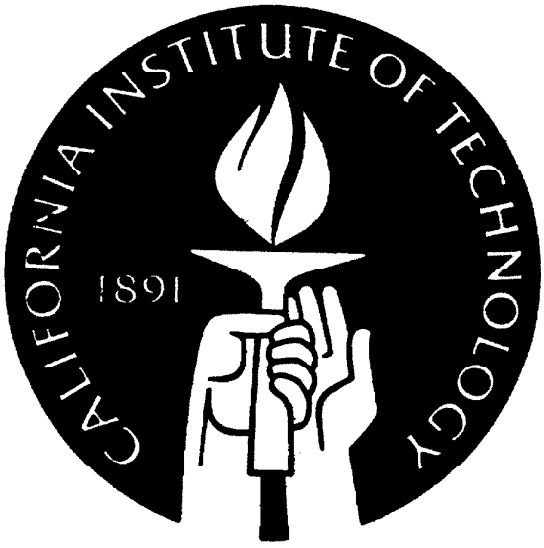

THE PROBE:
AN ADDITION TO COMMUNICATION PRIMITIVES

Alain J. Martin

Computer Science
California Institute of Technology

5124:TR:84

THE PROBE:
AN ADDITION TO COMMUNICATION PRIMITIVES

Alain J. Martin

Computer Science Department
California Institute of Technology

5124:TR:84

The research described in this paper was sponsored by
the Defense Advanced Research Projects Agency, ARPA Order No. 3771,
and monitored by the Office of Naval Research
under contract number N00014-79-C-0597

to appear in:
Information Processing Letters
vol. 20, no. 1, January 1985

The Probe: An Addition to Communication Primitives

Alain J. Martin
Computer Science Department
California Institute of Technology
Pasadena CA 91125, USA

Keywords: communication primitives, distributed processes, semaphores, bounded buffer, stack, fairness, CSP

Introduction

We consider a programming language kernel for distributed computation consisting of a sequential part--assignment, selection, repetition, recursion, sequential composition--extended with concurrent composition of processes and communication primitives. CSP and its variants are such a kernel [4].

We propose a simple addition to the communication primitives used in such a kernel. This extra primitive, called the probe, simplifies the semantics of the language by suppressing the need for communication actions in guards, while enhancing its expressive power.

After introducing the communication primitives and their semantics, the use of the probe is illustrated by a series of examples. Finally, implementation issues are discussed.

1. Sequential processes

For the sequential part, we use guarded commands [2], with CSP syntax: [...] instead of **if...fi**, and *[...] instead of **do...od**. We also simplify *[true --> S] to *[S].

A concurrent program is obtained by composing a set of sequential programs--called processes--by the usual concurrent composition operator ||. Processes do not share variables. They communicate and synchronize their activities by means of the communication primitives.

2. Communication primitives

Communication primitives have two semantic functions: they are synchronization primitives and they can be used to implement a form of distributed assignment. The communication command X in process p is paired with the communication command Y in process r by declaring the pair (X,Y) to be a channel between p and r--(X,Y) and (Y,X) denote the same channel.

2.1 Synchronization axioms

In [5], we show that any pair (X,Y) of synchronization primitives fulfills two semantic requirements. For an arbitrary command A, let cA denote the number of completed A actions (i.e. actions caused by the execution of a command A) at an arbitrary point of the computation. The boundedness requirement R1 states that the value of cX - cY is bounded, from above or from below or both.

The execution of a command X results either in the completion of an X action or in its suspension when its completion would violate R1. From suspension until completion an action is pending and the process executing the action is delayed at that action. We introduce the Boolean qX equal to the predicate "an X action is pending".

The progress requirement R2 states that the set of suspended actions is minimal, i.e. an action is suspended only if its completion would violate R1 (so as to maximize progress). Some of the examples will illustrate the difficulty of satisfying R2 at any point.

It is further shown in [5] that R1 can be realized in three ways, namely:

(i) $cX = cY$

(ii) $0 \leq cX - cY \leq S$, for some constant S, $S > 0$

(iii) $0 \leq cX - cY$.

These relations correspond to three different types of synchronization primitives. We shall use (i) for the communication primitives. Dijkstra's P and V operations on semaphores [3] use (iii). By analogy, we call primitives using (ii) P and V operations on bounded semaphores.

From the above, the synchronization axioms for a channel (X,Y) are

R1: $cX = cY$

R2: $\neg qX \vee \neg qY$

The nth X action is said to match the nth Y action. The completion of a matching pair of actions is called a communication.

2.2 Communication

A channel can be specified to consist of an input command and an output command by adjoining to them the symbols ? and !, respectively (e.g. X? and Y!).

Communication axiom: Let X?a and Y!b be matching, where a and b are process variables. If a,b = A,B before the communication, a,b = B,B after the communication.

3. Probes

Definition: Given the channel (X,Y), the **probe on X**, denoted $\bar{X}$, has the same value as qY, i.e. the same value as the predicate "a Y action is pending", and symmetrically for Y.

The probe primitive has two important properties. Let X belong to process p.

Property 1: The value **true** of $\bar{X}$ is **stable** in p, i.e. if $\bar{X}$ has the value **true**, it keeps it until the next X action. The value **false** of $\bar{X}$ is **unstable** in p: $\bar{X}$ can change from **false** to **true** at any point in p.

Consider the guarded command $B(\bar{X})$ --> S such that $B(\bar{X})$ ==> $\bar{X}$. Then the first X action in S, if any, is said to be probed.

Property 2: A probed action is never suspended.

Hence, $\bar{X}$ --> X guarantees that X is not suspended. But $\neg\bar{X}$ --> X does not guarantee that X is suspended.

Since probes will be used exclusively in guards, the following termination property of the selection command will be important. Let IF be the selection

$$[B_1 \text{ --> } S_1 \,[\!]\, B_2 \text{ --> } S_2 \,[\!]\, .. \,[\!]\, B_n \text{ --> } S_n] \ ,$$

with BB: (E i: $1 \leq i \leq n$: B_i). By definition, IF terminates only if BB holds. Therefore, the execution of IF with $\neg$BB as precondition can be viewed as suspending the completion of IF until BB holds. We say that IF is pending at the guards.

Obviously, in any matching pair of communication actions, at most one of the actions can be probed. Two probed matching actions means deadlock.

4. Channel selection

In its simplest form, the channel selection problem can be posed as follows. Process p communicates with two other processes via channels (X,Y) and (Z,V). (From now on, when a channel is declared together with a process, the first command of the channel belongs to the process.) We want to construct a statement SEL inside p such that the execution of SEL amounts to executing either X or Z, and SEL is suspended if and only if $\neg qY \wedge \neg qV$ holds.

Clearly, it is impossible to construct SEL with synchronizing commands only. With the probe construct, the solution is obvious.

$$SEL : [\bar{X} \text{ --> } X \;[\!]\; \bar{Z} \text{ --> } Z].$$

SEL meets the specification: as an alternative command, its execution amounts to the execution of either X or Z; SEL is suspended only if $\neg\bar{X} \wedge \neg\bar{Z}$ holds, i.e. $\neg qY \wedge \neg qV$.

5. Alignment

Process p communicates with n other processes via the channels (X1,Y1), (X2,Y2),..., (Xn,Yn). We want to implement a construct ALIGN inside p such that the execution of ALIGN is equivalent to the execution of X1, X2, ..., Xn in any order, and it is suspended if and only if $(E\ i: 1 \leq i \leq n: \neg qYi)$ holds. The implementation of such a construct in CSP-like language becomes quite cumbersome as soon as n is larger than 2. With probes, it is simply

$$[\bar{X}1 \wedge \bar{X}2 \wedge \ldots \wedge \bar{X}n \text{ --> } X1;\ X2;\ldots;\ Xn].$$

6. Timer

Given the channel (T,T'), a "timer" performs a communication action T "at regular intervals of time", so to speak. When there is a pending action T', the process p sharing the channel with the timer completes T' and performs S1. Otherwise p repeatedly performs S2.

$$p: *[\neg\bar{T}' \text{ --> } S2 \;[\!]\; \bar{T}' \text{ --> } T';\ S1].$$

7. Fairness

Consider the process

$$p: *[[\bar{X} \text{ --> } S1 \;[\!]\; \bar{Z} \text{ --> } S2]],$$

where S1 contains X but not Z, S2 contains Z but not X, from channels (X,Y)

and (Z,V). We want to guarantee that p is fair towards Y and V, i.e. if Y is pending, only a finite number of V actions are completed, and vice-versa. Obviously, the above version of p is not fair due to the non-deterministic choice of a guard when both guards are true. Negated probes make it possible to transform p into a fair version, namely

```
p': *[[X̄ --> S1; [Z̄ --> S2 [] ¬Z̄ --> skip]
     ]Z̄ --> S2; [X̄ --> S1 [] ¬X̄ --> skip]
     ]].                                          (0)
```

The execution of a guarded command of the outer selection when both guards are true results in the execution of S1 and S2 in some order. Another--only slightly different--version is

```
p": *[[X̄ ∧ ¬Z̄ --> S1
     ]Z̄ ∧ ¬X̄ --> S2
     []X̄ ∧  Z̄ --> S1; S2
     ]] .                                         (1)
```

(In the sequel, variants of (1) will be used in some examples to maximize progress.)

8. Bounded semaphores

We want to implement a process bsem using the channels (P,P') and (V,V') such that the pair of commands (P',V') fulfills the semantics of P and V operations on bounded semaphores, namely

R1: $0 \leq cV' - cP' \leq S$

R2: $(\neg qP' \vee cV' = cP') \wedge (\neg qV' \vee cV' = cP' + S)$.

A first version is

```
bsem: *[[P̄ ∧ 0 < s --> s:= s - 1; P
       ‖V̄ ∧ s < S --> s:= s + 1; V
       ]].
```
(2)

Since s = cV′ - cP′ holds as precondition of each guarded command, R1 holds at any point. Since bsem can only be delayed at a guard, when bsem is delayed ¬(P̄ ∧ 0 < s) ∧ ¬(V̄ ∧ s < S) holds, i.e. R2 holds. But the first guarded command can be selected S times while V̄ holds and the second guarded command can be selected S times while P̄ holds. Hence it can take up to S consecutive guarded commands executions before R2 is re-established. The following solution guarantees that R2 holds after each guarded command execution.

```
bsem: *[[P̄ ∧ ¬V̄ ∧ 0 < s --> s:= s - 1; P
       ‖V̄ ∧ ¬P̄ ∧ s < S --> s:= s + 1; V
       ‖P̄ ∧  V̄ ∧ 0 < s --> P; V
       ‖P̄ ∧  V̄ ∧ s < S --> V; P
       ]].
```
(3)

(Solution (3) is an improvement due to Jan van de Snepscheut of a previous program.)

9. General semaphores

We can adapt (2) and (3) to implement a process sem using the channels (P,P′) and (V,V′) such that (P′,V′) fulfills the semantics of P and V operations on general semaphores, namely

R1: $0 \leq cV' - cP'$

R2: $(\neg qP' \vee cV' = cP') \wedge \neg qV'$.

Adapting (2) for S infinite leads to

```
sem: *[[P̄ ∧ 0 < s --> s:= s - 1; P
      ▯V̄        --> s:= s + 1; V                    (4)
      ]].
```

R1 is satisfied at any point. However, the second guarded command can be selected an unbounded number of times while $\bar{P}$ holds, so that R2 may not hold for an unbounded number of steps. In the following solution, adapted from (3), R1 and R2 are satisfied after each guarded command execution.

```
sem: *[[P̄ ∧ ¬V̄ ∧ 0 < s --> s:= s - 1; P
      ▯V̄ ∧ ¬P̄        --> s:= s + 1; V
      ▯P̄ ∧ V̄         --> V; P                        (5)
      ]].
```

10. Bounded buffer

Program (2) can readily be adapted for implementing a process controlling an N-portion circular buffer b[i: $0 \leq i < N$]. Portions are added to b by the input command put?. Portions are removed from b by the output command get!.

```
buf: *[[put ∧ n < N --> put?b[i]; n,i:= n + 1, i + 1
      ▯get ∧ 0 < n --> get!b[j]; n,j:= n - 1, j + 1   (6)
      ]].
```

Operations on i and j are modulo N. If the buffer is empty initially, a proper initialization is n = 0, i = j.

This example shows that input and output commands are used symmetrically. Notice that (6) is similar to (2), and thus can be transformed into a version similar to (3).

11. Stack

We want to implement a stack S of size n, $n > 0$, as a string of n communicating processes defined as follows:

for $n = 1$: $S = h$
for $n > 1$: $S = [h \,||\, T]$

where h, the "head of S", is a process, and T, the "tail of S", is a stack of size n-1.

A process p, the "partner of S", adds elements of type t to S by the actions put! and removes elements of type t from S by the actions get?. The matching actions from the stack are the actions in? and out! from h. Further, the head of S is the partner of T.

A list L of elements of type t is associated with S in the following way:

$$b \wedge L = x{:}LT \vee \neg b \wedge L = LT \qquad (7)$$

where LT is the list associated with T, and LT is empty for $n = 1$,

: denotes concatenation,
b is a Boolean variable of h,
x is a variable of h of type t,
initially, all b's are false and thus L is empty.

Elements are added to and removed from L by actions put! and get? from h so as to satisfy:

$$\{x = xv \wedge L = Lv\}\ \text{put!}x\ \{L = xv{:}Lv\}$$
$$\{L = xv{:}Lv\}\qquad \text{get?}x\ \{x = xv \wedge L = Lv\} \qquad (8)$$

where xv and Lv denote values of the variables x and L.

Obviously, we must require that

$0 \leq$ cput - cget $\leq n$

be satisfied by the partner of S. Under this condition, the following program satisfies (7), (8), and $0 \leq$ cput - cget $\leq n-1$:

```
h: *[[b ∧ in  --> put!x; in?x
     [b ∧ out --> out!x; b:= ¬b
     [¬b ∧ in --> in?x ; b:= ¬b
     [¬b ∧ out--> get?x; out!x
    ]].
```

(I owe this program to Jan van de Snepscheut.)

12. Implementation

Discussing the implementation of probes in general is impossible, since the issue depends strongly on the implementation medium chosen. We can show how to solve the problem in a typical case.

We want to implement each communication on a channel (X,Y), by a so-called "four-phase handshaking protocol". Four Boolean variables are used: x and x′ are accessible to the process containing X; y and y′ are accessible to the process containing Y. Variables y and x on the one hand, x′ and y′ on the other hand are related by two processes called "wires" (and implemented as such):

```
wire(y,x) : *[y --> x↑ [] ¬y  --> x↓],
wire(x′,y′): *[x′--> y′↑ [] ¬x′ --> y′↓],
```

(a↑ and a↓ stand for a:= true and a:= false, respectively.)

With these definitions, a matching pair (X,Y), without probe, can be implemented

X: [x]; x′↑; [¬x]; x′↓
Y: y↑; [y′]; y↓; [¬y′] (9)

([a] stands for [a --> skip]). Initially, all handshaking variables are false.

Observe that the programs for X and Y are not symmetrical: X is said to be driven and Y to be driving. The choice is arbitrary, but a choice has to be made.

In order to implement the probe construct without restriction, we introduce two extra wires (px, py′) and (py, px′). And we extend the implementations of X and Y as follows:

px↑; [px′]; X; px↓; [¬px′]
py↑; [py′]; Y; py↓; [¬py′] (10)

Now the probe $\bar{X}$ can be implemented as px′, and the probe $\bar{Y}$ as py′.

One can avoid introducing the wires (px, py′) and (py, px′) by slightly restricting the use of probes. Consider the channel (X,Y) in a concurrent program p. We impose that one of the two commands of the channel never be probed in p. If none of the two commands is probed, the choice of driving and driven implementations is arbitrary. If one command--say, X--is probed, X is given the driven implementation, and Y the driving one. The probe $\bar{X}$ is implemented as x, and, e.g., the construct $\bar{X}$ -->..X as x -->..x↑; [¬x]; x′↓. Observe that this restriction is not as severe as forbidding probed output commands (equivalent to the CSP restriction of forbidding output commands in guards).

For instance, the bounded buffer would still be programmed as (6). However, the processes communicating with the buffer are not allowed to

probe communication actions with the buffer, which is obviously no restriction in this case.

Conclusion

The notion of suspension of an action is essential to describe the progress and fairness of a computation. Yet, none of the currently used communication primitive sets makes it possible to determine directly the suspended state of an action.

This omission might be traced back to the P and V synchronization primitives where this possibility is not present either. But in the case of P and V, the absence of probes is easily compensated by the use of shared variables recording the suspended state of an action. It is, for instance, impossible to implement general P and V operations with a fixed number of binary P and V operations and for an arbitrary number of processes without introducing shared variables to count the pending P operations. With the exclusion of shared variables, a primitive must be provided to determine whether an action is pending.

It is sometimes argued that the state in which an action is pending is not distinguishable from the state in which the initiation of the action is delayed. In our opinion, this is wrong: suspending an ation definitely changes the state of the computation. It is essential to be able to detect this change of state, and that is made possible by the probe.

By separating the testing of a pending communication action from the "firing" of the communication, the semantics are both simpler and more general. The communication commands are entirely symmetrical in input and output. Yet we do not encounter the semantic and implementation problems occurring in CSP-like languages when input and output are allowed in guards [1].

The negated probe considerably enhances the possibilities of the language by making the implementation of fairness properties and "timer-like" constructs possible and easy. The construct $\bar{X}$ --> S; X , which makes it possible to delay the firing of X until the completion of S, also enhances the possibilities of the language.

Finally, the implementation of the probe poses no major problem.

Acknowledgment. I am grateful to Jan van de Snepscheut, Martin Rem, Chuck Seitz, and Wlad Turski for valuable comments and criticisms. Jan also contributed improvements to the solutions of the bounded semaphore and the stack.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

References

[1] A.J. Bernstein, Output guards and nondeterminism in "Communicating sequential processes", ACM TOPLAS 2 (3) (1980) 234-238.

[2] E.W. Dijkstra, **A discipline of programming**, Prentice-Hall (1976).

[3] E.W. Dijkstra, Cooperating sequential processes, in: F. Genuys (ed.) **Programming Languages** (Academic Press, 1968).

[4] C.A.R. Hoare, Communicating sequential processes, Comm. ACM 21 (1978) 666-677.

[5] A.J. Martin, An axiomatic definition of synchronization primitives, Acta Informatica 16 (1981) 219-235.

28 February 1984