the one that increases j_k beyond r. However, due to $flag[r] > 2$, such a transition is disabled. For this argument to hold it is essential that the indices j in ℓ_3 are scanned in increasing order.

Lastly, we consider the transition of $P[r]$ from ℓ_{12} to ℓ_0, while resetting its flag value to 0. There are two possibilities. If r is the last process in $\ell_{8..12}$, then after the transition $L_{8..12}$ will become empty, causing B_1 to hold trivially. If r is not the last, there exists another process, say $P[t]$ in $\ell_{8..12}$. Then, due to B_2, which states that r is the minimal process in $\ell_{8..12}$, r must be 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.

5 Response Properties

Next to be considered is the class of *response* properties. The typical response property is expressed by the formula

$$p \Rightarrow\!\!\!\succ \Diamond\, q,$$

for assertions p and q. A sequence of states σ is said to satisfy the response formula $p \Rightarrow\!\!\!\succ \Diamond q$ if every p-position $i \geq 0$, is followed by a q-position $j \geq i$. Such a response formula is said to be valid over the program P (also called P-valid), denoted by $P \models (p \Rightarrow\!\!\!\succ \Diamond q)$, if all the computations of P satisfy the formula. This means that every occurrence of (a state satisfying) p in the execution of P, is followed by an occurrence of q. We will often omit the prefix $P \models$ when stating the validity of a response formula over P.

The temporal logic adepts will recognize $\Rightarrow\!\!\!\succ \Diamond$ as the combination of the two operators $\Rightarrow\!\!\!\succ$ and $\Diamond$ (see for example [MP89a]). However, for our purpose here it suffices to view it as a single binary temporal operator, whose semantics has been defined above. It is very similar to the *leads-to* operator of *Unity* ([CM88]).

The following axioms and rules identify the basic properties of the *response* operator $\Rightarrow\!\!\!\succ \Diamond$.

RFLX

(Reflexivity) axiom:

$$p \Rightarrow\!\!\!\succ \Diamond\, p$$

This axiom expresses the fact that every p-position is trivially followed by a p-position, namely itself.

TRNS

(Transitivity) rule:

$$\{p \Rightarrow\!\!\!\succ \Diamond\, q \ , \ q \Rightarrow\!\!\!\succ \Diamond\, r\} \quad \vdash \quad p \Rightarrow\!\!\!\succ \Diamond\, r$$

This rule states the transitivity of the *response* operator. It claims that if every p-position is followed by a q-position, and every q-position is followed by an r-position, then certainly every p-position must be followed by an r-position.

MON

(Monotonicity) rule:

$$\{p \Rightarrow\!\!\!\!\!\!\;\;\Diamond q \;,\; \tilde{p} \rightarrow p \;,\; q \rightarrow \tilde{q}\} \quad \vdash \quad \tilde{p} \Rightarrow\!\!\!\!\!\!\;\;\Diamond \tilde{q}$$

This rule allows us to replace in a valid response formula the antecedent p by a *stronger* assertion $\tilde{p}$, and the consequent q by a *weaker* assertion $\tilde{q}$, and obtain another valid formula.

DISJ

(Disjunction) rule:

$$\{p \Rightarrow\!\!\!\!\!\!\;\;\Diamond r \;,\; q \Rightarrow\!\!\!\!\!\!\;\;\Diamond r\} \quad \vdash \quad (p \vee q) \Rightarrow\!\!\!\!\!\!\;\;\Diamond r$$

This rule combines the two response formulae, $p \Rightarrow\!\!\!\!\!\!\;\;\Diamond r$ and $q \Rightarrow\!\!\!\!\!\!\;\;\Diamond r$, into the formula $(p \vee q) \Rightarrow\!\!\!\!\!\!\;\;\Diamond r$. It allows us to prove the last formula by separately considering the case that p holds and the case that q holds. In this way it supports proof by cases.

The Basic Response Rule

The axiom and three rules listed above are independent of the particular program analyzed, and describe the basic properties of the response operator. We now present a rule that enables us to establish the validity of a response formula over a program.

The rule singles out a particular transition τ_h, to which we refer as the *helpful* transition. It can establish response formulae $p \Rightarrow\!\!\!\!\!\!\;\;\Diamond q$, such that a single activation of the transition τ_h is sufficient to achieve q. We therefore refer to this rule as the *basic* or *single step* response rule.

$$\begin{array}{lll} \textbf{RESP} & \text{R1.} & p \rightarrow (q \vee \varphi) \\ & \text{R2.} & (\rho_\tau \wedge \varphi) \rightarrow (q' \vee \varphi') \quad \text{for every } \tau \in \mathcal{T} \\ & \text{R3.} & (\rho_{\tau_h} \wedge \varphi) \rightarrow q' \\ & \text{R4.} & \varphi \rightarrow (q \vee En(\tau_h)) \\ \hline & & p \Rightarrow\!\!\!\!\!\!\;\;\Diamond q \end{array}$$

Premise R1 ensures that p implies q or φ. Premise R2 states that any transition of the program, either leads from φ to q, or preserves φ. Premise R3 states that the helpful transition τ_h leads from φ to q. Premise R4 ensures that τ_h is enabled as long as φ holds and q does not occur.

It is not difficult to see that if p happens, say at position $i \geq 0$, but is not followed by a q, then φ must hold continuously beyond this position, and the helpful transition

τ_h is never taken beyond i. The latter fact follows from premise R3, which states that taking τ_h from a φ-state immediately leads to a q-state, contradicting the assumption that q never happens beyond i. However, due to R4, this means that τ_h is continuously enabled but never taken beyond position i, which violates the requirement of justice for τ_h.

Example

We will illustrate the application of this rule on the following program.

$$\textbf{out } x, y : \textbf{ integer where } x = 0\ ,\ y = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_0 : \textbf{ while } x = 0 \textbf{ do} \\ \quad [\ell_1 : \ y \ := \ y + 1] \\ \ell_2 : \end{array}\right] \quad || \quad P_2 :: \left[\begin{array}{l} m_0 : \ x \ := \ 1 \\ m_1 : \\ \qquad \cdots \end{array}\right]$$

This program consists of two processes, P_1 and P_2. Process P_1 continuously increments y while waiting for x to become non-zero. Process P_2 consists of a single statement, assigning 1 to x.

The response property we wish to establish for this program is that of termination. It can be expressed by the formula

$$(at\_\ell_0 \wedge at\_m_0) \Rightarrow\!\!\!\!\!\!\succ \Diamond(at\_\ell_2 \wedge at\_m_1),$$

that states that the event of being at the beginning of the program $(at\_\ell_0 \wedge at\_m_0)$ is eventually followed by the event of being at the end of the program $(at\_\ell_2 \wedge at\_m_1)$.

This property is established by a sequence of lemmas, each applying one of the rules presented above.

Lemma 1 (x eventually set to 1)

$$(at\_\ell_0 \wedge at\_m_0) \Rightarrow\!\!\!\!\!\!\succ \Diamond(at\_\ell_{0,1} \wedge at\_m_1 \wedge (x = 1))$$

This lemma claims that eventually the variable x is set to 1 by the process P_2, which then moves to m_1. When this happens, process P_1 is still executing within the loop region $\ell_{0,1}$.

To prove the lemma we choose

$$\begin{array}{rcl} p & : & at\_\ell_0 \wedge at\_m_0 \\ \varphi & : & at\_\ell_{0,1} \wedge at\_m_0 \wedge (x = 0) \\ \tau_h & : & \tau_{m_0} \\ q & : & at\_\ell_{0,1} \wedge at\_m_1 \wedge (x = 1) \end{array}$$

and apply the RESP rule.

It is not difficult to see that p implies φ, provided we prove first the obvious invariant $at\_m_0 \rightarrow (x = 0)$. It is also clear that taking τ_{m_0} from a φ-state leads to a state satisfying q, and taking any other transition, i.e., τ_{ℓ_0} or τ_{ℓ_1}, preserves φ. Obviously φ implies that τ_{m_0} is enabled.

Lemma 2 (From ℓ_0 to ℓ_2)

$$(at\_\ell_0 \wedge at\_m_1 \wedge (x = 1)) \Rightarrow\!\!\!\!\succ \Diamond (at\_\ell_2 \wedge at\_m_1)$$

Follows from the RESP rule, by taking $\varphi = p$ and $\tau_h = \tau_{\ell_0}$.

Lemma 3 (From ℓ_1 to ℓ_0)

$$(at\_\ell_1 \wedge at\_m_1 \wedge (x = 1)) \Rightarrow\!\!\!\!\succ \Diamond (at\_\ell_0 \wedge at\_m_1 \wedge (x = 1))$$

Follows from the RESP rule, by taking $\varphi = p$ and $\tau_h = \tau_{\ell_1}$.

Lemma 4 (From ℓ_1 to ℓ_2)

$$(at\_\ell_1 \wedge at\_m_1 \wedge (x = 1)) \Rightarrow\!\!\!\!\succ \Diamond (at\_\ell_2 \wedge at\_m_1)$$

Follows by transitivity (rule TRNS) from Lemma 3 and Lemma 2.

Lemma 5 (From $\ell_{0,1}$ to ℓ_2)

$$(at\_\ell_{0,1} \wedge at\_m_1 \wedge (x = 1)) \Rightarrow\!\!\!\!\succ \Diamond (at\_\ell_2 \wedge at\_m_1)$$

Follows by the DISJ rule from Lemma 4 and lemma 5, using the equivalence

$$\begin{array}{l} (at\_\ell_{0,1} \wedge at\_m_1 \wedge (x = 1)) \equiv \\ \quad \big((at\_\ell_0 \wedge at\_m_1 \wedge (x = 1)) \vee (at\_\ell_1 \wedge at\_m_1 \wedge (x = 1))\big). \end{array}$$

Lemma 6 (From $\{\ell_0, m_0\}$ to $\{\ell_2, m_1\}$)

$$(at\_\ell_0 \wedge at\_m_0) \Rightarrow\!\!\!\!\succ \Diamond (at\_\ell_2 \wedge at\_m_1)$$

This lemma which establishes the termination property follows by the TRNS rule from Lemma 1 and Lemma 5. ▪

The Well-Founded Rule for Response

The basic response rule supports the proof of response properties which are established by a *single* helpful step. As we have seen, even the simple example above requires several helpful steps to achieve its goal, i.e., termination. When the number of helpful steps required is small and fixed we can use a sequence of lemmas, each considering a single helpful step, and then combine their results by transitivity and case splitting. However, for the case that a large and a priori unknown number of helpful steps is required, we introduce below a more powerful rule that uses well-founded induction to combine the helpful steps.

We define a *well-founded (embedded) structure* $(\mathcal{A}, \mathcal{B}, \succ)$ to consist of the following components.

- $\mathcal{A}$ – A set of elements.
- $\mathcal{B}$ – A subset of $\mathcal{A}$.
- $\succ$ – A binary relation on $\mathcal{A}$, whose restriction to $\mathcal{B}$ is *well founded.* That is, there does not exist an infinite sequence of elements of $\mathcal{B}$; $\beta_0, \beta_1, \ldots$, such that

$$\beta_0 \succ \beta_1 \succ \ldots.$$

A typical example of a well-founded embedded structure is $(\mathcal{I}nt, \mathcal{N}at, >)$, where $\mathcal{I}nt$ are the integers (including the negative ones), $\mathcal{N}at$ are the natural numbers (including 0), and $>$ is the *greater than* relation. Clearly, $>$ is defined over all the integers but is well founded only over the natural numbers.

Given two well-founded structures, $(\mathcal{A}_0, \mathcal{B}_0, \succ_0)$ and $(\mathcal{A}_1, \mathcal{B}_1, \succ_1)$, we can form their *lexicographical product* $(\mathcal{A}, \mathcal{B}, \succ)$, defined by

- $\mathcal{A}$ is defined as $\mathcal{A}_0 \times \mathcal{A}_1$, i.e., the set of all pairs (α_0, α_1), such that $\alpha_0 \in \mathcal{A}_0$ and $\alpha_1 \in \mathcal{A}_1$.
- $\mathcal{B}$ is defined as $\mathcal{B}_0 \times \mathcal{B}_1$.
- $\succ$ is defined to hold between $(\alpha_0, \alpha_1) \in \mathcal{A}$ and $(\alpha_0', \alpha_1') \in \mathcal{A}$ iff

$$(\alpha_0 \succ \alpha_0') \vee \left[(\alpha_0 = \alpha_0') \wedge (\alpha_1 \succ \alpha_1')\right]$$

It is not difficult to prove that the lexicographical product of two well-founded structures is also a well-founded structure.

For $\succ$, an arbitrary binary relation over $\mathcal{A}$, we define its *reflexive extension* $\succeq$ to hold between $\alpha, \alpha' \in \mathcal{A}$ if either $\alpha = \alpha'$ or $\alpha \succ \alpha'$.

The following rule uses several *intermediate* assertions that hold at the positions lying between the position satisfying p and the position satisfying the goal q. We denote these assertions by φ_i, where i ranges over some finite index set $\mathcal{I}$, and denote their disjunction by $\varphi = \bigvee_{i \in \mathcal{I}} \varphi_i$. Each intermediate assertion φ_i is associated with a transition $\tau_i \in \mathcal{T}$, that is identified as *helpful* for φ_i.

The rule also requires the identification of a *distance function* δ_i, for each $i \in \mathcal{I}$. These functions map the states into the set $\mathcal{A}$ of a well-founded structure $(\mathcal{A}, \mathcal{B}, \succ)$. The intended meaning of these functions is that they measure the distance of the current state from the closest state that satisfies the goal q of the formula $p \Rightarrow \Diamond q$ which is the conclusion of the rule. We refer to the value of the distance function δ_i at a state satisfying φ_i as the *i-rank* of that state, or simply as the *rank* of the state if i is understood from the context.

Assuming that these constructs have been identified, the following rule establishes the P-validity of the formula $p \Rightarrow \Diamond q$.

WELL	W1.	$p \rightarrow (q \vee \varphi)$
	The following premises should hold for each $i \in \mathcal{I}$	
	W2.	for every $\tau \in \mathcal{T}$
		$(\rho_\tau \wedge \varphi_i) \rightarrow \Big(q' \vee \bigvee_{j \in \mathcal{I}}[\varphi'_j \wedge (\delta_i \succ \delta'_j)] \vee [\varphi'_i \wedge (\delta_i = \delta'_i)]\Big)$
	W3.	$(\rho_{\tau_i} \wedge \varphi_i) \rightarrow \Big(q' \vee \bigvee_{j \in \mathcal{I}}[\varphi'_j \wedge (\delta_i \succ \delta'_j)]\Big)$
	W4.	$\varphi_i \rightarrow \Big(q \vee (En(\tau_i) \wedge (\delta_i \in \mathcal{B}))\Big)$
		$p \Rightarrow \Diamond q$

Premise W1 requires that p implies that either q already holds, or the intermediate assertion φ (i.e., one of the φ_i's) holds. Premise W2 requires that taking any transition from a φ_i-state results in a next state which either satisfies q, or satisfies φ_j, for some $j \in \mathcal{I}$, and has a (j-) rank lower than that of the original state, or satisfies φ_i and has an equal rank. Premise W3 requires that taking the *helpful* transition τ_i from a φ_i-state, results in a next state which either satisfies q, or satisfies some φ_j with a lower rank. Premise W4 requires that any state s satisfying φ_i either satisfies q, or is such that τ_i is enabled on it, and the i-rank of s, $\delta_i(s)$, assumes a value in $\mathcal{B}$.

Assume that all the four premises hold. Consider a computation σ and a position m that satisfies p. We wish to prove that some later position satisfies q. Assume to the contrary that all positions later than m (including m itself) do not satisfy q. By W2 each of these positions must satisfy some φ_j and, according to W4, the value of δ_j for this position, to which we refer as the rank of the position, lie within $\mathcal{B}$. By W2, the value of δ_j can either decrease or remain the same. By the assumption that $\succ$ is well founded over $\mathcal{B}$, the value of δ_j can actually decrease only finitely many times. Therefore, there must exist some position $k \geq m$, beyond which δ_j never decreases.

Assume that φ_i is the assertion holding at position k. Since q is never satisfied and δ_j never decreases beyond position k, it follows (by W2) that φ_i holds continually beyond k. By W3, τ_i cannot be taken beyond k, because that would have led to a position satisfying q or to a decrease in δ. By W4, τ_i is continually enabled beyond k yet, by the argument above, it is never taken. This violates the requirement of justice for τ_i. It follows that if all the premises of the rule hold then $p \Rightarrow \Diamond q$ is P-valid.

In many cases, we may use the same ranking function δ for all $i \in \mathcal{I}$. We refer to these as the case of *uniform* ranking function. In these cases it is possible to use a simpler form for the premises W2 and W3, which is given by:

$$\begin{array}{ll} \text{W2.} & \text{for every } \tau \in \mathcal{T} \\ & (\rho_\tau \wedge \varphi_i) \rightarrow \Big(q' \vee [\varphi' \wedge (\delta \succ \delta')] \vee [\varphi_i' \wedge (\delta = \delta')]\Big) \\ \text{W3.} & (\rho_{\tau_i} \wedge \varphi_i) \rightarrow \Big(q' \vee [\varphi' \wedge (\delta \succ \delta')]\Big) \end{array}$$

Proving Accessibility

The main response property one usually wishes to prove for mutual exclusion programs is that of *accessibility*, by which whenever a process departs from its non-critical section it is guaranteed to eventually reach the critical section. In our case we will prove a stronger property which implies accessibility. The property we will prove is

$$(u \notin L_1) \Rightarrow \Diamond (u \in L_1).$$

This property, to which we refer as the *homing* property, states that from any location away from the non-critical section, each process $P[u]$ is guaranteed to home back to the non-critical section. Since in our case, when a process just departs from ℓ_1 it can return to ℓ_1 only via the critical section, the homing property implies accessibility. It also guarantees that processes do not get stuck in any of the locations following the critical section, such as ℓ_{11}. The way we establish the homing property is by a sequence of lemmas, each showing that a process cannot get stuck in any location, except perhaps in the non-critical section. The lemmas corresponding to locations which involve no tests, such as $\ell_0, \ell_2, \ell_4, \ell_6, \ell_8, \ell_{10}$, and ℓ_{12}, are trivial and will be omitted. We will concentrate on the testing locations.

The well-founded structures that we will use are either $(\mathcal{I}nt, \mathcal{N}at, >)$, or the lexicographic products of such structures.

Lemma 1 (Not Stuck at $\ell_{9..12}$)

$$(u \in L_{9..12}) \Rightarrow \Diamond (u \in L_0)$$

This lemma states that if the process $P[u]$ is anywhere within $\ell_{9..12}$, it will eventually return to ℓ_0.

To prove this lemma, we prove first two auxiliary lemmas.

Lemma 1.1 (Evacuation of the Waiting Room)

$$(u \in L_{9..12}) \Rightarrow \Diamond\Big((u \in L_0) \vee [(u \in L_{9..12}) \wedge (L_{5..8} = \phi)]\Big)$$

This lemma states that if $P[u]$ is currently at $\ell_{9..12}$ then either it will reach ℓ_0, or prior to that, the computation will reach a state in which $P[u]$ is still at $\ell_{9..12}$, but the waiting room $\ell_{5..8}$ is empty.

To prove this lemma we use the following intermediate assertions, uniform distance function, and helpful transitions:

$$\begin{array}{lcl} \varphi_{(k,i)} & : & (u \in L_{9..12}) \wedge (L_{5..8} \neq \phi) \wedge (i \in L_k) \\ \delta & : & \Big(4 \cdot N_5 + 3 \cdot N_6 + 2 \cdot N_7 + N_8\ ,\ \sum\limits_{r:r \in L_5} (n - j_r) + \sum\limits_{r:r \in L_7} ((u - j_r) \,\mathbf{mod}\, n)\Big) \\ \tau_{(k,i)} & : & \tau_k[i] \end{array}$$

for $k \in \{5..8\}$ and $i \in \{0..n-1\}$. Thus, we use for the index set $\mathcal{I}$ the set

$$\mathcal{I} \ :\ \Big\{(k,i) \mid k \in \{5..8\}\ ,\ i \in \{0..n-1\}\Big\}$$

Let us convince ourselves that taking any helpful transition decreases the distance function. Clearly a movement of process $P[i]$ from any location in the range $\ell_{5..8}$ to any other location decreases the first component of δ. For example, a movement of $P[i]$ from ℓ_6 to ℓ_7, removes i from L_6, where it has a weight of 3, and adds it to L_7 with a weight of 2. Consequently, the net change in the first component is -1.

Next, let us consider a transition that involves a compound test. Consider, for example, a transition of process $P[i]$ which currently resides at ℓ_5. According to $R_5[i]$ there are three possibilities. The first possibility is that $P[i]$ moves from ℓ_5 to ℓ_8, decreasing δ by $(3,0)$, i.e., 3 in the first component and 0 in the second component. The second possibility is that $P[i]$ moves from ℓ_5 to ℓ_6, decreasing δ by $(1,0)$. The last possibility is that j_i increases by 1, decreasing δ by $(0,1)$, due to the summand $n - j_i$ appearing in the second component of δ.

A somewhat more subtle argument is needed for the consideration of the transitions $\tau_7[i]$. Here there are two possibilities. Either $P[i]$ moves from ℓ_7 to ℓ_8, or j_i is incremented modulo n. In the first case δ decreases by $(1,0)$. In the second case, we have to show that $((u - j_i)\,\mathbf{mod}\, n)$ decreases. First, we observe that, since $u \in L_{9..12}$, $flag[u] = 4$, and therefore the test at ℓ_7 cannot fail for $j_i = u$. We conclude that the second possibility exists only if $j_i \neq u$. In that case we rely on the property of the integers, by which if $0 \leq j_i, u < n$ and $j_i \neq u$, then

$$((u - j_i)\,\mathbf{mod}\, n) > ((u - (j_i + 1))\,\mathbf{mod}\, n).$$

It follows that, in the second case, δ decreases by $(0,1)$.

Next let us show that any non-helpful transition either establishes $u \in L_0$, or at least preserves $\varphi_{(k,i)}$ and δ. Clearly, this is true for $\tau_{12}[u]$. The only other transitions that may be suspected of falsifying $\varphi_{(k,i)}$ or increasing δ are those that may cause new processes to join $\ell_{5..8}$. However, due to the assumption $u \in L_{9..12}$ and the invariant B_1, there are no processes at ℓ_4, and therefore, no new processes can join $\ell_{5..8}$. ▪

Lemma 1.2 (Progress within the Inner Sanctum)

$$[(u \in L_{9..12}) \wedge (L_{5..8} = \phi)] \Rightarrow\!\!\!\succ \diamond (u \in L_0)$$

This lemma claims that if now there is no process within the range $\ell_{5..8}$ then process u will eventually proceed to ℓ_0. Of course, for that to happen, all the processes with lower indices must arrive to ℓ_{10} first and depart via ℓ_{12}.

To prove the lemma we use the following intermediate assertions, uniform distance function, and helpful transitions:

$$\begin{aligned}
\varphi_{(k,i)} &: (u \in L_{9..12}) \wedge (L_{5..8} = \phi) \wedge (i \in L_k) \wedge (i = min_4) \\
\delta &: \Big(4 \cdot N_9 + 3 \cdot N_{10} + 2 \cdot N_{11} + N_{12}\ ,\ n - j_{min_4}\Big) \\
\tau_{(k,i)} &: \tau_k[i]
\end{aligned}$$

for $k \in \{9..12\}$ and $i \in \{0..n-1\}$, and where min_4 is defined to be the minimal element of $F_4 = L_{9..12}$, if that set is not empty, and 0 otherwise. In the case that $L_{9..12}$ is not empty, min_4 denotes the minimal index among all the processes currently residing at $\ell_{9..12}$ and (consequently) having a *flag* value of 4.

It is not difficult to see that the process with the minimal index is always enabled and causes a decrease in the value of the distance function, whatever transition in $\ell_{9..12}$ it takes. ▪

We may now return to the proof of Lemma 1. We proceed as follows:

1. $(u \in L_0) \Rightarrow\!\!\!\succ \diamond (u \in L_0)$ by **RFLX**
2. $\Big((u \in L_0) \vee [(u \in L_{9..12}) \wedge (L_{5..8} = \phi)]\Big) \Rightarrow\!\!\!\succ \diamond (u \in L_0)$ by **DISJ**, 1., and Lemma 1.2.
3. $(u \in L_{9..12}) \Rightarrow\!\!\!\succ \diamond (u \in L_0)$ by **TRNS**, Lemma 1.1, and 2.

This concludes the proof. ▪

Lemma 2 (Not Stuck at ℓ_7)

$$(u \in L_7) \Rightarrow\!\!\!\succ \diamond (u \in L_8)$$

To prove this lemma, we establish first an additional invariant, using the **INV** rule.

$$B_4 : \quad (L_{6,7} \neq \phi) \rightarrow (L_{3..5} \cup L_{8..12} \neq \phi)$$

This invariant guarantees that if there is some process in the region $\ell_{6,7}$, then there is also some process in $\ell_{3..5}$ or in $\ell_{8..12}$. It is not difficult to show that the assertion B_4 holds initially and is preserved by any transition. In particular, we may rely on B_3 to show that no process can leave $L_{8..12}$ while $L_{6,7}$ is non-empty.

Then we prove two auxiliary lemmas.

Lemma 2.1 (Entering $\ell_{9..12}$)

$$(u \in L_7) \Rightarrow \Diamond \Big((u \in L_7) \wedge (L_{9..12} \neq \phi) \Big)$$

Note that due to the invariant IF4, the set $L_{9..12}$ is precisely the set F_4, i.e., all the processes in this region have a $flag$ value of 4. To prove the lemma, we use the following intermediate assertions, distance functions, and helpful transitions:

$$\begin{array}{lcl} \varphi_{(3,i)} & : & (u \in L_7) \wedge (L_{9..12} = \phi) \wedge (L_{4..6,8} = \phi) \wedge (i \in L_3) \\ \delta_{(3,i)} & : & \big(5 \cdot N_{0..3} + 4 \cdot N_4 + 3 \cdot N_5 + 2 \cdot N_6 + N_8 \ , \ n - j_i\big) \\ \tau_{(3,i)} & : & \tau_3[i] \\ & & \text{For each } k \in \{4..6, 8\} \\ \varphi_{(k,i)} & : & (u \in L_7) \wedge (L_{9..12} = \phi) \wedge (i \in L_k) \\ \delta_{(k,i)} & : & \big(5 \cdot N_{0..3} + 4 \cdot N_4 + 3 \cdot N_5 + 2 \cdot N_6 + N_8 \ , \ n - j_i\big) \\ \tau_{(k,i)} & : & \tau_k[i] \end{array}$$

where i ranges over $\{0..n-1\}$.

As we see, the index set $\mathcal{I}$ is partitioned into the two subsets $\{(3,i) \mid i \in \{0..n-1\}\}$, and $\{(k,i) \mid k \in \{4..6,8\}, i \in \{0..n-1\}\}$. The transitions corresponding to the first subset are considered helpful (as we see from $\varphi_{(3,i)}$) only when $L_{4..6,8}$ is empty. This is necessary because $P[i]$ is guaranteed to progress when it is at ℓ_3 only if $L_{4..6,8}$ is empty. Otherwise, the test at ℓ_3 may cause j_i to decrease, or at least not to increase. The invariant B_4 is used to establish the premise

$$(u \in L_7) \rightarrow \bigvee_{(k,i) \in \mathcal{I}} \varphi_{(k,i)}.$$

Essential to the proof is the observation that some process can move from ℓ_7 to ℓ_8 only if $L_{9..12}$ is already non-empty. ▟

Lemma 2.2 (Escaping ℓ_7)

$$\Big((u \in L_7) \wedge (L_{9..12} \neq \phi) \Big) \Rightarrow \Diamond (u \in L_8)$$

To prove this lemma, we use the following single intermediate assertion, single distance function, and single helpful transition:

$$\begin{array}{rcl} \varphi_u & : & (u \in L_7) \wedge (L_{9..12} \neq \phi) \\ \delta & : & (min_4 - j_u) \mathbf{mod}\ n \\ \tau_u & : & \tau_7[u] \end{array}$$

It is not difficult to see that when $L_{9..12} \neq \phi$, $flag[min_4] = 4$, and therefore $P[u]$ will find $flag[j_u] = 4$, at the latest, when $j_u = min_4$. ▪

We may now return to the proof of Lemma 2. By transitivity, we may combine the results of Lemma 2.1 and Lemma 2.2 to obtain

$$(u \in L_7) \Rightarrow \Diamond (u \in L_8),$$

as claimed by Lemma 2. ▪

Lemma 3 (Not Stuck at ℓ_5)

$$(u \in L_5) \Rightarrow \Diamond (u \in L_{6,8})$$

This lemma is easily proven by taking

$$\begin{aligned} \varphi_u &: u \in L_5 \\ \delta &: n - j_u \\ \tau_u &: \tau_5[u] \end{aligned}$$

Progress in the execution of the compound test at ℓ_5 is guaranteed independently of the flag values encountered. ▪

Lemma 4 (Not Stuck at ℓ_3)

$$(u \in L_3) \Rightarrow \Diamond (u \in L_4)$$

We define the following sets of process indices

$$\begin{aligned} L_5(j > F_1) &: \{r \mid r \in L_5\ ,\ j_r > F_1\} \\ Block_3 &: L_{8..12} \cup L_5(j > F_1) \end{aligned}$$

where the inequality $j_r > F_1$ is defined to hold if F_1 is non-empty and j_r is greater than any element of F_1. Consequently, if F_1 is empty, then so is $L_5(j > F_1)$. Note that by the invariant B_0 it follows that if $L_5(j > F_1)$ is not empty, then $L_4 = \phi$, which implies $F_1 = L_3$.

The set $Block_3$ represents the set of processes that may potentially block the progress of any processes currently at ℓ_3 (including $P[u]$). Note that we have to add to $\ell_{8..12}$ also the processes that are in ℓ_5 and have already checked $flag[j]$ for all $j \in F_1$. This is because such processes may potentially move to ℓ_8. On the other hand, processes that are in ℓ_5 but have not checked $flag[j]$, for some $j \in F_1$, can only move to ℓ_6.

We prove the following auxiliary lemmas.

Lemma 4.1

$$(u \in L_3) \Rightarrow \Diamond \Big((u \in L_4) \vee [(u \in L_3) \wedge (L_5(j > F_1) = \phi)] \Big)$$

This lemma states that if $P[u]$ is currently at ℓ_3 then either it will reach ℓ_4, or prior to that, the computation will reach a state in which $P[u]$ is still at ℓ_3, but no process $P[i]$ is currently at ℓ_5 with $j_i > F_1$.

To prove the lemma, we use

$$\begin{aligned}
\varphi_i &: (u \in L_3) \wedge (i \in L_5(j > F_1)) \\
\delta_i &: (|L_5(j > F_1)| \ , \ n - j_i) \\
\tau_i &: \tau_5[i]
\end{aligned}$$

for $i \in \{0..n-1\}$. Thus, the relevant processes are those that are at ℓ_5 and have already checked $flag[j]$, for every $j \in F_1$. Note that no new processes can join $L_5(j > F_1)$ since any process checking $flag[j]$, for some $j \in F_1$ proceeds immediately to ℓ_6. ∎

Lemma 4.2

$$((u \in L_3) \wedge (L_5(j > F_1) = \phi)) \Rightarrow \Diamond \Big((u \in L_4) \vee [(u \in L_3) \wedge (Block_3 = \phi)]\Big)$$

This lemma establishes that if $P[u]$ does not reach ℓ_4, then at least the set $Block_3$ becomes empty. To prove the lemma, we use

$$\begin{aligned}
\varphi_{(7,i)} &: (u \in L_3) \wedge (L_5(j > F_1) = \phi) \wedge (i \in L_7) \wedge (L_{9..12} \neq \phi) \\
\delta_{(7,i)} &: \Big(8 \cdot N_5 + 7 \cdot N_6 + 6 \cdot N_7 + 5 \cdot N_8 + 4 \cdot N_9 + 3 \cdot N_{10} + 2 \cdot N_{11} + N_{12} \ , \\
&\qquad\qquad ((min_4 - j_i)\,\mathbf{mod}\; n)\Big)
\end{aligned}$$

For each $k \in \{5, 6, 8\}$

$$\begin{aligned}
\varphi_{(k,i)} &: (u \in L_3) \wedge (L_5(j > F_1) = \phi) \wedge (i \in L_k) \wedge (L_{8..12} \neq \phi) \\
\delta_{(k,i)} &: \Big(8 \cdot N_5 + 7 \cdot N_6 + 6 \cdot N_7 + 5 \cdot N_8 + 4 \cdot N_9 + 3 \cdot N_{10} + 2 \cdot N_{11} + N_{12} \ , \ n - j_i\Big)
\end{aligned}$$

For each $k \in \{9..12\}$

$$\begin{aligned}
\varphi_{(k,i)} &: (u \in L_3) \wedge (L_5(j > F_1) = \phi) \wedge (i \in L_k) \wedge (L_{5..8} = \phi) \wedge (i = min_4) \\
\delta_{(k,i)} &: \Big(8 \cdot N_5 + 7 \cdot N_6 + 6 \cdot N_7 + 5 \cdot N_8 + 4 \cdot N_9 + 3 \cdot N_{10} + 2 \cdot N_{11} + N_{12} \ , \ n - j_i\Big)
\end{aligned}$$

for $i \in \{0..n-1\}$. The overall range of k in the index set $\{(k, i)\}$ used in this lemma is $\{5..12\}$, and as usual $\tau_{(k,i)} = \tau_k[i]$.

Note that since $L_{8..12} \neq \phi$, no new processes can enter ℓ_5. ∎

Lemma 4.3

$$[(u \in L_3) \wedge (Block_3 = \phi)] \Rightarrow \Diamond (u \in L_4)$$

Note that when $Block_3$ is empty it cannot become non-empty as long as $P[u]$ stays at ℓ_3 with a $flag$ value of 1. At most, processes can accumulate at ℓ_7. Consequently, we use the following constructs:

$$\begin{array}{rcl}
\varphi_{(3,i)} & : & (u \in L_3) \wedge (Block_3 = \phi) \wedge (i \in L_3) \wedge (L_{4..6} = \phi) \\
\delta_{(3,i)} & : & \left(4 \cdot N_{0..3} + 3 \cdot N_4 + 2 \cdot N_5 + N_6 \ , \ n - j_i\right) \\
 & & \text{For each } k \in \{4..6\} \\
\varphi_{(k,i)} & : & (u \in L_3) \wedge (Block_3 = \phi) \wedge (i \in L_k) \\
\delta_{(k,i)} & : & \left(4 \cdot N_{0..3} + 3 \cdot N_4 + 2 \cdot N_5 + N_6 \ , \ n - j_i\right)
\end{array}$$

for $i \in \{0..n-1\}$.

Note that when $L_{4..6}$ is empty, any transition $\tau_3[i]$ is helpful. ▪

It is not difficult to combine the results of Lemmas 4.1, 4.2, and 4.3, using reflexivity, disjunction, and transitivity, to obtain the result of Lemma 4, namely:

$$(u \in L_3) \Rightarrow \Diamond (u \in L_4)$$

▪

This concludes the proof of the homing property for the MUTEX program.

6 Precedence Properties

Next, we consider properties that are expressed by the formula

$$p \Rightarrow q_0 \,\mathsf{U} \ldots \,\mathsf{U}\, q_{r-1} \,\mathsf{U}\, q_r,$$

for any $r > 0$. Adepts in temporal logic will recognize this formula as a nested *unless* formula. For our purposes here, it suffices to consider it as a temporal operator of $r+2$ arguments.

To define the semantics of this operator, we deal with *half-open intervals* of the form $[i..j)$, for $i \leq j$. Such an interval consists of all the positions k, such that $i \leq k < j$. Note that if $i = j$, the interval is empty. For the two intervals $[i..j)$ and $[j..k)$, we say that the second interval is adjacent to (or follows) the first, and observe that their union is also a half-open interval, given by $[i..k)$. For infinite computations, we allow also intervals of the form $[i..\omega)$ for an integer i and the interval $[\omega, \omega)$, which by definition is empty.

Given a computation $\sigma : s_0, s_1, \ldots$, we say that the interval $[i..j)$ is a *p-interval* if for every $k \in [i..j)$, s_k satisfies p. By definition, an empty interval is a p-interval for every assertion p.

A computation σ is said to satisfy the *precedence* formula $p \Rightarrow q_0 \,\mathsf{U} \ldots \,\mathsf{U}\, q_{r-1} \,\mathsf{U}\, q_r$ if for every p-position i there exists a sequence of positions $i = i_0 \leq i_1 \leq \ldots \leq i_r \leq |\sigma|$, such that $[i_0..i_1)$ is a q_0-interval,...,$[i_{r-1}..i_r)$ is a q_{r-1}-interval, and finally, if $i_r < |\sigma|$, then i_r is a q_r-position. That is, it requires that any p-position initiates a q_0-interval, which is followed by a succession of $q_1, \ldots, q_{r-1}$-intervals, where the q_{r-1}-interval either extends to the end of the computation or is terminated by a q_r-position. Note that this definition allows some of the intermediate intervals to be empty, and any of them to extend to the end of the computation $|\sigma|$ (which may also be ω), and this forces all the succeeding intervals to have the form $[|\sigma|..|\sigma|)$, and therefore to be empty.

The precedence formula $p \Rightarrow q_0 \mathsf{U} \ldots \mathsf{U} q_{r-1} \mathsf{U} q_r$ is said to be P-valid if it satisfied by all computations of the program P.

Let us see how the property of *linear wait* as claimed in [Szy88] for the MUTEX program, can be expressed by a precedence formula. Consider the precedence formula

$$[(u \in L_3) \wedge (v \in L_{1,2})] \Rightarrow (v \notin L_{10}) \mathsf{U} (v \in L_{10}) \mathsf{U} (v \notin L_{10}) \mathsf{U} (u \in L_{10})$$

This formula considers the question of how many times can the process $P[v]$ *overtake* the process $P[u]$ on its way to the critical section. It considers a starting position in which $P[u]$ has already made public its intention to proceed to the critical section (by setting $flag[u]$ to 1, while $P[v]$ has not done so yet. In this starting position $P[u]$ is somewhat ahead of $P[v]$. The precedence formula predicts that, following such a position, there will be an interval in which $P[v]$ is not critical (i.e., not in the critical section ℓ_{10}), followed by an interval in which $P[v]$ is critical, followed by an interval in which $P[v]$ is again non-critical, followed by a position in which $P[u]$ is critical. Consequently, it claims that between the starting position and the entry of $P[u]$ to the critical section, there can be at most one visit of $P[v]$ to the critical section. Note that the interval of $P[v]$ being critical can also be empty. This is why we say *at most* once. Note that this property does not guarantee that $P[u]$ will eventually get to the critical section, because any of the preceding intervals may extend to the end of the computation. In [MP83] this property is called 1-*bounded overtaking.*

First let us consider two rules that characterize some of the basic properties of the precedence operator.

MON (Monotonicity)

$$\frac{\begin{array}{c} p \Rightarrow q_0 \mathsf{U} \ldots \mathsf{U} q_{r-1} \mathsf{U} q_r \\ \tilde{p} \rightarrow p, \; q_0 \rightarrow \tilde{q}_0, \ldots, q_r \rightarrow \tilde{q}_r \end{array}}{\tilde{p} \Rightarrow \tilde{q}_0 \mathsf{U} \ldots \mathsf{U} \tilde{q}_{r-1} \mathsf{U} \tilde{q}_r}$$

This rule allows us to replace in a valid precedence formula the antecedent p by a *stronger* assertion $\tilde{p}$, and the assertions $q_0, \ldots, q_r$ appearing in the consequent by *weaker* assertions $\tilde{q}_0, \ldots, \tilde{q}_r$, and obtain another valid formula.

For the next rule we introduce the following notations

$$\begin{array}{lcll} q_{i_1, i_2, \ldots, i_m} & = & q_{i_1} \vee q_{i_2} \vee \ldots \vee q_{i_m} & \\ q_{i..k} & = & q_i \vee q_{i+1} \vee \ldots \vee q_k & \text{for } i < k \end{array}$$

TEL

(Telescoping) rule:

$$\begin{array}{c} \text{For each } i = 0, \ldots, r-1 \\ p \Rightarrow \cdots q_i \,\mathcal{U}\, q_{i+1} \cdots \quad \vdash \quad p \Rightarrow \cdots q_{i,i+1} \cdots \end{array}$$

For the case of $i < r-1$, this rule allows us to replace (telescope) the prediction of a q_i-interval followed by a q_{i+1}-interval, by the prediction of a single $(q_i \vee q_{i+1})$-interval (i.e., a $q_{i,i+1}$-interval). For the end case of $i = r-1$, the rule allows us to replace the prediction of a q_{r-1}-interval followed by a q_r-position, by the prediction of a $(q_{r-1} \vee q_r)$-position (i.e., a $q_{r-1,r}$-position).

The next rule is the main proof rule for establishing precedence properties of a given program.

$$\begin{array}{ll} \textbf{PREC} & \text{R1.} \quad p \rightarrow q_{0..r} \\ & \text{For each } i = 0, \ldots, r-1, \text{ and each } \tau \in \mathcal{T} \\ & \text{R2.} \quad (q_i \wedge \rho_\tau) \rightarrow q'_{i..r} \\ \hline & p \Rightarrow q_0 \,\mathcal{U} \ldots \mathcal{U}\, q_{r-1} \,\mathcal{U}\, q_r \end{array}$$

Proving Bounded Overtaking

We are now ready to prove the property of 1-bounded overtaking, or linear wait, for the program **MUTEX**.

For our case, we take $r = 6$ and define as follows:

$$p : (u \in L_3) \wedge (v \in L_{1,2})$$

The assertion q_0 is given by

$$q_0 : (u \in L_3) \wedge (Block_3 \neq \phi) \wedge (v \in L_{1..3})$$

where $Block_3$ is as defined before, i.e. $Block_3 = L_{8..12} \cup L_5(j > F_1)$.

The assertions $q_1, \ldots, q_6$ are given by

$$\begin{array}{lcl} q_1 & : & (u \in L_{3,4}) \wedge (Block_3 = \phi) \wedge \Big((v \in L_{1..4,6,7}) \vee [(v \in L_5) \wedge (j_v \leq u)]\Big) \\ q_2 & : & (u \in L_{5..7}) \wedge (L_{8..12} = \phi) \wedge (v \in L_{1..7}) \\ q_3 & : & (u \in L_{5..9}) \wedge (L_{8..12} \neq \phi) \wedge (L_4 = \phi) \wedge (v \in L_{5..9}) \\ q_4 & : & (u \in L_{5..9}) \wedge (L_{8..12} \neq \phi) \wedge (L_4 = \phi) \wedge (v \in L_{10}) \\ q_5 & : & (u \in L_{5..9}) \wedge (L_{8..12} \neq \phi) \wedge (L_4 = \phi) \wedge (v \in L_{0..3,11,12}) \\ q_6 & : & (u \in L_{10}) \end{array}$$

It is beyond the scope of this paper to check the second premise for $i = 0, \ldots, 5$ and all the transitions. We will, however, indicate in the table below what transitions $\tau_k[i]$ may lead from q_f to q_t for $f = 0, \ldots, 5$ and $t = 0, \ldots, 6$. Note that the same transition may lead from q_f to two or more q_t's. By observing that the only non-empty entries in this table correspond to $f \leq t \leq 6$, we are convinced that the second premise of the PREC rule is valid. In computing such successors, we may rely on any of the previously proven invariants.

From	To: q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_0	$\tau_{0..12}$	$\tau_{2,12}$					
q_1		$\tau_{0..12}$	$\tau_5[u]$				
q_2			$\tau_{0..12}$	τ_5		τ_5	
q_3				$\tau_{0..12}$	$\tau_9[v]$		$\tau_9[u]$
q_4					$\tau_{0..12}$	$\tau_{10}[v]$	
q_5						$\tau_{0..12}$	$\tau_9[u]$

We may conclude, by the PREC rule, that the precedence formula

$$p \Rightarrow q_0 \,\mathsf{U}\, q_1 \,\mathsf{U}\, q_2 \,\mathsf{U}\, q_3 \,\mathsf{U}\, q_4 \,\mathsf{U}\, q_5 \,\mathsf{U}\, q_6$$

is valid over the program MUTEX.

Next, we apply the monotonicity rule with $\tilde{p} = p$, $\tilde{q}_0 = \tilde{q}_1 = \tilde{q}_2 = \tilde{q}_3 : (v \notin L_{10})$, $\tilde{q}_4 : (v \in L_{10})$, $\tilde{q}_5 : (v \notin L_{10})$, and $\tilde{q}_6 : (u \in L_{10})$. This application is justified by observing that $\tilde{p} = p$, and getting easily convinced that q_i implies $\tilde{q}_i$ for $i = 0, \ldots, 6$. The application yields the formula

$$p \Rightarrow \tilde{q}_0 \,\mathsf{U}\, \tilde{q}_1 \,\mathsf{U}\, \tilde{q}_2 \,\mathsf{U}\, \tilde{q}_3 \,\mathsf{U}\, \tilde{q}_4 \,\mathsf{U}\, \tilde{q}_5 \,\mathsf{U}\, \tilde{q}_6.$$

Observing that $\tilde{q}_0 = \ldots = \tilde{q}_3$, we may telescope the first four intervals together. This yields the formula

$$p \Rightarrow \tilde{q}_0 \,\mathsf{U}\, \tilde{q}_4 \,\mathsf{U}\, \tilde{q}_5 \,\mathsf{U}\, \tilde{q}_6,$$

which, when substituting the assertions standing for p and $\tilde{q}_i$, leads to

$$[(u \in L_3) \wedge (v \in L_{1,2})] \Rightarrow (v \notin L_{10}) \,\mathsf{U}\, (v \in L_{10}) \,\mathsf{U}\, (v \notin L_{10}) \,\mathsf{U}\, (u \in L_{10}).$$

Acknowledgment

We gratefully acknowledge the help rendered by Rajeev Alur, Ed Chang, and Tom Henzinger who critically read various versions of this manuscript. Special thanks are due to Roni Rosner for his dedicated technical help and most helpful suggestions.

References

[AS85] B. Alpern and F.B. Schneider, Defining liveness, *Info. Proc. Lett.* **21**, 1985, pp. 181–185.

[CM88] K.M. Chandy and J. Misra, *Parallel Program Design*, Addison-Wesley, 1988.

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

[MP83] Z. Manna and A. Pnueli, Proving precedence properties: The temporal way, *Proc. 10th Int. Colloq. Aut. Lang. Prog.*, Lec. Notes in Comp. Sci. 154, Springer, 1983, pp. 491–512.

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

[MP89a] Z. Manna and A. Pnueli, The anchored version of the temporal framework, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, eds.), Lec. Notes in Comp. Sci. 354, Springer, 1989, pp. 201–284.

[MP89b] Z. Manna and A. Pnueli, Completing the temporal picture, *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, Lec. Notes in Comp. Sci. 372, Springer, 1989, pp. 534–558.

[MP89c] Z. Manna and A. Pnueli, *An Exercise in the Verification of Multi – Process Programs*, Technical Report, Stanford University, 1989. To appear in a book dedicated to E.W. Dijkstra.

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

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