PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

A COMPARISON OF TWO NOTATIONS FOR PROCESS COMMUNICATION

Jim Welsh*, Andrew Lister and Eric J. Salzman

Department of Computer Science
University of Queensland

ABSTRACT

This paper compares the mechanisms for process communication, synchronisation and non-determinism in recent language proposals by Hoare and Brinch Hansen, by both qualitative and quantitative analyses. A significant variation in effectiveness with program class is shown.

1. INTRODUCTION

Advances in hardware technology have made networks of loosely connected processors, each with its own local storage, an attractive economic possibility. In doing so, they have created new problems for programming-language designers. Parallel processing itself is not new, and languages such as Concurrent Pascal (Brinch Hansen 1975), Modula (Wirth 1977) and Pascal-Plus (Welsh and Bustard 1979) have allowed the description of parallel processes for some time. However these languages allow co-operation between processes by means of monitors, a concept developed in the early '70s by Brinch Hansen and Hoare (Brinch Hansen 1973, Hoare 1974). A monitor ensures well-ordered access to shared variables by the processes which share them, and is easily implemented when processes are executed by a single processor, or by multiple processors with access to a common store. The monitor concept is less natural, and its implementation is less obvious, on processors without common store, so language designers have sought alternative solutions.

A significant milestone has recently been passed with the publication, by the originators of the monitor, of two new language proposals which seek to overcome the problem. Hoare has suggested a set of language primitives for the description of *communicating sequential processes* (Hoare 1978), which is hereafter referred to as CSP. Brinch Hansen has outlined a language for *distributed processes* (Brinch Hansen 1978), hereafter referred to as DP. While the objectives of Hoare's proposal are somewhat broader than those of Brinch Hansen, the two proposals provide direct alternatives in the application area chosen for DP, i.e. real-time systems implemented

---

*on leave of absence from the Queen's University of Belfast

by networks of processors with distributed storage. It is interesting to examine their effectiveness in this area, as demonstrated by the example programs chosen by their authors.

Both proposals adopt the process as the fundamental notion in program construction, but differ significantly in their mechanisms for process communication, process synchronisation, and non-determinism within processes. Section 2 of this paper illustrates these differences, by comparing corresponding versions of two of the program examples given in the original papers, and makes a qualitative assessment of the significance of these differences in program construction. Comparative versions of the complete range of examples used in the original papers are given as appendices to this paper and these form the basis of a quantitative assessment presented in Section 3. While the basis of this quantitative assessment may not be universally accepted, an interesting correlation between the qualitative and quantitative findings is demonstrated.

The paper is concerned only with the mechanisms for process communication, synchronisation and non-determinism in the two proposals. In practice they also differ on data structures, on process structures and on process termination. Where such differences would complicate the quantitative comparison of the communication features, we have taken the liberty of eliminating them by adjustment of one language or the other. The changes made are in no way a comment on the language features involved and we apologise to their authors for making them.

2. A QUALITATIVE COMPARISON

2.1 Communication

Both CSP and DP allow processes to communicate only by explicit commands, but the forms chosen for these are somewhat different.

In CSP a process *X* outputs, or sends, information to a process *Y* by executing a command of the form

Y ! tag(values)

and process *Y* receives, or inputs, the information by executing a command of the form

X ? tag(variables)

Both the tags and the value and variable lists must correspond if communication is to succeed.

In DP, communication is accomplished by one process *X* say, executing a command of the form

call Y.P (values, result variables)

where process *Y* contains a procedure declaration of the form

proc P (value parameters # result parameters) ... body of P ...

The value parameters carry information from process *X* to process *Y*, the result parameters carry information back from *Y* to *X*.

One difference which is immediately apparent is that in CSP each process must name the other in order to communicate while in DP the process defining a procedure need not identify the processes which call it. Hoare argues that this is not a significant semantic difference, which Brinch Hansen confirms by indicating that the implementation of a DP process must make suitable provision for the calls on its procedures by other processes. Depending on how the network is connected and the storage economy required, this provision may involve identifying, or at least enumerating, the callers of each procedure. Thus CSP makes explicit what a DP implementation must deduce from each program.

The DP convention may not even be a significant user convenience, for two reasons:

(1) In some cases programming advantage is obtained from not having to name calling processes - the parallel array adder suggested as an exercise by Brinch Hansen is such a case - but in others a process may require other processes which call its procedures to pass their identity as parameters. Brinch Hansen's shortest-job-next scheduler (A4) is such a case. In the latter the programmer is forced to duplicate information which the implementation will deduce by other means, without any check on its equivalence. This is a first, if modest, symptom that the language design is at a higher level than the application requires, since the abstraction has removed information which the programmer is forced to recreate.

(2) As Brinch Hansen points out, a complete language (based on his DP proposal) should provide additional notation to limit the access rights of individual processes to the procedures of other processes. This is precisely what CSP's process naming achieves - each process defines precisely the processes with which it is intended to communicate, and an attempt by any other process to do so is detectable during compilation of the program.

Clearly CSP's input and output commands are lower level primitives than the composite procedure mechanism of DP. One might expect therefore greater flexibility in CSP but greater convenience (and error security) in DP. This convenience is apparent in program examples where no real parallelism is intended, such as Brinch Hansen's vending machine (A9). It is less obvious in programs where parallelism is significant, because of the additional synchronisation role which the communication mechanisms play there.

2.2 Synchronisation

In CSP the corresponding input and output commands by which two processes communicate are executed 'simultaneously', so whichever process reaches its command first must wait until the other process reaches the corresponding command. When the communication is completed successfully both processes continue in parallel.

In DP a process which calls a procedure of another process is held up until the requested execution of the procedure is complete. This involves waiting until the called process

(a) decides to execute the procedure on the caller's behalf, and

(b) does so.

Throughout this period the calling process cannot engage in any other activity.

The additional waiting (b) which is inherent in a DP procedure call complicates the programmer's task in situations where the action requested by a call can and should be carried out in parallel with further activity by the caller. Such a situation is well illustrated by Hoare's set of integers example, which in its original form is as follows:

```
S::
content : (0..99) integer ; size : integer ; size := 0;
*[ n : integer ; X ? has(n) → SEARCH ; X ! reply(i<size)
□ n : integer ; X ? insert(n) → SEARCH;
                                [ i<size → skip
                                □ i=size ; size<100 →
                                    content(size) := n ; size := size+1
]                               ]
where SEARCH ≡ i:integer ; i := 0 ; *[ i<size ; content(i) ≠ n → i := i+1]
```

The user process X sends a query *has* and then waits for a reply by means of two consecutive commands

$$....S\ !\ has(x)\ ;\ S\ ?\ reply(b)\ ....$$

but to request an insertion a single command

$$....S\ !\ insert(x)....$$

is all that is necessary. Once the process S accepts the input x the user process X continues in parallel with the insertion activity within S.

A comparable process in DP might be written as follows:

```
process S
    content : array [100] int ; size, i : int ;
    insertionrequested : boolean ; insertionvalue : int ;
    proc search (n : int)
         begin i := 1 ; do (i≤size) & (content[i]≠n) : i: i+1 end end

    proc has (n : int # answer : boolean)
         begin call S.search(n) ; answer := i ≤ size end

    proc insert (n : int)
         begin insertionrequested := true ; insertionvalue := n end

    begin
         size := 0 ;
         insertionrequested := false ;
         cycle insertionrequested :
                   call S.search(insertionvalue) ;
                   if i≤size : skip |
                      (i>size)&(size<100) :
                           size: = size+1 ; content[size]:=insertionvalue
                   end ;
                   insertionrequested :=false
         end
    end
```

Since a process calling the procedure *has* wishes to wait for the answer to its query the determination of this answer can be coded as the procedure body itself. However to allow the process calling *insert* to continue while the insertion is carried out, the procedure body is written simply to record that an insertion has been requested. When execution of this body is completed the calling process continues, and in parallel with this the infinite cycle in the body of process S is resumed, to detect that an insertion has been requested, carry out the insertion and reset the state variable *insertionrequested*.

Thus to achieve the required parallelism between insertions and the process requesting them, all of the code enclosed in boxes has to be introduced. In no sense can this code be dismissed as the useful redundancy of a higher level notation - it is an additional logical framework which the program writer must conceive and the program reader must unravel, and creates an additional area of potential error for either. It arises because DP's abstraction of process communication, the procedure call, includes a second phase of waiting which this particular application, the set insertion operation, does not require.

2.3 Non-determinism

In fact the DP process *S* is still incorrect. Because of the non-deterministic way in which processing switches between the cycle in the process body and the calls made on its procedures there is no

guarantee that the cycle will be resumed to carry out a requested insertion before a further call of *has* or *insert* is accepted. To guard against this possibility the *begin* at the start of each of the procedures must be replaced by

when not insertionrequested :

This all-too-easy programming error underlines a significant shortcoming of DP as a transparent programming language - the way in which non-determinism is incorporated.

Both CSP and DP recognise the need for non-determinism in programming processes which respond to unpredictably ordered external events. Both adopt Dijkstra's guarded command (Dijkstra 1976) as the means of expressing non-determinism, with trivial differences in syntax :

CSP	DP
[*guard* → *command*	*if guard : command*
□ *guard* → *command*	| *guard : command*
□	| .....
]	*end*
*[*guard* → *command*	*do guard : command*
□	| .....
]	*end*

To enable (possibly) non-deterministic waiting CSP allows an input command *X?t( )* to appear as a guard ; such a guard is true only when process *X* executes a corresponding output command, is false if process *X* has terminated, but otherwise implies waiting to determine the result.

Waiting in DP is expressed as *when* and *cycle* variants of the *if* and *do* commands, which imply waiting for a boolean expression guard to become true.

In CSP the guarded alternative and repetitive commands are the only source of non-determinism within a process. In DP however there is an additional non-determinism in the way in which processing switches between the process body and the external calls on its procedures. As in a conventional monitor, the points at which this switching may occur are imbedded as wait operations (i.e. *when* and *cycle* commands) within the procedures and process body itself, with no explicit structural representation of the non-determinism involved. Furthermore this non-determinism is not at the same level as that defined within the *if*, *do*, *when* and *cycle* commands themselves. Thus processing will remain within a *cycle* command in the process body as long as any of the alternatives of the cycle allow it to continue. Only when this is not so may a procedure call be started or resumed. However the process body has no priority to resume processing when a procedure call is completed or held up. As we have seen, other procedure calls may intervene and the programmer must take care of this possibility.

Both CSP and DP may be criticised for forcing the programmer to use non-deterministic constructs to express deterministic behaviour. However the familiar deterministic constructs *if..then..else* and *while..do* are obvious special cases of the corresponding guarded constructs in either language, and could easily be restored by trivial language extension or pre-processing. This is not so with the non-determinism between the procedures and the body of a DP process. To impose a deterministic sequence here the programmer must use guards composed of state variables which are reset at appropriate points in the code. In doing so he has no syntactic indication of the extent of the non-determinism involved, and the presence of additional explicitly non-deterministic constructs may complicate his task.

2.4 Scheduling

Hoare's set-of-integers example highlights the worst features of the communication/synchronisation mechanism adopted in DP but similar problems and logical overheads arise in example programs used in Brinch Hansen's own paper, such as the shortest-job-next scheduler (A4) and the sort array (A8).

Under what conditions do the disadvantages of DP's communication mechanism not apply? Since the semantics of the DP procedure call require that the calling process must wait, it seems ideal for implementing processes whose job is to make other processes wait, i.e. schedulers. From the examples given in the papers this generalisation must be qualified in two cases :

(1) If the scheduling decision for each type of request is expressible as a boolean expression whose value is determined by the sequence of requests already serviced, then the scheduler may be expressed as easily in CSP - by prefixing the input guard representing the request with the boolean expression. The resource scheduler (A3) and the readers and writers scheduler (A5) are such cases, though in practice the resource scheduler can be expressed even more simply in CSP.

(2) If the scheduling decisions are to be taken in parallel with resource/user activity wherever possible, then for the reason already illustrated by the set of integers example the structure used in the DP scheduler is equivalent to that required in CSP and there is little difference in the volume of program required by either language. The shortest-job-next scheduler (A4) is such a case.

One example in which DP does have a significant advantage is Brinch Hansen's alarm clock process, which is as follows :

```
process alarm
    time : int
    proc wait (interval : int)
        due : int
        begin
            due := time+interval
            when time = due : skip end
        end
proc tick ; time := time+1
time := 0
```

A comparative solution in CSP might be as follows, assuming user processes as shown:

```
[ User (i:1..n) :: ..... alarm!wait(t) ; alarm?wakeywakey()...

// alarm::
      time:integer ; time:=0;
      due:(1..n) integer ;
      i:integer ; i := 1 ; *[ i ≤ n → due(i):= -1 ; i := i+1 ];

      *[(i:1..n) interval:integer ; user(i)?wait(interval)→
                          due(i) := time+interval
       □(i:1..n) due(i)=time →
                          user(i)!wakeywakey() ; due(i) := -1
       □ realtimeclock?tick() →
                          time := time+1
]     ]
```

Both solutions assume that the tick step is sufficiently long for the alarm clock to service all necessary users between ticks.

This problem is particularly suited to DP for the following reasons:

(1) The DP procedure *wait* encapsulates the waiting requirement of each user process clearly and concisely, without reference to the other processes involved.

(2) The procedure *tick* produces the only changes necessary in the alarm clock environment, so no explicit cycle is required as the process body.

(3) The separate copies of the local variable *due*, which are created for the processes calling *wait*, provide an implicit data structure over exactly those processes with calls outstanding - in CSP an explicit array over all user processes is declared.

(4) In determining which processes may continue, the implementation of the DP alarm clock logic inspects only the *due* variables of those actually waiting; in the CSP version all elements of the *due* array are inspected and must therefore be initialised and reset to default value for non-waiting processes.

All four of these factors make the DP program easier to construct, easier to understand, and less prone to error than the CSP version. Advantages (3) and (4) are significant in that they are not illustrated elsewhere in the set of program examples considered. Note also that they are dependent on the procedural encapsulation of communication.

As far as efficiency is concerned points (3) and (4) deserve further consideration. The explicit array *due* in the CSP version is not a storage overhead since the DP implementation must set aside at least as much storage for the activation records as the user processes may require by all waiting concurrently. The reduction in computation (4) which the DP program achieves is an exploitation of the underlying set of suspended procedure executions which the DP implementation must maintain. A comparable efficiency could be achieved in CSP by explicitly creating a similar set of waiting users, but at additional programming effort. This efficiency could affect the feasibility of such a clock process in a real-time environment.

2.5 Classes of distributed process

In brief the alarm clock example epitomises the advantages of DP over CSP while the set of integers example epitomises its disadvantages. However factor (2) on the alarm clock above gives a significant clue to the conditions under which DP is effective.

The example DP processes considered in the evaluation may be sub-divided into three classes:

Class 1: A DP process which consists of a set of procedures, and a body which terminates before any procedures calls are allowed, is a monitor in the conventional sense, but with explicit wait and implicit signal operations as provided by guarded regions.

Class 2: A DP process which contains no procedures and consists solely of a body is a process in the conventional sense whose only communication is by synchronised procedure calls to other (monitor-like) processes.

Class 3: By bringing together the process and monitor as a single "distributed process" concept DP creates a third class of process in which the sequential execution of the process body is non-deterministically interleaved with execution of calls on its monitor-like procedures.

It is the class 3 process which realises the full potential of each processor in a distributed network - for asynchronous parallel activity interleaved with synchronised communication sequences. However the retention of the monitor procedure as the only means of communication in DP enforces a textual and logical split between the synchronised communication sequence and its asynchronous consequence within the process, interposing a subtle non-determinism which the programmer often has to override by additional code.

In contrast a CSP process adopts monitor-like behaviour by executing a non-deterministic loop with an appropriate input guard for each monitor call. In some cases the waiting required of the 'caller' is expressible as a boolean precondition to the input guard; in others it may be expressed as the boolean guard of a second 'signalling' command in the same non-deterministic loop. The extent to which synchronisation persists once an input 'call' is accepted is freely determined by the position of the corresponding output response, if any, and the asynchronous consequence follows immediately and deterministically in the process text. Any additional activity by a class 3 process in parallel with its users can be added to the same non-deterministic loop without any additional structure.

Thus while DP seems to graft the monitor and process concepts together to allow a clumsy expression of class 3 'processes', CSP enables the characteristics of each class to be synthesised from a single concept, the guarded loop, by the inclusion of separate input and output commands, and input guards.

3. A QUANTITATIVE COMPARISON

The qualitative comparison given in section 2 is illustrated using two programs only, but is based on experience of translating the complete set of example programs given in each proposal into the notation advocated in the other. The appendices A1-16 show the results of that effort.

Such a collection of small but significant programs in two languages invites some quantitative comparison. A quantitative method for measuring program characteristics by examination of their source text has been developed by Halstead under the general title Software Science (Halstead 1977). Despite the simplistic and sometimes startling hypotheses on which it is based some significant success has been claimed for the approach (Fitzsimmons and Love 1978). Halstead's measures are obtained by counting

the number of distinct operators n_1,
the number of operator occurrences N_1,
the number of distinct operands n_2,
and the number of operand occurrences N_2,

in each program. From these the program 'volume' V is given by

$$V = (N_1+N_2)\ log_2(n_1+n_2)$$

Thereafter Halstead postulates that.

(1) The level of abstraction in a program is given by

$$L = V^*/V$$

where V^* is the minimal volume possible for the algorithm.

(2) An appropriate measure of language level is

$$\lambda = LV^* = (V^*)^2/V$$

(3) The effort involved in writing a program is measured by

$$E = V/L = V^2/V^*$$

Thus, in comparing versions of the same algorithm in two programming languages, the ratio of the measured volumes V_1/V_2 is an inverse measure of the relative language levels (and the square of this ratio is a measure of the relative effort involved in the program's construction).

To compare the effect of the different mechanisms for communication, synchronisation and non-determinism in CSP and DP by use of Halstead's measures it is essential that other differences in the two languages do not interfere. To this end the programs given in the appendices differ from the originals in the following ways:

(1) The CSP proposal makes no suggestions on the range of data types and data structures that a complete language should provide, while the DP proposal includes arrays, sets and bounded sequences, with appropriate operations for each, together with a *for* statement enabling the processing of arrays and sets, component by component. For comparison purposes the CSP examples have been rewritten assuming an exactly equivalent set of features.

(2) The DP proposal makes the use of semicolon delimiters optional. To make counting easier all semicolons which denote explicit sequencing in a DP process have been made explicit in the process text.

(3) In his discussion Hoare suggests an abbreviated notation for declaring input variables, writing *X?T(n:integer)* rather than *n:integer ; X?T(n)*. This abbreviation has been used throughout the CSP examples as it corresponds more closely to an input parameter declaration in DP.

The precise strategy to be used in counting operators and operands in the CSP and DP programs also required careful consideration. The accessible papers on software science give limited examples, in languages very different from CSP and DP. However from these the following general strategies are apparent:

(1) Only executable program text is counted, all declarative text being ignored.

(2) Composite combinations of symbols such as *if* *then* or *while* *do* are considered to be single operators.

(3) Where control operations refer to other points in a program text, e.g. *goto* or *call*, each combination of operation and distinct label or name is considered a distinct operator.

For DP and CSP these strategies were interpreted as follows:

(1) In DP all declarations and procedure headings were ignored. In CSP all declarations were ignored, including those within input commands such as *X?T(n:integer)*; however in a subsequent input to an existing variable such as *X?T(n)* the variable *n* was counted as an operand occurrence.

(2) In DP the combinations

if : *end*
do : *end*
when : *end*
cycle : *end*

were considered to be four distinct operators, with imbedded occurrences of

| :

denoting a further auxiliary operator. Likewise the combination

for *x* *in* : *end*

was considered a single operator (and declaration of *x*). Similar arguments give the following combinations as operators in CSP:

[→]
*[→]
[] →
*[*x* *in* →]

(3) In DP each distinct combination of the form

call P.t( )

was considered a distinct operator. In CSP each of the forms

P!t( )
P?t( )

was considered a distinct operator.

One problem area was the treatment of bound variables in CSP's guarded arrays, such as:

(i : 1..n) P(i)?t(.....) → .....i.....i.....

How should the introduction and use of *i* be counted? After some consideration the following procedure was adopted:

(1) The initial *(i : 1..n)* was considered to be declarative and was therefore ignored.

(2) Each subsequent occurrence of *i* on either side of the → was counted as an operand occurrence.

(3) The parentheses() enclosing *i* in *...P(i)?....* were not counted as a subscripting operation, since the process array name P cannot occur without them - a similar attitude was taken on all purely syntactic parentheses, commas, etc. throughout the two languages.

The details of the precise counting method to be used for any language are clearly open to dispute. Experiments by Elshoff (1978) suggest that, while some software science measures are sensitive to minor variations in counting strategy, the volume measure *V* is not. What matters in comparing CSP and DP is that the two counting strategies used should be consistent and should reflect the significant programming differences imposed by the two notations. It is on this assumption of fairness that interpretation of the results depends.

Application of the chosen counting strategy to the programs given as Appendices A1-A16 produced the results shown in Table 1.

Program example	DP					CSP					λDP / λCSP
	n_1	N_1	n_2	N_2	V	n_1	N_1	n_2	N_2	V	
A1. Message buffer	8	10	3	7	58.8	13	17	3	8	100	1.70
A2. Character stream	14	27	9	24	231	14	26	8	22	214	0.93
A3. Resource scheduler	4	6	3	8	39.3	3	3	1	2	10.0	0.25
A4. S-J-N scheduler	20	49	10	43	451	20	48	10	44	451	1.00
A5. Readers and writers	8	15	3	20	121	15	20	4	14	144	1.19
A6. Alarm clock	7	10	5	10	71.7	11	21	9	20	177	2.47
A7. Dining philosophers	14	16	6	13	125	16	20	5	13	145	1.16
A8. Sorting array	16	47	10	50	456	17	43	6	42	385	0.84
A9. Vending machine	13	37	10	36	330	16	33	8	30	289	0.88
A10.Copier	4	10	6	12	73.1	3	3	1	1	8.00	0.11
A11.Squasher	9	37	10	42	336	8	17	3	13	104	0.31
A12.Disassembler	5	7	4	5	38.0	5	6	3	3	27.0	0.71
A13.Division 1	8	13	6	12	95.2	8	15	6	14	110	1.16
A13.Division 2	10	30	13	32	280						0.39
A14.Set of integers	18	47	14	40	435	16	34	9	28	288	0.66
A15.Integer semaphore	5	7	3	10	51.0	9	12	5	12	91.4	1.79
A16.Eratosthenes' sieve	16	31	8	29	275	14	22	5	20	178	0.65

Table 1 : Program examples with software science data and resultant λ ratio

Programs A1 to A9 are those used by Brinch Hansen to illustrate the features of DP; programs A10 to A16 are used by Hoare to illustrate the features of CSP. The following additional programs in Hoare's paper were discarded for the reasons given:

(1) The ASSEMBLE process is too dependent on process termination in CSP to be represented fairly in DP.

(2) The recursive FACTORIAL array of processes is unrealistic.

(3) The bounded buffer is equivalent to Brinch Hansen's message buffer (A1).

(4) The matrix multiplication array is too dependent on features not provided in DP.

In translating a CSP process which accepts input from another process P and later returns an output response, one must decide in DP whether the process P is to wait for the response, or is to be allowed to continue in parallel and request the response later - in CSP this is determined solely by the "calling sequence" used in P. Since this distinction is crucial to the qualitative assessment, Hoare's division process (A13) was translated into two DP forms which demonstrate the difference involved. Division 1 provides a single

procedure which forces the user process P to wait; Division 2 provides one procedure which inputs the dividend and divisor, and another which collects the remainder and quotient when they are available.

The following comments may be made on the results shown in Table 1:

(1) The resource scheduler (A3) and the copier process (A10), which is a one-character buffer, are represented so nearly optimally in CSP that they may be discounted as pathological cases.

(2) For several of the examples, such as the readers and writers scheduler (A5), the dining philosophers (A7) and the integer semaphore (A15), the actions coded in DP and CSP are identical. The advantage shown for DP by the λ ratio is due entirely to the fact that in DP the initial execution of the process body followed by the repeated execution of calls on its procedure is implicit in the process syntax, while in CSP it must be expressed explicitly by coding of the form

```
...initialisation...;
*[ inputguard → ......
 □ inputguard → ......
.....................
 ]
```

This inherent advantage of DP applies throughout the coding of all class 1 processes.

(3) The λ-ratios show an extreme advantage for DP in the alarm clock process, and a significant advantage for CSP in the set of integers process, in accordance with the preceding qualitative assessment. Given that software science is a statistical science, and that the counting strategy used may be open to criticism, the precise value of the λ ratio for each example is not highly significant, but the variation of λ ratios with process class is. Discarding the pathological cases, and re-ordering the examples in descending λ-ratio order gives the results shown in Table 2.

Program	λDP/λCSP	Class
Alarm clock	2.47	1
Integer semaphore	1.79	1
Message buffer	1.70	1
Readers and writers	1.19	1
Dining philosophers	1.16	1
Division 1	1.16	1
Shortest job next scheduler	1.00	3
Character stream	0.93	2
Vending machine	0.88	1
Sort array	0.84	3
Disassembler	0.71	2
Set of integers	0.66	3
Sieve	0.65	3
Division 2	0.39	3
Squasher	0.31	3

Table 2 Program examples in descending λ ratio order showing correlation with process class

Table 2 clearly shows that of program examples for which DP is either better or comparable to CSP by λ measurement all but one are of class 1 or 2, i.e. either conventional monitors or conventional processes. All but one of the examples in which CSP has a significant advantage are of class 3. Thus these quantitative results show a close agreement with the qualitative assessment in section 2.

4. CONCLUSION

The faults found with DP in comparison to CSP are typical of those to which any higher level notation is liable, namely that the higher level abstraction removes from the user either access to information, or a freedom, which his application requires. From the examples considered, the constraints of a procedural communication mechanism make class 3 processes more difficult to construct in DP than in the lower level CSP. This is borne out by both the qualitative and quantitative analyses of the examples chosen. It seems therefore that CSP has struck a better level of language features for such programs, and creates less unwanted non-determinism than DP.

Is this the whole story? The example programs were presumably chosen by their authors to illustrate each language feature, rather than to provide a representative sample of the potential program set of each language. We in turn have used them simply because they were there. In the end the success of a language is determined by the actual application programs which have to be written in it, not by the toy programs beloved by language designers and academics. Of the examples considered, one showed advantages of DP which were not illustrated elsewhere, and which are dependent on its procedural

encapsulation of communication. If the real world's application programs of class 3 exploit these advantages more than the examples used in the papers then the relative effectiveness of the two languages may be different from that found here.

ACKNOWLEDGEMENTS

The contents of this paper are the result of a series of group discussions with our colleagues Paul Bailes, Bob Buckley, Rick Stevenson and Greg Smith. Their contribution to the final outcome is gratefully acknowledged.

REFERENCES

Brinch Hansen P.[1973] "Operating System Principles", Prentice-Hall, Englewood Cliffs, N.J.

Brinch Hansen P.[1975] "The programming language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol 1, No 2.

Brinch Hansen P.[1978] "Distributed processes", Comm.ACM, Vol 21, No 11.

Dijkstra E.W.[1976] "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J.

Elshoff J.L.[1978] "An investigation into the effects of the counting method used on software science measurements", ACM Sigplan Notices, Vol 13, No 2.

Fitzsimmons A., Love T.[1978] "A review and evaluation of software science", ACM Computing Surveys, Vol 10, No 1.

Halstead M.[1977] "Elements of Software Science", Elsevier North-Holland Inc., N.Y.

Hoare C.A.R.[1974] "Monitors: an operating system structuring concept", Comm.ACM, Vol 17, No 10.

Hoare C.A.R.[1978] "Communicating sequential processes", Comm.ACM, Vol 21, No 8.

Welsh J., Bustard D.W.[1979] "Pascal-plus - another language for modular multiprogramming", Australian Computer Science Communications, Vol. 1, No. 1.

Wirth N.[1977] "Modula : A programming language for modular multiprogramming", Software - Practice and Experience,Vol 7, No 1.

APPENDICES

A1 : Message buffer

Brinch Hansen's example of a bounded buffer is as follows:

```
process buffer
     s : seq[n]char
     proc send(c:char)  when not s.full : s.put(c) end
     proc rec(#v:char)  when not s.empty : s.get(v) end
     s := []
```

A comparable CSP solution is as follows:

```
buffer ::
     s : seq(n)char ;
     s := () ;
    *[ ¬ s.full ; X?send(c:char) → s.put(c)
     [] ¬ s.empty ; Y?rec() → v:char ; s.get(v) ; Y!rec(v)
     ]
```

Comments: (i) The CSP solution requires an explicit loop *[....[]....] to express the non-deterministic sequencing of send and receive operations ; in CSP this is implicitly expressed. A similar difference is observable in all class 1 (monitor) programs.

(ii) The CSP solution requires a two-step calling sequence in the receiver process Y

```
buffer!rec() ; buffer?rec(x)
```

to simulate a call to a procedure with a result parameter.

A2 : An input character stream

Brinch Hansen programs a process which reads card images and sends a corresponding stream of characters to a buffer, as follows:

```
process stream
     b : array[80]char ; n,i : int
     do true :
        call cardreader.input(b) ;
        if b=blankline : skip |
           b≠blankline :
                i := 1 ; n := 80 ;
                do b[n]=space : n := n-1 end ;
                do i≤n : call buffer.send(b[i]) ; i :=i+1 end
        end ;
        call : buffer.send(newline)
     end
```

A comparable CSP process is as follows:

```
stream ::
*[ cardreader?input(b:(1..80)char) →
      [ b=blankline → skip
      □ b≠blankline →
            n,i : integer ;
            i := 1 ; n := 80 ;
           *[ b(n)=space → n := n-1] ;
           *[ i≤n → buffer!send(b(i)) ; i := i+1]
      ] ;
      buffer!send(newline)
]
```

Comment: The CSP version replaces *do true : input call ; ....* by **[inputguard → .....* which in CSP also takes care of the termination of the inputting (cardreader) process, but otherwise the solutions are equivalent in all but syntax.

A3 : A resource scheduler

Brinch Hansen gives the following DP coding of a monitor which schedules the use of a single resource:

```
process resource
     free : bool
     proc request  when free : free := false end
     proc release  if not free : free := true end
     free := true
```

Assuming user processes *user (1..n)*, a suitable CSP scheduler is as follows:

```
resource ::
   *[ (i:1..n) user(i)?request() → user(i)?release() ]
```

Comments:

(i) The CSP solution allows only a non-deterministic choice between competing user requests. In DP the non-determinism also includes the release operations, and the state variable free has to be introduced.

(ii) The DP scheduler stops if a release operation is attempted when the resource is already free; the CSP version ignores such operations.

(iii) The CSP requirement for naming user processes means that the scheduler accepts a release operation only from the user who last acquired it; to achieve the same control in DP requires a user identity parameter in each procedure.

A4 : shortest-job-next scheduler

Brinch Hansen gives the following (class 3) process as a shortest-job-next scheduler, in which the next job is decided in parallel with the current job's use of the resource whenever possible:

```
process sjn
    queue : set[n]int ; rank : array[n]int
    user,next,min : int

    proc request(who,time:int)
        begin queue.include(who) ; rank[who] := time ;
              next := nil ; when user=who : next := nil end
        end

    proc release  user := nil

    begin queue := [] ; user := nil ; next := nil ;
          cycle
              not queue.empty & (next=nil) :
                    min := maxint ;
                    for i in queue :
                       if rank[i]≥min : skip |
                          rank[i]<min : next := i ; min := rank[i]
                       end
                    end |

              (user=nil) & (next≠nil) :
                     user := next ; queue.exclude(user)
          end
    end
```

Assuming user processes *users(1..n)* an equivalent CSP process is as follows:

```
sjn ::
    queue : set(n)integer ; rank : (1..n)integer ;
    user,next : integer ;
    queue := () ; user := nil ; next := nil ;

   *[ (i:1..n) users(i)?request(time:integer) →
                 queue.include(i) ; rank(i) := time ; next := nil
    □ (i:1..n) users(i)?release() → user := nil

    □ next=nil ;  queue.empty →
              min : integer ; min := maxint ;
             *[ i in queue →
                    [ rank(i)≥min → skip
                    □ rank(i)<min → next := i ; min := rank(i)
              ]     ]
    □ user=nil ; next≠nil →
              users(next)!ok() ; user := next ;
              queue.exclude(user) ; next := nil
```

Comments: (i) Each solution involves four non-deterministically interleaved code fragments, whose interleaving is controlled by the same variables. In CSP the four fragments form a single non-deterministic command.

(ii) In CSP reactivation of a user is separated from the code which accepted the request; however this separation allows this reactivation to precede, rather than follow, the corresponding housekeeping within the scheduler, and precludes the interleaving of any further request processing, so a higher degree of parallelism is possible.

A5 : A readers and writers scheduler

Brinch Hansen shows the following solution to the readers and writers problem:

```
process resource
      s : int
      proc startread  when s≥1 : s := s+1 end
      proc endread    if   s>1 : s := s-1 end
      proc startwrite when s=1 : s := 0   end
      proc endwrite   if   s=0 : s := 1   end
      s := 1
```

The following CSP process solves the same problem:

```
resource ::
     readers : set(n)int ; readers := () ;
   *[ (i:1..n)    (i in readers) ; user(i)?startread()  →
                                        readers.include(i)
    □ (i:1..n)     i in readers  ; user(i)?endread()    →
                                        readers.exclude(i)
    □ (i:1..n)     readers.empty ; user(i)?startwrite() →
                                        user(i)?endwrite()
    ]
```

Comments: (i) As in the resource scheduler the deterministic sequence for *startwrite* and *endwrite* operations is expressed directly in CSP; however non-deterministic selection between *startreads*, *endreads* and *startwrites* is required.

(ii) By replacing the state variable *s* by the set *readers* the CSP version validates user identity at little extra cost; to do the same in DP requires an identity parameter in each procedure, a set of readers, and a boolean flag for writing.

(iii) Because the waiting condition is expressible as a boolean expression prefixing an input guard the CSP solution encapsulates the waiting process as neatly as in DP.

A6 : An alarm clock

Brinch Hansen suggests the following process to enable other user processes to wait for a specified interval of time, where the actual progress of time is signalled by a regular tick operation:

```
process alarm
     time : int
     proc wait(interval:int)
          due : int
          begin due := time+interval ;
                when time=due : skip end
          end
     proc tick  time := time+1
     time := 0
```

The same effect is provided by the following CSP process:

```
alarm ::
    time : integer ; due : (1..n)integer ;
    time := 0 ; *[ d in due → d := -1 ] ;
   *[ (i:1..n) user(i)?wait(interval:integer) →
                                          due(i) := time+interval
    [] (i:1..n) due(i) = time →
                          user(i)!wakeywakey() ; due(i) := -1
    [] realclock?tick() → time := time+1
    ]
```

Comments: see text.

A7 : The dining philosophers problem

Brinch Hansen gives the following solution to the dining philosophers problem:

```
process Philosopher[5]
    do true : ..... call table.join(this)....
                    call table.leave (this).....end
process table
    eating : set[5]int
    proc join(i:int)
        when ([i⊖1,i⊖1] & eating)=[] : eating.include(i) end
    proc leave(i:int)
         eating.exclude(i)
    eating := [ ]
```

An exactly equivalent solution is possible in CSP:

```
Philosopher (i:1..5) ::
    *[true → ....table!join()......table!leave().... ]

table ::
     eating : set(5)integer ; eating := () ;
    *[ (i:1..5) ((i⊖1,i⊕1) & eating)=() ;
                Philosopher(i)?join() → eating.include(i)
     □ (i:1..5) Philosopher(i)?leave() → eating.exclude(i)

     ]
```

Comment: The table process is another example of a scheduler in which the waiting condition is expressible as a boolean expression prefixing an input guard.

A8 : Sorting array

Brinch Hansen's version of this example is confusing in that it uses a sequence of length ≤ 2 to hold the items within each element process at any moment, but also accesses and rearranges them by subscript as if the sequence is an array. The following version uses two variables and a load count instead of the sequence, and gives a smaller software science volume than Brinch Hansen's original:

```
process Sort[n]
     x,temp,load,rest : int

     proc put(c:int)  when load=0 : x := c ; load := 1 |
                           load=1 : temp := c ; load := 2 end

     proc get(#c:int)  when load=1 : c:= x ; load := 0 end

     begin
          load := 0 ; rest := 0 ;
          cycle
               load=2 :
                    if x≤temp : call Sort[succ].put(temp) |
                       x>temp : call Sort[succ].put(x) ; x:=temp
                    end;
                    rest := rest+1 ; load := 1 |

               (load=0) & (rest≠0) :
                    call Sort[succ].get(x) ;
                    rest := rest-1 ; load := 1
          end
     end
```

An equivalent CSP array is as follows:

```
Sort(i:1..n) ::

*[ Sort(i-1)?put(x:integer) →

        count : integer ; count := 1 ;
       *[ count>0 ; Sort(i-1)?put(temp:integer) →
                  [ x≤temp → Sort(i+1)!put(temp)
                  □ x>temp → Sort(i+1)!put(x) ; x:=temp
                  ] ;
                  count := count+1
        □ count>0 ; Sort(i-1)?get() →
                  Sort(i-1)!got(x) ; count := count-1
                  [ count=0 → skip
                  □ count>0 → Sort(i+1)!get() ;
                              Sort(i+1)?got(x)
]       ]       ]
```

Comments: (i) In the CSP version the first *put* operation which loads a previous empty process element is distinguished from those which follow; the count variable ensures that the process reverts to this initial empty state when the last value has been extracted.

(ii) In the CSP version the balancing actions required of each process after it services a *get* or *put* request are coded to follow the servicing code deterministically. In the DP version the load variable controls the non-deterministic interleaving of the four code fragments required.

A9 : A vending machine

Brinch Hansen gives the following process defining the behaviour of a single vending machine:

```
process vending machine
     items,paid,cash : int
     proc insert(coin:int)  paid := paid+coin
     proc push(#change,goods:int)
         if (items>0) & (paid≥price) :
              change := paid-price ; cash := cash+price ;
              goods := 1 ; items :=items-1 ; paid := 0 |
           (items=0 or (paid<price) :
              change := paid ; goods := 0 ; paid := 0
         end
     begin items := 50 ; paid := 0 ; cash := 0 end
```

A similar process in CSP is as follows:

```
Vending machine ::
    items,paid,cash : integer ;
    items :=50 ; paid :=0 ; cash := 0 ;
   *[ user?insert(coin : integer) → paid := paid+coin
    [] user?push() →
          [ items>0 ; paid≥price →
                    user!deliver(paid-price,1) ;
                    cash := cash+price ; items := items-1
          [] (items=0) ∨ (paid<price) →
                    user!deliver(paid,0)
          ] ;
          paid := 0
    ]
```

Comment: The CSP solution is slightly neater because the return of the results of a *push* operation can be coded as direct output commands, rather than as assignments to formal parameters.

A10 : A single character buffer

Hoare gives the following CSP process which acts as a single character buffer between processes *west* and *east*:

```
x :: *[ west?put(c:char) → east!get(c)]
```

An equivalent buffer might be programmed in DP as follows:

```
process x
      c : char ; cready : bool ;
      proc put(u:char)  when not cready :
                                c := u ; cready := true end
      proc get(#v:char)  when cready :
                                v := c ; cready := false end
      cready := false
```

Comments: (i) As with the resource scheduler (A3) the deterministic sequence of *put* and *get* operations is expressed explicitly in CSP. In DP it must be superimposed on an inherently non-deterministic sequence by means of the state variable *cready*.

(ii) A slightly shorter if obtuse solution can be expressed in DP by declaring *c* as a bounded sequence of maximum length 1, thus

```
c : seq[1]char
```

(cf. A1) but comment (i) still applies.

All : A squasher

Hoare also gives the following example, which is again a single character buffer, but which replaces each pair of consecutive asterisks by an upward arrow:

```
x :: *[ west?put(c:char) →
          [ c≠asterisk → east!get(c)
          [] c=asterisk →
              west?put(c) ;
              [ c≠asterisk → east!get(asterisk) ; east!get(c)
              [] c=asterisk → east!get(upwardarrow)
    ]     ]   ]
```

A DP solution to the same problem is as follows:

```
process x
      c : char ; state : int
      proc put(u:char)  when state=cempty :
                                        c := u ; state := cfull end
      proc get(#v:char)
           when state=cready : v := c ; state := cempty |
                state=asteriskfirst : v := asterisk ;
                                      state := cready
           end
      begin
          state := cempty ;
          cycle state=cfull :
              if c≠asterisk : state := cready |
                 c=asterisk :
                   state := cempty ;
                   when state=cfull :
                      if c≠asterisk : state := asteriskfirst |
                         c=asterisk : c:= upwardarrow ;
                                      state := cready
                      end
                   end
              end
          end
      end
```

Comments: (i) In the CSP solution the state of the buffer at any moment is represented by its point of execution, with distinct input and output commands for each state. In DP all input and output must pass through the procedures *put* and *get* with the process body keeping track of the state in which any particular call of *put* or *get* occurs.

(ii) A DP solution using a two character buffer to solve the same problem may be simpler, but the above is typical of the coding required of any 'intelligent' buffer which vets the data received before transmitting it.

A12 : A card image disassembler

Hoare gives the following process to read card images and output them character by character to a process x inserting an extra space at the end of each card. It is similar to, but simpler than, Brinch Hansen's character stream (A2). Using the for construct imported from DP it is trivially programmed in CSP as follows:

```
east ::
   *[ cardfile?read(cardimage:(1..80)char) →
                *[ c in cardimage → x!put(c) ] ;
                 x!put(space)
    ]
```

The DP solution is as follows:

```
process east
      cardimage : array[80]char
      do true :
           call cardfile.read(cardimage) ;
           for c in cardimage : call x.put(c) end ;
           call x.put(space)
      end
```

A13 : A division routine

Hoare gives the following process as a means of providing an integer division facility for a user process X :

```
Division ::
    *[ X?div(x,y:integer) →
             quot,rem : integer ;
             quot := 0 ; rem := x ;
            *[ rem≥y → rem := rem-y ; quot := quot+1 ] ;
             X!ans(quot,rem)
     ]
```

This can be implemented in DP in two ways according to whether the user process X is to be forced to wait for the answer. If it is, then a process providing a single procedure suffices:

```
process Division 1
     proc div(x,y:integer#quot,rem:integer)
       begin
           quot := 0 ; rem := x ;
           do rem≥y : rem := rem-y ; quot := quot+1 end
       end
      skip
```

However if the user process is to be allowed to continue while the answer is being computed, a process with two procedures and an interleaved process body is required:

```
process Division 2
        Y,R,Q : int ; state : int
        proc div(x,y:integer)
             when state=idle : R := x ; Y := y ;
                                    state := active end
        proc ans(#quot,rem:integer)
             when state=done : quot := Q ; rem := R ;
                                    state := idle end
        begin
            state := idle ;
            cycle state=active :
                 Q := 0 ;
                 do R≥Y : R := R-Y ; Q := Q+1 end ;
                 state := done
            end
        end
```

Comment: The two processes illustrate clearly the differences between class 1 and class 3 processes in DP, and the programming overheads which it imposes on the latter.

A14 : A set of integers

Hoare gives the following process as a representation for a set of integers, in which insertions are carried out in parallel with continued execution of the requesting process x. For ease of comparison an imbedded procedure-like formulation has been adopted for the search subroutine.

```
S :: content : (0..99)integer ; size,i : integer ;

     search(n:integer) ≡
          ( i := 0 ; *[ i<size ; content(i)≠n → i := i+1 ] )

     size := 0 ;
    *[ x?has(n:integer) → search(n) ; x!reply(i<size)

     □ x?insert(n:integer) → search(n) ;
                             [ i<size → skip
                             □ i=size ; size<100 →
                                content(size) :=n ; size := size+1
     ]                       ]
```

A DP process allowing the same parallelism is as follows:

```
process S
     content : array[100]int ; size,i : int ;
     insertionrequested : boolean ; insertionvalue : int ;
     proc search(n:int)
          begin i := 1 ;
                do (i≤size)&(content[i]≠n) : i:=i+1 end end

     proc has(n:int#answer:boolean)
          when not insertionrequested :
               call S.search(n) ; answer := i≤size end
     proc insert(n:int)
          when not insertionrequested :
               insertionrequested := true ;
               insertionvalue := n end

     begin
          size := 0 ;
          insertionrequested := false ;
          cycle insertionrequested :
                  call S.search(insertionvalue) ;
                  if i≤size : skip |
                     (i>size)&(size<100) :
                          size := size+1 ;
                          content[size] := insertionvalue
                  end ;
                  insertionrequested := false
          end
     end
```

Comments: see text.

A15 : An integer semaphore

Hoare implements an integer semaphore as follows:

```
S :: val : integer ; val := 0 ;
    *[ (i:1..100) X(i)?V() → val := val+1
     □ (i:1..100) val>0 ; X(i)?P() → val := val-1
]
```

The DP solution is very similar:

```
process S
     val : int
     proc V  val := val+1
     proc P  when val>0 : val := val-1 end
     val := 0
```

Comment: This is a third example of a scheduler in which the waiting condition is trivially expressed as a boolean prefix to an input guard.

A16 : A prime number sieve

Hoare gives the following process array as the core of a sieve to find prime numbers, in which the processes deal concurrently with the different trial numbers percolating through the sieve. For comparison purposes the boundary processes are omitted as these involve different techniques for their definition in CSP and DP.

```
Sieve(i:1..100) ::
     p,mp : integer ;
     Sieve(i-1)?test(p) ;
     print!prime(p) ;
     mp := p ;
    *[Sieve(i-1)?test(m:integer) →
           *[ m>mp → mp :=mp+p ] ;
            [ m=mp → skip
            ] m<mp → Sieve(i+1)!test(m)
     ]      ]
```

A DP process array with the same effect is as follows:

```
process Sieve(100)
     p,mp,m :int ; testing : bool
     proc test(t:int) when not testing : m := t ;
                                         testing := true end
     begin
         testing := false ;
         when testing :
            p := m ;
            call print.prime(p) ;
            mp := p ;
            testing := false ;
            cycle testing :
                 do m>mp : mp := mp+p end ;
                 if m=mp : skip |
                    m<mp : call Sieve[succ].test(m)
                 end ;
                 testing := false
            end
         end
     end
```

Comment: Again the significant difference is that in the DP process all input must pass through the procedure *test*, and that the interleaving of this and the process body must be controlled by the state variable testing.