Nondeterminism and the Correctness of Parallel Programs

Lawrence Flon¹ and Norihisa Suzuki

May 1977

Carnegie-Mellon University
Pittsburgh, Pennsylvania

Approved for public release; distribution unlimited.

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b). Distribution is unlimited.
A. D. BLOSE
Technical Information Officer
We present weakest preconditions which describe weak correctness, blocking, deadlock, and starvation for nondeterministic programs. A procedure for converting parallel programs to nondeterministic programs is described, and the correctness of various example parallel programs is treated in this manner. Among these are a busy-wait mutual exclusion scheme, and the problem for the Five Dining Philosophers.
Non-determinism and the Correctness of Parallel Programs

Lawrence Flon and Norihisa Suzuki

May 1977

Carnegie-Mellon University
Pittsburgh, Pennsylvania

Abstract: We present weakest pre-conditions which describe weak correctness, blocking, deadlock, and starvation for nondeterministic programs. A procedure for converting parallel programs to nondeterministic programs is described, and the correctness of various example parallel programs is treated in this manner. Among these are a busy-wait mutual exclusion scheme, and the problem of the Five Dining Philosophers.


This work was supported in part by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074, and in part by the National Science Foundation under grant DCR74-24573, and monitored by the Air Force Office of Scientific Research.

1Author's address 8/77: Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.
1. Introduction

Some of the most difficult to find bugs in systems programs arise in situations of concurrent access to data structures. Thus the need for a precise understanding of the semantics of parallel programs is clear. The attainment of a suitable formal definition of parallel program semantics will allow construction of automatic verification tools. These should help to eliminate the frustrating, "irreproducible" bugs which usually plague an operating system.

The earliest attempts at verifying parallel programs, e.g. [Habermann 72, Brinch Hansen 72] were basically informal and concerned primarily with weak correctness. [Flon 76] describes a semi-formal approach to verifying concurrently accessed abstract data types which contain path expressions. Hoare [Hoare 74] gave a formal axiomatization for monitors. These latter two papers take a data-oriented view of parallelism which, though quite reasonable for the problems treated, is not particularly suited for proof of such strong correctness properties as safety from blocking and deadlock. By data-oriented we mean with regard only to concurrent access to data structures, independent of the control structures of the actual processes.

The approach of Owicki and Gries [Owicki 76] is a process-oriented extension of Hoare's axiom system for sequential programs [Hoare 69] - close attention is paid to the form of the actual processes. Our goals are closely related to those of Owicki and Gries, although our approach is quite different. There have been similar approaches, notably [Keller 76, Lamsveerde 76]. The work presented here is an outgrowth of some ideas discussed in [Flon 77].
We shall begin by using Dijkstra's predicate transformers, primarily the concept of weakest pre-condition, to describe the semantics of nondeterministic programs which differ from Dijkstra's [Dijkstra 76] in that they are not necessarily required to terminate. In particular we will discuss the weak correctness of such nondeterministic programs, along with the strong correctness issues of blocking, deadlock, and starvation.

Subsequently we discuss the relationship between nondeterminism and parallelism. We show how parallel programs can be effectively transformed into nondeterministic programs, so that the results obtained for nondeterministic programs can be indirectly used to verify parallel programs. Several examples are treated, including the problem of the Five Dining Philosophers, which relies on a synchronization primitive, and an old Mutual Exclusion scheme which relies only on the indivisibility of access to memory.

2. The Nondeterministic Command Rep

Dijkstra [Dijkstra 76] describes the semantics of the repetitive guarded command, written

\[
\text{do } B_1 \rightarrow S_1 \parallel B_2 \rightarrow S_2 \parallel \ldots \parallel B_n \rightarrow S_n \text{ od}
\]

in terms of the weakest (i.e. necessary and sufficient) pre-condition to the command which guarantees termination with a given post-condition. The intent of DO is that the guards \( B_j \) are evaluated, a true one is chosen (nondeterministically), and its corresponding statement \( S_j \) is executed. The process is continued until no \( B_j \) is true, at which point DO terminates. Formally,
wp(DO, R) = (3k ≥ 0)H_k(R)

H_0(R) = R \land \neg(3j ∈ 1..n)B_j

H_{k+1}(R) = (3j ∈ 1..n)B_j \land (Yj ∈ 1..n)(B_j \Rightarrow wp(S_j, H_k(R)))

where H_k(R) gives the weakest pre-condition which assures that R will be established and the loop will terminate in at most k steps. We define for our purposes a similar command, REP, denoted

rep B_1 → S_1 || B_2 → S_2 || ... B_n → S_n per

REP has similar behavior to DO, with the exception that REP will not terminate unless the command "exit" is executed. Thus, if no guard evaluates to true, REP will "hang" - i.e. it will not terminate. If there is in general no final state, how are we to define the semantics of REP?

3. Weak Correctness

3.1 Invariance

One way to represent the weak correctness of REP is in terms of a set of assertions \{P_j \mid 1 ≤ j ≤ n\}, where n is the number of guards and P_j is guaranteed to hold whenever S_j is executable. That is, a selection is about to be made from among the guards and B_j is true. If we denote the fact that P must always hold whenever B_j evaluates to true as wp_{P_j}(REP, P), we have

\[ wp_{P_j}(REP, P) = (Yk ≥ 0)W_k^j(P) \]

\[ W_0^j(P) = B_j \Rightarrow P \]

\[ W_{k+1}^j(P) = (Yi ∈ 1..n)(B_i \Rightarrow wp(S_i, W_k^i(P))) \]
\( W^j_k(P) \) is the weakest condition which guarantees \( B_j \Rightarrow P \) after exactly \( k \) statements have been executed, if it is possible to execute that many. The predicate \( \omega_l p(S, R) \) is Dijkstra's weakest liberal pre-condition, and is equivalent to \( \omega p \) without requiring termination.

As an example, consider

```ruby
rep
l=0 \rightarrow x=x/2; \textbf{if} \; \text{odd}(x) \; \textbf{then} \; l+1 \; \textbf{else} \; \textbf{skip} \; \textbf{fi} ||
\textbf{if} \; l=1 \; \rightarrow \; x=x-1 \; \textbf{per}
```

Let \( P_1 \) be \( \text{even}(x) \) and \( P_2 \) be \( \text{odd}(x) \). Then by the above recurrence,

\[
\omega_l p_1(REP, \text{even}(x)) = (l=0 \Rightarrow \text{even}(x))
\]

\[
\omega_l p_2(REP, \text{odd}(x)) = (l=1 \Rightarrow \text{odd}(x))
\]

### 3.2 Potentiality

Let us also consider the question of whether or not a given guard can ever (has the potential to) evaluate to true. That is, what is the weakest pre-condition to \( REP \) that guarantees the existence of a finite execution sequence which leads to the truth of \( B_j \)? Let

\[
\omega_p o t(REP, P) = (\exists k \geq 0)V_k(P)
\]

\[
V_0(P) = P
\]

\[
V_{k+1}(P) = (\exists j \in 1..n)(B_j \land \omega p(S_j, V_k(P)))
\]

Then \( \omega p o t(REP, B_j) \) is the desired weakest pre-condition. In the above recurrence, \( V_k(P) \) gives the weakest pre-condition that guarantees the existence of a length-\( k \) execution sequence which establishes \( P \).

For example, if \( REP \) is

```ruby
rep A \rightarrow x=0 || B \rightarrow x=1 || x=1 \rightarrow x=x+1 \textbf{per}
```
then guard 3 \((x=1)\) has the potential to become true iff \(wpot(REP, x=1)\), which evaluates to \((x=1 \lor B)\), is true at entry.

4. Strong Correctness

4.1 Blocking

Neither \(wp_j\) nor \(wpot\) is sufficient to guarantee that REP will do anything useful. For example, it may be that eventually no guards will be true and the command will "hang". We shall say that REP is blocked if such a state is reached, and is blocking-free if no such state can possibly be reached. The weakest pre-condition that guarantees that REP is blocking-free is

\[
wp(REP) = \forall k \exists G_k
\]

\[
G_0 = \text{true}
\]

\[
G_{k+1} = (\exists j \in 1..n) B_j \land (\forall j \in 1..n) (B_j \rightarrow wp(S_j, G_k))
\]

Here, \(G_k\) is the weakest pre-condition that guarantees at least a length-\(k\) execution sequence. For example, if REP is

\[
\text{rep \ } x>y \rightarrow x+x-y \parallel y>x \rightarrow y+y-x \text{ per}
\]

then by the above recurrence

\[
wp(REP) = x\neq y \land (x\leq 0 \lor y\leq 0)
\]

4.2 Deadlock

Deadlock in a system of parallel processes is defined in [Holt 72] as a state in which "one or more processes are blocked forever because of requirements that can never be satisfied." We will call a nondeterministic program deadlocked if it reaches a state from which, for any guard, there is no possible execution sequence which will
lead to its truth. A *deadlock-free* program is one in which it is not possible to reach a
deadlock state.

Consider the predicate $\wpot(REP, B_j)$ for some REP command. That predicate
gives the weakest pre-condition that guarantees the existence of an execution
sequence which leads to the truth of guard $j$. Suppose $\wpot(REP, B_j)$ is always true
whenever a guard selection is made, and that REP is blocking-free. Then REP can
never reach a deadlock state with respect to guard $j$, since $B_j$ always has the potential
to evaluate to true. This condition, denoted $\wpd(j)(REP)$ is precisely defined by

$$\wpd(j)(REP) = \wpb(\text{REP}) \land (\forall k \in \{1..n\}) \wpk(REP, \wpot(REP, B_j))$$

### 4.3 Starvation

The phenomenon of *starvation* in a system of parallel processes is another
strong correctness issue that we must consider in addition to blocking and deadlock.
Dijkstra [Dijkstra 71] briefly discusses this issue with respect to the problem of the
Five Dining Philosophers. In a nondeterministic program, we say that a particular
statement may *starve* if it is possible for the program to reach a state in which the
statement’s guard is false, and state transitions which leave it false may continue to be
taken indefinitely. Thus a given guard $j$ is starvation-free if it is not possible to reach
a state from which there is an execution sequence which forever maintains $\neg B_j$. Let
this condition be denoted $\wsp(j)(REP)$. Then

$$\wsp(j)(REP) = \wpb(\text{REP}) \land \neg \wpot(REP, U\neg(B_j))$$

- $U(R) = (\forall k \geq 0) \ U_k(R)$
- $U_0(R) = R$
- $U_{k+1}(R) = (\exists i \in \{1..n\})(B_i \land \wp(S_i, U_k(R)))$

$U_k(R)$ gives the weakest pre-condition which assures the existence of a length-$k$
execution sequence during which \( R \) is always true. \( U(R) \) requires that there be an unbounded execution sequence which maintains the truth of \( R \). Thus \( \text{wsp}_j(\text{REP}) \) gives the weakest pre-condition which denies the possibility of reaching a state in which \( U(-B_j) \) holds.

4.4 Invariants

It is not always necessary to compute all of the stated recurrences in the previous sections in order to verify a given program. The following theorems follow from the previously defined recurrences:

Theorem 1
\[
[(\forall k \in 1..n)(f \land B_k \Rightarrow \text{wp}(S_k, f))] \land (f \Rightarrow (B_j \Rightarrow P))
\]
\[\vdash f \Rightarrow \text{wlp}_j(\text{REP}, P)\]
That is, any predicate \( f \) which is invariant across each statement and implies \( (B_j \Rightarrow P) \) will suffice to guarantee \( \text{wlp}_j(\text{REP}, P) \).

Theorem 2
\[
[(\forall k \in 1..n)(f \land B_k \Rightarrow \text{wp}(S_k, f))] \land (f \Rightarrow (\exists k \in 1..n)B_k)
\]
\[\vdash f \Rightarrow \text{wp}(\text{REP})\]
Similarly, an invariant predicate which implies the existence of a true guard must be sufficient to guarantee absence of blocking.

Theorem 3
\[
[(\forall k \in 1..n)(f \land B_k \Rightarrow \text{wp}(S_k, f))] \land (f \Rightarrow (\exists k \in 1..n)B_k) \land (f \Rightarrow \text{wpo}(\text{REP}, B_j))
\]
\[\vdash f \Rightarrow \text{wlp}_j(\text{REP})\]
Any invariant predicate which implies both safety from blocking and the potential for \( B_j \) to be established must guarantee guard \( j \) to be safe from deadlock.
Theorem 4

\[ \left( \forall k \in 1 \ldots n \right) \left( J \land B_k \Rightarrow wp(S_k, J) \right) \land \left( J \Rightarrow \left( \exists k \in 1 \ldots n \right) B_k \right) \land \left( J \Rightarrow \neg U(-B_j) \right) \]

\[ \vdash J \Rightarrow wsp_j(REP) \]

If an invariant predicate implies that no unbounded execution sequence exists which keeps \( B_j \) false, then guard \( j \) must be free from starvation.

4.5 Example

Consider the command

\[
\begin{align*}
\text{rep} & \quad x > 0 \land y < n \rightarrow x+x-1; y+y+1 || \\
& \quad y > 0 \land z < n \rightarrow y+y-1; z+z+1 || \\
& \quad z > 0 \land w < n \rightarrow z+z-1; w+w+1 || \\
& \quad w > 0 \land x < n \rightarrow w+w-1; x+x+1
\end{align*}
\]

\[
\text{per}
\]

Let \( J \) be

\[
0 < x < n \land 0 < y < n \land 0 < z < n \land 0 < w < n
\]

We will show that if \( J \) is true at entry, the command is safe from blocking. By Theorem 2, \( J \Rightarrow wp(REP) \) because

1) \( J \land B_1 \Rightarrow wp(S_1, J) \)

\[
0 < x < n \land 0 < y < n \land 0 < z < n \land 0 < w < n
\]

\[
\Rightarrow 1 < x < n + 1 \land -1 < y < n - 1 \land 0 < z < n \land 0 < w < n
\]

\[
\land 0 < x + y + z + w < 4n
\]

(the others follow by symmetry)

2) \( J \Rightarrow (\exists k \in 1 \ldots n) B_k \)

\[
0 < x < n \land 0 < y < n \land 0 < z < n \land 0 < w < n
\]

\[
\Rightarrow (x > 0 \land y < n) \lor (y > 0 \land z < n) \lor (z > 0 \land w < n) \lor (w > 0 \land x < n)
\]

5. Reduction of parallel programs to nondeterministic programs

We have seen that such problems as weak correctness, blocking, deadlock, and
starvation can be formalized for nondeterministic programs. These results can be applied to parallel programs if we can effectively convert parallel programs to equivalent nondeterministic programs. Here the meaning of equivalence is that for any execution sequence of one program, there is a corresponding execution sequence of the other such that values of variables in the parallel program have the same history sequence. (Note that we will have to introduce program counters to transform parallelism to nondeterminism, so they are only equivalent in this sense.) For a rigorous treatment of models of parallel computation see [Karp 69].

We will outline a procedure for transforming parallel programs made up of the \texttt{cobegin}-\texttt{coend} construct with conditional critical regions for synchronization [Brinch Hansen 73]. The sequential parts of these programs consist of assignment, conditionals, while-loops, compounds, and sequencing. We will use the notion of \textit{effective indivisibility} of program segments. A segment is effectively indivisible if the final values of variables are always determined only by their initial values. That is, they are not affected by the other processes. For example, in the program

\begin{verbatim}
cobegin x+x+1 // x+x+1 coend
\end{verbatim}

\texttt{x+x+1} is not indivisible because the final value of \texttt{x} may be either 1 or 2 more than the initial value. If we convert the program to

\begin{verbatim}
cobegin c-; x+c+1 // d-; x+d+1 coend
\end{verbatim}

then each assignment is indivisible. The conditional critical region itself is by definition effectively indivisible, so the program

\begin{verbatim}
cobegin
  with x when true do x+x+1 //
  with x when true do x+x+1
coend
\end{verbatim}

is in indivisible form.
We first convert the entire parallel program to indivisible form - that is, a program in which every assignment statement which is outside of a conditional critical region is indivisible. This is accomplished by introducing variables local to each process as in the previous example. Next, each statement is converted as follows:

1) \texttt{cobegin P}_1//\ldots//\texttt{P}_n\texttt{coend}

The program skeleton is converted to

\begin{verbatim}
\texttt{P}_1\leftarrow\ldots\texttt{P}_n\leftarrow0;
\texttt{rep}
\begin{verbatim}
<guarded commands for \texttt{P}_1> ||

<guarded commands for \texttt{P}_2> ||

<guarded commands for \texttt{P}_n> ||
\end{verbatim}
\begin{verbatim}
c_1=m_1\land\ldots\land c_n=m_n\rightarrow \texttt{exit}
\end{verbatim}
\begin{verbatim}
\texttt{per}
\end{verbatim}
\end{verbatim}

Here, <guarded commands for \texttt{P}_i> consists of a number of guarded commands of the form

<condition> \rightarrow <statement list>

and \texttt{m}_i is the largest value of \texttt{c}_i assigned in <guarded commands for \texttt{P}_i>. The "exit" command is introduced to provide a means for termination.

2) Sequencing of the form \texttt{S}_1;\ldots;\texttt{S}_n

Statement lists are converted to

<guarded command> ||

<guarded command> ||

<guarded command> ||

3) Assignment of the form \texttt{x=e}

If the program counter for the previous statement is \texttt{n}, this is transformed to
\[ p_j = n+1 \rightarrow x=e; \quad p_j = n+2 \]

4) Conditionals of the form \textit{if} \( B \) \textit{then} \( S_1 \) \textit{else} \( S_2 \) \textit{fi}

Let \( k_1 \) be the number of different values of the program counter resulting from the transformation of \( S_1 \), and \( k_2 \) be the same for \( S_2 \). If \( n \) is the program counter for the previous statement, the conditional is transformed to

\[
\begin{align*}
    p_j &= n+1 \land B \rightarrow p_j = n+2 \\
    p_j &= n+1 \land \neg B \rightarrow p_j = n+k_1+2 \\
    p_j &= n+2 \\
    \langle \text{guarded commands for } S_1 \rangle \\
    p_j &= n+1+k_1 \\
    p_j &= n+2+k_1 \\
    \langle \text{guarded commands for } S_2 \rangle \\
    p_j &= n+1+k_1+k_2 \\
\end{align*}
\]

5) Loops of the form \textit{while} \( B \) \textit{do} \( S \) \textit{od}

Let \( k \) be the number of different values of the program counter which result from converting \( S \). If \( n \) is the previous program counter,

\[
\begin{align*}
    p_j &= n+1 \land B \rightarrow p_j = n+2 \\
    p_j &= n+1 \land \neg B \rightarrow p_j = n+2+k \\
    p_j &= n+2 \\
    \langle \text{guarded commands for } S \rangle \\
    p_j &= n+1+k \\
\end{align*}
\]
6. Application

Even though by conversion to nondeterminism we have obtained formal definitions for the various correctness issues associated with parallel programs, the question remains as to whether this approach is practical. In this section we shall treat various examples to show the power of our method. The approach of Owicki [Owicki 75] is both practical and highly dissimilar to ours, so by way of comparison some of the examples we discuss have been previously handled by her method.

6.1 Weak correctness

The following example appears in [Owicki 75], where its weak correctness cannot be proved without the addition of auxiliary variables. The discovery of the right set of auxiliary variables (in general) requires much intellectual effort.

```plaintext
cobegin
  with x when true do x=x+1 //
  with x when true do x=x+1
coend
```

The transformation to a nondeterministic program also introduces auxiliary variables (program counters), but in a uniform manner. Even though some of the program counters may be superfluous, we remove the burden of inventing auxiliary variables and the corresponding operations upon them. The nondeterministic version is

```plaintext
P_1+P_2=0;
rep
  P_1=0 \rightarrow x=x+1; P_1=1 ||
  P_2=0 \rightarrow x=x+1; P_2=1 ||
  P_1=1 \land P_2=1 \rightarrow \text{exit}
per
```

It is easily seen that the weak correctness invariant is $x=x_0+P_1+P_2$, where $x_0$ is the initial value of $x$. Since $P_1=1 \land P_2=1$ at the exit, the program will establish $x=x_0+2$. 
6.2 Mutual exclusion

We consider a solution to a mutual exclusion problem discussed by Dijkstra in 1965 [Dijkstra 65]. Two processes A and B have critical sections which must be excluded from one another. No synchronization primitive other than the indivisibility of a single access to memory is allowed. The following solution is discussed in [Flon 77]:

```plaintext
var inA, inB: boolean initially false,
prty: (A,B) initially A;

processA: while true do
  <think>
inA=true;
while inB do
  if prty=B then inA=false;
  while prty=B do skip od;
inA=true
  fi
  od;
  <critical section>
inA=false;
prty=B
od

processB: while true do
  <think>
inB=true;
while inA do
  if prty=A then inB=false;
  while prty=A do skip od;
inB=true
  fi
  od;
  <critical section>
inB=false;
prty=A
od
```

The nondeterministic version of this parallel system is
To show that the critical sections cannot both be active at the same time, it suffices to prove that the guards which reflect entry to the critical sections cannot both be true at the same time. Thus,

\[(p_1 = 2 \land \lnot \text{inB}) \land (p_2 = 2 \land \lnot \text{inA})\]

must be invariantly false. Consider the predicate

\[L = (p_1 = 3 \Rightarrow \text{inA}) \land (p_2 = 3 \Rightarrow \text{inB}) \land [(p_1 = 2 \Rightarrow \text{inA}) \lor (p_2 = 2 \Rightarrow \text{inB})]\]

(The last conjunct is the negation of the previous formula.) \(L\) is invariant across all of the commands, and

\[L \Rightarrow \lnot (p_1 = 2 \land \lnot \text{inB}) \lor \lnot (p_2 = 2 \land \lnot \text{inA})\]

so clearly the critical sections cannot be active simultaneously. This program is proved correct in [Flon 77], using an extension of Owicki’s methodology, but auxiliary variables are needed in a non-trivial way. The proof presented here is much simpler.
6.3 Deadlock

We will show that the following program is subject to deadlock:

\[ r1 \leftarrow r2 \leftarrow 1; \]
\[ \text{cobegin} \]
\[ \quad \text{while true do} \]
\[ \quad \quad \text{with } r2 \text{ when } r2 = 1 \text{ do } r2 \leftarrow r2 - 1 \text{ od; } \]
\[ \quad \quad \text{with } r1 \text{ when } r1 = 1 \text{ do } r1 \leftarrow r1 - 1 \text{ od; } \]
\[ \quad \quad \text{with } r1, r2 \text{ when true do } r1 \leftarrow r2 + 1 \text{ od} \]
\[ \text{od} // \]
\[ \quad \text{while true do} \]
\[ \quad \quad \text{with } r1 \text{ when } r1 = 1 \text{ do } r1 \leftarrow r1 - 1 \text{ od; } \]
\[ \quad \quad \text{with } r2 \text{ when } r2 = 1 \text{ do } r2 \leftarrow r2 - 1 \text{ od; } \]
\[ \quad \quad \text{with } r1, r2 \text{ when true do } r1 \leftarrow r2 + 1 \text{ od} \]
\[ \text{od} // \]
\[ \quad \text{while true do skip od} \]
\[ \text{coend} \]

The nondeterministic version is

\[ r1 \leftarrow r2 \leftarrow 1; \ p1 \leftarrow p2 \leftarrow p3 \leftarrow 0; \]
\[ \text{rep} \]
\[ \quad \ p1 = 0 \wedge r1 = 1 \rightarrow r1 \leftarrow r1 - 1; \ p1 \leftarrow 1 \ || \]
\[ \quad \ p1 = 1 \wedge r2 = 1 \rightarrow r2 \leftarrow r2 - 1; \ p1 \leftarrow 2 \ || \]
\[ \quad \ p1 = 2 \rightarrow r1 \leftarrow r2 - 1; \ p1 \leftarrow 0 \ || \]
\[ \quad \ p2 = 0 \wedge r2 = 1 \rightarrow r2 \leftarrow r2 - 1; \ p2 \leftarrow 1 \ || \]
\[ \quad \ p2 = 1 \wedge r1 = 1 \rightarrow r1 \leftarrow r1 - 1; \ p2 \leftarrow 2 \ || \]
\[ \quad \ p2 = 2 \rightarrow r1 \leftarrow r2 - 1; \ p2 \leftarrow 0 || \]
\[ \quad \ p3 = 0 \rightarrow \text{skip} \]
\[ \text{per} \]

We can establish the possibility of deadlock by finding an invariant \( f \) which implies \( \neg B_j \)
for some \( j \), and then showing that there is a way to arrive at \( f \). The latter property is
expressed by

\[ \text{wpo}(\text{REP}, f) \]

For the above program

\[ p1 = 1 \wedge r2 = 0 \wedge p2 = 1 \wedge r1 = 0 \]
is an invariant. It is also possible to achieve this by first executing command 1 and
then command 4. That is,
which evaluates to
\[ r1=1 \land r2=1 \land p1=0 \land p2=0 \]

which is established by the initialization. Furthermore, \( f \) prevents all but the last guard from running, so the original parallel program may deadlock.

### 6.4 Starvation

In this section we treat the problem of the Five Dining Philosophers [Dijkstra 71]. The following parallel program is a solution to the problem which is known to have the possibility of starvation:

```plaintext
f0=f1=f2=f3=f4=1;
cobegin
    philosopher 1: while true do
        with f0,f1 when i0=1 \land f1=1 do f0=f1=0 od;
        "eat";
        with f0,f1 when true do f0=f1=1 od
    od //

    philosopher 5: while true do
        with f4,f0 when f4=1 \land f0=1 do f4=f0=0 od;
        "eat";
        with f4,f0 when true do f4=f0=1 od
    od

cend
```

Below is the corresponding nondeterministic program. It is evident that the program does not terminate, so we have deleted the superfluous exit statement.
To prove that the program is not free from starvation, we will use the following theorem:

\[ \text{wpot}(\text{REP}, P) \land (P \Rightarrow \text{wpot}(\text{REP}, B_j)) \land [P \Rightarrow -B_j \land (\exists k \in 1..n)(B_k \land \text{wp}(S_k, P))] \]

\[ \vdash -\text{wp}(\text{REP}) \]

The theorem states that REP is subject to starvation whenever it is possible to reach a state \( P \), from which it is both possible to establish \( B_j \) and possible not to.

For the Dining Philosophers program, let \( P \) be

\[
(f4=0 \land f0=0 \land f1=0 \land f2=0 \land p5=1 \land p1=0 \land p2=1) \lor \\
(f4=1 \land f0=1 \land f1=0 \land f2=0 \land p5=0 \land p1=0 \land p2=1) \lor \\
(f4=0 \land f0=0 \land f1=1 \land f2=1 \land p5=1 \land p1=0 \land p2=0)
\]

Clearly \( P \Rightarrow -B_1 \). Furthermore,

\[ P \Rightarrow (\exists k \in 1..n)(B_k \land \text{wp}(S_k, P)) \]

because

\[ P \Rightarrow (B_3 \land \text{wp}(S_3, P)) \lor (B_4 \land \text{wp}(S_4, P)) \lor (B_9 \land \text{wp}(S_9, P)) \lor (B_{10} \land \text{wp}(S_{10}, P)) \]

To prove that the state \( P \) is reachable, we must show \( \text{wpot}(\text{REP}, P) \). This clearly holds because statement 3 may be executed first, and

\[ B_3 \land \text{wp}(S_3, P) = p2=0 \land p1=0 \land p5=0 \land ((f4=1 \land f0=1) \lor (f4=0 \land f0=0)) \]
is implied by the initialization. Finally, we show that \( P \Rightarrow \text{wpot}(\text{REP}, B_1) \). This is guaranteed by

\[
P \Rightarrow (B_4 \land \text{wp}(S_4, B_1)) \lor (B_{10} \land \text{wp}(S_{10}, B_1)) \lor (B_4 \land \text{wp}(S_4, B_{10} \land \text{wp}(S_{10}, B_1)))
\]

Thus, there is a possibility that process 1 (and by symmetry any process) may starve.

7. Practical considerations for program verification

Even though it is sometimes possible to compute the weakest pre-condition recurrences, in general we will have to use less complex techniques if parallel program verification is to be practical. If we can discover sufficiently strong invariants for the nondeterministic programs, we can use the theorems of section 4.4 to verify weak correctness, safety from blocking, and safety from deadlock rather easily.

Verification of safety from starvation can be done by showing that starting from a given invariant, all possible execution sequences must arrive at a state satisfying the guard in question. This may be done by proving safety from blocking and defining an integer function of the program state which is bounded from below and which is decreased by every statement other than the one in question. Following is an example of a proof of starvation-freeness.

Consider the program

\[
\begin{align*}
\text{rep} & \quad x>0 \land y<n \Rightarrow x+x-1; y+y+1 \\
& \quad y>0 \land z<n \Rightarrow y+y-1; z+z+1 \\
& \quad z>0 \land x<n \Rightarrow z+z-1; x+x+1
\end{align*}
\]

For this program, a loop invariant which guarantees safety from blocking is

\[
0 \leq x \leq n \land 0 \leq y \leq n \land 0 \leq z \leq n \land 0 < x+y+z < 3n
\]

Since all three statements are symmetric, we need only prove absence of starvation...
for the first one. To do that we will show that it is not possible for the second and the third processes to execute continuously without eventually making the first guard true. Suppose the program is in a state in which the first guard is false - i.e. \( x=0 \lor y=n \). Consider the state function \( W = 3y+2z+x \). \( W \) is decreased by exactly 1 by both the second and third statements. When \( x=n \land y=0 \), \( W \) can no longer decrease and the last two statements cannot execute, guaranteeing that the first statement will execute.

8. Summary

For some time we have been lacking an effective formalism for parallel program semantics. The approach discussed in this paper was motivated by two observations - that the important correctness issues for parallel programs have counterparts in nondeterministic sequential programs, and that parallel programs can be effectively transformed to nondeterministic ones. We have therefore presented formal definitions for the weakest pre-conditions which guarantee weak correctness, absence of blocking, absence of deadlock, and absence of starvation for nondeterministic programs, along with a procedure for the conversion of parallel programs to nondeterminism.

As a demonstration of the usefulness of our formalism we have proved various properties of several programs, including a busy-wait mutual exclusion scheme and the problem of the Five Dining Philosophers. It remains to be seen to what degree these techniques will apply to actual operating system examples, although we have tried to present various methods which reduce the burden of computation in exchange for some intellectual creativity, such as the discovery of invariant predicates and monotonically decreasing state functions.
References


