Implementations of the CSP Notation for Concurrent Systems

M. ELIZABETH C. HULL
Department of Computing Science, University of Ulster at Jordanstown, Shore Road, Newtownabbey, Co. Antrim, BT37 0QB, Northern Ireland

In his paper 'Communicating sequential processes', C. A. R. Hoare introduced a concept for the design of concurrent systems. His proposal was intended as a notation and little consideration was given by Hoare to subsequent implementation. Much discussion has taken place concerning associated problems, and recommendations for enhancement have been put forward.

This paper surveys the languages which have resulted from his proposal and discusses three of these languages.

Received February 1985

1. INTRODUCTION

In 'Communicating sequential processes' (CSP),[1] Hoare outlines an approach to program design by presenting a notation for processes executing concurrently, and communicating directly by means of input and output commands. However, the emphasis in the paper is very much on notation, and little consideration is given to subsequent implementation.

Previous papers have attempted to highlight some of the problem areas associated with CSP, and to propose solutions. This paper will deal with these issues in detail. However, before progressing to that stage, Section 2 gives an overview of the CSP notation. Section 3 considers the proposals for enhancement ranging from problems of nondeterminism and synchronisation and the introduction of communication channels to the consideration of output guards. Section 4 discusses the proposed implementations and how they have incorporated the proposals already discussed. Finally, Section 5 summarizes the major issues and comments on the way forward.

2. OVERVIEW OF CSP

CSP is a notation for programming concurrent systems. Central to it are input and output commands as basic primitives, and the concept of Dijkstra's guarded commands.[2] A CSP program is a fixed number of parallel processes, each of which consists of a series of sequential statements.

The input command takes the following format
⟨*source*⟩ ? ⟨*variable*⟩
where *source* is the name of a process and *variable* will store the result of the input, for example
producer ? item

If the source process is terminated the input fails, otherwise the communicating process waits until the source process is ready. In order for the input not to fail the value must match and is assigned to the variable.

The output command takes the following format
⟨*destination*⟩ ! ⟨*expression*⟩
where *destination* is the name of a process and *expression* gives the value to be output, for example
consumer ! item

If the destination process is terminated the output fails, otherwise the communicating process waits until the destination process is ready. The expression is then evaluated and its value transmitted. In order for the output not to fail the value must match the variable to which it is assigned.

The rules for matching structures are twofold:
(i) any simple variable will match any value of its own type;
(ii) if the variable is structured then
(*a*) the tags are the same, and
(*b*) the variable lists are the same length, and
(*c*) each variable must match its counterpart value.

In a CSP program, the processes of the parallel command execute concurrently with one another. The parallel command terminates when all its constituent processes have terminated.

To complete the overview, the use of guarded commands must be considered for controlling nondeterminism. The alternative and repetitive commands are based on this concept. A guard is an evaluation of a boolean expression. If the guard succeeds, i.e. if the boolean expression is true, then the statement is executed. If the boolean expression is false the guard fails. An extension of this is represented in CSP by the alternative command, which consists of a number of guarded commands, exactly one of which is to be executed, e.g.

```
 [⟨guard₁⟩ → ⟨statement₁⟩
□⟨guard₂⟩ → ⟨statement₂⟩
  .
  .
  .
  .
□⟨guardₙ⟩ → ⟨statementₙ⟩
]
```

A repetitive command is simply an alternative command repeatedly executed
 * [⟨*alternative command*⟩]

Two examples will be used throughout this paper to illustrate various points. The first is the well-known problem of the producer and consumer acting on a common buffer. The outline structure for this problem in CSP is as follows

```
producer_consumer::
[producer::  {produces item for input to buffer}
//
buffer::  {holds array of items}
//
consumer::  {receives item and uses it}
]
```

Three processes are required playing the role of the producer, the buffer and the consumer respectively.

The other example which will be used in this paper is that of a simple batch operating system. An absurdly simple view of such a system might be three processes executing in parallel. A *cardreader* process reads cards from the reader and sends the card image to the *execute* process which performs the computation. The *execute* process then passes the generated output line to the *lineprinter* process for output to the printer. The overall structure is as follows:

```
operating_system::
[cardreader::  {read card from reader and pass image to
                execute process}
//
execute::  {accepts images, performs computation and
            passes generated output to lineprinter
            process}
//
lineprinter::  {takes images from execute process and
                prints line on printer}
]
```

Communication between a pair of processes occurs by means of input/output commands. One process names the other from which it is prepared to receive input and the other names the first as its destination process for output

```
buffer::
  [....; consumer ! content(i) ....]
consumer::
  [.... buffer ? item; ....]
```

These operations take place simultaneously, having the effect of the input and output commands described above.

The facilities of CSP are demonstrated in the following CSP solutions to the examples being considered.

Example 1. Producers and consumers problem

```
producer_consumer::
  [producer::
    *[{generate item} → buffer ! item]
  //
  buffer::
    [content: (0..n-1) item;
      incount, outcount: integer;
      incount: = 0; outcount: = 0;
      *[incount < outcount+n; producer ? content
          (incount mod n) → incount: = incount+1
      []outcount < incount; consumer ? request() →
          consumer ! content(outcount mod n);
          outcount: = outcount+1
      ]
    ]
  //
  consumer::
    *[buffer ! request(); buffer ? item; {use item}]
  ]
```

Example 2. A simple operating system

```
operating_system::
  [cardreader::
    *[{read a card from reader} → execute ! card]
  //
  execute::
    *[cardreader ? card → {process card and generate
        line} lineprinter ! line
    ]
  //
  lineprinter::
    *[execute ? line → {print line on printer}]
  ]
```

3. PROPOSED ENHANCEMENTS

3.1 Communication Channels

The main features of CSP concerned with communication are as follows.

(i) Communication between processes is achieved by means of input/output commands. This is the central feature to the CSP notation.

(ii) To communicate, a process must name a destination process explicitly.

(iii) Processes do not share data, but do have their own local data structures.

Silberschatz[3] has proposed an alternative to the explicit naming approach. His proposal is that a process should name a **port** through which communication would take place. Each process would declare local port names:

send, *receive* : **port**;

The process which declares these ports is designated as the *owner* of the ports. A process which does not own a port may use it as follows:

use (*send*, *receive*);

In general a port has only one owner and several users.

If two processes wish to communicate they must be connected by a common port, the latter being owned by one of the processes. In Silberschatz's proposal input/output commands are used for communication as in CSP, the only difference being that port names are substituted for process names:

receive ? ⟨*target variable*⟩
send ! ⟨*expression*⟩

We can see this more clearly by considering the problem of the producers and consumers.

```
producer::
  use request;
  *[{generate item} → request ! item]
//
buffer::
  request : port;
  content : (0..n-1) item;
  incount, outcount : integer;
    incount: = 0; outcount: = 0;
  *[incount < outcount+n; request ? content
      (incount mod n) →
        incount: = incount+1
  [ ]outcount < incount; request ! content
      (outcount mod n) →
        outcount: = outcount+1
  ]
//
consumer::
  use request;
  *[request ? item → {use item}]
```

The process *buffer* is the owner of the channel *request*, which is used by the *producer* and *consumer* processes for communication.

As a further example we can consider the problem of the simple batch operating system.

```
cardreader::
  cardimage: port;
  *[{read a card from reader} → cardimage ! card]
//
execute::
  use cardimage, lineimage;
  *[cardimage ? card → {process card and generate line}
    lineimage ! line
   ]
//
lineprinter::
  lineimage : port;
  *[lineimage ? line → {print line on printer}]
```

In this case the *execute* process uses two ports *cardimage* and *lineimage* owned by the processes *cardreader* and *lineprinter* respectively.

The introduction of ports suggests that there is an alternative to the direct naming concept, i.e. that every input and output command must name its source and destination process explicitly. Instead processes communicate through ports which are named. This can be further extended to allow arrays of ports to be declared, distributed as necessary over several processes.

3.2 Synchronisation and nondeterminism

In a CSP program activities are synchronised by means of inter-process communication. Implicit synchronisation restricts parallelism, as the source process cannot send a communication until the destination process is ready to accept it, which in turn leads to a degradation in performance. To overcome this synchronisation, the need to buffer output from a process has been suggested.[4] This leads to the formation of input/output ports which have already been discussed in the previous section and facilitates asynchronous communication.

The constructs available in CSP for selection may be criticised for forcing the programmer to use non-deterministic constructs to express deterministic behaviour. That is, the programmer cannot control the selection in an alternative command but instead relies on the fairness of the implementation. It must be said, however, that the constructs are useful when the programmer cannot determine the order in which concurrent processes will be ready to execute or the programmer is not concerned about the order in which certain actions are performed.

It is recognised that the introduction of boolean variables into the guard is not the answer,[5] and suggestions have therefore been made to allow nondeterminism to take place only between equal priority guards.

3.3 Output guards

In CSP output commands may not appear in guards. Bernstein[5] and others have given reasons in opposition to this idea. Many examples can be given of processes which must execute an output statement, and since it cannot appear as part of a guard it must wait until a corresponding input statement is executed. This can lead to unnecessary delay for a process. Another example is the all too frequent, awkward structure of a 'double' output/input command. For example, in the producer consumer problem:

```
buffer ! request(); buffer ? item
```

where the *request*( ) signal informs the *buffer* process that the *consumer* process is ready to receive another *item*.
In the *buffer* process itself there is the following statement:

```
outcount < incount ; consumer ? request() →
    consumer ! content (outcount mod in);
    outcount: = outcount + 1
```

Neither it nor the previous statements would be necessary if the following was permitted:

```
outcount < incount ; consumer ! content
    (outcount mod in) → outcount: = outcount + 1
```

Thus the output guard avoids the need for the *request*( ) signal in the consumer process.

Buckley and Silberschatz[6] concur with this approach and propose an implementation for such a construct. They have identified four criteria which influence the efficiency of an implementation of an alternative command with input/output. These are:

(i) only a small number of processes should be involved in synchronising two processes which wish to communicate;

(ii) the information required by processes for a decision to be made about communication should be minimal;

(iii) there should be a time limit on how long two processes take to establish communication;

(iv) the message overhead between processes to establish communication should be small.

Buckley and Silberschatz present an algorithm supporting this argument.

4. THE IMPLEMENTATIONS

This section considers three separate implementations which have been directly influenced by CSP. It is not intended that the following sections should give a complete description of the languages, but rather an overview, with a clear explanation of how much has been influenced by CSP and, more importantly, how each has taken into account the proposed enhancements for implementation already discussed in the previous section. Two of the languages considered come from the academic community and the third is a commercially available language.

4.1 A communicating sequential process language

T. J. Roper and C. J. Barter from the University of Adelaide developed a language described in 'A communicating sequential process language and implementation' (COSPOL).[7] This language is strongly influenced by CSP and includes the parallel command, process communication by messages and the use of Dijkstra's guarded commands for controlling non-determinism. However, in the area of process communication this proposal differs significantly. Messages are selected for input, by a process, on their 'construction'. Most significantly, the sending process plays no part in the message reception. Indeed communication is built on the concept of a port, an input port being associated with more than one output port. Automatic buffering of messages is assumed rather than the synchronous communication in CSP.

A program consists of a parallel command which is a number of communicating sequential processes. The command list specifies sequential execution of its constituent commands and, apart from those associated with message passing, they take a similar form to those in CSP. The messages consist of a constructor applied to a set of slots. They are received by input commands, being specified by the notation of message construction. A process executing an output command need not wait until the message sent is received before continuing – a process outputs a message of construction to a process label. In this sense the concept of process naming is supported.

To illustrate the language COSPOL, consider a solution to the producers and consumers problem:

```
[producer_consumer::
  [producer::
    message send (item:char);
    *[{generate item_in} → buffer ! send (item ~ item_in)]
  //
  buffer::
    const n = 10;
    var in, out:integer;
    message send (item:char);
        request (from:ref);
        receive (item:char);
    in: = 0; out: = 0;
    *[in < out+n; ? send →
        content (in mod n): = send.item
    [ ]out < in; ? request →
      request.from ! receive (item ~ content(out mod n));
      out: = out+1
    ]
  //
  consumer::
    message receive(item : char);
        request (from : ref);
    *[buffer ! request( ); ? receive; {use receive.item}]
  ]
]
```

The communication interfaces between the *producer*, *buffer* and *consumer* process are defined by the process labels and the message constructions *send*, *request* and *receive* – the input commands *?send* and *?request* in the guards of the buffer process provide the two interfaces, and the message slots are used to pass value parameters. The destination for the *receive* message is dynamically determined, and hence the *buffer* process is independent of the processes with which it communicates.

In summary, the state of COSPOL is as follows.

(i) It has made some movement towards the ideas of port communication and the buffering of messages.

(ii) The alternative command provides the sole method for command selection.

(iii) No attempt has been made to introduce output commands in guards.

(iv) Following the conventions of Pascal, type and constant definitions have been introduced, together with the declaration of ordinary variables.

(v) Standard input and output processes represented by Pascal-like text files have been introduced.

The implementation was written in Pascal for the DEC VAX-11/780 running under VMS, and has proved a very satisfactory base on which to develop CSP-like programs.

4.2 CSP/80

CSP/80[8] was also developed by academics, who wished to develop a language based on CSP. Like COSPOL it differs significantly from its parent.

A program consists of a number of separately compiled processes which communicate with each other using unidirectional channels. A channel connects two ports, one in each process. Input and output statements reference a port which is typed. When such statements are executed the process is suspended until its partner process executes the corresponding command. Synchronisation of processes therefore takes place. As in CSP, CSP/80 has two non-deterministic constructs, the alternative and repetitive commands. One of the main differences is that CSP/80 supports the use of output commands as part of a guard, however only one of a pair of input/output commands may appear in a guard when the ports are connected by the same channel.

To illustrate the language CSP/80, consider a solution to the producers and consumers problem. This shows quite clearly the declaration of a port to be guarded, and that the linker can verify that only one end of such a channel is guarded.

```
process producer::
output char port send;
char ch;
*[{generate ch} → ! send = ch;
 ]
end process
process buffer::
guarded input char port send;
guarded output char port receive;
const n = 10;
int in;
int out;
char content[n];
in = out = 0;
*[in < out+n; ? content [in mod n] = send → in++;
 [ ]out < n; ! receive = content [out mod n] → out++;
 ]
end process
process consumer::
input char port receive;
char ch;
*[true → ? ch = receive; {use item}]
end process
/* linker instructions */
char channel from produce.send to buffer.send;
char channel from buffer.receive to consumer.receive;
```

In summary, the state of CSP/80 is as follows.

(i) It has introduced the idea of ports proposed by Silberschatz.

(ii) It supports integer, character scalars and array types, and this is currently being extended to any type supported by the language C.

(iii) It is strongly typed, hence the introduction of ports.

(iv) It supports modularity and the development of libraries of processes.

(v) It has introduced the use of output commands in guards.

(vi) Alternative and repetitive commands are the two non-deterministic constructs.

The implementation was written in the C language for the DEC PDP-11/45 running under UNIX.

4.3 Occam

Occam[9] is the language developed by Inmos Ltd, based on the concepts of concurrency and communication, and designed for the professional programmer. It thus develops the single idea of CSP, that for concurrent systems one must define independent entities operating in parallel with communication between such entities. In Occam a process represents such an activity and a channel forms the basic communication link between two processes. This is an unbuffered structure which allows information to pass in one direction only – processes communicate with each other by sending and receiving messages via a channel. A sending process may have to hang up until a receiving process is ready and vice versa; that is, a receiving process can read from a channel which is full and a sending process can send a message when a channel is empty.

Apart from processes, the basic mechanisms are sequential, parallel and alternate statements together with the WHILE construction for looping. Deterministic choice is provided in Occam by means of the IF construction. Occam is untyped, a value being one word regardless of its meaning.

Occam is also intended as a design tool. Consider the example of a single operating system outlined earlier in this paper.

Starting with a diagram of the subsystems that form the operating system, the design appears to be a network of boxes, representing the functions connected by labelled lines, representing the interaction. A designer can map this directly into an Occam program:

```
OPERATING_SYSTEM
  PAR
    PROCESS-cardreader
    PROCESS-execute
    PROCESS-lineprinter
```

Next the meaning and form of the Occam processes can be defined in logical terms:

```
chan cardimage, lineimage:
par
  while true
    seq
      {read a card from cardreader}
      cardimage ! card
  while true
    seq
      cardimage ? card
      {process card and generate line}
      lineimage ! line
  while true
    seq
      lineimage ? line
      {print line on printer}
```

As a second example, consider the problem of the producers and consumers. The following Occam program results:

```
chan send, request, receive:
par
  while true
    var item:
    seq
      {generate item}
      send ! item
  while true
    def n = 10:
    var content[n], in, out:
    seq
      in: = 0
      out: = 0
      alt
        in < out+n & send ? content [in mod n]
          in: = in+1
        out < in & request ? any
          seq
            receive ! content [out mod n]
            out: = out+1
  while true
    var item:
    seq
      request ! any
      receive ? item
      {use item}
```

In summary, the state of Occam is as follows.

(i) Processes communicate via channels.

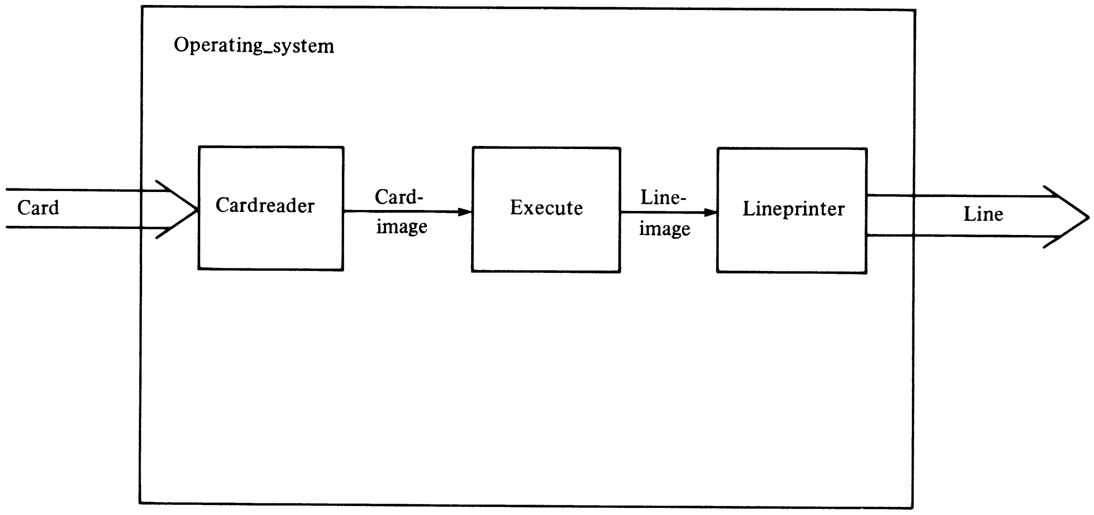

Figure 1

(ii) Communication is synchronous – it can only occur when both the input and output processes are ready.

(iii) Non-determinism is introduced by the **alt** command.

(iv) Deterministic choice is possible using the **if** construction.

(v) Output commands are not permitted in guards.

(vi) It is untyped – a value being one word regardless of its meaning.

The Occam Evaluation Kit was available under the UCSD P-system for such machines as the Apple, Sirius, IBM PC, VAX-11/780 and the LSI-11/23. The Occam Development System is currently available for the VAX-11/780 under VMS.

5. CONCLUSION

This paper has considered the major enhancements considered necessary to convert a notation proposal into an acceptable implementation language. As can be seen from the three languages studied, all the enhancements (in some form) have been utilised, but not all in the one language. The most striking feature must be the addition of ports (or channels) for process communication and the move away from synchronous communication to achieve an acceptable implementation. The concept of deterministic choice is one which one feels should be explicit in a programming language, and a certain movement in this direction has been recognised. The introduction of output guards has gained least favour.

The development of such implementations leads directly to proposals for methodologies to support the development of programs written in such languages. It is claimed that Occam is a language for design as well as implementation. However development tools, in general, are required to support such progress.

REFERENCES

1. C. A. R. Hoare, Communicating sequential processes. *CACM* **21** (8), 666–667 (1978).
2. E. W. Dijkstra, Guarded commands, nondeterminacy, and formal derivation of programs. *CACM* **18** (8), 453–457 (1975).
2. A. Silberschatz, Port directed communication. *The Computer Journal* **24** (1) 78–82 (1981).
4. R. E. Kieburtz and A. Silberschatz, Comments on 'communicating sequential processes'. *ACM TOPLAS* **1** (2), 218–225 (1979).
5. A. J. Bernstein, Output guards and nondeterminism in 'Communicating sequential processes'. *ACM TOPLAS* **2** (2), 234–238 (1980).
6. G. N. Buckley and A. Silberschatz, An effective implementation for the generalised input–output construct of CSP, *ACM TOPLAS* **5** (2), 223–235 (1983).
7. T. J. Roper and C. J. Barter, A communicating sequential process language and implementation, Software – Practice and Experience. **11**, 1215–1234 (1981).
8. M. Jazayeri *et al.* CSP/80: a language for communicating sequential processes. In *Proc. Fall IEEE COMPCON 80*, pp. 736–740. IEEE, New York (1980).
9. Inmos Ltd, *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, N.J. (1984).