Acta Informatica 16, 219–235 (1981)

An Axiomatic Definition of Synchronization Primitives

Alain J. Martin

Philips Research Laboratories, 5600 MD Eindhoven, The Netherlands

Summary. The semantics of a pair of synchronization primitives is characterized by three fundamental axioms: boundedness, progress, and fairness. The class of primitives fulfilling the three axioms is semantically defined. Unbuffered communication primitives, the symmetrical *P* and *V* operations, and the usual *P* and *V* operations are proved to be the three instances of this class. The definitions obtained are used to prove a series of basic theorems on mutual exclusion, producer-consumer coupling, deadlock, and linear and circular arrangements of communicating buffer-processes. An implementation of *P* and *V* operations fulfilling the axioms is proposed.

1. Introduction

The purpose of this paper is to define the semantics of any pair of synchronization primitives in terms of three fundamental axioms called boundedness, progress, and fairness. Synchronization constructs that are not a pair of primitives, like "conditional critical regions", are not considered. The class of constructs considered comprises, besides Dijkstra's *P* and *V* operations [1], the different forms of communication primitive pairs (mostly called "send-receive" or "input-output" operations) used in distributed programming. Only the synchronization property of communication primitives will be of interest to us; the other semantic property - the "distributed assignment" - will be ignored.

It will be shown that the class defined by these axioms comprises three different types of primitives according to the degree of synchronization freedom (we call it the "slack") allowed. The primitives with the largest (infinite) slack correspond to *P* and *V* operations, and to "send" and "receive" operations via a channel with infinite buffering capacity. The primitives with a finite but non-zero slack correspond to symmetrical *P* and *V* operations, and to "send" and "receive" operations via a buffer with finite (positive) buffering capacity. The ones with zero slack correspond to Hoare's primitives [5] for communication via a channel without buffering capacity.

In the first part, starting from the most general formulation of the three axioms, an algebraic semantic definition is derived for each of the three types of primitives. In the second part, the usefulness of the semantic definitions obtained is demonstrated by proving a series of fundamental theorems. Most of them concern programs using P and V operations; one is about linear and circular arrangements of buffer processes communicating by means of Hoare's primitives. The alternative between communication primitives with infinite slack and those with finite slack is discussed in relation with the producer-consumer problem.

Finally, in order to verify that the axiomatic definition proposed for P and V operations does not differ fundamentally from traditional operational definitions, an implementation is derived from the axioms. The reader may convince himself that this implementation indeed corresponds to one (the best!) of the traditional definitions.

2. The Synchronization of two Actions

For the sequential program parts, besides the usual assignment statement, we use Dijkstra's alternative and repetitive constructs [2], with a slightly different syntax. Following C.A.R. Hoare, we write [...] for **if**...**fi**, and *[...] for **do**...**od**.

Non-terminating (cyclic) programs of the form

$$*[true \rightarrow S]$$

are very frequent in concurrent programming. In such a case, we shall omit the guard ("*true*") and simply write:

$$*[S].$$

We view a sequential computation as a (finite or infinite) sequence of (hence totally ordered) actions; when necessary we shall name the actions after (the names of) the program commands that provoke them. (The precise "granularity" of actions not concerned with synchronization is irrelevant for the purpose of this paper.)

We view a concurrent computation as a number of simultaneous sequential computations without any intrinsic ordering between actions from different sequential computations; in this context the constituent sequential computations will be referred to as "*processes*".

Let X be a command provoking an "X action" when executed. With X we associate three fundamental non-negative integer variables, denoted by cX, qX, and tX. (The convention consisting of denoting these variables by the name of the action prefixed by the letter c, q, or t will be used all through the paper.)

- The value of cX is defined as the number of X actions *completed* since the beginning of the computation.
- The value of qX is defined as the number of X actions currently *suspended* or *queued.*
- For brevity's sake we shall use tX defined as $cX + qX$.

For a given pair (X, Y) of commands, the corresponding X actions and Y actions are *synchronized* when there exists an invariant relation between cX and cY. (Of course, tautologies of the form $cX+cY \geqq cX$ are excluded as relations.)

With the usual interpretation of the semicolon, execution of

$$*[X; Y]$$

maintains the truth of $0 \leqq cX-cY \leqq 1$. But, by definition of a concurrent computation, $cX-cY$ has a priori no bounds if the X actions and the Y actions occur in different processes. When, however, $cX-cY$ is by definition bounded while the X actions and the Y actions may occur in different processes, we call X and Y a pair of *synchronization primitives.*

Hence the sequencing of actions inside a process and the use of synchronization primitives among different processes are the two means for synchronizing actions.

3. The Semantic Characterization of Synchronization Primitives

We have met for a pair (X, Y) of synchronization primitives the first requirement, to be called "*boundedness requirement*",

R1: *The value of $cX-cY$ is bounded, either at one side or at both sides.*

The requirement of maintaining R1 introduces the notion of suspension of an action. In order not to violate the bounds, the completion of an initiated synchronization action may have to be postponed; from initiation until completion, such a synchronization action is then called *suspended.*

Note that the completion of an X action - i.e. $cX:=cX+1$ - does not influence the value of $cY+qY$: it may be accompanied by $cY, qY:=cY+1, qY-1$, but this does not influence their sum tY. And vice versa.

A trivial way of maintaining the truth of (the initially true) R1 would be to prevent the completion of any X or Y action. But we would like to ensure progress of the computation. Hence we impose upon synchronization primitives a second requirement, to be called "*progress requirement*",

R2: *The set of suspended actions is minimal, i.e. the completion of any non-empty subset of suspended actions would violate* R1.

Corollary. *Suspended X actions exclude suspended Y actions, and vice versa.*

The process to which a suspended action belongs is said to be *delayed* at that action.

When a computation has reached the point where an X action is suspended and no Y action of the same pair occurs in any future history of the computation, we say that there is a *deadlock*: the X action will never be completed, i.e. a process will be delayed forever at X. (If some future histories of the computation contain Y actions and some do not, we say that there is a possible deadlock.)

When a computation has reached the point where an X action is suspended in a process i, and there exists a future history of the computation containing an unbounded number of Y actions completed while i is delayed at X, we say that there is a danger of starvation of i at X. (The concept of starvation has been introduced by Dijkstra in [3].) Observe that, by definition of R2, the number of X actions and the number of Y actions completed while an X action of the same pair is suspended are equal. We could therefore as well define starvation in terms of the number of X actions completed while process i is delayed at X. The starvation of i at X is caused by the non-deterministic choice of the suspended X action to be completed upon the completion of a Y action, and vice versa.

A third requirement, to be called "*fairness requirement*", can be introduced to exclude the danger of starvation, namely

R3: *During the delay of a process at a synchronization action, only a bounded number of actions of the same pair can be completed.*

We observe the following hierarchy among the three requirements: R2 reduces the implementation freedom of R1 in order to guarantee the progress of the whole computation. R3 reduces the implementation freedom of R2 in order to guarantee the progress of each individual process.

Two primitive actions which satisfy R1 and R2, but not necessarily R3 are called *weak synchronization primitives.*

Two primitive actions which satisfy R1, R2, and R3 are called *strong synchronization primitives.*

4. An Algebraic Formulation of the Synchronization Requirements

4.1. Boundedness Requirement

R1: *There exist two integer constants kX and kY such that:*
- *at least one of the two constants is finite,*[1]
- $-kY \leqq cX - cY \leqq kX$.

Corollary. *The two constants are non-negative.*

Proof. R1 holds at initialization, and thus $-kY \leqq 0 \leqq kX$ holds. (*End of proof*)

4.2. Progress Requirement

From the corollary of R2, R2 implies:

$$qX = 0 \vee qY = 0. \tag{1}$$

[1] The aversion of one referee for infinite constants is understandable; yet it is the only way to give a unique formulation of R1 and R2

Assume that $qX > 0$ holds. Then, if R2 is to hold, $cX = cY + kX$ must hold (any increase of cX would violate R1). Hence, R2 implies:

$$qX = 0 \vee cX = cY + kX \tag{2}$$

and symmetrically,

$$qY = 0 \vee cY = cX + kY. \tag{3}$$

It is easy to prove that, conversely, (1), (2), and (3) imply R2. Hence:

R2: $(qX = 0 \vee cX = cY + kX) \wedge (qY = 0 \vee cY = cX + kY) \wedge (qX = 0 \vee qX = 0)$.

(We shall see that, for each of the synchronization primitive pairs used in programming, R2 can be drastically simplified.)

4.3. Fairness Requirement

To each process i, an integer variable d_i is attached such that $d_i = 0$ if i is not delayed, and if i is delayed - say, at an X action - d_i is the number of X actions completed since i has started the suspended X action. Hence, $d_i \geqq 0$. (By definition of R2, d_i is also the number of Y actions completed since i has started the suspended X action.)

R3: *For each process i, d_i has an upper bound.*

Consider a computation consisting of N processes sharing a given pair (X, Y) of synchronization primitives, and let D be the upper bound of d_i for any process i. Let us assume that the computation is deadlock-free for (X, Y) weak. In this case, there can be at most $N-1$ processes simultaneously delayed at the same synchronization primitive, say, X. One suspended action, say in process i, will be the last one to be completed: for process i, d_i will be increased by at least $N-2$.

Hence, a necessary condition for R3 *not to introduce deadlock is that D be at least equal to $N-2$.*

There exist strategies for choosing the suspended action to be completed such that the above condition is also sufficient. In [3], Dijkstra has described such a strategy. From the above, we conclude that fairness cannot be guaranteed for computations comprising an unbounded number of processes.

4.4. An Alternative Definition of Weak Synchronization Primitives

Theorem. *X and Y are weak synchronization primitives if and only if:*

$$cX = \min(tX, tY + kX) \wedge cY = \min(tY, tX + kY).$$

Proof. By definition:

$$cX \leqq tX \wedge cY \leqq tY.$$

R1 is equivalent to:

$$cX \leqq cY + kX \wedge cY \leqq cX + kY.$$

R2 is equivalent to:

$$(cX = tX \vee cX = cY + kX) \wedge (cY = tY \vee cY = cX + kY) \wedge (cX = tX \vee cY = tY).$$

Since, by definition, $a = \min(b, c)$ is equivalent to

$$a \leqq b \wedge a \leqq c \wedge (a = b \vee a = c),$$

R1 $\wedge$ R2 is equivalent to:

$$cX = \min(tX, cY + kX) \wedge cY = \min(tY, cX + kY) \wedge (cX = tX \vee cY = tY).$$

Which is equivalent to:

$$cX = \min(tX, tY + kX) \wedge cY = \min(tY, tX + kY) \wedge (cX = tX \vee cY = tY).$$

But the first two factors imply the third one. (*End of proof*)

5. The Three Types of Synchronization Primitives

5.1. The Slack

Let K be the sum $kX + kY$; K is called the "*synchronization slack*" ($kX + kY \geqq 0$): the larger the value of K, the more the synchronized processes are allowed to get out of step.

The class of synchronization primitives defined by R1, R2, and R3 can be divided into three disjoint subclasses according to whether K is zero, positive and finite, or infinite. These three subclasses define precisely the three types of synchronization primitive pairs used in programming:

- The case $K = 0$ corresponds to the communication primitives such as proposed by Hoare in [5] for communication via channels without buffering capacity: since a channel cannot buffer the messages sent, the completion of a send action must coincide with the completion of the corresponding receive action.
- In the case where K is positive and finite, the primitives, which we call "*symmetrical P and V operations*", correspond to send and receive primitives via channels with finite (but positive) buffering capacity: the value of the slack equals the maximum number of messages a channel can buffer.
- The case where K is infinite corresponds to the usual P and V operations on a semaphore. It also corresponds to send and receive primitives via a channel with infinite buffering capacity: a send action is thus never delayed.

Hence the above three types of primitives are the only synchronization primitives defined by R1, R2, and R3. The specific value of the slack fully determines the type of primitives; the particular values of kX and kY for a given value of K are only a matter of initialization (they are defined by the initial value of the semaphore or by the number of messages initially present in the channel).

5.2. Unbuffered Communication Primitives

For $K=0$, which implies

$$kX=0 \wedge kY=0,$$

R1 reduces to

$$cX=cY.$$

As to R2,

$$(qX=0 \vee cX=cY+kX) \wedge (qY=0 \vee cY=cX+kY)$$

is implied by R1. Hence the following

Definition. The primitives X and Y are unbuffered communication primitives when the following two requirements are fulfilled:

R1: $cX=cY$,
R2: $qX=0 \vee qY=0$.

It is clear that the above definition applies to Hoare's communication primitives. In [5], Hoare states: "there is *no* automatic buffering: In general, an input or output command is delayed until the other process is ready with the corresponding output or input. Such delay is invisible to the delayed process."

5.3. The P and V Operations

Definition a. Synchronization primitives with an infinite slack are called P and V operations. Let kX be finite and kY infinite: X is then a P operation and Y a V operation.

By substituting the values of kX and kY in R1 and R2, we obtain:

R1: $cX \leqq cY+kX$,
R2: $(qX=0 \vee cX=cY+kX) \wedge qY=0$.

From R2 we immediately conclude that a V operation is never suspended since $qY=0$ always holds.

Observing that, in R1 and R2, the variables cX, cY, and the constant kX only occur in the expression

$$kX-cX+cY,$$

we can simplify the definition of P and V operations by introducing an integer variable s equal to $kX-cX+cY$. We obtain:

R1: $s \geqq 0$,
R2: $(qX=0 \vee s=0) \wedge qY=0$.

The variable s is called the *semaphore* associated with the given pair (P, V). By definition of s, kX is equal to the initial value of s, say s_0. Furthermore, since a V operation is never suspended, the factor $qY=0$ can be omitted from R2.

Thus we arrive at a second, equivalent, definition of P and V operations:

Definition b. P and V operations are synchronization primitives such that, given the following variables associated with a pair (P, V):

- s is an integer variable called semaphore, the initial value of which is s_0,
- cP is the number of completed P operations,
- cV is the number of executed V operations,
- qP is the number of suspended P operations,

the relations:

A0: $s \geqq 0$,
A1: $cP + s = cV + s_0$,
A2: $qP = 0 \vee s = 0$,

are invariantly true.

(In the case several pairs (P, V) are simultaneously used, the name of the associated semaphore uniquely identifies a given pair, and the different instances of the variables cP, cV, qP are distinguished from each other by postfixing the identifiers cP, cV, qP with the name of the semaphores, e.g. cPs, cVs, qPs.)

Remark. From the alternative definition given in 4.4., we can immediately deduce that A0, A1, and A2 are equivalent to:

$$\text{A}': \ cP = \min(tP, cV + s_0).$$

This characterization of the semantics of P and V operations has already been proposed by A.N. Habermann [4].

5.4. *The Symmetrical P and V Operations*

Definition a. Synchronization primitives for which the slack is finite and positive are called symmetrical P and V operations.

An equivalent alternative definition can be derived from R1 and R2 by introducing a semaphore variable as in the case of the usual P and V operations.

Definition b. Symmetrical P and V operations are synchronization primitives such that, given the following variables associated with a pair (P', V'):

- s is an integer variable called semaphore, the initial value of which is s_0,
- cP' and cV' are the number of completed P' and V' operations, respectively,
- qP' and qV' are the number of suspended P' and V' operations, respectively,
- s_m is a positive integer constant,

the relations:

B0: $0 \leqq s \leqq s_m$
B1: $cP' + s = cV' + s_0$
B2: $qP' = 0 \vee s = 0$
B3: $qV' = 0 \vee s = s_m$

are invariantly true.

(Note that since $kX + kY > 0$, the factor $qX = 0 \vee qY = 0$ is implied by the first two factors of R2.)

So far, we have given a theoretical justification for the set of axioms chosen. We shall now supplement this with a practical justification: we shall show how these axioms can simplify the formal treatment of a number of synchronization problems.

6. Some Classical Theorems

We shall use the classical notation for identifying pairs of P and V operations - namely, $P(s)$ and $V(s)$ - where s is the name of the semaphore associated with this particular pair.

6.1. Mutual Exclusion with Split and Simple Semaphores

Theorem. *Consider an arbitrary number of concurrent processes, each consisting for what concerns synchronization in a strict alternation of P and V operations, starting with a P and ending with a V, on any of the n* $(n > 0)$ *semaphores* s_i*:* $0 \leqq i < n$. *If np is the number of processes having completed a P and not yet completed the following V, then* $np \leqq \sum_i s0_i$, *where* $s0_i$ *is the initial value of* s_i.

Proof. By definition

$$np = cP - cV,$$

i.e.:

$$np = \sum_i (cPs_i - cVs_i),$$

so, by A1:

$$np = \sum_i (s0_i - s_i),$$

and next by A0:

$$np = \sum_i s0_i. \qquad \textit{(End of proof)}$$

Corollary. *If* $\sum_i s0_i = 1$, *there is at most one process at a time inside the program part enclosed by a P and the following V.*

The set of semaphores s_i forms a so-called "split semaphore". When $n=1$, we obtain the well-known mutual exclusion theorem on a simple semaphore.

6.2. Producer-Consumer

Theorem. *Given an arbitrary number of concurrent processes of the "producer" type defined by the program text:*

$$*[\ldots P(s);\ PUT;\ V(r);\ldots]$$

and an arbitrary number of concurrent processes of the "consumer" type defined by the program text:

$$*[\ldots P(r);\ GET;\ V(s);\ldots],$$

where PUT and GET are arbitrary atomic actions, then:

$$-r_0 \leqq cPUT - cGET \leqq s_0$$

holds, where s_0 and r_0 are the initial values of s and r, respectively.

Proof. Thanks to the sequencing of producer process actions:

$$cPs \geqq cPUT \geqq cVr.$$

Thanks to the sequencing of consumer process actions:

$$cPr \geqq cGET \geqq cVs.$$

And thus:

$$cVr - cPr \leqq cPUT - cGET \leqq cPs - cVs.$$

Thanks to A0 and A1:

$$cPs - cVs \leqq s_0 \quad \text{and}$$

$$cPr - cVr \leqq r_0. \qquad \textit{(End of proof)}$$

6.3. A Theorem on Deadlock

Theorem. *Given n processes defined by the program text:*

$$P(a);\ P(b);\ V(a);\ V(b)$$

and m processes defined by the program text:

$$P(b);\ P(a);\ V(b);\ V(a)$$

with $m+n>0$, the computation is deadlock-free if the following relation holds on the initial values a_0 and b_0 of a and b:

$$a_0 > 0 \wedge b_0 > 0 \wedge (a_0 > n \vee b_0 > m).$$

Proof. Assume there is a deadlock. Each process either is ready or is delayed at $P(a)$ or $P(b)$; and at least one process is delayed:

$$qPa + qPb > 0. \tag{4}$$

From the sequencing in the processes:

$$(cPa - cVa \leqq n) \wedge (cPb - cVb \leqq m),$$

and thus from A1:

$$(a_0 - a \leqq n) \wedge (b_0 - b \leqq m). \tag{5}$$

Assume $qPa = 0$: either a process is ready or it is delayed at $P(b)$. In both cases, from the sequencing of actions in the processes we deduce $cPb = cVb$, i.e. $b_0 = b$. And symmetrically if $qPb = 0$, hence:

$$(qPa > 0 \vee b_0 = b) \wedge (qPb > 0 \vee a_0 = a). \tag{6}$$

From A2:

$$(qPa = 0 \vee a = 0) \wedge (qPb = 0 \vee b = 0). \tag{7}$$

From (4), (5), and (6), (7) is equivalent to:

$$a_0 = 0 \vee b_0 = 0 \vee (a_0 \leqq n \wedge b_0 \leqq m). \qquad \textit{(End of Proof)}$$

7. Linear and Circular Arrangements of Buffer Processes

In this exercise, we use unbuffered communication primitives with the following syntax, similar to Hoare's. A pair of primitives comprises an input command $C?x$ and an output command $C!y$. The name C (the "channel name") uniquely identifies a given pair of commands. When a matching pair $(C?x, C!y)$ is executed, the assignment $x := y$ is performed.

Since we are interested in synchronization aspects only, we shall omit the variable names at the right-hand side of the exclamation and question marks. We consider the concurrent computation consisting of n $(n > 0)$ processes

$$B_i\colon\ 0 \leqq i < n\colon\ *[(i)?;\, (i+1)!].$$

From the sequencing of actions in the processes:

$$\forall i\colon\ 0 \leqq i < n\colon\ 0 \leqq c(i)? - c(i+1)! \leqq 1. \tag{8}$$

And thus each process individually can be viewed as a one-place buffer. But from the semantics of unbuffered synchronization primitives, we have:

$$\forall i\colon\ 0 < i < n\colon\ c(i)? = c(i)! \tag{9}$$

By summing the n inequalities of (8), and by simplification due to (9), we deduce:

$$0 \leqq c(0)? - c(n)! \leqq n. \tag{10}$$

And thus the linear arrangement of n one-place buffers forms an n-place buffer.

Let us add an extra process

$$B_n\colon\ *[(n)?;\ (0)!]$$

so as to form a ring of processes. From the sequencing of actions in B_n:

$$0 \leqq c(n)? - c(0)! \leqq 1. \tag{11}$$

From A0, (10) is equivalent to:

$$0 \leqq c(0)! - c(n)? \leqq n. \tag{12}$$

From (11) and (12), we deduce:

$$c(n)? = c(0)!,$$

which means that B_n never completes an input command. Since the arrangement is symmetrical, this holds for all processes: there is a total deadlock. (It is left as an exercise to the reader to prove that a circular arrangement of buffer processes is deadlock-free if it contains processes both of type B_i and of type $B'_i\colon *[(i+1)!;\ (i)?]$.)

8. Symmetrical *P* and *V* Operations, Producer-Consumer Coupling, and Message-Exchange Primitives

Let us introduce in the producer-consumer problem the integer variable t and the actions $P'(t)$ and $V'(t)$ as:

$$t\colon\ cPUT - cGET + r_0$$

$$P'(t)\colon\ P(r);\ GET;\ V(s) \tag{13}$$

$$V'(t)\colon\ P(s);\ PUT;\ V(r). \tag{14}$$

We shall prove that the actions P' and V' thus introduced obey definition b of symmetrical P and V operations such as given in 5.4.

Theorem. *If $P'(t)$ and $V'(t)$ are considered as indivisible actions, the following relations are invariantly true:*

$$\begin{aligned}
&\text{C0: } 0 \leqq t \leqq s_0 + r_0\\
&\text{C1: } cP' + t = cV' + r_0\\
&\text{C2: } qP' = 0 \vee t = 0\\
&\text{C3: } qV' = 0 \vee t = s_0 + r_0.
\end{aligned}$$

Proof. C0 holds by definition of t, and as a consequence of the producer-consumer theorem.

By definition of cP' and cV', and since P' and V' are indivisible actions:

$$cP' = cGET \wedge cV' = cPUT.$$

By substitution in the definition of t, C1 follows.

By definition of qP' and qV', and since P' and V' are indivisible actions:

$$qP' = qPr \wedge qV' = qPs.$$

And thus, from the axioms of classical P and V operations on r and s:

$$qP' = 0 \vee r = 0,$$

and

$$qV' = 0 \vee s = 0.$$

From A1:

$$r = 0 \Leftrightarrow cPr - cVr = r_0$$

and

$$s = 0 \Leftrightarrow cPs - cVs = s_0.$$

Considering P' and V' as indivisible actions,

$$cPr - cVr = cGET - cPUT,$$

and

$$cPs - cVs = cPUT - cGET.$$

And thus:

$$r = 0 \Leftrightarrow cGET - cPUT = r_0 \Leftrightarrow t = 0$$

and

$$s = 0 \Leftrightarrow cPUT - cGET = s_0 \Leftrightarrow t = s_0 + r_0$$

which establishes C2 and C3. (*End of proof*)

Observing that r_0 is the initial value of t, and introducing t_m for $r_0 + s_0$, we can eliminate r_0 and s_0, and rewrite:

$$\begin{aligned}
&\text{C0: } 0 \leqq t \leqq t_m \\
&\text{C1: } cP' + t = cV' + t_0 \\
&\text{C2: } qP' = 0 \vee t = 0 \\
&\text{C3: } qV' = 0 \vee t = t_m
\end{aligned}$$

thus establishing the equivalence with the definition b of symmetrical P and V operations.

Consider two processes A and B exchanging messages via a channel of finite positive capacity: sending a message amounts to putting the message into a buffer (the channel), and is thus equivalent to a producer action; receiving a message amounts to removing the message from the buffer, and is thus equivalent to a consumer action. From the above theorem, we can conclude that the acts of sending and receiving a message via a channel of finite, positive, capacity are – for what concerns synchronization – semantically equivalent to symmetrical V and P operations, respectively. The capacity of the channel equals the upper bound on the semaphore.

When this upper bound becomes infinite, definition b of symmetrical P and V operations reduces to definition b of usual P and V operations. Hence, send and receive operations via a buffer of infinite buffering capacity are – for what concerns synchronization – semantically equivalent to usual V and P operations, respectively.

The above implementation - (13) and (14) - of symmetrical P and V operations in terms of usual P and V operations also provides an implementation of the communication between A and B in terms of infinite-slack primitives. We see that the transmission of a message from A to B requires the transmission of two messages: the message proper is sent by A by the action "*PUT*; $V(r)$", and received by B by the action "$P(r)$; *GET*". Another message called "acknowledgement" is sent by B by the action $V(s)$, and received by A by the action $P(s)$.

An advantage of choosing send and receive operations semantically equivalent to symmetrical P and V operations is that the transmission of acknowledgements is made implicit.

9. An Implementation of *P* and *V* Operations

In order to verify that the definition of P and V operations in terms of the axioms A0 through A2 does not differ fundamentally from the traditional operational definitions, we shall implement the P and V operations on a general semaphore - i.e. a semaphore the value of which is any non-negative integer - in terms of P and V operations on binary semaphores only - i.e. semaphores the values of which are zero or one.

Two pairs of P and V operations $(P(m), V(m))$ and $(P(z), V(z))$ are given for which we *postulate* that they fulfil the axioms A0, A1, and A2, and for which we shall *prove* that $m \leqq 1$ and $z \leqq 1$. We shall prove that the following implementation of the P and V operations on the general semaphore s in terms of the P and V operations on the binary semaphores m and z also fulfil A0, A1, and A2.

```
$P(s)$: $P(m)$;
    [$s>0 \rightarrow s, p := s-1, p+1$
    ▯$s=0 \rightarrow q := q+1$; $V(m)$; $P(z)$
    ];
    $V(m)$.
$V(s)$: $P(m)$;
    [$q=0 \rightarrow s, r := s+1, r+1$
    ▯$q>0 \rightarrow q, p, r := q-1, p+1, r+1$; $V(z)$; $P(m)$
    ];
    $V(m)$.
```

Initially: $s = s0$, $m, z = 1, 0$ $p, q, r = 0, 0, 0$.

The variables p and r, which appear neither in a guard nor in an assignment to other variables, are semantically redundant variables introduced for the purpose of the proof. They are called "auxiliary" or "ghost" variables. (For a formal treatment of auxiliary variables, see [7].)

Let np be the number of processes that have completed a P operation, and not yet completed the following V operation. Thanks to the mutual exclusion theorem, the relation:

$$m + z + np = 1 \tag{15}$$

holds. And thus:

$$m \leqq 1,\ z \leqq 1,\ np \leqq 1. \tag{16}$$

It is easy to verify that each guarded command in isolation leaves the relation:

$$\begin{aligned}&(s \geqq 0) \wedge\\ &(p+s=r+s0) \wedge\\ &(q=0 \vee s=0)\end{aligned} \tag{17}$$

invariantly true, and that (17) holds initially provided $s0$ is chosen non-negative. Since $np \leqq 1$ implies that the guarded commands are mutually exclusive actions, any concurrent execution of $P(s)$ and $V(s)$ leaves (17) invariantly true.

Let nq be the number of processes that have completed the $V(m)$ action in the third line of $P(s)$, and not yet started the following $P(z)$. The state in which $np=0 \wedge nq=0$ is called the "*stable state*". We shall prove the

Theorem T. *In the stable state,* $p, r, q = cPs, cVs, qPs$.

T together with (17) establishes that A0, A1, and A2 hold for $P(s)$ and $V(s)$ in the stable state. Which completes the proof.

Proof of T.

a) We shall prove that the relation:

$$q = nq + qPz - z \tag{18}$$

holds when $np=0$.

Initially, (18) holds and we shall prove that each atomic action modifying a variable of (18) leaves (18) invariant. (We shall write $x{:}+1$ or $x{:}-1$ to indicate an increase or a decrease of x by one as a result of the action considered.)

"$q:=q+1$; $V(m)$"	:		$q{:}+1, nq{:}+1$
"$P(z)$"	:	if $z=0$	$nq{:}-1, qPz{:}+1$
		if $z>0$	$nq{:}-1, z{:}-1$
"$q, p, r:= \ldots$; $V(z)$"	:	if $qPz=0$	$q{:}-1, z{:}+1$
		if $qPz>0$	$q{:}-1, qPz{:}-1.$

b) From (18), we deduce:

$$nq>0 \vee q=qPz-z. \tag{19}$$

Since $qPz=0 \vee z=0$ holds from A2, if $z>0$ the second term of (19) reduces to $q=-z$. But this is impossible since $q \geqq 0$. Hence, (19) is equivalent to:

$$nq>0 \vee z=0 \wedge q=qPz. \tag{20}$$

c) In the stable state, $m=1 \wedge z=0 \wedge q=qPz$ holds as a consequence of (15) and (20). Hence a process can only be delayed at $P(z)$, from which follows that:

$$q=qPs$$

and obviously

$$r = cVs$$

hold in the stable state.

d) Let us replace the variable p in $P(s)$ by an extra variable $p1$, and in $V(s)$ by an extra variable $p2$. Initially $p1 = p2 = 0$, and thus $p = p1 + p2$ by construction. In the stable state,

$$cPs = p1 + cPz:$$

cPs equals the number of times the first guarded command of $P(s)$ has been completed, plus the number of times the second guarded command of $P(s)$ has been completed.

But:

$cPz = cVz$

and

$cVz = p2$ hold in the stable state.

Hence

$cPs = p$ holds in the stable state. (*End of proof of T*)

For $V(s)$, we could have chosen the less symmetrical but somewhat simpler version:

"$P(m)$; $[q = 0 \rightarrow s, r := \ldots; V(m) \,\square\, q > 0 \rightarrow q, p, r := \ldots; V(z)]$".

The above correctness proof holds also for this version. The two versions are thus strictly equivalent.

10. Conclusion

It has been shown that the semantics of synchronization primitives can be fully defined as a set of axioms embodying three fundamental requirements: boundedness, progress, and fairness.

The statement that these three axioms are necessary and sufficient for all synchronization problems is presented here as a conjecture. But the choice of a linear boundedness requirement of the form $-kY \leqq cX - cY \leqq kX$ can be justified: by the very nature of cyclic sequential processes, the synchronization relation should be insensitive to the process histories, which implies that the relation between cX and cY should not be modified by the increase $cX, cY := cX + a, cY + a$.

Whether it is possible to define usable synchronization primitive pairs relying on alternative boundedness relations is an open question.

We have chosen a "strict" progress requirement rather than a "liberal" one of the form:

"A set of initiated synchronization actions is put in the suspended state if and only if its completion would violate R1. A set of suspended actions whose completion would not violate R1 will *eventually* be completed."

There are several reasons for this choice. First, the strict progress requirement guarantees maximal progress of the computation, which is one of our main goals. Second, the semantics of the "eventual completion" is difficult to formalize and to manipulate in correctness proofs. Third, the liberal progress requirement introduces extra danger of starvation.

For example, with liberal *P* and *V* operations the completion of a *V* operation while *P* operations of the same pair are suspended does not imply the immediate completion of a *P* operation. It is thus possible for one process to starve all others, whereas at least two "conspiring" processes are necessary to starve all others with strict primitives. This difference is more important than it seems at first sight: J.M. Morris [6], has proposed a starvation-free solution to the mutual exclusion problem using weak *P* and *V* operations. The correctness of the solution critically depends on the strict progress property of *P* and *V* operations.

The notion of slack could help the programmer to choose between the different types of primitives: the question "under which conditions can one type of primitives be replaced by another one?" can be reformulated as "under which conditions can the slack be decreased (without introducing deadlock) or increased (without introducing unacceptable non-determinism)?"

Anyone who compares the correctness proofs given for, e.g., the mutual exclusion, the producer-consumer, or the deadlock problem with alternative ones to be found in the literature, should be convinced of the suitability of these axioms for correctness proofs. The guiding principle for using these axioms is to consider a pair of synchronization primitives not as two independent primitives, each with its own semantics, but as *one* semantic construct.

Acknowledgement. I am deeply indebted to C.S. Scholten for several important improvements, in particular the unique formulation of the general progress requirement for all values of the slack. Acknowledgement is also due to Edsger W. Dijkstra, F.E.J. Kruseman Aretz, I.S. Herschberg, Martin Rem, and W.M. Turski for invaluable comments and criticisms.

References

1. Dijkstra, E.W.: Co-operating sequential processes. In: Programming languages, pp. 43–112 (F. Genuys, Ed.), New York: Academic Press, 1968
2. Dijkstra, E.W.: Guarded commands, non-determinancy, and formal derivation of programs. Comm. ACM **18**, 453–457 (1975)
3. Dijkstra, E.W.: A class of allocation strategies inducing bounded delays only. Proc. S.J.C.C., pp. 933–936 (1972)
4. Habermann, A.N.: Synchronization of communicating processes. Comm. ACM **15**, 171–176 (1972)
5. Hoare, C.A.R.: Communicating sequential processes. Comm. ACM **21**, 666–677 (1978)
6. Morris, J.M.: A starvation-free solution to the mutual exclusion problem. Information Processing Lett. **8**, 76–80 (1979)
7. Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Comm. ACM **19**, 279–285 (1976)

Received July 3, 1979 / June 21, 1981