A Guide to Communicating Sequential Processes

by

Shan S. Kuo*,

Michael H. Linck**

and

Sohrab Saadat***

Technical Monograph PRG-14

August 1978

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
Oxford.

* On leave from the University of New Hampshire.
** On leave from the University of Natal.
*** On leave from Pahlavi University, Shiraz.

c/o Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford.
OX2 6PE.

ABSTRACT

This report is a tutorial introduction to a technique of programming which involves the communication between two or more concurrently executing processes. The notations of communicating sequential processes (CSP), suggested by C.A.R. Hoare, are presented in detail.

This report is chiefly intended for 'educated scientists' who are encountering the subject of parallel processing for the first time. For pedagogical reasons, a large number of examples of increasing conceptual complexity are given and solved throughout the report.

CHAPTER 1

INTRODUCTION

The simultaneous execution of two or more sets of sequential computer instructions is called parallel processing, concurrent programming, or multiprocessing. For example, many modern computers facilitate some degree of parallel operations by providing two processors. The first processor is commonly known as the central processing unit (CPU) used to execute 'regular' instructions; and the second is an Input/Output(I/O) processor, sometimes known as a channel, to process I/O commands[18]. The CPU and I/O processor work in parallel. The present-day trend in computer design is to have as many system components as possible operating in parallel. The advent of inexpensive microprocessors has helped to accelerate this trend.

Despite the complexity involved in reliably controlling parallel processing, two important and interrelated areas of research seem to stand out.

One of them is the development of a notational system, also known as an abstract language or even a programming language, for expressing the program structure and the data structure. One of the goals of a notational system is to enable workers in the computer field to express their thoughts on programs or algorithms in a simple, precise, and transparent manner, so that a colleague will understand. Since parallel processing is a complex topic, it is of the utmost importance to develop a good notational system. The communicating sequential processes (CSP) notation, using a single structuring method, is a significant step in this direction [14].

The other research area seems to focus on the identification of some fundamental, or 'primitive', operations. Such primitive operations are akin to an operator as used in mathematics (e.g. +,-,x,÷). It is useful to classify primitives into two groups: executable primitives and structural primitives. Examples in each group are listed in Table 1.1.

Structural primitives	Executable primitives	
	CSP,Fortran,Pascal,etc	Assembler language
Sequential composition	Assignment	Load
Parallel composition	Expressions	Jump
Alternative	I/O	Store
Go to(Jump)		
Concitional Go to		

Table1.1 Examples of Primitives.

This technical note is a tutorial exposition of the recent work in these areas by C.A.R. Hoare [14]. It is chiefly intended for those who are encountering the subject for the first time. The reader is expected to have experience in programming in at least one high-level language.

In Chapter 2, we commence with some simple examples illustrating how the well-known programming constructs--- sequential composition, selection, and repetition--- are related to Hoare's CSP notational system.

In Chapter 3, the parallel command and examples of parallel composition are studied. Communication between two processes using I/O commands is introduced for the first time.

Chapter 4 deals with synchronization and buffering. It shows how a buffer can be implemented as a process. The classical consumer/producer problem [6] is reviewed and its solution, expressed in CSP, is studied in depth.

Chapter 5 introduces the concept of an array of processes. It is illustrated by several examples including the well-known factorial and prime number problems.

Finally, it is worth noting that the CSP notational system, like many other computer languages, rapidly gives rise to a host of extensions and modifications. In this report, we study only the topics which are presented in the original paper[14]. In particular, the topics of recursion and procedures are not treated. A tutorial exposition of the recent extensions by C.A.R. Hoare and C.M. Holt will be published in a separate report[15].

CHAPTER 2

ALTERNATIVE AND REPETITIVE COMMANDS

This chapter is mainly devoted to the conditional and repetitive commands as expressed in CSP. These commands are based on Dijkstra's guarded command [8]. A large number of examples is presented. Whenever practicable, Algol 60 [19] statements are listed side by side with the CSP commands.

2.1 A Simple Example for CSP

Let us consider the simple example of the swapping of two real numbers a and b.

```
begin
     real a,b;
     a:=3; b:=5;
     begin
         comment swap values
                   of a and b;
          real t;
          t:=a;
          a:=b;
          b:=t
     end
end
```

(a)Solution expressed in Algol.

```
[
   a,b:real;
   a:=3; b:=5;
   [
      comment swap values
                of a and b;
        t:real;
        t:=a;
        a:=b;
        b:=t
   ]
]
```

(b)Solution expressed in CSP.

Figure 2.1 Swapping of two numbers.

From Fig. 2.1 we see that the symbols '[' and ']' in CSP are respectively analogous to begin and end in Algol. The block structure and scope rules for variables used in CSP are similar to those used in Algol. The semicolons are used to indicate sequential execution. We note that the five assignment commands appearing in both languages are identical. We also note the minor difference in the declaration of the three variables a,b and t. There are four 'standard' types available in CSP: real,integer,boolean,and character. Also an array is declared in CSP as follows:

```
p:(1..100)integer;
```

where p is a one dimensional array of type integer having 100 elements.

2.2 Alternative Command

Let us consider the function

$$y=\begin{cases} 2x & \text{if } x<0 \\ x^2 & \text{if } x\geq 0 \end{cases}$$

To write this function in CSP one can use the following alternative command:

```
[ x<0 → y:=2*x
 □x≥0 → y:=x*x
]
```

We summarize in Table 2.1 terminologies for the various components of an alternative command.

Terminologies	CSP representation
Alternative command	[x<0 → y:=2*x □ x≥0 → y:=x*x]
Guarded commands	x<0 → y:=2*x x≥0 → y:=x*x
Guards	x<0 x≥0
Commands	y:=2*x y:=x*x

Table 2.1 Summary of components of an alternative command.

This command consists of two guarded commands separated by a symbol '□' (read as 'fat bar', and interpreted as an OR). This command may be translated as follows: If the guard $x<0$ is true, then compute y:=2*x. If this guard is false, then the command y:=2*x is not executed. Similarly, if the guard $x\geq0$ is true, then compute y:=x*x. If it is false, then the command y:=x*x is not executed.

It is possible for the evaluation of the two guards, $x<0$ and $x\geq0$, to start at the same time and to continue in parallel. As soon as either guard is true, the following two events will take place:

1) any further evaluation of the other guard is discontinued; and
2) the command corresponding to the successful guard is executed.

In this particular example, one of the two guards is always true. There are situations where 1) no guard is true; or 2) more than one guard is true. These will be discussed in the next two examples.

Case 1: No guard is true.

Let us consider the function

$$y=\begin{cases} 2x & \text{if } x<0 \\ x^2 & \text{if } x>0 \end{cases}$$

In CSP this function may be expressed as:

```
[ x<0 → y:=2*x
 □x>0 → y:=x*x
]
```

If x=0, both the guards $x<0$ and $x>0$ will fail. As a result neither of the commands y:=2*x and y:=x*x will be executed. The alternative command fails and the program which contains it will abort. This sequence of actions is shown in Fig. 2.2.

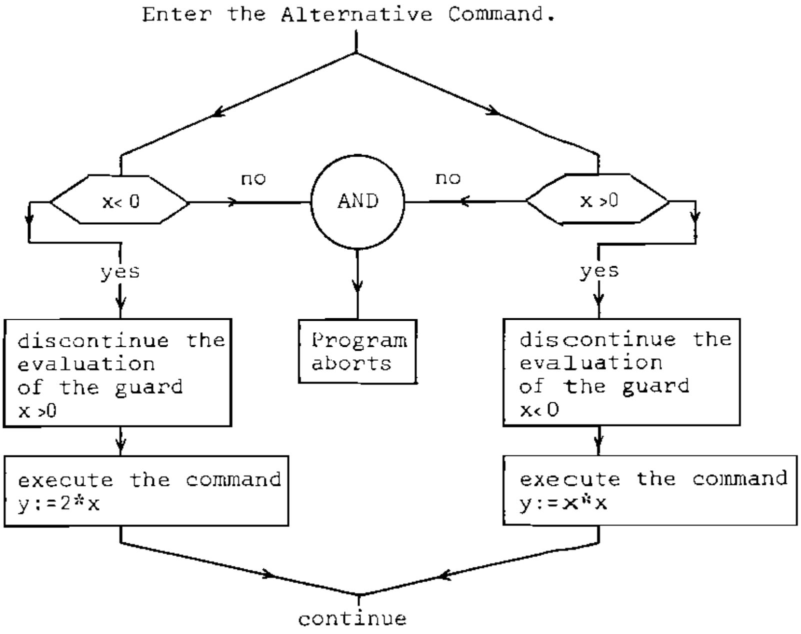

Enter the Alternative Command.
x< 0
no
AND
no
x >0
yes
yes
discontinue the evaluation of the guard x >0
Program aborts
discontinue the evaluation of the guard x< 0
execute the command y:=2*x
execute the command y:=x*x
continue

Figure 2.2 Diagram showing details of the execution of the Alternative command.

```
[ x< 0 → y:=2*x
 □x >0 → y:=x*x
]
```

Case 2: More than one guard is true.

Let us take the above example again, except that now both '=' signs are included in the function:

$$y=\begin{cases}2x & \text{if } x\le 0\\ x^2 & \text{if } x\ge 0\end{cases}$$

The corresponding CSP representation is as follows:

```
[ x≤0 → y:=2*x
 □x≥0 → y:=x*x
]
```

In the case that x=0, both guards are true. It should be strongly emphasized that we have no knowledge as to which guard will succeed first. As a consequence, we do not know which one of the following commands:

```
y:=2*x or y:=x*x
```

is executed. This is known as <u>nondeterminism</u>. Fortunately, in this case, it just does not matter which of the commands is executed.

So far, in our examples, each guard contains only one boolean expression. However a guard may consist of two or more boolean expressions, separated by semicolons.

Take the example of evaluating the following function:

$$y=(x+12)/((x-1)(x+2)). \quad (\ x\ne 1 \text{ and } x\ne -2\ )$$

The CSP representation of this function takes the form:

```
[ x≠1 ; x≠-2 → y:=(x+12)/((x-1)*(x+2)) ]
```

The guard is evaluated as follows:

Step 1. Check if the guard x≠1 is true. If so, go to step 2. Otherwise the program aborts.

Step 2. Check if the guard x≠-2 is true. If so, evaluate the command y:=(x+12)/((x-1)*(x+2)). Otherwise the program aborts.

2.2.1 Alternative Command with Range

We shall now show how several guarded commands with subscripted variables may be written in a compact form. As a specific example, consider the three guarded commands shown in Fig. 2.3(a). This can be written simply as shown in Fig. 2.3(b), where the index variable (also known as the bound variable) k is used. The expression (k:1..3) is known as the range. In the compact form, the fat bar symbol is not used.

```
[  a(1)>0 → a(1):=a(1)-1
  □a(2)>0 → a(2):=a(2)-2
  □a(3)>0 → a(3):=a(3)-3
]
```

(a) Expanded form.

```
[(k:1..3)a(k)>0 → a(k):=a(k)-k]
```

(b) Compact form.

Figure 2.3 An alternative command consisting of three guarded commands.

2.2.2 Summary

In Table 2.2 we summarize the evaluation of guards in an alternative command. It also lists which command, if any, is to be executed.

Result of evaluation of the guards	Which command, if any, is to be executed
All guards are false	None; the program aborts.
Only one of the guards is true	The corresponding command is executed
Two or more guards are true.	Exactly one of the commands, depending on implementation.

Table 2.2 Guards and guarded commands.

2.3 Repetitive Command

We turn now to describe how the repetitive construct is expressed in CSP. Two examples will follow. The first example will deal with the sum of the integers:

$$s = \sum_{i=1}^{100} i.$$

The computation as expressed in Algol and in CSP, is listed in Fig. 2.4. In Fig. 2.4(b), the last command is called a repetitive command. It consists of an alternative command preceded by the symbol '$\underline{*}$' . This symbol may be interpreted as: repeatedly execute the following alternative command until all its guards fail. When all the guards fail the repetitive command terminates and control is transferred to the next command.

```
integer s,k;
s:=0;
for k:=1 step 1 until 100 do
    s:=s+k
```
	```
	s,k:integer;
	s:=0 ; k:=1;
	*[ k≤100 → s:=s+k ; k:=k+1 ]
	```

(a) In Algol.	(b) In CSP.

Figure 2.4 Sum of the integers.

As the second example, consider arrays a and b. We wish to interchange a_k and b_k if $a_k > b_k (1 \leq k \leq 100)$. The solution shown in Fig. 2.5 needs some explanation. First, there are 100 guarded commands, one for each k value, in the alternative command. Second, the alternative command is repeatedly executed until all 100 guards fail. Third, for each iteration, the 100 guards are evaluated concurrently, the command list (on the right hand side of the arrow) corresponding to the first successful guard is then executed. During any iteration, however, if all the 100 guards fail, then none of the guarded commands is executed. The repetitive command terminates, and control is transferred to the next command. Finally, the number of iterations may vary from 1 to 100, depending on how often $a_k > b_k$ occurs. Fig. 2.6 shows the detail of evaluation of this repetitive command.

```
a,b:(1..100)real;

    comment Assume that random
    values have been assigned
    to the array elements;

    *[ (k:1..100)a(k)>b(k) → t:real;
                               t:=a(k);
                               a(k):=b(k);
                               b(k):=t
     ]
```

Figure 2.5 Swapping of a(k) and b(k) if a(k)>b(k) [1≤k≤100.]

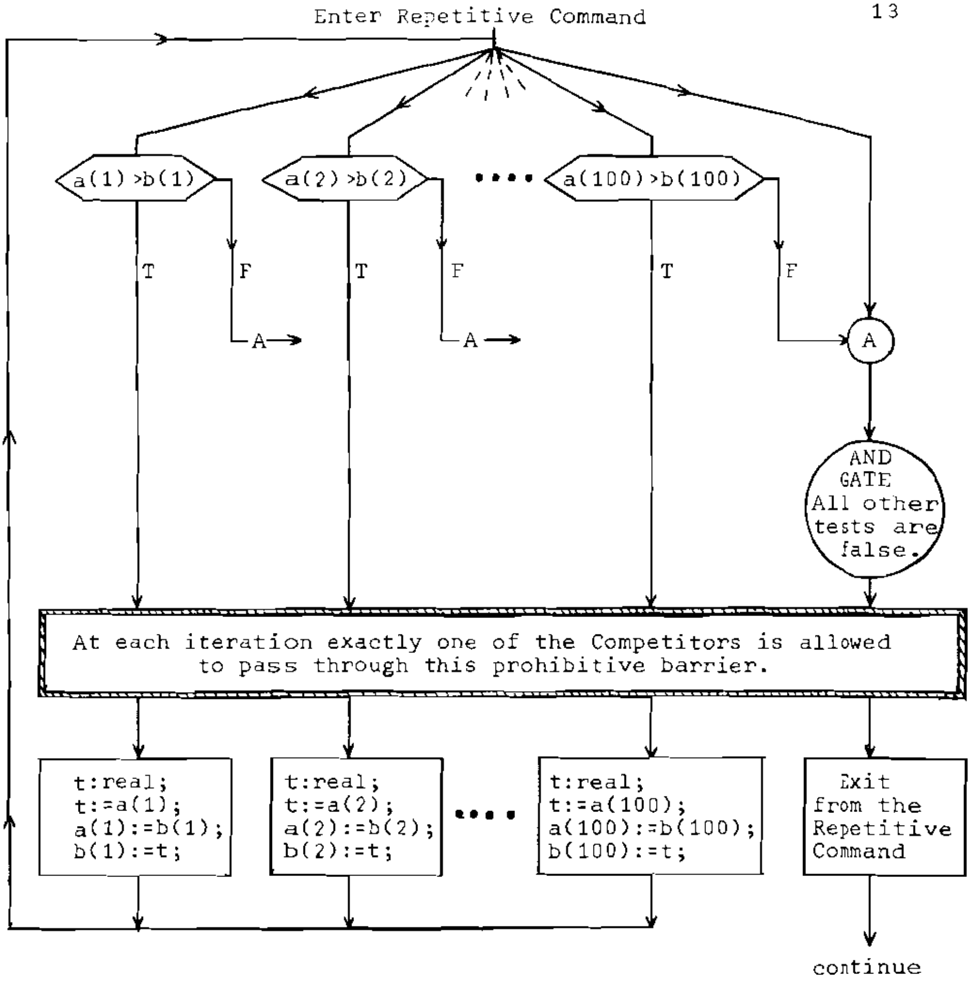

Enter Repetitive Command
a(1)>b(1)
a(2)>b(2)
a(100)>b(100)
T
F
A
AND GATE All other tests are false.
At each iteration exactly one of the Competitors is allowed to pass through this prohibitive barrier.
t:real; t:=a(1); a(1):=b(1); b(1):=t;
t:real; t:=a(2); a(2):=b(2); b(2):=t;
t:real; t:=a(100); a(100):=b(100); b(100):=t;
Exit from the Repetitive Command
continue

Figure 2.6 Diagram showing the detail and execution of the Repetitive Command given in Fig. 2.5.

CHAPTER 3

PARALLEL COMMANDS AND INPUT/OUTPUT COMMANDS.

In this chapter we first introduce the parallel command. Its purpose is to define two or more processes that will run concurrently. We then introduce input and output commands and show how they are used to effect communication between processes.

3.1 Parallel Commands.

Consider the evaluation of the function

$$y=(x+1)\sin(x+1) + \cos(x)\cos(2x)\cos(3x) \qquad \text{for } x=3.$$

A possible CSP description is shown in Fig. 3.1.

```
        x,y,p,prod:real;
        i:integer;
        x:=3;
        [   C:: p:=x+1;                                  } Process C
                p:=p*sin(p)

         || D:: prod:=1;i:=1;                            } Process D
                *[i≤3 → prod:=prod*cos(i*x);
                        i:=i+1
                 ]
        ];
        y:=p+prod
```

Figure 3.1 CSP evaluation of the function
$y=(x+1)\sin(x+1) + \cos(x)\cos(2x)\cos(3x)$

The shaded command in Fig. 3.1 is known as a parallel command and consists of two processes (or sets of commands)[4,5]. The commands used to evaluate the Sine term are collectively known as Process C. The process label C must be followed by the symbol '::'. Similarly, the commands for the Cosine term are labelled Process D. Processes C and D are separated by the symbol '||', which indicates that they are executed in parallel.

The parallel command, shaded in Fig. 3.1, is executed in the following manner:

1) The execution of the two processes labelled C and D, start at the same time and continue in parallel ;
2) The parallel command is sucessfully completed only when the execution of process C and process D are both completed; and
3) No assumptions, at all, are made about the relative speeds at which the commands in process C and those in process D are executed.

In general, a parallel command consists of two or more processes,enclosed between a pair of square brackets '[' and ']', and separated by the symbol '||'. All the processes of this command are executed in parallel.

There is a non-local variable p which appears on the left hand side of two different assignment commands in process C. Its value changes first to 4 then to 4sin4. This variable p must not be used in process D. The variable x appears in both process C and D. This is acceptable because its value is not changed in either process (x occurs on the right hand side of both the assignment commands in which it appears). In general, each process of a parallel command must be disjoint from every other process of the command, in the sense that it does not mention any variable to which a value is assigned in any other process.

3.2 Input/Output Commands

In CSP the symbol '?' means input and is used in the input command. The symbol '!' means output and is used in the output command.

Fig. 3.2 shows how the input and the output commands are used to send the value of x from process A to process B.

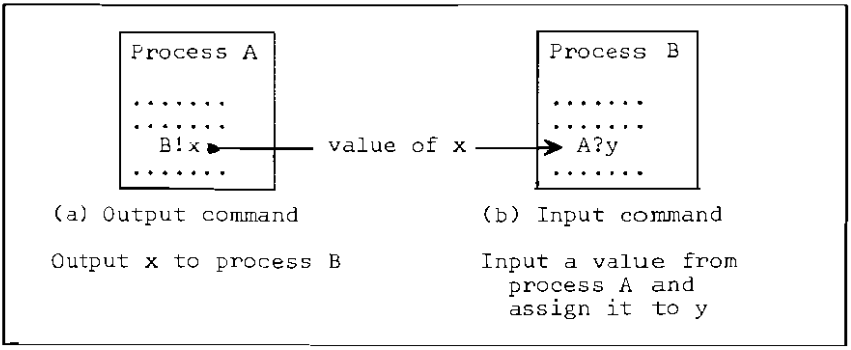


Figure 3.2 Input and Output Commands in CSP.

The input command A?y consists of 3 parts:

1) A, is a process name, specifying the source of the input;
2) ?, is the symbol that means input; and
3) y, is a variable name, the target, which is to receive the input value.

A?y is interpreted as follows:

From the process A input a value and assign that value to the target variable y.

The output command B!x also consists of 3 parts:

1) B, is a process name specifying the destination of the

output;

2) !, is the symbol that means output; and

3) x, is an expression.

B!x is interpreted as follows:

To the process named B output the value of x.

Let us reconsider the evaluation of the function

y=(x+1)sin(x+1) + cos(x)cos(2x)cos(3x)   for x=3.

In order to illustrate how the input/output commands are used, we will take two processes called C and D. They are used respectively to compute the first and second term of the function y. The final sum of the two terms is computed in process C. The details, expressed in CSP, are shown in Fig. 3.3.

```
[C::x,y,p,q:real;                   ||D::prod,val:real;
                                         i:integer;
    x:=3;                                prod:=1;
    D!x;>-------- value of x ------->    C?val;
    p:=x+1;                              i:=1;
    p:=p*sin(p);                         *[i≤3 →
                                          prod:=prod*cos(i*val);
                                          i:=i+1
                                          ];
    D?q; <-------- value of prod ---<    C!prod ]
    y:=p+q
```

Figure 3.3 Input/Output commands in CSP.

Process C and D use no non-local variables. In order to send the value of 'x' from C to D, C uses the output command D!x and D uses the input command C?val to receive this value. Similarly, to send the value of the term cos(x)cos(2x)cos(3x) from D to C, D uses the output command C!prod and C uses the input command D?q.

The interaction between the output command D!x in process C and the input command C?val in process D can be explained as follows:

1) The first command encountered will be delayed until the other command is ready;
2) The output command, D!x , names the process to which x is to be sent (and in which the input command C?val occurs);
3) The input command, C?val , names the process from which a value is required (and in which the output command D!x occurs);
4) The type of the variable in the input command (<u>val</u>) must match the type of expression in the output command (<u>x</u>);
5) When conditions 1,2,3 and 4 are met the input and output commands are said to <u>correspond</u>. They are executed simultaneously. In this example their combined effect is to assign the value of x to the target variable val;
6) Should an input and an output command not correspond then both commands fail and the processes that contain them are both aborted; and
7) Should an output command specify a destination process that has terminated, then the output command fails and the process that contains it is aborted, and similarly for an input command.

Finally, we emphasise that communication between processes is strictly synchronised in CSP (i.e. there is no buffering.)

A guard was introduced in Chapter 2. It consists of one or more boolean expressions. We now allow a guard also to contain an input command.

Consider the example of calculating the sum of the negative integers contained in a 10-element array. Two processes, called COMPARE and COUNT, are used to solve this problem. Process COMPARE outputs to process COUNT all the negative integers found in the array x, while COUNT sums all the values sent to it. A CSP description is shown in Fig. 3.4 and illustrates several points:

1) The process COMPARE will terminate after the 10 integers of the array x have been processed;
2) The input command COMPARE?y, shaded in Fig. 3.4, is used as a guard. This guard becomes true after COMPARE?y has been executed. (COUNT!x(i) is executed simultaneously with COMPARE?y and the value of x(i) is assigned to y). After this guard becomes true, the assignment command sum:=sum+y is executed; but after the process COMPARE has terminated this guard fails; and
3) As soon as the guard fails, the repetitive command, $\underline{*}$[COMPARE?y → sum:=sum+y] will terminate (It does not abort under these conditions).
   In general, a repetitive command will terminate on failure of all the guards contained in it.

```
[ COMPARE::i:integer;x:(1..10)integer;
           Comment We assume that array x has 10
                   random integer values;
           i:=1;
           *[i≤10  →  [ x(i)≥0 → skip
                       □x(i)<0 → COUNT!x(i)
                      ];
                      i:=i+1
           ]
 || COUNT::y,sum:integer;
           sum:=0;
           *[ COMPARE?y  → sum:=sum+y]
]
```

Figure 3.4 An input command as a guard.

It is possible for a guard to consist of one or more boolean expressions followed by a single input command. The boolean expressions must precede the input command. Consider the following problem: Process A sends the values 1,2,...,10 to process B. Process B receives these 10 values and sums them. A CSP description, which uses a guard comprising a boolean expression and an input command, is shown in Fig. 3.5.

```
[  A:: i:integer;
       i:=1;
       *[i≤10 → B!i;  i:=i+1]

 ||B:: j,x,total:integer;
       j:=1;  total:=0;
       *[j≤10;A?x → total:=total+x;  j:=j+1]
]
```

Figure 3.5 A guard consisting of a boolean expression followed by an input command.

The repetitive command, shaded in Fig. 3.5 , terminates after 10 iterations because the boolean expression j≤10 is false, which causes the guard to fail. (We also observe that, on this 11th evaluation of the guard the input command A?x would also fail, because process A has already terminated.)

A repetitive command may contain multiple guards. The following problem solution illustrates this construct: Three processes A, B and C respectively send 5,10 and 15 integer values to process SUM. Each process uses a simple function to generate its values. Process SUM (1) accepts each value sent to it; (2) adds each value to the cumulative total and (3) after all 30 values have been received, sends the value of the total to be printed. A CSP description is shown in Fig. 3.6.

```
[  A::  i:integer;
        i:=1;
        *[i<6 → SUM!i;  i:=i+1]

 ||B::  j:integer;
        j:=1;
        *[j<11 → SUM!j*j;  j:=j+1]

 ||C::  k:integer;
        k:=1;
        *[k<16 → SUM!k*k*k;  k:=k+1]

 ||SUM::x,total:integer;
        total:=0;
        *[ A?x → total:=total+x
          □B?x → total:=total+x
          □C?x → total:=total+x
         ];
        print!total
]
```

Figure 3.6 An example showing a repetitive command with multiple guards.

This description illustrates the following points:

1) The repetitive command, shaded in Fig. 3.6, will accept all the integers sent to it by processes A,B and C.(In this example, 5 values from A, 10 from B and 15 from C);
2) The order in which integers are accepted from processes A, B and C is not determined because it depends on the relative speeds of these three processes;
3) After process A has sent 5 values to process SUM it terminates. This termination causes the guard A?x to fail. Similarly the guards B?x and C?x fail after processes B and C terminate. When all the guards A?x, B?x and C?x of the repetitive command have failed the command itself terminates and the next command ,print!total, is executed; and

4) The output command print!total, that occurs in process SUM, is interpreted as sending the value of 'total' to process 'print' which prints this value.

CHAPTER 4

SYNCHRONIZATION AND BUFFERING

4.1 Communication between two processes revisited

In chapter 3 we studied the input/output commands. At the risk of repetition, we shall briefly review this important topic by means of a concrete example.

If process A wishes to pass the value 7 to process B, we may use the pair of I/O commands shown in Table 4.1, where the value 7 is 'assigned to' the target variable S.

Commands	Used in process	A	B
		B ! 7 destination expression	A ? S source target variable
	type	output	input
'loose' translation		To process B, output 7	From process A, input to S.

Table 4.1 Example of I/O commands.

Before such communication can be completed, the two processes A and B must meet the following three conditions:

1) Process A must contain an output command, which specifies the process B as its destination;
2) Process B must have an input command, specifying the

process A as its source;

3) The type of target variable in the input command must match that of the expression in the output command.

If they satisfy the conditions, the output command in process A and the input command in process B are called a pair of corresponding commands. In CSP, some waiting is usually necessary for one of the corresponding commands. For example, if the output command in process A is ready before the input command in process B, then the process A must wait until the input command in process B becomes ready. Similarly, if the input command in process B is ready before the output command in process A, then the process B must wait until the output command in process A becomes ready. This waiting for simultaneous execution of a pair of corresponding commands is known as synchronization.

In the remaining part of this chapter, we shall further illustrate the important concept of synchronization of I/O commands with three more examples. The first two deal with a single buffer and double buffer respectively; the third, a solution of the classical Producer/Consumer problem using a bounded buffer of K-slots.

4.2 Buffers

Communication between two corresponding processes is synchronized, i.e. executed simultaneously by both processes. There is no third party involved, such as a buffer. In a computer system a buffer consists of some permanent storage used to smooth out temporary variations in the rate of flow of data, when transmitted from one process to another.

Consider the case of sending a series of numbers, in this example the integers 1 to 10, from a producer process to another process called the consumer.

A CSP solution containing no buffer is shown in Fig. 4.1. Once the producer has produced and sent a value to the consumer it can work on the next value but can not send it until the consumer is ready to accept it.

```
[  PRODUCER:: i:integer;
              i:=1;
              *[i≤10 → CONSUMER!i; i:=i+1 ]

|| CONSUMER:: x:integer;
              *[PRODUCER?x → .... ]
]
```

Figure 4.1 Unbuffered communication between the PRODUCER and the CONSUMER.

We now introduce a buffer process as an intermediary between the producer and the consumer. The CSP description is shown in Fig. 4.2.

```
[  PRODUCER:: i:integer;
              i:=1;
              *[i≤10 → BUFFER!i; i:=i+1 ]

|| BUFFER::   slot:integer;
              *[PRODUCER?slot → CONSUMER!slot ]

|| CONSUMER:: x:integer;
              *[BUFFER?x → .... ]
]
```

Figure 4.2 CSP solution for a single buffer.

The action of the buffer can be described as follows:

a) Each value produced is transferred to the buffer as soon as possible. After the buffer has been 'filled' this value is sent to the consumer as soon as possible.

b) When the producer process is ready to output, it outputs to the buffer; it does not have to wait for the consumer ; instead, it can then continue processing until it produces the next integer. Then, there are two cases:
   1) If the buffer has managed, by that time, to output the previous value to the consumer, it can then input the next integer; or
   2) If the buffer has not yet output the previous value, the producer must wait until it does.

4.3 Double buffers

We can improve the efficiency of a single buffer discussed in the previous section by adding one more buffer. We shall describe two possible methods of double buffering.

In the first method we simply put another buffer between the first buffer and the consumer. Thus the producer will have to wait only when both buffers are full. This method is pictured in Fig. 4.3 and the CSP description shown in Fig. 4.4. This simple and elegant solution has the feature that each value passes through both of the buffers.

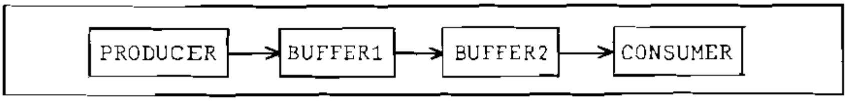


Figure 4.3 Double Buffer

```
[  PRODUCER:: i:integer;
              i:=1;
              *[i≤10 → BUFFER1!i; i:=i+1 ]

|| BUFFER1::  slot:integer; *[PRODUCER?slot → BUFFER2!slot ]

|| BUFFER2::  slot:integer; *[BUFFER1?slot → CONSUMER!slot ]

|| CONSUMER:: x:integer; *[BUFFER2?x → .... ]

]
```

Figure 4.4 A CSP solution for a double buffer.

The second method is to send alternate values to buffer1 and buffer2. A picture of this method is shown in Fig. 4.5 and a CSP description in Fig. 4.6. This solution obviates the necessity of passing each value through each of the buffers. However, it requires a complex alternative command to send alternate values to buffer1 and buffer2. This command also ensures the correct termination of the producer process irrespective of whether an odd or an even number of values is produced.

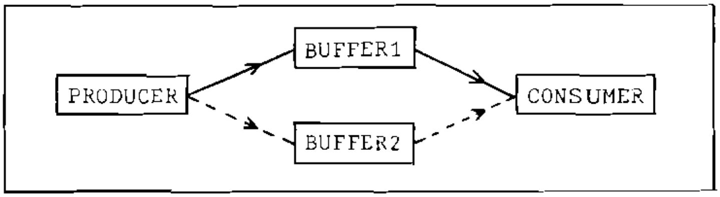


Figure 4.5 Double Buffer.

```
[  PRODUCER:: switch,i:integer;
              i:=1; switch:=1;
              *[i≤10 → [ switch=1 → BUFFER1!i; switch:=2
                         □switch=2 → BUFFER2!i; switch:=1
                        ];
                        i:=i+1
               ]

|| BUFFER1::  slot:integer; *[PRODUCER?slot → CONSUMER!slot ]

|| BUFFER2::  slot:integer; *[PRODUCER?slot → CONSUMER!slot ]

|| CONSUMER:: switch,x:integer;
              switch:=1;
              *[ switch=1; BUFFER1?x → switch:=2; ....
                 □switch=2; BUFFER2?x → switch:=1; ....
               ]
]
```

Figure 4.6 A CSP solution for a double buffer.

4.4 Producer/Consumer Problem

We shall now describe the well-known Producer/Consumer problem, leaving its solution (expressed in CSP) until the next section. This section may be skipped without loss of continuity, if the reader is already familiar with the problem.

For simplicity, we shall first illustrate the bounded buffer solution to this problem with an idealized hamburger stand where the cook produces only <u>one</u> hamburger at a time. Once cooked, he places it sequentially in one of K slots of the warmer (a K-slot buffer) ready for a customer. This is shown in Fig. 4.7. A customer can order only one hamburger at a time, but he may repeat his order. The problem is how to coordinate the cook and his customers to ensure that:

1) The cook does not prepare more than k hamburgers before any customer buys one (called <u>overflow</u>);

2) The customer does not get ahead of the cook (called <u>underflow</u>.)

The Producer/Consumer problem as it arises in the area of computer operating systems may have the following interpretation. The producer process may represent an input operation; it is placing the data into a buffer (of k records), one record at a time. The consumer process, perhaps representing the evaluation of some mathematical expressions, fetches one record at a time from the buffer.

Although we will not discuss the classical solution using the p-v semaphores by Dijkstra[6], it suffices to say that, in that solution, only <u>two</u> processes (i.e., a producer process and a consumer process) are working in parallel. In contrast, in the solution suggested by Hoare and presented in the next section, the buffering action is described as a third process which uses a buffer of K-slots. In other words, <u>three</u> processes are working in parallel; they are the producer, the consumer, and the buffer. A buffer of a finite number of slots is known as a <u>bounded</u> buffer.

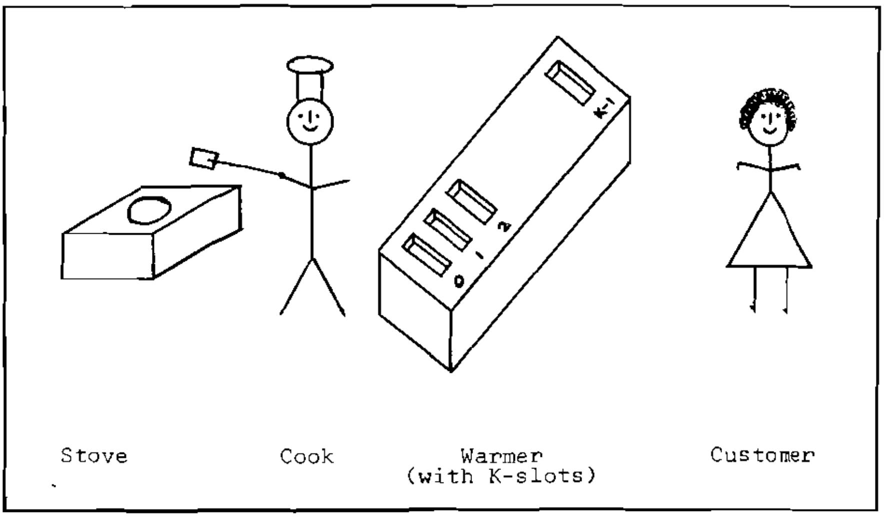


Figure 4.7 A Hamburger Stand to Illustrate the bounded-buffer problem.

4.5 Bounded Buffers

This section presents a solution, written in CSP, for the consumer/producer problem. Three processes are used; They are the consumer, the producer, and the buffer. The last process, acting as a K-slot buffer, is shown in Fig. 4.8.

```
BUFFERACTION::
buffer:array of K slots;
nrmade,nrbought,order:integer;
temp:slot;

nrmade:=0;
nrbought:=0;

comment At all times we ensure that:
        0 ≤ nrbought ≤ nrmade ≤ nrbought+K;

*[ nrmade < nrbought+K; producer ? temp →
                        buffer(nrmade mod K) := temp;
                        nrmade := nrmade+1
        step 2(a)

 □ nrbought<nrmade;consumer ? order →
                        consumer ! buffer(nrbought mod K);
                        nrbought := nrbought+1
 ]      step 2(b)
```

Figure 4.8 Process to describe the buffering action

This process contains only two basic steps:

Step 1 Initialize the number of hamburgers made(nrmade) to zero and that bought (nrbought) to zero.

Step 2(a) Check to see if the following 'no-overflow' condition:

$$nrmade < nrbought+K$$

is satisfied. If so we then check to see whether the process producer has cooked a hamburger. If so, it will be placed in slot number J where J = nrmade mod K and nrmade is incremented by 1.

Or

Step 2(b) Check to see if the following 'no underflow' condition:

$$nrbought < nrmade$$

is satisfied. If so, we then check to see whether the consumer has ordered a hamburger. If he has, then the customer gets one from the slot number I where I = nrbought <u>mod</u> K and nrbought is incremented by 1.

If both of the following situations occur:

1) The process producer terminates (which causes the input guard producer ? temp to fail); and
2) The process consumer terminates (which causes the input guard consumer ? order to fail)

then the repetitive command in Fig 4.8 will terminate, after which the process BUFFERACTION will also terminate.
Observe that BUFFERACTION will also terminate if the producer has terminated and the buffer is empty, or, if the consumer has terminated and the buffer is full.

4.5.1 Hand trace for a 3-slot buffer

To be more specific, let us take K = 3 and study the details shown in Table 4.2 (on page 35).

At time t0, step 1 yields the following initialization:

nrmade := 0 and nrbought := 0.

At t1 the buffer is empty thus the only option possible is for the producer to make a hamburger and put it in slot0 (step2(a)).

At t2, t3, t4, t5 and t7 there are hamburgers available in the buffer and the buffer is not full. It is thus possible either for the cook to put another hamburger in the buffer or for the customer to buy one. Which of these actions take place depends on:

1) Whether the cook <u>only</u> is ready to put one into the buffer (producer ? temp), in which case he does; or
2) Whether <u>only</u> a customer wishes to buy one, in which case he does; or
3) Whether,<u>simultaneously</u>, both the cook is ready to put one into the buffer and the customer wishes to buy one. In this case it is undetermined who gets in first although both will eventually succeed.

In this hand-trace the cook puts one in at t2, t4, t5 and t7 while the customer buys one at t3.

At t6 and t8 the buffer is full and the only action possible is for the customer to buy a hamburger. The cook is prevented from putting another one in because the condition nrmade < nrbought + K is not satisfied.

Time	Cook	nrmade	nrbought	Customer	Buffer Configuration
t0		0	0		slot 0 1 2 □□□
t1	Make 1 [O.K. as 0< 3] Place in slot 0	1	0		full ▨□□
t2	Make1 [O.K. as 1< 3] Place in slot 1	2	0		▨▨□
t3		2	1	Take one from slot 0 [O.K. as 0< 2]	□▨□
t4	Make 1 [O.K. as 2< 1+3] Place in slot 2	3	1		□▨▨
t5	Make 1 [O.K. as 3< 1+3] Place in slot 0	4	1		▨▨▨
t6		4	2	Take one from slot 1 [O.K. as 1< 4]	▨□▨
t7	Make 1 [O.K. as 4< 2+3] Place in slot 1	5	2		▨▨▨
t8	Must wait as [5≮2+3]				

Table 4.2 Parallel trace for the 3-slot buffer.

4.5.2 Time-history of some I/O commands.

We conclude this chapter by presenting a time-history of the three communicating processes: producer, buffer, and consumer, described in Fig. 4.2. It serves to focus on some fine points in the model for synchronization and buffering.

In Fig. 4.9 each t shown on the vertical axis indicates the time when either an I/O command begins to try to execute or when it succeeds. For example, at time t1 the value 1 (from the producer) is 'assigned to' the variable slot (in the buffer) without any delay. Immediately after this, at t2, the value in slot is transmitted to the variable x (in the consumer) without delay.

At t3 the value 2 is sent from the producer to the buffer without delay. However, at t4 when the buffer tries to pass this value to the consumer it is not able to do so. and has to wait until t6 when the consumer is ready to accept it.

At t5 the producer is ready to send the value 3 to the buffer but has to wait until t7 when the buffer is able to accept this value. At t8 the buffer is ready to send this value to the consumer but has to wait until t9 when consumer is ready to accept this value. Etcetera.

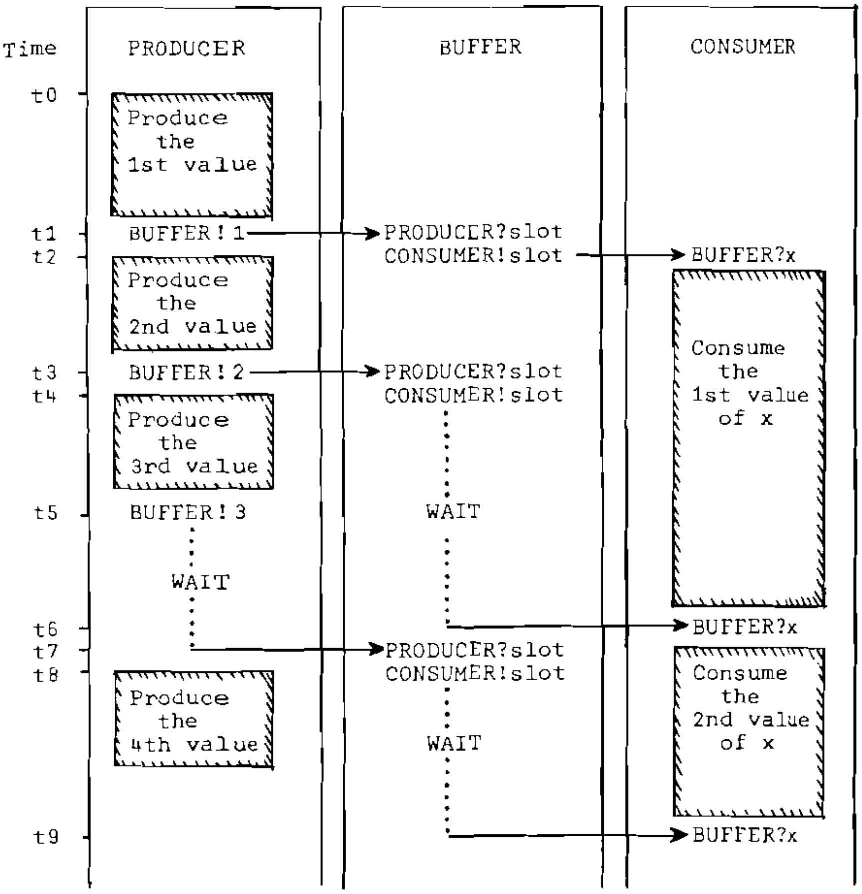


Figure 4.9 Time-history of three communicating processes with the shaded areas indicating computations.

CHAPTER 5

ARRAY OF PROCESSES

When a number of similar processes is involved, it is convenient to specify them as an array of processes. In this chapter this concept is illustrated with several examples. One of the examples serves to show how each process in such an array is used to represent a set of data items. The factorial and prime number problems are also studied in depth.

5.1 Subscripted Process Names

Thus far a process name is used to identify only one process. It is possible to use an identifier (as used for arrays) to name a group of processes. Thus

```
name(k:1..n)::command list
```

declares a one dimensional array of n processes: with names name(1), name(2), .. ,name(n). In long hand they are:

```
   name(1)::command list
|| name(2)::command list
|| name(3)::command list
        .
        .
|| name(n)::command list
```

where each command list may involve the index k which ranges between 1 and n. We illustrate this concept by an example. Given a character string, count the number of occurrences of the characters 'u','v' and 'w' respectively.

A possible solution in CSP is shown in Fig. 5.1(a). The parallel command consists of five processes named g,x,y,z and p. The process g inputs the characters from the typewriter one at a time. Each character is then sent to the three processes x,y and z which serve to count the number of the characters 'u','v' and 'w' respectively. In the parallel command, the termination of the process g causes the termination of the processes x,y and z which in turn causes the termination of the process p.

In Fig. 5.1(a) the processes x,y and z have similar command lists. To take advantage of this similarity, the CSP representation in Fig. 5.1(a) can be rewritten in the form shown in Fig. 5.1(b). Two major changes are made. First, the process names x,y,z have been replaced by s(1),s(2) and s(3). Second, the subscripted variables char(1) through char(3) are used to hold the characters 'u','v' and 'w'. The scope of this array extends throughout the parallel command.

Fig. 5.1(c) shows a shorthand version of that in Fig. 5.1(b). s(k:1..3) stands for the three process names s(1),s(2) and s(3) and k is known as the bound variable. Except for the indices, the same command list is used in all three processes. The output commands s(1)!c, s(2)!c and s(3)!c have been given the process names m(1), m(2) and m(3) respectively. This labelling allows us to rewrite the command [s(1)!c || s(2)!c || s(3)!c] as [m(j:1..3):: s(j)!c].

```
[   g:: c:character;
        *[typewriter?c → [ x!c || y!c || z!c ]]

  ||x:: c:character; t:integer; t:=0;
        *[g?c → [ c='u' → t:=t+1
                 □c≠'u' → skip]
          ];
        p!('u',t)

  ||y:: c:character; t:integer; t:=0;
        *[g?c → [ c='v' → t:=t+1
                 □c≠'v' → skip]
          ];
        p!('v',t)

  ||z:: c:character; t:integer; t:=0;
        *[g?c → [ c='w' → t:=t+1
                 □c≠'w' → skip]
          ];
        p!('w',t)

  ||p:: c:character; n:integer;
        *[ x?(c,n) → lineprinter!(c,n)
          □y?(c,n) → lineprinter!(c,n)
          □z?(c,n) → lineprinter!(c,n)
          ]
]
```

Figure 5.1(a) Solution without subscripted process names and array elements.

```
    char:(1..3)character;
    char(1):='u'; char(2):='v'; char(3):='w';

      [  g:: c:character;
            *[typewriter?c → [ s(1)!c || s(2)!c || s(3)!c ]]

       ||s(1)::  c:character; t:integer; t:=0;
                 *[g?c → [ c=char(1) → t:=t+1
                          □c≠char(1) → skip]
                  ];
                 p!(char(1),t)

       ||s(2)::  c:character; t:integer; t:=0;
                 *[g?c → [ c=char(2) → t:=t+1
                          □c≠char(2) → skip]
                  ];
                 p!(char(2),t)

       ||s(3)::  c:character; t:integer; t:=0;
                 *[g?c → [ c=char(3) → t:=t+1
                          □c≠char(3) → skip]
                  ];
                 p!(char(3),t)

       ||p:: c:character; n:integer;
             *[  s(1)?(c,n) → lineprinter!(c,n)
               □s(2)?(c,n) → lineprinter!(c,n)
               □s(3)?(c,n) → lineprinter!(c,n)
              ]
      ]
```

Figure 5.1(b) Solution in which 3 similar processes have subscripted process names and array elements.

```
      char:(1..3)character;
      char(1):='u'; char(2):='v'; char(3):='w';
       [g:: c:character;
            *[typewriter?c → [m(j:1..3)::s(j)!c]]

        ||s(k:1..3):: c:character; t:integer; t:=0;
                      *[m(k)?c → [ c=char(k) → t:=t+1
                                  □c≠char(k) → skip]
                       ];
                      p!(char(k),t)

        ||p:: c:character; n:integer;
              *[(j:1..3)s(j)?(c,n) → lineprinter!(c,n)]
       ]
```

(c) Short form of solution given in Fig. 5.1(b).

Figure 5.1 Programs that count the number of characters 'u', 'v','w' in a character string.

5.2 Bounded Buffer Using an Array of Processes

In this section we use an array of processes to implement the bounded buffer which was introduced in chapter 4.

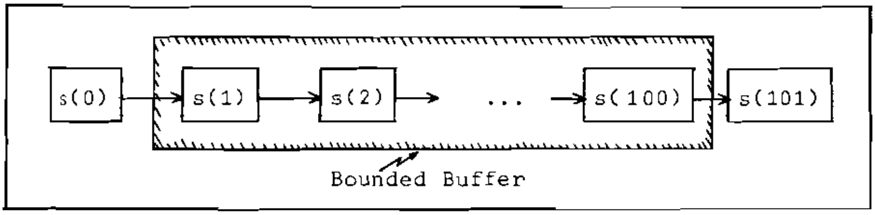


Figure 5.2 The process s(0) outputs integers one at a time. The processes s(1),s(2), ... ,s(100) have the effect of a bounded buffer. The process s(101) computes the sum of these integers and prints it.

As shown in Fig. 5.2 we input integers one at a time from a producer process s(0). Each integer in turn is passed through all the processes of the bounded buffer: s(1),s(2), ... ,s(100). For $1 \le i \le 100$, the integer in the process s(i) will be passed to the process s(i+1) only when the latter is ready to receive this integer. The process s(101) (consumer) is used to compute and print the sum of all the incoming integers. The CSP representation is shown in Fig. 5.3.

```
[ s(0)::            p:integer;
                    *[ cardreader ? p → s(1) ! p ]

 ||s(i:1..100):: q:integer;
                    *[ s(i-1) ? q → s(i+1) ! q ]

 ||s(101)::         r,sum:integer;sum:=0;
                    *[ s(100) ? r → sum:=sum+r ];
                    lineprinter ! sum
]
```

Figure 5.3 The above CSP representation finds the sum of a set of the integers and prints this sum.

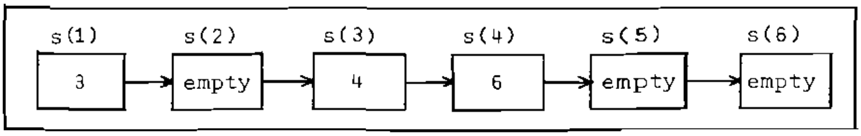
We note that:

1) The processes s(1) to s(100) all serve the same purpose: to push an integer away from the producer towards the consumer. As shown in Fig. 5.3, these processes have identical command lists, varying only in the value of i.
2) Each of the processes s(1) to s(100) represents a slot in the bounded buffer. If the process s(1) cannot accept a new integer, the process s(0) must wait until the process s(1) is ready. Similarly, the process s(101) cannot receive an integer unless the process s(100) is ready to send it. The same is true for any s(i) and s(i+1).

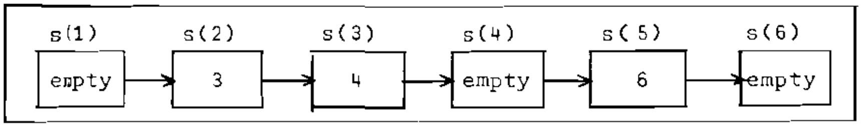
3) For the sake of simplicity, let us take a 'snapshot' of the six buffer slot s(1) through s(6) as shown in Fig. 5.4(a). All values used in this intermediate state are arbitrarily assumed. The details of these six processes, working in parallel, are as follows:

1. s(1) is ready to output the value 3 to s(2) and s(2), being empty, is ready to input this value from s(1).
2. s(2) is empty and, as a result, cannot transmit any result value to s(3) and s(3), not being empty, cannot accept any input from s(2).
3. s(3) is ready to output the value 4 to s(4) but s(4), still holding the value 6, cannot accept any input from s(3).
4. s(4) is ready to output the value 6 to s(5) and s(5), being empty, is ready to input this value from s(4).
5. s(5) is empty and, as a result, cannot transmit any value to s(6) but s(6), being empty, is ready to input from s(5).

Fig. 5.4(b) gives a snapshot of the 6-slot buffer after both the possible communications have been completed. Gradually, all the integers in any buffer are pushed to the right.



(a) An intermediate state of the 6-slot buffer.



(b) The state of the above 6-slot buffer after the two possible communications have taken place.

Figure 5.4 Communications between slots of the bounded buffer.

5.3 Case Study 1: Factorial Using Recursion

This section deals with a study of a CSP solution for the factorial function:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n*\text{factorial}(n-1) & \text{if } n>0 \text{ and } n \le \text{limit}. \end{cases}$$

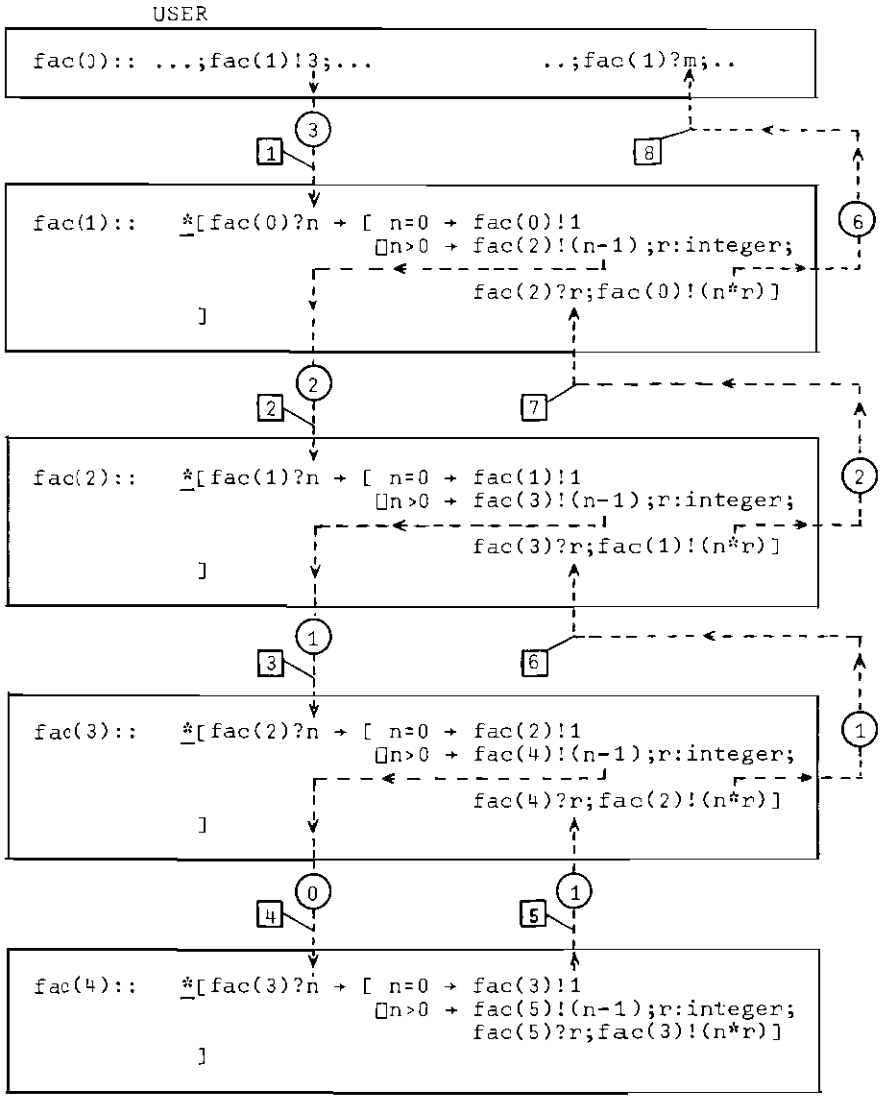
Hoare's solution [14] is reproduced in Fig. 5.5.

```
[ fac( i:1..limit)::
     *[ n:integer; fac(i-1) ? n →
                      [ n=0 → fac(i-1)!1
                       □n>0 → fac(i+1)!(n-1); r:integer;
                              fac(i+1)?r; fac(i-1)!(n*r)
                      ]
      ]

  || fac(0)::USER
]
```

Figure 5.5 Solution to the factorial function.

As the solution shows, each process fac(i) of the array inputs the value of n from its predecessor (process fac(i-1)) and outputs the value of factorial(n) back to its predecessor (process fac(i-1)). If n is not equal to zero, it requires the assistance of its successor (process fac(i+1)) to compute the value of factorial (n-1). fac(0) is the user program which initiates the calculation and obtains the final result.

For the purpose of illustration, we take n=3. The interrelationship of the five processes involved in the calculation is depicted in Fig. 5.6. The arrowed lines show the communication between two processes; and the trans-



```
fac(0):: ...;fac(1)!3;...          ..;fac(1)?m;..

fac(1)::   *[fac(0)?n → [ n=0 → fac(0)!1
                          □n>0 → fac(2)!(n-1);r:integer;
                                 fac(2)?r;fac(0)!(n*r)]
           ]

fac(2)::   *[fac(1)?n → [ n=0 → fac(1)!1
                          □n>0 → fac(3)!(n-1);r:integer;
                                 fac(3)?r;fac(1)!(n*r)]
           ]

fac(3)::   *[fac(2)?n → [ n=0 → fac(2)!1
                          □n>0 → fac(4)!(n-1);r:integer;
                                 fac(4)?r;fac(2)!(n*r)]
           ]

fac(4)::   *[fac(3)?n → [ n=0 → fac(3)!1
                          □n>0 → fac(5)!(n-1);r:integer;
                                 fac(5)?r;fac(3)!(n*r)]
           ]
```

Fig.5.6 The five processes involved in evaluating factorial(3).
The arrowed lines show the communication pattern.

mitted integers are shown in circles. The step number, enclosed in a box, shows the order in which the communication takes place between any two processes. There are in total 8 steps involved.

5.4 Case Study 2: Generation of Prime Numbers

In this section we examine in detail a CSP solution to the well-known problem of finding prime numbers by Eratosthene's sieve method [9,11]. For the sake of simplicity, we shall only consider the sequence of integers from 2 to 25 :

s1: 2,3,4,...,25.

For the uninitiated, we shall illustrate below in four steps how all the prime numbers in s1 can be obtained.

1. Take the first integer (i.e. 2, a prime number). Eliminate all its multiples from the sequence s1. We now obtain a new sequence:

   s2: 3,5,7,9,11,13,15,17,19,21,23,25.

   and note that the first number eliminated by 2 is 2*2=4.
2. Similarly, take the first integer (i.e. 3, a prime number) in the sequence s2. Eliminate all its multiples to produce a new sequence:

   s3: 5,7,11,13,17,19,23,25.

   We note that the first number eliminated by 3 is 3*3=9, (because 3*2 has already been eliminated).

3. Again, take the first integer in the sequence s3 (i.e. 5, a prime number). Eliminate all its multiples to produce the sequence:

   s4: 7,11,13,17,19,23.

   We note that the first number eliminated by 5 is 5*5=25.
4. Finally, take the first number in the sequence s4 (i.e. 7, a prime number). We note that the first number to be eliminated would be 7*7=49 (because all the earlier multiples of 7 must already have been eliminated). As a result no elimination will be made. The remaining integers:

   11,13,17,19,23

   are all primes. In summary, the primes in the sequence are as follows:

   2,3,5,7,11,13,17,19 and 23.

5.4.1 CSP Solution for Primes Using Eratosthenes Sieve

The CSP solution to the sieve problem (for integers from 2 to 25) is given in Fig. 5.7. There are 8 processes, namely SIEVE(0), SIEVE(1) to SIEVE(5), SIEVE(6) and print. The purpose of SIEVE(0) is threefold. The first purpose is to print the prime number 2 (from our previous knowledge). The second is to eliminate all the even numbers. The third is to pass all the odd numbers to SIEVE(1).

The array of 5 processes SIEVE(i:1..5) is rather involved. In Fig. 5.8 is listed each command used and its interpretation. Several important points are not self-

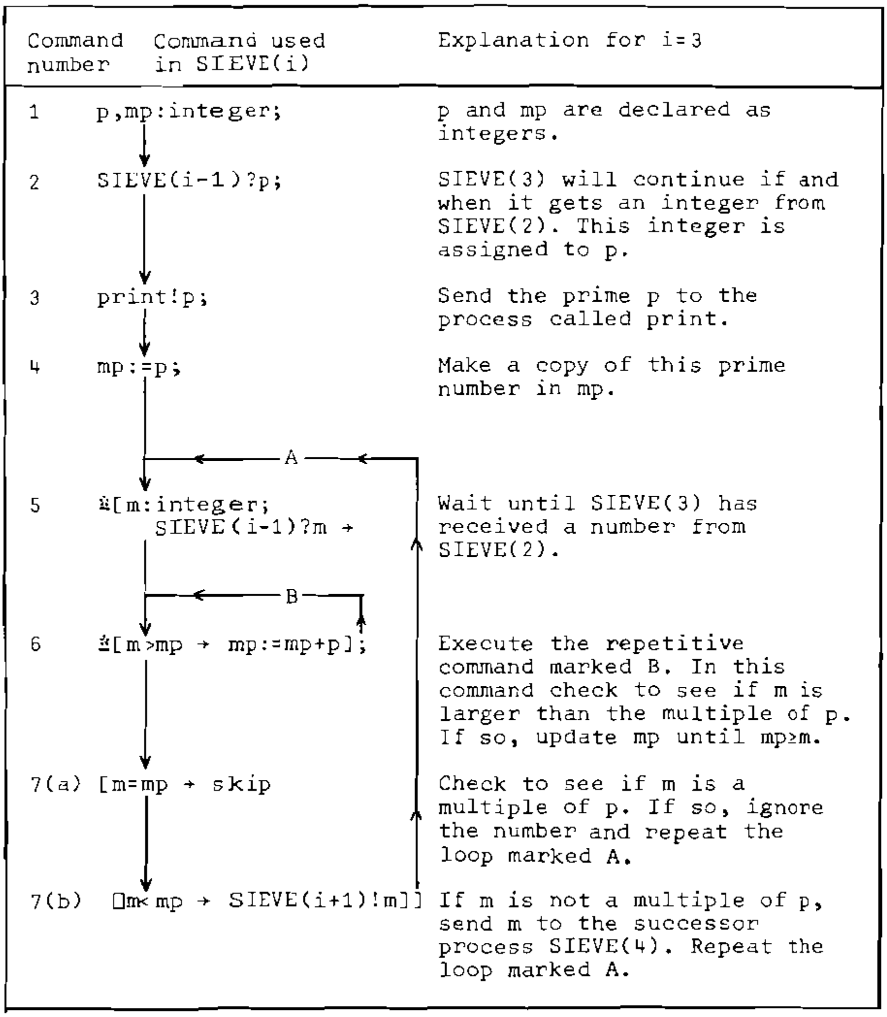
```
[ SIEVE(i:1..5)::
        p,mp:integer;
        SIEVE(i-1) ? p;
        print  ! p;
        mp:=p ; comment mp is a multiple of p;

        *[ m:integer; SIEVE(i-1) ? m →
                                          *[ m>mp → mp:=mp+p ];

                                           [ m=mp → skip
                                            □m<mp → SIEVE(i+1) ! m
                                           ]
         ]

 || SIEVE(0)::print!2; n:integer; n:=3;
              *[ n≤25 → SIEVE(1) ! n; n:=n+2 ]

 || SIEVE(6)::*[ n:integer; SIEVE(5) ? n → print ! n ]

 || print::*[  (i:0..6)n:integer; SIEVE(i) ? n → ...   ]

]
```

Fig. 5.7 A CSP solution to generate and print in ascending order all primes less than 25[14]. Note there are 8 processes involved.

explanatory in Fig. 5.8. The first number received by a process, and placed in p (command 2), is a prime number and will be retained by that process. All the subsequent numbers received by the same process (command 5), will be passed on to the next process if they are not a multiple of the said prime p. For example, SIEVE(1) retains the first prime number 3 and passes on the remaining numbers 5,7,11,13,17,etc. The numbers 9,15,etc. are multiples of 3. As such, they are eliminated by SIEVE(1) (see Fig. 5.9).

Similarly SIEVE(2) retains the second prime 5 and passes on the remaining numbers 7,11,13,17,etc. Moreover, the only input number to be eliminated is 25. This is also shown in Fig. 5.9.

The question arises as to why 6 SIEVE processes, namely SIEVE(0) to SIEVE(5) are used in Fig. 5.7. The reason is in this specific example (where n=25) we only need 3 processes i.e., SIEVE(0),SIEVE(1), and SIEVE(2) as explained at the beginning of this section and in Fig. 5.9. However to use more will do no harm. When n is large, it is convenient and safe to use $\sqrt{n}$+1 SIEVE processes as follows: SIEVE(0),SIEVE(1),...,SIEVE($\sqrt{n}$). This rule of thumb is used in Fig. 5.7.



Command number	Command used in SIEVE(i)	Explanation for i=3
1	p,mp:integer;	p and mp are declared as integers.
2	SIEVE(i-1)?p;	SIEVE(3) will continue if and when it gets an integer from SIEVE(2). This integer is assigned to p.
3	print!p;	Send the prime p to the process called print.
4	mp:=p;	Make a copy of this prime number in mp.
5	*[m:integer; SIEVE(i-1)?m →	Wait until SIEVE(3) has received a number from SIEVE(2).
6	*[m>mp → mp:=mp+p];	Execute the repetitive command marked B. In this command check to see if m is larger than the multiple of p. If so, update mp until mp≥m.
7(a)	[m=mp → skip	Check to see if m is a multiple of p. If so, ignore the number and repeat the loop marked A.
7(b)	□m<mp → SIEVE(i+1)!m]]	If m is not a multiple of p, send m to the successor process SIEVE(4). Repeat the loop marked A.

Fig.5.8 Explanation of each command used in the SIEVE(i) where 1≤i≤5. In this example i is taken as 3.

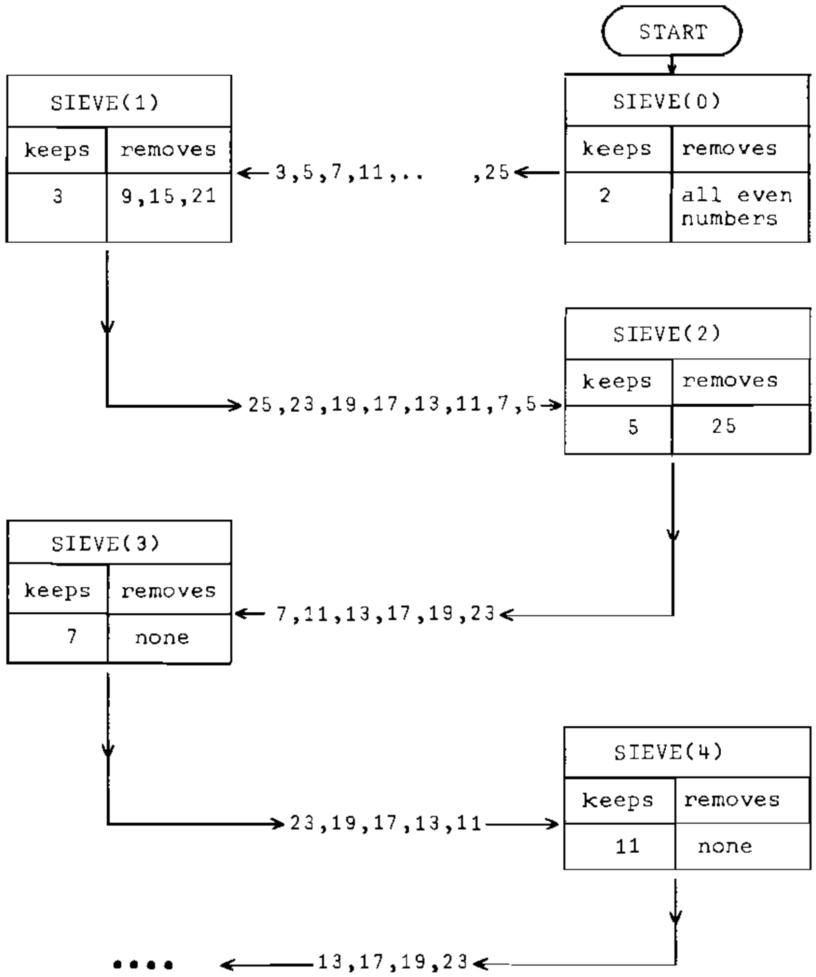


Fig.5.9 With each process is associated a prime number. Each process eliminates those numbers which are a multiple of its prime. The diagram shows the range of numbers $2 \le n \le 25$.

CHAPTER 6

CONCLUDING REMARKS

The examples of parallel processes in this report are, for the most part, familiar programming problems that have been recast in a somewhat different way using parallel composition, input/output primitives and guarded commands. the use of the CSP language conveniently leads to the rapid and clear development of complex parallel processes.

The CSP language has already been used in operating systems design [13] and simulation [17]. Among the areas CSP could be used are Numerical Analysis, Process Control and On-line system design.

The implementation aspect of CSP has deliberately not been discussed in the main body of this report because research is still in progress [10]. The overheads involved in creating processes in CSP is of particular interest and importance.

Many schemes have been proposed to implement synchronization. They include the events of PL/I [16]; the queues and monitors [12] in CONCURRENT PASCAL [1,2] and PASCALPLUS [3]; the interface module and signal in MODULA [21]. None of these seems, however, to be as conceptually simple as the synchronized I/O commands of CSP.

Various questions remain open when CSP is used. They include:

(a) The method used to describe a data structure. For example, the K-slot buffer, used in Fig. 4.8, represents a data structure. How does this representation, using one process for the entire buffer, compare with K processes each representing a slot as used in Fig. 5.2?

(b) What is the overhead associated with each of the representations mentioned in (a)

(c) Should the repetitive command have a specific terminator i.e.

```
*[ i≤10 → total:=total+i;i:=i+1
  []i>10 → exit from the loop
 ]
```

ACKNOWLEDGEMENTS

The authors wish to thank the academic members of the Programming Research Group, Oxford University, Professor C. A. R. Hoare and Mr. J. Stoy, for welcoming them for their sabbatical leave and for providing a stimulating environment to work in.

Specifically we thank:

Professor Hoare for providing his assistance and advice. He has patiently read the complete draft twice and offered valuable suggestions and detailed criticism.

Mr. C. Holt for being directly helpful in many hours of constructive discussions, and for illuminating various difficult points. He has read and criticized the complete draft.

Professor J. D. Lipson of University of Toronto for his perceptive readings of the manuscript and many helpful suggestions.

Mr. M. K. Harper for his critical readings of the final draft and for his help on the local computer system and Mr. A. M. R. Newman who provided us with valuable help in connection with the local text editor. Last but not least we thank Mrs. A. Bampton, who assisted with the preparation of many of the figures.

REFERENCES

1. Brinch Hansen, P.
   The Programming Language Concurrent Pascal.
   IEEE Trans. Soft. Eng. 1,2 (June 1975), 199-207.

2. Brinch Hansen, P.
   The Architecture of Concurrent Programs.
   Prentice-Hall (1977), 47-65.
   Concurrent Pascal is described and used to develop three non-trivial concurrent programs: a single-user operating system, a Job-Stream system and a real time scheduler.

3. Bustard, D.W.
   A user manual for PASCAL PLUS.
   Internal Publication, Department of Computer Science, The Queen's University of Belfast.

4. Coffman, E.G. and Denning, P.J.
   Operating Systems Theory.
   Prentice-Hall (1973), 7-9.
   A process is described and several references are given to definitions offered by other authors.

5. Colin, A.J.T.
   Introduction to Operating Systems.
   MacDonald/American Elsevier Computer Monographs no. 17. (1971), 10-17.
   Describes the difference between programs and processes.

6. Dijkstra, E.W.
   Co-operating Sequential Processes, in
   Programming Languages (ed. F. Genuys),
   Academic Press (1968), 43-112.

7. Dijkstra, E.W.
   The Structure of the T.H.E. Multiprogramming System.
   Comm.ACM, 11,5 (May 1968), 341-346.

8. Dijkstra,E.W.
   Guarded Commands, nondeterminancy, and formal derivation of programs.
   Comm.ACM, 18,8 (Aug. 1975), 453-457.
   This paper introduces the guarded command and is regarded as a prerequisite of the CSP paper.

9. Halberstam, H. and Richard, H.E.
The Sieve of Eratosthenes: Formulation of the General sieve problem in Sieve Methods.
Academic Press (1974), 12-36.
The first three pages provides an informal description of the sieve method. The remaining description is a rigorous mathematical treatment of the subject.

10. Hales, T.F. and Holt, C.M.
Verbal communication.

11. Hoare, C.A.R.
Notes on Structured Programming, in
Structured Programming,Academic Press(1972), 127-130.
The powerset data type is used to solve the prime number problem.

12. Hoare, C.A.R.
Monitors: an Operating System Structuring Concept.
Comm.ACM, 17,10 (Oct. 1974), 549-557.

13. Hoare, C.A.R. and McKeag, R.M.
Structure of an Operating System.
Unpublished.
This paper suggests that the structure of an operating system can be clearly expressed as a hierarchy of communicating sequential processes.

14. Hoare, C.A.R.
Communicating Sequential Processes.
Comm.ACM, 21,8 (Aug. 1978).

15. Hoare, C.A.R. and Holt, C.M.
Verbal communication.

16. IBM System/360 PL|I Reference Manual.
IBM Corp., C28-8201.0 .

17. Kaubisch, W.H. and Hoare, C.A.R.
Discrete Event Simulation Based on Communicating Sequential processes.
Unpublished.

18. Kuo, S.S.
Assembler language for Fortran, Cobol and PL|1 Programmers.
Addison-Wesley Company, Reading, Massachusetts (1974).

19. Naur, P. (ed.).
Report on Algorithmic Language ALGOL 60 .
Comm.ACM, 3,5 (May 1960), 299-314.

20. Madnick, S.E. and Donovan, J.J.
Operating Systems.
McGraw-Hill (1974), 247-248.
An excellent book on operating systems.

21. Wirtn, N.
Modula: a Programming Language for Modular Multiprogramming. Software - Practice and Experience, 7, 3—35, (Jan—Feb 1977).
Modula includes general multiprocessing facilities, namely processes, interface modules and signals.

APPENDIX

FORMAL SYNTAX OF CSP COMMANDS

In this appendix BNF notation is used to describe the CSP commands. The curly braces { } have been introduced into BNF to denote none or more repetitions of its contents.

Types of Commands

```
<command> ::= <simple command>|<structured command>
<simple command> ::= <null command>|<assignment command>
                    |<input command>|<output command>
<structured command> ::= <alternative command>
                        |<repetitive command>
                        |<parallel command>
<null command> ::= skip
<command list> ::={<declaration>;|<command>;} <command>
```

1 Alternative and Repetitive Commands

```
<repetitive command> ::= *<alternative command>
<alternative command> ::= [<guarded command>{□<guarded command>}]
<guarded command> ::= <guard> → <command list>
                   |(<range>{,<range>})<guard> → <command list>
<guard> ::= <guard list>|<guard list>;<input command>
           |<input command>
<guard list> ::= <guard element>{;<guard element>}
<guard element> ::= <boolean expression>|<declaration>
<range> ::= <bound variable>:<lower bound>..<upper bound>
<lower bound> ::= <integer constant>
<upper bound> ::= <integer constant>
```

2 Parallel Commands

```
<parallel command> ::= [<process>{||<process>}]
<process> ::= <process label><command list>
<process label> ::= <empty>|<identifier> ::
        |<identifier>(<label subscript>{,<label subscript>}) ::
<label subscript> ::= <integer constant>|<range>
<integer constant> ::= <numeral>|<bound variable>
<bound variable> ::= <identifier>
```

3 Assignment Commands.

```
<assignment command> ::= <target variable> := <expression>
<expression> ::= <simple expression>|<structured expression>
<structured expression> ::= <constructor>(<expression list>)
<constructor> ::= <identifier>|<empty>
<expression list> ::= <empty>|<expression>{,<expression>}
<target variable> ::= <simple variable>|<structured target>
<structured target> ::= <constructor>(<target variable list>)
<target variable list> ::= <empty>|<target variable>
                                        {,<target variable>}
```

4 Input and Output Command.

```
<input command> ::= <source>?<target variable>
<output command> ::= <destination>!<expression>
<source> ::= <process name>
<destination> ::= <process name>
<process name> ::= <identifier>|<identifier>(<subscripts>)
<subscripts> ::= <integer expression>{,<integer expression>}
```

(cont'd.)

PRG-9 Christopher Strachey and Joseph Stoy.
The Text of OSPub
(£3.50)

PRG-10 Christopher Strachey.
Varieties of Programming Language
(£0.50)

PRG-11 Christopher Strachey and Christopher P. Wadsworth.
Continuations: A Mathematical Semantics for Handling Full Jumps
(£0.60)

PRG-12 Peter Mosses.
The Mathematical Semantics of Algol 60
(£1.00)

PRG-13 Robert Milne.
The Formal Semantics of Computer Languages and their Implementations

Available as:
Technical Microfiche TCF-2.
A set of 6 Microfiche (£4.00)
or
Photocopy PRG-X13 (£10.00)

PRG-14 Shan S. Kuo, Michael H. Linck and Sohrab Saadat.
A Guide to Communicating Sequential Processes
(£1.00)