"indivisible" action. For instance, \( R \) might be the three instructions or actions:

\[
\begin{align*}
\text{begin} \\
r &\leftarrow x; \\
\text{increment } r; \\
x &\leftarrow r; \\
\text{end;}
\end{align*}
\]

Assuming that \( R \) is uninterruptible or indivisible reduces \( R \) to the single instruction:

\[
x \leftarrow x + 1;
\]

In contrast to (1), it is usually not easy to prove that (2) \( P \) has property \( \Sigma \) when \( R \) is interruptible.

The basic question considered in this paper is: When are assertions (1) and (2) equivalent?

Define \( P/R \) to be the parallel program obtained from \( P \) by reducing \( R \) to one indivisible action (i.e. \( R \) is considered to be uninterruptible). \( P/R \) is called the reduction of \( P \) by \( R \). Then the type of result we obtain is:

(3) \( P/R \) has property \( \Sigma \) iff \( P \) has property \( \Sigma \).

In proving (3), restrictions must be placed on \( R \). These restrictions, however, are satisfied by a wide range of statements. These results are then used as follows. Suppose that one desires to prove that \( P \) has property \( \Sigma \). \( P \) is then reduced to \( P' \), \( P' \) is reduced to \( P'' \), and so on, finally yielding \( Q \). Now \( Q \) is shown to have property \( \Sigma \); thus several applications of (3) show that \( P \) also has property \( \Sigma \). The reason this method is fruitful is that \( Q \) is usually much "simpler" than \( P \). There are two ways in which \( Q \) is simpler: (i) \( Q \) has fewer actions than \( P \). It follows that a proof that \( Q \) has property \( \Sigma \) must consider fewer cases than a proof that \( P \) has property \( \Sigma \). (ii) Assertions about \( Q \) are often simpler than assertions about \( P \). For example, we will later investigate an example where in \( Q \) the sum of two variables \( a + b \) is always a constant, while in \( P \), \( a + b \) is a complex function of the state of \( P \). This difference in the assertions that can be made about \( a + b \) is important: the fact that \( a + b \) is constant allows an easy proof that \( Q \) does not halt.

The previous proof procedures for parallel programs have consisted essentially of Floyd's assertion method [3] adapted to parallel programs (Ashcroft [1], Lauer [5], and Levitt [6]). The basic drawback to this method is that because of the many possible computations in a parallel program, the assertion method tends to involve the consideration of many cases. The arbitrary interleaving of a parallel program is then a major obstacle for the assertion method. It seems to lead to complex assertions of the form if process-1 is at statement \( h_1 \) and process-2 is at statement \( h_2 \) and ..., then ... This should be compared with the main advantage of the reduction method: The computations of \( P/R \) are a proper subset of the computations of \( P \). Of course, the reduction method can be used in conjunction with the assertion method.

This paper is composed of five sections. In Section 2
the basic definitions of parallel programs and indivisibility are presented. In Section 3 the concept of reduction is presented. It is then proved that certain reductions, called D-reductions, preserve a number of properties such as halting. In Section 4 several examples that demonstrate the power of the reduction method are presented.

2. Parallel Programs

For the basic definition of parallel programs in this paper, Algol or a similar language will be supplemented with the parallel statement parbegin ... end of Dijkstra [2]. The effect of

```
parbegin S_1 ; . . . ; S_n , parend
```

is then to interleave the statements $S_1, . . . , S_n$ in some arbitrary order until no further execution is possible. A computation is then a sequence $t_1, . . . , t_m$ of statements such that $t_1$ is executed first, then $t_2$ is executed, and so on until the last statement $t_m$ is executed. Since an $S_i$ may be a compound statement, $m > k$ is possible. For example, if $S_1$ is

```
begin x := x + 1 ; y := if w = 1 then y else 1 ; end
```

then $t_i$ might be the statement $x := x + 1$ or the statement

```
y := if w = 1 then y else 1
```

Indeed one might even allow $t_i$ to be a "part" of one of these statements. Thus $t_i$ might be the action that computes the value of the Boolean expression $w = 1$. The reason it is not necessary to say exactly what $t_i$ can be is that in the majority of cases it simply does not matter. In some cases, however, it is extremely important that some statement be considered indivisible, i.e. that no $t_i$ be a part of it. For this reason we add to the parbegin . . . parend notation as follows: A statement $S$ (we assume $S$ has a single entry and a single exit) is indivisible if it is enclosed in brackets to form $[S]$. The semantics of $[S]$ are then:

1. In a given state of the parallel program, $[S]$ can execute provided in this state control (in the normal sense) is ready to enter $S$ and after $S$ is applied control has left $S$.
2. In a given state of the parallel program, the effect of applying $[S]$ (provided it can execute) is the same as that of $S$.

The key to the definition of $[S]$ is that we can never apply it when it cannot fully complete its execution. For example, consider the indivisible statement

```
[L : if a > 0 then a := a - 1 else goto L]
```

It can execute iff $a > 0$; if $a \leq 0$, then control remains ready to enter and so the indivisible statement cannot be applied. The effect of this statement is always to decrement $a$ by 1. This statement will be later denoted by $P(a)$; it corresponds to the "wait" primitive of [2].

Second, consider the indivisible statement

```
[a := a + 1]
```

Clearly, it can execute iff true, i.e. it can always execute. The effect of this statement is always to increment $a$ by 1. This statement will be later denoted by $V(a)$; it corresponds to the "signal" primitive of [2]. Note this second example is not equivalent to $a := a + 1$

Without enclosing $a := a + 1$ in brackets it is possible to "lose counts," i.e. in

```
integer a ; (a = 0) ;
parbegin a := a + 1 ; a := a + 1 ; parend ;
```

the value of $a$ can be 2 or 1.

Finally, consider the indivisible statement

```
[P(a) ; r := r + 1 ; if r = 1 then P(b) ; V(a)]
```

It can execute iff $a > 0$ and $(r \neq 0$ or $b > 0$). The effect of this statement is to leave $a$ unchanged ($P(a)$ decrements $a$ and $V(a)$ increments $a$); always to increment $r$ by 1; and to replace $b$ by

```
if r = 0 then b := b + 1 else b
```

The expression for $b$ follows from the observation that the then statement is evaluated only if $r = 0$ on entry to the indivisible statement.

In order to complete the basic definitions of this paper the notion of computation is extended to programs with indivisible statements. Define $a[S]$ to be a computation provided $a$ is a computation and $[S]$ can execute in the state that results after $a$ is executed. Thus in the program

```
integer a ; (a = 0) ;
parbegin A : P(a) ; B : V(a) ; parend ;
```

the only computations are the sequences $B$ and $B$, $A$. The sequence $A$, $B$ is not a computation since $P(a)$ cannot execute initially, for $a = 0$. This is an important point, which must be stressed: Computations are sequences of statements that execute; no statement can occur in a computation if it would "block" in the sense of [2]. The reason that this assumption can be made is that only properties of programs that depend on their states (i.e. the values of their program variables will be studied). Now the key to this assumption is that the reachable states in a program with or without "blocking" are the same.

3. Reductions

The concept of reduction is now defined. It is then shown that D-reductions, a class of reductions, preserve a number of interesting properties, including for instance, halting.

**Definition.** Suppose that $P$ is a parallel program with statement $S$. Then define $P/S$, the reduction of $P$ by $S$, of the ACM Number 12 Communications December 1975 Volume 18
to be the parallel program that results when \( S \) is replaced by \([S]\).

The fundamental question is: what is the relationship between \( P \) and \( P/S \)? In particular, let us consider the question: is it true that \( P/S \) halts iff \( P \) halts? A program halts if there is some computation \( a \) such that \( a \) is not a computation for all statements \( f \). A parallel program that does not halt is often called “deadlock free” [2]. In analyzing parallel programs, as found in operating systems, it is often important to prove that they never halt, i.e. that they are deadlock free. This follows since operating systems are often never-ending tasks and hence must be proved never to halt.

The most optimistic conjecture to make is that all reductions preserve halting; more exactly,

\[(4) \ P/S \text{ halts iff } P \text{ halts.}\]

This is, however, false. Consider the parallel program \( \text{EX1} \):

\[
\text{integer } a, b \ (a = b = 1);
\text{parbegin}
\text{repeat } P(a); P(b); V(a); V(b); \text{end;}
\text{repeat } P(b); P(a); V(b); V(a); \text{end; paren;}
\]

It is easy to see that this program halts. Just let both repeat’s execute their first \( P \)'s; then \( a = b = 0 \) and the program has halted. Now consider the following program \( \text{EX1}/S \):

\[
\text{integer } a, b \ (a = b = 1);
\text{parbegin}
\text{repeat } [P(a); P(b); V(a); V(b)]; \text{end;}
\text{repeat } P(b); P(a); V(b); V(a); \text{end; paren;}
\]

Clearly, \( \text{EX1}/S \) does not halt. This follows since the effect of

\[
[P(a); P(b); V(a); V(b)]
\]

is to leave both \( a \) and \( b \) fixed. Therefore, assertion (4) is false. The failure of assertion (4) can be explained as follows. In \( \text{EX1} \) it is possible to enter \( S \) and not ever be able to leave it. This observation leads to one restriction on statements \( S \):

\[(R1) \text{ If a statement } S \text{ is ever entered, then it must be possible eventually to exit } S.\]

This restriction appears to be strong; as demonstrated later, however, it is satisfied by a wide class of statements.

Restriction (R1) alone is not sufficient to ensure the truth of assertion (4). For example, consider the parallel program \( \text{EX2} \):

\[
\text{integer } x, y \ (x = y = 0);
\text{parbegin } x \leftarrow 0; \text{repeat } A : x \leftarrow 1; B : y \leftarrow x; P(y); \text{end; paren;}
\]

It is easy to see that \( \text{EX2} \) halts. Now \( S = A : x \leftarrow 1; B : y \leftarrow x \); satisfies restriction (R1). The program \( \text{EX2}/S \) is:

\[
\text{integer } x, y \ (x = y = 0);
\text{parbegin } x \leftarrow 0; \text{repeat } [A : x \leftarrow 1; B : y \leftarrow x]; P(y); \text{end; paren;}
\]

Clearly, this does not halt; this follows since \( [A : x \leftarrow 1; B : y \leftarrow x] \) always sets \( y \) to 0. Thus, restriction (1) is not sufficient to imply assertion (4). This example fails to satisfy assertion (4) because the effect \( A : x \leftarrow 1 \) and \( B : y \leftarrow x \) when “separated” and when “together” is not the same. When together \( y \) is always set to 1; when separated \( y \) can be set to 0 or 1. This observation leads to another restriction:

\[(R2) \text{ The effect of the statements in } S \text{ when together and separated must be the same.}\]

This restriction may appear to be difficult to capture precisely and perhaps just as difficult to satisfy, but this is not the case. The following is the key definition.

**Definition.** Suppose that \( f \) and \( g \) are statements in a parallel program. Then

(a) \( f \) is a right mover provided for any \( a \) a computation where \( f \) and \( h \) lie in different processes (in \( \text{parbegin } S_1 ; \ldots ; S_y ; \text{parend} \) the statements of each \( S_i \) form a distinct process), then \( ahf \) is also a computation; moreover, the values of all the program variables in \( ahf \) and \( ahf \) are the same;

(b) \( g \) is a left mover provided for any \( ahg \) a computation where \( h \) and \( g \) lie in different processes, then \( ahg \) is also a computation; moreover, the values of all the program variables in \( ahg \) and \( ahg \) are the same.

Essentially, a right mover is a statement that performs a “seize” while a left mover is a statement that performs a “release” of a “resource.”

In order to see this, consider first the case of a left mover. If \( ahf \) is a computation and \( f \) performs a “release,” then \( ahf \) is also a computation provided \( f \) and \( h \) lie in different processes (recall here our restriction on what is a computation, i.e. no blocking can occur in a computation):

1. \( ahf \) is a computation since \( h \) could execute after \( a \) and \( f \) did not seize any resource (i.e. any demand of \( h \) can still be fulfilled).

Second, consider the case of a right mover. If \( ahg \) is a computation and \( g \) performs a “seize,” then \( ahg \) is a computation provided \( g \) and \( h \) lie in different processes:

1. \( ah \) is a computation; argue as before.
2. \( ahg \) is a computation. If \( h \) is a “release” this follows immediately by the first case. Thus assume that \( h \) is a “seize” and the result follows by a symmetry argument.

The “proofs” above can be stated exactly for PV parallel programs. A program \( P \) is a PV parallel program provided there is a distinguished subset of the program variables \( a_1, \ldots, a_n \) called semaphores with integer values such that they can be used only in either \( P(a_i)'s \) or \( V(a_i)'s \). Then we have essentially proven the following theorem.

**Theorem 1.** In any PV parallel program all \( P(a)'s \) are right movers and all \( V(a)'s \) are left movers.
D-reductions can now be defined.

Definition. Let \( S_1, \ldots, S_k \) be such that \( \{ S_1, \ldots, S_k \} \) is a \( D \)-reduction provided, for some \( i, S_i \), \( S_{i-1} \) are right movers and \( S_{i+1}, \ldots, S_k \) are left movers (\( S_i \) is unconstrained) and each \( S_i, \ldots, S_k \) can always execute.

Restriction (R2) corresponds to the fact that the first \( i - 1 \) statements are right movers and the last \( k - i \) are left movers. Restriction (R1) corresponds to the fact that the last \( k - 1 \) statements can always execute. For example, in a \( PV \) parallel program \( \{ S_1, \ldots, S_n \} \) is always a \( D \)-reduction provided \( S_2, \ldots, S_n \) are \( V \)'s. This follows from Theorem 1 and the fact that any \( V \) in a \( PV \) program can always execute.

**Theorem 2.** Suppose that \( S \) is a \( D \)-reduction in \( P \). Then \( P \) halts if \( p/S \) halts.

Proof. Clearly if \( P/S \) halts, then \( P \) halts. This follows since any state of \( P \) is also a state of \( P \). It will now be shown that if \( P \) halts, then \( P/S \) halts. To this end, assume that \( P \) halts; moreover, let \( \alpha \) be a computation in \( P \) such that \( \alpha \) halts. It will now be shown that \( S = S_1, \ldots, S_n \). The plan of the proof is to construct a computation \( \beta \) such that all the program variables agree after \( \alpha \) and \( \beta \) are executed and \( S_1, \ldots, S_n \) always occur as "consecutives of blocks of statements in \( \beta \)," i.e., \( \beta_i \) is the \( i \)-th element of the sequence \( \beta_i, \beta_j \) if \( \beta_i = S_j \) and \( j < n \), then \( \beta_{i+1} = S_{i+1} \); \( \beta_{i+1} = S_{i+1} \) if \( \beta_i = S_j \) and \( j > 1 \), then \( \beta_{i+1} = S_{i+1} \). In order to avoid complex notation it will be assumed there are no goto's in \( S_1, \ldots, S_n \). Now two simple lemmas are needed. Lemma 2 encodes the key "trick" used in our proof.

**Lemma 1.** Suppose that \( \alpha S \beta \) is a computation in \( P \) with \( i > 1 \). Then \( \lambda \alpha S_{i-1} \mu \) where no statement from the process of \( S_i \) is in \( \mu \).

Proof of Lemma. This follows easily from the fact that \( S_1, \ldots, S_n \) has a single entry and the assumption that no goto's occur in our programs.

**Lemma 2.** Suppose that \( \alpha S \beta \) is a computation that halts in \( P \) with \( i < n \). Then \( \beta = \lambda S_{i+1} \mu \) where no statement from the process of \( S_i \) is in \( \lambda \).

Proof of Lemma. If \( f \) occurs in \( \beta \) where \( f \) is in the process of \( S_i \), then the first such \( f \) must be \( S_{i+1} \). Thus assume that no such \( f \) is in \( \beta \). In \( \alpha S \beta \) control must be ready to enter \( S_{i+1} \); therefore \( \alpha S \beta S_{i+1} \) is a computation, which is a contradiction (recall \( \alpha S \beta \) halts). Note \( \alpha S \beta S_{i+1} \) is a computation since by the definition of \( D \)-reduction \( S_{i+1} \) can always execute.

If no \( S_i \) is in \( \alpha \), then \( \alpha \) is already in the desired form (i.e., \( \beta = \alpha \)). Therefore suppose that some \( S_i \) is in \( \alpha \). By repeated applications of Lemmas 1 and 2,

\[
\alpha = \lambda S_1 \alpha \ldots \lambda S_n \mu
\]

where no statement from the process of \( S_i \) in any \( \alpha_j \) \( (j = 2, \ldots, n) \). By the definition of \( D \)-reduction for some \( k \),

\[
\delta = \lambda \alpha \ldots \lambda S_1 \ldots S_n \alpha \ldots \alpha_{k+1} \ldots \alpha_{n} \mu
\]

is a computation and it agrees with \( \alpha \) on all the program variables. This argument can be repeated to form the desired computation \( \beta \).

Now \( \beta = \beta_1 S_1 \ldots S_{n} \beta_2 \ldots \beta_{n-1} S_1 \ldots S_n \beta_n \)

where no \( S_i \) is in any \( \beta \) and \( \alpha \) and \( \beta \) agree on all program variables. Then \( \delta = \beta_1 S_2 \ldots \beta_{n-1} S_\beta \)

is a computation in \( P/S \); moreover, \( \alpha \) and \( \beta \) and \( \delta \) agree on all program variables. Then \( \delta \) is a computation in \( P/S \); further assume that \( \delta h \) is a computation in \( P/S \). If \( h \neq S_i \), then \( \alpha h \) is a computation in \( P \); if \( h = S_i \), \( \alpha S_i \), \ldots, \( S_n \) is a computation in \( P \). These assertions follow since \( \alpha \) and \( \delta \) agree on all program variables. In either case we have reached a contradiction.

The proof of Theorem 2 actually establishes that for each \( \alpha \) that halts in \( P \) there is a \( \beta \) that halts in \( P/S \) such that \( \alpha \) and \( \beta \) agree on all program variables. Thus if \( S \) is a \( D \)-reduction, then

(5) The final states of \( P \) equal the final states of \( P/S \).

(A final state of a program is a state that results after an \( \alpha \) is executed where \( \alpha \) is a halting computation.)

Theorem 2 is then seen to be a special case of (5). It states that \( P \) has a final state if \( P/S \) has a final state. In general, \( D \)-reduction then preserves any property that depends only on a program's final state.

**4. Applications**

The reduction method is now demonstrated by two examples. In both cases applications of Theorems 1 and 2 show that a parallel program does not halt, i.e. it is deadlock free.

The first example is based on the program \( EX_3 \):

\[
\begin{align*}
\text{integer} & \ a, b, c \ (a = b = c = 1); \\
\text{parbegin} & \ P(a); P(b); V(a); V(b); \text{end}; \\
& \ \text{repeat} P(b); P(c); V(b); V(c); \text{end}; \\
& \ \text{repeat} P(a); P(c); V(a); V(c); \text{end}; \text{parend}; \\
\end{align*}
\]

Essentially this program is the "smoker's example" except for initial conditions. By Theorems 1 and 2, \( EX_3 \) halts if the following program halts:

\[
\begin{align*}
\text{integer} & \ a, b, c \ (a = b = c = 1); \\
\text{parbegin} & \ P(a); P(b); V(a); V(b); \text{end}; \\
& \ \text{repeat} P(b); P(c); V(b); V(c); \text{end}; \\
& \ \text{repeat} P(a); P(c); V(a); V(c); \text{end}; \text{parend}; \\
\end{align*}
\]

The two indivisible statements behave as follows:

1. \( \{ P(c); V(b); V(c) \} \) can execute iff \( c > 0 \); its effect is to increment \( b \) by 1 and leave \( a \) and \( c \) unchanged.

2. \( \{ P(c); V(a); V(c) \} \) can execute iff \( c > 0 \); its effect is to increment \( a \) by 1 and leave \( b \) and \( c \) unchanged.

Thus \( c \) is always equal to 1. It follows that

\[
\left[ P(c); V(b); V(c) \right] = V(b)
\]

and

Communications of the ACM

December 1975

Volume 18

Number 12
\[ P(c); V(a); V(c); ] = V(a) \]

In summary, EX3 halts iff the following program halts:

\begin{verbatim}
  integer a, b (a = b = 1);
  parbegin repeat P(a); P(b); V(a); V(b); end;
    repeat P(b); V(b); end;
    repeat P(a); V(a); end; parenend;
\end{verbatim}

Once again Theorems 1 and 2 can be applied; hence, EX3 halts iff the following program halts:

\begin{verbatim}
  integer a, b (a = b = 1);
  parbegin repeat P(a); [P(b); V(a); V(b); ] end;
    repeat [P(b); V(b); ] end;
    repeat P(a); V(a); end; parenend;
\end{verbatim}

As with variable c, it is now the case that b is always equal to 1. Therefore, EX3 halts iff the following program halts:

\begin{verbatim}
  integer a (a = 1);
  parbegin repeat P(a); V(a); end
    repeat P(a); V(a); end; parenend;
\end{verbatim}

Finally, this program trivially never halts. The second process runs forever, doing nothing! Thus EX3 does not halt.

Our second example is based on the program EX4:

\begin{verbatim}
  integer a, b (a = 0, b = N);
  parbegin repeat P(a); V(b); end;
    repeat P(b); V(a); end parenend;
\end{verbatim}

The integer \( N > 0 \) represents the amount of "buffer space" available. This is essentially the bounded buffer example of [4]. Each process consumes from one buffer and produces elements for the other buffer. The value of \( a + b \) intuitively represents the number of elements in the buffers. One would like to argue that \( a + b \) is always equal to \( N \), but it clearly is not. Indeed \( a + b \) can equal \( N \) or \( N - 1 \) or \( N - 2 \). Now let us apply Theorems 1 and 2. Then EX4 halts iff the following program halts:

\begin{verbatim}
  integer a, b (a = 0, b = N);
  parbegin repeat [P(a); V(b); ] end;
    repeat [P(b); V(a); ] end; parenend;
\end{verbatim}

The effect of \([ P(a); V(b); ] \) is to decrement \( a \) by 1 and increment \( b \) by 1; the effect of \([ P(b); V(a); ] \) is to decrement \( b \) by 1 and increment \( a \) by 1. Thus, \( a + b \) is "conserved" and is always equal to \( N \). But \([ P(a); V(b); ] \) can execute iff \( a > 0 \) and \([ P(b); V(a); ] \) can execute iff \( b > 0 \). Since \( a + b = N > 0 \), it is not possible for the program above to halt; hence EX4 does not halt.

5. Conclusions

That reduction aids in a correctness proof seems to be clear. Essentially reduction is nothing more than analyzing a parallel program by collapsing pieces of the program together. It is interesting to note that the same technique has long been used in sequential programs (e.g. macros or procedures). But in the parallel case it was not at all clear that reduction was possible. The main achievement of Theorems 1 and 2 is the realization that in a wide number of nontrivial instances reduction preserves important properties. Indeed, Theorem 1 can be extended to show that left and right movers exist in great abundance in parallel programs. In any \( PV \) program—and even the restriction to \( PV \) can be weakened—that allows processes to share only global variables with critical sections [2], any statement that is not a \( P \) or a \( V \) is both a left and a right mover. The ramification of this generalization is that reduction can be applied to a very wide class of statements.

References