JSD EXPRESSED IN CSP

by

K.T.Sridhar C.A.R.Hoare

Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD

Technical Monograph PRG-51

July 1985

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

[1]Supported by a fellowship from UNDP and on leave from

NCSDCT
Tata Institute of Fundamental Research
Homi Bhabha Road
Bombay 400 005
India

[2]Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Abstract

System development in JSD is done by building a model of the real world taking into account the time-ordering of events, and then extending this model to satisfy the functional requirements. The resulting specification is transformed into programs more efficiently executable on current computer systems. We study the relationship between JSD and CSP and suggest that CSP provides a theoretical basis for the concepts and methods of JSD. Constructive specifications in CSP are given for problems solved using JSD. Efficient implementations for conventional sequential processors may be derived from the parallel CSP solutions using the algebraic laws governing the CSP operators.

1. Introduction

In the introduction to his book on System Development [J], Michael Jackson acknowledges the influence of earlier work on communicating sequential processes. The similarity of JSD with CSP is due to their common aim to describe, specify, develop and implement systems whose subject matter has a strong time dimension. These include embedded systems, switching systems, control systems, and all kinds of data processing systems, both on-line and batch processing.

Michael Jackson's book is written as a practical guide to practicing programmers. It helps them to organise their thoughts, their projects, and their teams in a more effective fashion. The development process is split into six steps, starting with a description of the system environment and the capture of customer requirements and culminating in the production of efficient code in a particular language for a particular machine. The method involves liberal use of diagrams, and is illustrated by five amusing and instructive examples.

A later book on Communicating Sequential Processes [H2] takes a much more theoretical approach. The notations are modelled on mathematical formulae; they are given a mathematical definition, and they are governed by mathematical laws. No advice is given on development steps; and although an implementation is given, it uses a functional programming language and cannot be recommended for any practical purpose. Diagrams play a very subordinate role, and the examples, though numerous, are very much smaller.

The purpose of this paper is to clarify the relationship between JSD and CSP. We have tried to show that CSP provides a promising theoretical basis for the practical concepts and methods used in JSD. We hope that this may suggest fruitful directions for research in the practical application of CSP, and that it may lead to a better appreciation of JSD and perhaps suggest further improvements.

Many of the most apparent differences between JSD and CSP are only superficial, and arise from the differences described above in the orientation and target readership of the two books. In order to strip away these superficialities we have developed CSP solutions for each of the five examples used in [J] to illustrate JSD. Such non-trivial case studies still seem to offer the best research technique for elucidating the similarities and differences, and the merits and defects, of proposed methods for design and development of computer programs.

This paper is designed to be self-contained, but readers familiar with CSP and/or JSD

will find it much easier to follow. The next section contains a brief review of CSP: it describes the CSP operators used in the remainder of the paper, and gives their mathematical definition in terms of the trace model [H2].

Sections 3 to 7 contain a treatment of the five main examples of [J]. In each example we follow Jackson's strategy of first solving a simple version, and then adding further functions and complications. However, it is not possible to pursue all the complications of [J] in the span of this shorter article.

In section 8, we show how the algebraic laws of CSP can be used to transform a highly parallel program into one with the same specification of its observable behaviour, but which can be executed efficiently on a conventional sequential processor. This is a more mathematical version of Jackson's recommendation for the implementation steps in his development process. Section 9 summarises the main technical differences between CSP and JSD and the final section indicates directions for further work.

In an appendix we have given a summary of all the main notational and terminological differences between JSD and CSP. A reader familiar with JSD may wish to study the appendix first as an aid to familiarisation with CSP; other readers are recommended to consult the appendix as an aid to understanding Jackson's book.

2. CSP

The formalisms used in this paper are taken from [H2]. However, we make the simplification that processes are described only in terms of traces of their behaviour, ignoring problems of non-determinism. This simplification is justified perhaps by the fact that we are more interested in the specification of processes than their implementation. Thus we merely specify that the system must not deadlock; and we do not at this stage need to introduce such complexities as refusal sets in order to prove that a particular implementation avoids such problems.

The behaviour of a process is described in terms of externally observable *events* drawn from its *alphabet*. A trace of a process is a sequence of events it can engage in up to some moment in time. `traces(P)` denotes the set of all possible traces of a process P. `initials(P)` is the set of events in which P can engage right on its first step

```
initials(P) ≙ {a | <a> ∈ traces(P)}
```

The following two sections summarise the most important concepts and operators of CSP. For further information the reader is referred to [BHR], [H1] or [H2].

2.1 Operations on Traces

Since traces play a crucial role in understanding the behaviour of processes it is necessary first to define a number of operators on traces.

A trace is denoted by a sequence of symbols (which stand for events) separated by commas and enclosed within angular brackets. $\langle a, b\rangle$ is a trace of two events a and b. $\langle\rangle$ denotes the empty trace. The letters s and t are frequently used to stand for traces. Catenation of two traces is denoted by the symbol $^\frown$, for example

$$\langle a, b\rangle^\frown\langle c\rangle = \langle a, b, c\rangle$$

$s\restriction A$ is the trace s restricted to the events in the set A

$$\langle a, b, c, a\rangle\restriction\{a, c\} = \langle a, c, a\rangle$$

The length or cardinality of a trace is denoted by #s

$$\#\langle a, b, c\rangle = 3$$

The first event in a non-empty trace s is denoted by s_0, while s' is the rest of the trace s after removing s_0

$$\langle a, b, c\rangle_0 = a$$
$$\langle a, b, c\rangle' = \langle b, c\rangle$$

The partial ordering relation $s \leqslant t$ means that s is a *prefix* (initial subsequence) of t

$$\langle a, b\rangle \leqslant \langle a, b, c, a\rangle$$

The empty trace is a prefix of all traces.

Sequential composition of traces s and t is denoted s;t. A special symbol ✓ is introduced to stand for the event of successful termination of a process, and so it can occur only at the end of a trace. If s does not contain the event ✓, then

$s;t = s$
$(s^\frown\langle\surd\rangle);t = s^\frown t$

For any set of events A, A^* denotes the set of all possible traces obtained using the events of the set A. Let f be a function from a set of events to another set of events, $f:A\rightarrow B$. Then f^* is a function from the set of traces A^* to B^*, and is defined by the two equations

$f^*(\langle\rangle) = \langle\rangle$
$f^*(\langle a\rangle^\frown s) = \langle f(a)\rangle^\frown f^*(s)$

2.2 CSP Operators

The syntax of the main CSP operators that we use is given below in BNF style. In the following P and Q are processes; s is a trace; a is an event; A and B are sets of events; i is a process label; $Bool$ is some boolean condition; αP is the alphabet of process P.

$$\begin{aligned} CSP ::= {} & a \rightarrow P \mid STOP_A \mid SKIP_A \mid RUN_A \\ & \mid P \parallel Q \mid P \,[]\, Q \mid P;Q \mid *P \\ & \mid P/s \mid P\backslash B \mid f(P) \mid i:P \\ & \mid P^\frown Q \mid P \,\lhd Bool\rhd\, Q \mid \mu X.P \end{aligned}$$

The prefixing operator $\rightarrow$ is used to define a process, $a \rightarrow P$, which first engages in the event a and then behaves like P. A process definition is *guarded* if it begins with a prefix

$traces(a \rightarrow P) = \{s \mid s=\langle\rangle \vee (s_0=a \wedge s' \in traces(P))\}$

$STOP_A$ is a deadlocked process which can engage in no event from its alphabet A. The suffix denoting the alphabet is often dropped

$\alpha STOP_A = A$
$traces(STOP_A) = \{\langle\rangle\}$

$SKIP_A$ is a process which has terminated successfully and is defined as

$SKIP_A \triangleq \surd \rightarrow STOP_A$
$\alpha SKIP_A = A \qquad$ provided $\surd \in A$

$traces(SKIP_A) = \{<>, <\surd>\}$

RUN_A is a process which is willing to engage in any of the events of the alphabet A

$$\alpha RUN_A = A$$
$$traces(RUN_A) = A^*$$

P ‖ Q is the parallel combination of the two processes P and Q. Events common to the alphabets of P and Q require simultaneous participation of both P and Q. However, P may engage independently in those events of its alphabet which are not in the alphabet of Q and *vice versa*

$$\alpha(P \parallel Q) = \alpha P \cup \alpha Q$$
$$traces(P \parallel Q) = \{s \mid s \in (\alpha P \cup \alpha Q)^* \wedge s \restriction \alpha P \in traces(P) \wedge s \restriction \alpha Q \in traces(Q)\}$$

We use the deterministic choice operator (denoted by |) when we know that no first event of P is also possible for Q. Process R defined below offers a deterministic choice between events a and b and subsequently behaves like either P or Q, according to this choice

$$\text{If } R \mathrel{\hat{=}} a \rightarrow P \mid b \rightarrow Q \qquad \text{where } a \neq b$$
$$\text{then } traces(R) = \{s \mid s = <> \vee (s_0 = a \wedge s' \in traces(P)) \vee (s_0 = b \wedge s' \in traces(Q))\}$$

[] is the general choice operator, which allows the environment to choose between P and Q. This choice is exercised by the environment only on the first action by selecting an event allowed by one and not allowed by the other. In this paper we shall ignore the problem of non-determinism. When no first event of P is also possible for Q, [] reduces to the deterministic choice operator |

$$traces(P \mathbin{[]} Q) = traces(P) \cup traces(Q)$$

P;Q is the sequential composition of the two processes P and Q. It behaves like P until P terminates, after which it behaves like Q. The occurrence of $\surd$ at the end of P is automatic and is not observable externally, *i.e.* this $\surd$ does not appear in any trace of P;Q

$$traces(P;Q) = \{s;t \mid s \in traces(P) \wedge t \in traces(Q)\}$$

*P is a process that behaves like an infinite sequential composition of the process P. It is the same as

$$P;P;P;\ldots$$

P/s (P after s) is the behaviour of process P after it has engaged in the events of trace s. P/s is defined only if $s \in traces(P)$.

P\B is the process P with events in the set B hidden from its environment. The events in set B do not appear in the traces of P\B. In the absence of divergence (see [BHR])

$$\alpha(P\backslash B) = \alpha P - B$$
$$traces(P\backslash B) = \{s\restriction(\alpha P - B) \mid s \in traces(P)\}$$

For simplicity we give here a definition which ignores the problem of divergence.

f(P) is the direct image of process P under the injection $f:\alpha P \rightarrow A$. The process f(P) performs the event f(a) whenever P performs the event a

$$traces(f(P)) = \{f^*(s) \mid s \in traces(P)\}$$

i:P is the process derived from P by labelling all events of P by i ; as a result, each event a of P becomes i.a for the process i:P. Process labelling is defined using the direct image operator as

$$i:P = f_i(P)$$
$$\text{where } f_i(x) = i.x \text{ for all } x \in \alpha P$$

The operator ^ denotes the interrupt operator. P^Q behaves like P until the occurrence of an event in initials(Q), whereupon it behaves like Q

$$traces(P\hat{}Q) = \{s\hat{}t \mid s \in traces(P) \wedge t \in traces(Q)\}$$

P⊧Bool⊦Q is a different notation for the familiar if-then-else. If the boolean condition Bool evaluates to true then P⊧Bool⊦Q behaves like P, otherwise Q.

μX.P denotes a process defined using guarded recursion. If F(X) is a guarded expression using the process name X (whose alphabet is A) then [BHR] shows that the equation

$$X = F(X)$$

has a unique solution with alphabet A. This solution is denoted by $\mu X{:}A.F(X)$. Whenever the alphabet is obvious, A is dropped and it is written as $\mu X.F(X)$. For example,

$$\mu X.(b \rightarrow X)$$

is a process which engages in an indefinitely long series of b events. It is the same as $RUN_{\langle b \rangle}$

$$traces(\mu X.(b \rightarrow X)) = \{b\}^*$$

In most of our examples we find it necessary to use arrays of processes and use indices in their definition. We therefore augment the earlier syntax of CSP with indexed operators for the three operators $\|$, $\Box$ and $;$

$$\|_{inex}\ P \mid \Box_{inex}\ P \mid ;_{inex}\ P$$

where inex is an index expression such as $i \geq 0$, $j \in X$, etc. We illustrate the indexed operators by the following examples.

$(\|_{i \geq 0}\ i{:}P)$ is the parallel combination of an infinite number of processes $0{:}P$, $1{:}P$, $2{:}P$,...

$$(\|_{i \geq 0}\ i{:}P) = 0{:}P \| 1{:}P \| 2{:}P \| \ldots$$

$(\Box_{i \geq 0}\ i.a \rightarrow P_i)$ is a concise notation for

$$0.a \rightarrow P_0 \Box 1.a \rightarrow P_1 \Box 2.a \rightarrow P_2 \Box \ldots$$

$(;_{0 \leq i < 10}\ (i.a \rightarrow p!i \rightarrow SKIP))$ is a concise notation for the sequential composition

$$(0.a \rightarrow p!0 \rightarrow SKIP);(1.a \rightarrow p!1 \rightarrow SKIP);\ldots;(9.a \rightarrow p!9 \rightarrow SKIP)$$

? and ! are used for input and output communications respectively with their usual meaning. $(in?x \rightarrow P)$ is a process which engages in the event $in.y$ for any y communicable as a single message on the input channel in and then behaves like P; while $(out!(x+1) \rightarrow P)$ is a process which engages in event $out.37$ if the value of x is 36 and subsequently behaves like P.

A process P with state x can be defined by a mutual recursion, in which each equation defines P_x for a different value of the state x. The subscript x may range over a small, large, or even infinite set. Here is a simple example, the process Count, which counts the number of occurrences of event a and is always willing to communicate this value on the channel out

$$Count \triangleq P_0$$
$$P_x \triangleq a \rightarrow P_{x+1} \mid out!x \rightarrow P_x$$

3. Modelling a bank

The first example we treat is a simplified version of a system which models the behaviour of customers in a bank. On opening an account, a customer joins a deposit scheme run by the bank. Subsequently, the customer may deposit or withdraw money as many times as required (overdrafts are permitted) until the account is terminated. Four events {invest, payin, withdraw, terminate} are required to describe the behaviour of a customer, which is represented by process Customer in the system. The meanings of these four events are

invest	open an account
payin	deposit money into the account
withdraw	take money out of the account
terminate	close an account

In this simplified version, we have chosen to ignore the amount that is being deposited, or withdrawn, and the balance of the account.

Since opening an account is the obvious first action of a customer, we write the process Customer as a guarded expression with prefix invest. The subsequent behaviour of a customer can be expressed using guarded recursion and the process STOP

$$Customer \triangleq invest \rightarrow \mu X.(payin \rightarrow X \mid withdraw \rightarrow X \mid terminate \rightarrow STOP)$$

A bank has many customers, each independently and concurrently interacting with the bank. To model this, each Customer process is labelled by the name of the customer. For example, the behaviour of a collection of three customers named a, b and c is described by

a:Customer ||| b:Customer || c:Customer

Since there is no limit to the number of customers, it is convenient to use natural numbers to name each of them uniquely. The indexed parallel combinator is used below with i denoting the identity of each customer

Bank $\hat{=}$ ($|||_{i \geq 0}$ i:Customer)

Our model of the bank consists apparently of an infinite number of processes, all waiting to be called into existence by the action invest. But at any given time only a finite number of them are actually started, one active process for each customer of the bank who has opened an account. On closing the account, the corresponding process stops and thereafter performs no further actions.

In an implementation starting and terminating processes could be very complicated and expensive actions; the advantage of mathematical description is that it ignores such complication and expense. The "infinite" array of processes presents no more conceptual difficulty than the potenitially unbounded number of calls of a procedure in a normal programming language.

3.1 Adding Functions

In this section, we explain how the JSD step of adding functions can be carried out in CSP. Extending the model to meet functional requirements often results in modification of existing processes or sometimes inclusion of new processes to meet the functional requirements. We illustrate this by extending the model of the bank to cater for the following two functions

(1) Whenever a customer overdraws, produce an overdraft report. We shall assume that the bank is generous enough to let the customers have an unlimited overdraft.

(2) On an enquiry specifying a customer identifier list, print balances of all customers specified in the list.

In order to meet these requirements, process Customer must be rewritten to store the current balance of a customer's account in its state. Further the events invest, payin and withdraw now become communication events in which these are channel names; and the amount that is being invested, paid in or withdrawn is the value that is

communicated

$$\begin{aligned}
&\mathsf{Customer} \triangleq \mathsf{invest?x} \rightarrow \mathsf{CUST}_x \\
&\mathsf{CUST}_x \triangleq (\mathsf{payin?y} \rightarrow \mathsf{CUST}_{x+y} \\
&\qquad\quad \mid \mathsf{withdraw?y} \rightarrow \mathsf{CUST}_{x-y} \\
&\qquad\quad \mid \mathsf{terminate} \rightarrow \mathsf{STOP})
\end{aligned}$$

To provide the exception reports, we need to further modify Customer by introducing a new communication event which uses channel overdraft. The process sends a pair of values along this channel whenever the balance goes negative. The first value is the withdrawn amount and the second is the subsequent balance. We use the if-then-else operator to check the status of the balance after every withdraw.y event

$$\begin{aligned}
&\mathsf{Customer} \triangleq \mathsf{invest?x} \rightarrow \mathsf{CUST}_x \\
&\mathsf{CUST}_x \triangleq (\mathsf{payin?y} \rightarrow \mathsf{CUST}_{x+y} \\
&\qquad\quad \mid \mathsf{withdraw?y} \rightarrow \mathsf{CUST}_{x-y} \ \{x \geq y\} \ (\mathsf{overdraft!}(y, x-y) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{CUST}_{x-y}) \\
&\qquad\quad \mid \mathsf{terminate} \rightarrow \mathsf{STOP})
\end{aligned}$$

For the second function ((2) above) we need to include a new process List whose job is to interrogate selected customers for their balance and print it out. This process List will input on channel in a list L of customer account numbers for which the balance is to be printed on channel print. List sequentially interrogates all customer processes with account numbers contained in the list L. Since List has to interrogate Customer about its balance, process Customer needs another communication event report.x to send its current balance x along channel report

$$\begin{aligned}
&\mathsf{Customer} \triangleq \mathsf{invest?x} \rightarrow \mathsf{CUST}_x \\
&\mathsf{CUST}_x \triangleq (\mathsf{payin?y} \rightarrow \mathsf{CUST}_{x+y} \\
&\qquad\quad \mid \mathsf{withdraw?y} \rightarrow \mathsf{CUST}_{x-y} \ \{x \geq y\} \ (\mathsf{overdraft!}(y, x-y) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathsf{CUST}_{x-y}) \\
&\qquad\quad \mid \mathsf{terminate} \rightarrow \mathsf{STOP} \\
&\qquad\quad \mid \mathsf{report!x} \rightarrow \mathsf{CUST}_x)
\end{aligned}$$

Process List uses a compound channel name i.report to input from the channel report of the customer whose name is i

$$\mathsf{List} \triangleq \mu X.(\mathsf{in?L} \rightarrow (\|_{i \in L}(\mathsf{i.report?x} \rightarrow \mathsf{print!}(i,x) \rightarrow \mathsf{SKIP})); X)$$

However this definition of List suffers from the problem of deadlock: a customer in

the list L may choose to perform the terminate event and stop before List has input this customer's balance. The problem can be solved by modelling reality more closely, taking into account the opening and closing of the bank for customers. The balance list is generated during the closed hours of the bank when no customer can engage in any of the actions invest, pay-in, withdraw or terminate. Two new events open and close are added to the alphabet of List which is rewritten as

```
List ≙ μX.(close → μY.(in?L → (⫴_{i∈L}(i.report?x → print!(i,x)
                                                  → SKIP);Y)
                          | open → X))
```

The effect of close and open on customers can be formalised by including another process OC with the following alphabet

```
αOC = {i.invest.x, i.payin.x, i.withdraw.x, i.terminate,
                          i.report.x | i≥0, x ∈ N} ∪ {open, close}
```

Process OC monitors the actions of all customers by jointly participating in them

```
OC ≙ O
O ≙ ([]_{i≥0}(i.invest.x → O
          | i.payin.x → O
          | i.withdraw.x → O
          | i.terminate → O
          | i.report.x → O)
     | close → C)
C ≙ ([]_{i≥0}(i.report.x → C)
     | open → O)
```

When the bank is open a customer may engage in any of the common events, but when it is closed OC permits only the common event i.report.x for customer i. This ensures that no customer may operate his/her account during closed hours. Notice that the communication event on channel overdreft is private to αCustomer and the customer process may still perform this event during closed hours. OC runs in parallel with the customer processes and List, and guarantees that List will not deadlock

```
Bank ≙ (⫴_{i≥0} i:Customer) ‖ List ‖ OC
```

3.2 Customers with Many Accounts

The model of the bank described in the previous sections considers customers with only one account (assuming that the same customer does not have two different names). In general, a customer may operate many accounts concurrently. Such a customer may be modelled employing the same technique used in section 3 to represent many customers in a bank, if the accounts of a customer are also identified by a name or number. Each account of customer i is now given an identity n

$$ManyAccCustomer \triangleq (\|_{n \geq 0}\ n{:}Customer)$$

Process Customer remains as before while Bank (without the functions) becomes

$$Bank \triangleq (\|_{i \geq 0}\ i{:}ManyAccCustomer)$$

Note that each action of a customer now has two labels, i for the identity of the customer and n for his/her account: the event $20.3.withdraw.x$ corresponds to customer with identity 20 withdrawing amount x from his/her account number 3.

4. Ruritanian Army

Jackson introduces the example of the *Ruritanian Army* to illustrate a form of concurrency where the sequential behaviour of one entity has several aspects. The ordering constraints on the events in its life are such that more than one Jackson *structure diagram* (refer appendix) is needed. In this section we show how the $\|$ combinator of CSP nicely solves this problem.

4.1 First Version

Jackson presents two versions of the problem. The salient features of the first version are:

> The Ruritanian Army has only three ranks: Private, Captain and General. On enlisting, a soldier becomes a Private and works his way up the hierarchy. Soldiers at all ranks may need to attend courses. All soldiers enrolled in a course complete it successfully. Promotions are given only between courses.

Four events are adequate to describe the behaviour of a soldier in the Ruritanian Army

$$\alpha Soldier = \{enlist, enrol, complete, promote\}$$

A straightforward, but inelegant, definition for the process Soldier can be given by mutual recursion, using one equation for each of the ranks

$$Soldier \triangleq enlist \rightarrow Private$$
$$Private \triangleq enrol \rightarrow complete \rightarrow Private \mid promote \rightarrow Captain$$
$$Captain \triangleq enrol \rightarrow complete \rightarrow Captain \mid promote \rightarrow General$$
$$General \triangleq enrol \rightarrow complete \rightarrow General$$

Notice that a soldier can get only two promotions, which always occur between courses. As enrol is a possible initial event for all three processes, Private, Captain and General, Soldier satisfies the requirement of possibly attending any number of courses at all ranks.

The mutually recursive solution can be modularised by observing that the life of a soldier has two aspects: his course career and his promotion career. These two careers may themselves be represented as processes evolving in parallel: process Course for his course career and process Rank for his promotion career. Since a soldier may have to attend many courses at any rank, process Course becomes a recursive process with only two events in its alphabet, enrol and complete

$$Course \triangleq \mu X.(enrol \rightarrow complete \rightarrow X)$$

As there are only three ranks in the army, a soldier can get at most two promotions and we reflect this by using STOP in the definition of Rank

$$Rank \triangleq promote \rightarrow promote \rightarrow STOP$$

It would be nearly correct now to describe the behaviour of a soldier using a parallel combination of Course and Rank as

$$Soldier \triangleq enlist \rightarrow (Course \parallel Rank)$$

Here the alphabets of Course and Rank have no events in common, so when they evolve in parallel, their events are arbitrarily interleaved. This fails to satisfy an aspect of the specification met by the previous solution, namely that a soldier may be

promoted only between two courses and not in the middle of a course. To satisfy this requirement, we use another process Life whose alphabet is a union of the alphabets of Course and Rank. Its effect is to prevent promotion from occuring between enrolment and completion

$$Life \triangleq \mu X.(enrol \rightarrow complete \rightarrow X \mid promote \rightarrow X)$$

The process Soldier can now be correctly defined as the parallel combination of the two processes Life and Rank preceded, of course, by enlistment

$$Soldier \triangleq enlist \rightarrow (Life \parallel Rank)$$

4.2 Second Version

In the second version of the problem, five ranks are introduced in the army: Private, Acting Captain, Captain, Acting General and General in that order of hierarchy. Once again a soldier has to do courses, but he may now have to re-enrol in a course due to unsatisfactory performance. However, the army believes that it is not necessary to repeat a course more than once. Finally, promotions may occur at any time.

We include another event reenrol in the alphabet of Course to describe the requirement of repeating a course which the soldier has failed. A soldier may, after enrolling in a course, either complete it, or re-enrol in the course and subsequently complete it. The course career of a soldier in this version becomes

$$Course \triangleq \mu X.(enrol \rightarrow (complete \rightarrow X \mid reenrol \rightarrow complete \rightarrow X))$$

Since the army now has five ranks, the number of promotions that a soldier may obtain increases to four. Process Rank of the previous version can be suitably altered to reflect this increase in number of promotions

$$Rank \triangleq promote \rightarrow promote \rightarrow promote \rightarrow promote \rightarrow STOP$$

Since promotions can occur at any time, we do not require the process Life of the previous section. Soldier is therefore specified as

$$Soldier \triangleq enlist \rightarrow (Rank \parallel Course)$$

It is possible to express the solution for the second version of the problem also as a set of mutually recursive equations. But now there are sixteen states, so sixteen equations are necessary to specify the problem, which makes such an exercise less attractive.

4.3 Final Version

As a final concession to realism we recognise that a soldier's career can be prematurely terminated due to the event death. Whatever the current status of the process Soldier, this event causes its termination. Our model of the Ruritanian Army (for either version) can be easily extended to reflect such an eventuality by using the interrupt operator ^. We show this for the second version of Soldier

$$MortalSoldier \triangleq enlist \rightarrow ((Rank \parallel Course)\hat{}(death \rightarrow SKIP))$$

Since a person becomes a soldier in the army only after enlisting, the above model deliberately ignores the possibility of a person dying before the event enlist.

5. The Daily Racket Competition

The statement of the *Daily Racket* problem is reproduced from Jackson [J]

> To boost circulation, the *Daily Racket* plans to run a competition open to subscribing readers. Once a reader has become a subscriber, he may enter the competition as often as he wishes, sending in one or more entries on each occasion that the newspaper publishes details of the competition. Each entry must be accompanied by an entry fee. The competition is judged periodically, by a panel of television celebrities, and the best entries received since the preceding judgement are awarded prizes....
>
> Some hidden rules are operated, designed to simplify the task of the judging panel, who are not very clever. No competitor can win more than once; no more than one entry from each competitor is submitted for judging in any one session of the panel. Entries which cannot win because of these rules are not returned to the competitors; instead the accompanying fees are retained by the *Daily Racket*, and the entries are quietly ignored. The editor's decision is final.

Each participant in the competition is represented by a process Reader which has only two events, subscribe and enter, in its alphabet. The only action of a participant after subscribing to the publication is to keep sending entries to the competition. Submission describes the process of submitting a single entry

$$\mathsf{Submission} \mathrel{\hat{=}} \mathsf{enter} \rightarrow \mathsf{SKIP}$$

As a participant may submit any number of entries process Reader is written using the indexed ; operator with Submission

$$\mathsf{Reader} \mathrel{\hat{=}} \mathsf{subscribe} \rightarrow (;_{i \geq 0}\ i{:}\mathsf{Submission})$$

where each entry is indexed by its serial number i. Notice that Reader is a non-terminating process and the hidden rules of the competition do not affect it in any way. The very fact that the rules are hidden indicates that process Reader should not be aware of them.

The existence of many participants in the competition is conveniently expressed by the indexed $\|$ combinator as

$$(\|_{n \geq 0}\ n{:}\mathsf{Reader})$$

where n gives the identity of the participant.

The process Panel models the behaviour of the panel which meets periodically and judges entries received and then disperses. We introduce a separate process Meeting to describe the behaviour of the panel during a session

$$\mathsf{Panel} \mathrel{\hat{=}} \mathsf{meet} \rightarrow \mathsf{Meeting}$$

During a judging session Panel receives an identification for the single chosen entry for participant n through channel n.entry. For each participant n who has an entry for the current session the judging activity of the panel is

$$\mathsf{n.entry?e} \rightarrow (\mathsf{n.win} \rightarrow \mathsf{Meeting} \mid \mathsf{n.reject} \rightarrow \mathsf{Meeting})$$

where the panel awards a prize by the event win and discards an entry by reject. Including the event disperse to signify the conclusion of a meeting

$$\begin{aligned}\mathit{Meeting} \;\hat{=}\; (&\Box_{n\geq 0}\ (n.entry?e \rightarrow (n.win \rightarrow \mathit{Meeting}\\ &\qquad\qquad \mid n.reject \rightarrow \mathit{Meeting}))\\ &\mid disperse \rightarrow Panel)\end{aligned}$$

We have not yet used the hidden rules of the competition in defining Panel, and it remains to show how the entries are communicated to it according to these rules. Though a reader may submit many entries between two judging sessions only one entry per reader is chosen for the panel's viewing. We use a process Clerk, one for each Reader, to pick the entry which will be considered at the next meeting of the panel. Initially this will be the the first entry submitted after subscribing; but later it will be the first entry after failing to win a prize.

The process Clerk carries the number of the chosen entry in its state and picks an entry i after the event i.enter which is common to the alphabets of Clerk and Reader

$$\mathit{Clerk} \;\hat{=}\; (\Box_{i\geq 0}\ i.enter \rightarrow C_i)$$

Once it has chosen an entry, Clerk accepts and ignores all subsequent entries until the panel meets and asks for the chosen entry. To ensure that a participant wins only one prize in his/her lifetime, the process Clerk also needs to know of the panel's decision for the chosen entry. We achieve this by making

$$\{n.win, n.reject\} \subseteq \alpha(n{:}Clerk)$$

Clerk never submits an entry to the panel after a win but it continues receiving further entries. The behaviour of Clerk after choosing an entry i is either to accept and ignore a new entry, or submit the chosen entry to the panel, and take appropriate action on the panel's decision

$$\begin{aligned}C_i \;\hat{=}\; \mu X.((&\Box_{j\geq 0}\ j.enter \rightarrow X)\\ &\mid entry!i \rightarrow (win \rightarrow \mu Y.(\Box_{j\geq 0}\ j.enter \rightarrow Y)\\ &\qquad\qquad \mid reject \rightarrow Clerk))\end{aligned}$$

Neither the process Panel nor the process Reader reflects the hidden rules. The process Clerk which serves as a link between the panel and the participants is the only one responsible for the hidden rules of the competition. The complete system can now be built using the $\|$ operator with one Reader and one Clerk process for each participant, and a process for the panel

$$System \triangleq (\|_{n\geq 0}\ n{:}(Reader \parallel Clerk)) \parallel Panel$$

The most unrealistic aspects of this model are (1) there seem to be as many clerks as readers, and (2) the readers' act of submitting an entry occurs simultaneously with that of the clerk receiving it. The first problem is easily solved: in CSP parallel composition is symmetric and associative, so the solution quoted is identical to

$$(\|_{n\geq 0}\ n{:}Reader) \parallel (\|_{n\geq 0}\ n{:}Clerk) \parallel Panel$$

Here it can be quite reasonably understood that a single clerk might carry out all the tasks described by an apparently unbounded array of processes.

The second problem is solved by changing the action enter in $\alpha Reader$ to sendentry (leaving the event enter to stand for receipt of the entry by the process Clerk), and then by interposing a process which models the postal service. The new reader is defined

$$NReader \triangleq f(Reader)$$
$$\text{where } f(subscribe) = subscribe$$
$$f(i.enter) = i.sendentry \qquad \text{for } i\geq 0$$

The post office can be modelled by a standard buffer, which stores in its state the sequence of undelivered entries

$$PO \triangleq P_{\langle\rangle}$$
$$P_{\langle\rangle} \triangleq (\|_{n,i\geq 0}\ n.i.sendentry \rightarrow P_{\langle n.i\rangle})$$
$$P_{\langle n.i\rangle^\frown s} \triangleq (\|_{m,j\geq 0}\ m.j.sendentry \rightarrow P_{\langle n.i\rangle^\frown s^\frown\langle m.j\rangle}$$
$$\mid n.i.enter \rightarrow P_s)$$

The system is now

$$(\|_{n\geq 0}\ n{:}NReader) \parallel PO \parallel (\|_{n\geq 0}\ n{:}Clerk) \parallel Panel$$

In practice, the mail service can reorder the messages it receives and deliver them in an order different from that in which they were posted. In practice also the clerk will have to reject any entries sent before but received after the specified closing date for each panel meeting. A solution to these problems can be formulated in CSP, but introduction of the mail service also introduces non-determinism, a complexity we have decided to avoid in this paper.

5.1 Adding Functions

In this section we extend the processes of the previous section to include the functions added to the system in [J]. Inevitably, we need to modify the processes by extending their alphabet and sometimes we even introduce new processes.

The simplest function to be added to the system is the one corresponding to

(1) Acknowledge each entry received

Either Reader or Clerk (or both) can be extended to provide this function. We do it for Reader by adding the event ack to Submission

$$\begin{aligned} &\mathsf{Submission} \mathrel{\hat{=}} \mathsf{enter} \rightarrow \mathsf{ack} \rightarrow \mathsf{SKIP} \\ &\mathsf{Reader} \mathrel{\hat{=}} \mathsf{subscribe} \rightarrow (\$_{i \geq 0}\ i{:}\mathsf{Submission}) \end{aligned}$$

If entries and acknowledgements are buffered, this simple solution is inadequate. The next two functions to be added require the introduction of new processes that store information in their state

(2) On request, list the number of entries for each reader received so far.

(3) Print the total number of entries for each week and the cumulative total over the weeks along with the current week number.

Since a count of the number of entries is to be kept for these two functions, the processes introduced should participate in the event i.enter of Reader. For function (2), each Reader process has another process ReaderSum which stores in its state the cumulative count of entries sent by this reader. The "on request" part of this function is captured by the fact that ReaderSum sends the value on channel out whenever its environment is willing to accept it

$$\begin{aligned} &\mathsf{ReaderSum} \mathrel{\hat{=}} \mathsf{RSUM}_0 \\ &\mathsf{RSUM}_x \mathrel{\hat{=}} ((\mathbin{[\!]}_{i \geq 0}\ i.\mathsf{enter} \rightarrow \mathsf{RSUM}_{x+1}) \\ &\qquad\qquad \mathbin{[}\ \mathsf{out}!x \rightarrow \mathsf{RSUM}_x) \end{aligned}$$

For function (3) we use a process WeekSum which stores three values in its state, the week number (w), cumulative sum of entries across weeks (x) and the total number of entries received in the current week from all readers

(c). WeekSum also participates in the enter event, but we now need two labels as qualifiers for this event: n (for readers) and i (for the entries of reader n). The printing of the values is caused by the event weekend

$$\begin{aligned}&\mathit{WeekSum} \triangleq \mathit{Sum}_{1,0,0}\\&\mathit{Sum}_{w,x,c} \triangleq (\mathit{weekend} \rightarrow \mathit{print}!(w, x+c, c) \rightarrow \mathit{Sum}_{w+1,x+c,0}\\&\qquad\qquad \Box\ (\Box_{n,i\geq 0}\ n.i.\mathit{enter} \rightarrow \mathit{Sum}_{w,x,c+1}))\end{aligned}$$

We finally add two more functions

(4) Print all entries chosen each week for the panel's viewing.

(5) The panel should produce a list of its results.

Printing of the list of chosen entries on a channel list is the responsibility of a new eavesdropping process L, which listens to communications on the channel entry as they pass between the Clerk process and the Penel. These events therefore occur with the participation of three processes; this is in full accordance with the definition of the ‖ combinator in CSP, though it is not a feature to be lightly included in a programming language

$$\begin{aligned}&\alpha L = \{n.\mathit{entry}.i \mid n, i\geq 0\}\\&L \triangleq (\Box_{n,i\geq 0}\ n.\mathit{entry}.i \rightarrow \mathit{list}!(n,i) \rightarrow L)\end{aligned}$$

The reports required from the panel are got by extending the alphabet of Penel to include communication events which use channels winlist and rejectlist to list the winning and rejected entries respectively. The values output are the identity of the reader and serial number of the entry. A simple change to Meeting provides the required reports

$$\begin{aligned}&\mathit{Meeting} \triangleq (\Box_{n\geq 0}\ (n.\mathit{entry}?e \rightarrow\\&\qquad\qquad (n.\mathit{win} \rightarrow \mathit{winlist}!(n,e) \rightarrow \mathit{Meeting}\\&\qquad\qquad\ \Box\ n.\mathit{reject} \rightarrow \mathit{rejectlist}!(n,e) \rightarrow \mathit{Meeting}))\\&\qquad \Box\ \mathit{disperse} \rightarrow \mathit{Panel})\\&\mathit{Panel} \triangleq \mathit{meet} \rightarrow \mathit{Meeting}\end{aligned}$$

The system after the addition of all the above functions becomes

$$(\|_{n\geq 0}\ n{:}(\mathit{Reader} \parallel \mathit{Clerk} \parallel \mathit{ReaderSum})) \parallel \mathit{Panel} \parallel L \parallel \mathit{WeekSum}$$

6. Widget WareHouse System

This example develops a system for the allocation of product stock to customer orders of the Widget Warehouse Company. The problem statement from [J]

> The company's customers order products from the company, often by telephone but sometimes by other means such as mail or personal visit to the company's warehouse. There is a company rule that separate orders are required for separate products...
>
> Customers sometimes amend their orders, changing the quantity or the requested delivery date. Occasionally a customer may cancel an order.
>
> The company employs a clerk whose job is to deal with the customers and to allocate the available stock to outstanding orders. This clerk has access to information about the available stock of each product. This enquiry is usually answered with reasonable reliability... We will be developing only the sales system, handling customer orders.

Jackson presents two solutions to this problem, a non-automated and an automated system. Here we shall present only the automated version where we include processes for orders, products and to perform allocation of stock. We introduce a simplification by ignoring amendment of requested delivery dates.

Consider the possible events in the life of an order placed by a customer. After being placed, it may be amended or cancelled; and if the product is allocated to the customer's satisfaction it may then be delivered. We choose four events for the process Order

place	place an order for a product
amend	amend the quantity of an order
cancel	cancel the order
deliver	deliver the order

Since allocation of an order depends on the size of the order, it is necessary to store this value as the state of the process that models the behaviour of an order. Of the four events in the alphabet of Order, we model placing and amending as communications of the relevant quantity.

place.x	place an order of size x
amend.y	amend an order to size y

A very simple definition of Order may be given without saying anything about the allocation or delay due to unavailability of stock

$$\begin{aligned} &Order \triangleq place?x \rightarrow ORD_x \\ &ORD_x \triangleq (amend?y \rightarrow ORD_y \\ &\qquad \mid cancel \rightarrow STOP \\ &\qquad \mid deliver \rightarrow STOP) \end{aligned}$$

Since the company deals with many customers and each customer places orders for many products we shall use two labels, p (for product) and c (for customer), with process Order

p:c:Order	order for product p by customer c

We have, for ease of presentation, restricted orders to one per product per customer. But our solution can be easily extended to multiple orders by using another label i for the ith order for product p by customer c.

In order to allocate stock it is necessary to have access to the stock status information which may itself be modelled as a process Product (one per product p). Current stock status is stored in the state of Product. The environment communicates delivery of fresh stock to this process through channel fresh. Product sends the current stock status to its environment on channel stock, and then expects input on channel supply of the quantity of items taken from stock

$$\begin{aligned} &Product \triangleq P_0 \\ &P_x \triangleq (stock!x \rightarrow supply?y \rightarrow P_{x-y} \\ &\qquad \mid fresh?q \rightarrow P_{x+q}) \end{aligned}$$

To link the product with the order we now design an allocator process, a separate one for each product the company supplies. Depending on the stock availability process Allocator may either delay an order or allocate the quantity requested for. With the addition of the allocator process, the simple model of an order given earlier becomes inadequate, as the following three events (common with the allocator process) must be added to αOrder

allocate	allocation of an order
delay	delay an order due to unavailability of stock
howmuch.x	communication event by which allocator process finds out the size of the order

Process Order is rewritten for interaction with the allocator process

$$\begin{aligned}
&\mathtt{Order} \triangleq \mathtt{place?x} \rightarrow \mathtt{ORD}_x \\
&\mathtt{ORD}_x \triangleq (\mathtt{amend?y} \rightarrow \mathtt{ORD}_y \\
&\qquad \Box\ \mathtt{cancel} \rightarrow \mathtt{STOP} \\
&\qquad \Box\ \mathtt{howmuch!x} \rightarrow (\mathtt{allocate} \rightarrow \mathtt{deliver} \rightarrow \mathtt{STOP} \\
&\qquad\qquad\qquad \Box\ \mathtt{delay} \rightarrow \mathtt{ORD}_x))
\end{aligned}$$

Notice that the above model of an order has introduced a constraint on customers: once allocated a customer is forbidden from cancelling or amending an order. Finally, the allocator process (one per product) may be written as

$$\begin{aligned}
&\mathtt{Allocator} \triangleq \mu X.(\Box_{c \in Cp}\ (\mathtt{c.howmuch?x} \rightarrow \mathtt{stock?y} \rightarrow \\
&\qquad\qquad\qquad\qquad (\mathtt{c.allocate} \rightarrow \mathtt{supply!x} \rightarrow X) \\
&\qquad\qquad\qquad\qquad\quad \natural y \geq x \natural \\
&\qquad\qquad\qquad\qquad (\mathtt{c.delay} \rightarrow \mathtt{supply!0} \rightarrow X)))
\end{aligned}$$

where Cp is the set of all customers. Allocator guarantees that process Product will not have any inconsistent values (such as negative values) of stock status. Whenever it delays an order, Allocator sends the value 0 to Product as the quantity supplied. The complete system for this version is

$$\mathtt{System} \triangleq (\|_{p \in Pr,\ c \in Cp}\ \mathtt{p:c:Order}) \parallel (\|_{p \in Pr}\ \mathtt{p:(Product} \parallel \mathtt{Allocator)})$$

where Pr is the set of all products supplied by the company and Cp is the set of all customers.

6.1 Adding Functions

To the solution of the previous section we shall add the following function

> For a specified product-id, list the names of all customers who have outstanding orders for that product, and, against the name of each such

customer, the total quantity outstanding. An order is outstanding if it has been placed but not yet allocated or cancelled.

To provide this listing a new process ProductList which communicates with Order to obtain the ordered quantity is added to the system. Once again we are forced to modify Order to communicate the quantity along channel size to ProductList

```
Order ≙ place?x → ORD_x
ORD_x ≙ (amend?y → ORD_y
        | cancel → STOP
        | howmuch!x → (allocate → deliver → STOP
                      | delay → ORD_x)
        | size!x → ORD_x)
```

The process for listing all customer orders for a given product is

```
ProductList ≙ μX.(in?p → (||_{c∈L}(p.c.size?x → print!(c,x)
                                                → SKIP));X)
```

where L is the list of all customers with outstanding orders for product p. This raises the problem: how does this process discover which customers are in the list L? Jackson solves the problem by assuming that the information is made available in some underlying data base. We can model this by an eavesdropping process, which participates in the events

```
{place.x, cancel, deliver, send.L}
```

and stores a set B of all customers who have outstanding orders, *i.e.* placed an order but not yet cancelled or delivered it. This process Spy also outputs a serialised list L of the set B on channel send whenever required

```
Spy ≙ S_{}
S_B ≙ (||_{c∈Cp}(c.place.x → S_{B ∪ {c}}
              | c.cancel → S_{B - {c}}
              | c.deliver → S_{B - {c}}
              | send!list(B) → S_B))
```

where list(B) is a serialised list of the set B. Process ProductList is rewritten to

communicate with the process Spy to obtain the list of customers with outstanding orders

$$\mathtt{ProductList} \triangleq \mu X.(\mathtt{in?p} \rightarrow \mathtt{p.send?L} \rightarrow (\natural_{c \in L}(\mathtt{p.c.size?x} \rightarrow \mathtt{print!(c,x)} \rightarrow \mathtt{SKIP})); X)$$

The system with the addition of this function becomes

$$\mathtt{System} \triangleq (\|_{p \in Pr,\ c \in Cp}\ \mathtt{p:c:Order}) \| (\|_{p \in Pr}\ \mathtt{p:(Product} \| \mathtt{Allocator} \| \mathtt{Spy})) \| \mathtt{ProductList}$$

where Pr is the set of all products supplied by the company and Cp is the set of all customers.

The process ProductList is still unsatisfactory as it suffers from a deadlock problem similar to the function process List used in the bank example. A solution may be formulated along the same lines as in section 3.1.

7. Elevator Problem

Problem statement from Jackson [J]

> The Hi-Ride Elevator Company is installing elevators in a small building of six floors. At each floor, except the top floor, there is a button which users can press to summon an elevator to take them upwards; at each floor, except the ground floor, there is a similar button for downwards travel. Inside each elevator there are six buttons marked with floor numbers. There is a pair of doors at each floor, and another pair on each elevator. The elevators are raised and lowered by cables which are wound and unwound by motors positioned above the top floor. At each floor, in each elevator shaft, there is a sensor operated by a small wheel attached to the elevator: when the elevator is within 15 cms of the home position at that floor, the sensor is depressed by the wheel and closes an electrical switch...
>
> The computer system will schedule the travel of the elevators according to the users' requests for service, and will produce commands for the motors and the lights which are associated with the buttons. In the usual way, when a button is pressed, the associated light must be turned on.

We shall represent buttons and lift as processes in the system. Consider the process Button. Once the button is depressed, the light is to be turned on and is to be turned off when the request has been serviced. In addition to these three events, depress, lighton and lightoff, the process Button also communicates with the lift process sending the value 1 when the button has been pressed, 0 otherwise

$$\mathsf{Button} \triangleq \mu X.(\mathsf{lift!0} \rightarrow X \mid \mathsf{depress} \rightarrow \mathsf{lighton} \rightarrow \mathsf{lift!1} \rightarrow \mathsf{lightoff} \rightarrow X)$$

Note that the light goes off automatically when the lift reads the value 1.

There are three kinds of buttons in the system: buttons on floors which control the upward motion of the lift, buttons on floors which control the downward motion of the lift and buttons inside the elevator. We can use the process naming operator with the obvious labels {u, d, e} to denote these three types of buttons in the system. The corresponding processes are

u:Button processes for buttons on floors controlling upward motion of elevator
d:Button processes for buttons on floors controlling downward motion of elevator
e:Button processes for buttons inside the elevator

These processes have to be further qualified by the floor they relate to - buttons inside the elevator also correspond to a particular floor. We shall use another label i where i is the floor number. For n floors, and

for $1 \leq i < n$ i:u:Button
for $1 < i \leq n$ i:d:Button
for $1 \leq i \leq n$ i:e:Button

Buttons inside the elevator can be further distinguished by the elevator within which they are located. We shall for the moment ignore the second elevator and develop a solution for a single elevator system. Initially we simplify the problem by ignoring the buttons inside the elevator, and sacrifice efficiency by assuming that the elevator always travels from the ground floor to the top floor and back servicing any requests in intermediate floors.

7.1 Perpetual Motion: No Buttons inside Elevator

Consider the behaviour of the elevator in any of the intermediate floors. Impending arrival at a floor is detected by the sensor in the elevator shaft and the corresponding event is `atfloor`. After communication with the floor button the elevator decides whether to stop at the floor or not

```
Floor ≙ atfloor → floorbutton?x → SKIP ⨽x=0⨾ (halt → dir
                                                → SKIP)
```

After a delay it starts the motor by the event `dir` which sets the motor polarity according to the chosen direction of motion. We have ignored the events of opening and closing the doors.

For the intermediate floors we can define two processes $\mathtt{Upward}_i$ and $\mathtt{Downward}_i$ using direct image operators with the process `Floor`

$$\mathtt{Upward}_i = f_{u,i}(\mathtt{Floor}) \qquad \text{for } 1 < i < n$$
$$\mathtt{Downward}_i = f_{d,i}(\mathtt{Floor}) \qquad \text{for } 1 < i < n$$

The alphabet transformations for the functions $f_{u,i}$ and $f_{d,i}$ are:

$f_{u,i}$(floorbutton.x) = i.u.lift.x
$f_{u,i}$(dir) = up
$f_{u,i}$(atfloor) = atfloor
$f_{u,i}$(halt) = halt

$f_{d,i}$(floorbutton.x) = i.d.lift.x
$f_{d,i}$(dir) = down
$f_{d,i}$(atfloor) = atfloor
$f_{d,i}$(halt) = halt

The behaviour of the lift at the top and ground floors is slightly different as it always stops at these two floors and reverses direction, and is described by process `TerminalFloor`

```
TerminalFloor ≙ atfloor → floorbutton?x → halt → dir → SKIP
```

The processes corresponding to the behaviour of the lift at the ground and top floors can be defined from `TerminalFloor` using the direct image operators $f_{u,i}$ and $f_{d,i}$ setting $i = 1$ and $i = n$ respectively

$$\mathtt{Ground} = f_{u,1}(\mathtt{TerminalFloor})$$
$$\mathtt{Top} = f_{d,n}(\mathtt{TerminalFloor})$$

One complete motion of the elevator where it starts from the ground floor, travels to the top floor servicing any requests on the way and returns back can be described in terms of the processes defined earlier and sequential composition

$$\mathsf{UpDownLift} \triangleq \mathsf{Ground};(\sharp_{1<i<n}\mathsf{Upward}_i);\mathsf{Top};(\sharp_{1<i<n}\mathsf{Downward}_{n-i+1})$$

We can now construct an elevator which relentlessly keeps going up and down irrespective of the existence or absence of requests

$$\mathsf{Elevator} \triangleq *(\mathsf{UpDownLift})$$

This runs in parallel with the external buttons, which control its stopping at each floor

$$\mathsf{System} \triangleq \mathsf{Elevator} \parallel \mathsf{ExtButtons}$$
$$\mathsf{ExtButtons} \triangleq (\|_{1\leq i<n}\ i{:}u{:}\mathsf{Button}) \parallel (\|_{1<i\leq n}\ i{:}d{:}\mathsf{Button})$$

7.2 Perpetual Motion: Buttons inside Elevator

If we now introduce buttons within the elevator, the definitions of the various processes remain similar in spirit to those given earlier except that at each floor two buttons have to be checked: one on the floor and one inside the elevator. Floor and TerminalFloor have to be suitably altered by introducing another communication event elevbutton.y

$$\mathsf{Floor} \triangleq \mathsf{atfloor} \rightarrow \mathsf{floorbutton?x} \rightarrow \mathsf{elevbutton?y} \rightarrow \mathsf{SKIP}\ \sharp x{=}y{=}0\sharp\ (\mathsf{halt} \rightarrow \mathsf{dir} \rightarrow \mathsf{SKIP})$$
$$\mathsf{TerminalFloor} \triangleq \mathsf{atfloor} \rightarrow \mathsf{floorbutton?x} \rightarrow \mathsf{elevbutton?y} \rightarrow \mathsf{halt} \rightarrow \mathsf{dir} \rightarrow \mathsf{SKIP}$$

The alphabet transformations $f_{u,i}$ and $f_{d,i}$ have to be augmented to include elevbutton.y while the transformations for the other events in the alphabet remain the same as before

$$f_{u,i}(\mathsf{elevbutton.y}) = \mathsf{i.e.lift.y} \qquad f_{d,i}(\mathsf{elevbutton.y}) = \mathsf{i.e.lift.y}$$

The processes Ground, Top, Upward_i and $\mathsf{Downward}_i$ are defined similar to the earlier versions but using the new definitions of Floor and TerminalFloor. UpDownLift and Elevator also remain as before, but use the new definitions of the processes describing floor behaviour. Process Elevator now runs in parallel with

Buttons which is itself a parallel combination of all the three types of button processes in the system

$$System \triangleq Elevator \parallel Buttons$$
$$Buttons \triangleq ExtButtons \parallel (\parallel_{1 \leq i \leq n} i{:}e{:}Button)$$
$$ExtButtons \triangleq (\parallel_{1 \leq i < n} i{:}u{:}Button) \parallel (\parallel_{1 < i \leq n} i{:}d{:}Button)$$

7.3 Elevator System

The next attempt at a solution to the elevator problem should eliminate the inefficiency introduced by unnecessary travel to top and bottom floors. In this section we only indicate how this may be done without presenting a solution.

The elevator is to normally wait at the ground floor. If there is a request from any of the floors above, the elevator journeys upwards to service the request. On any occasion during its upward travel the elevator may reverse direction and move downwards if it finds that there are no pending requests in any of the floors above. Similarly, during its downward travel it may reverse direction and move upwards if there are no pending requests in the floors below and there is a request from one of the floors above.

With these additional requirements, the elevator has to poll the buttons above (or below) the current floor to choose the direction of motion. The behaviour of the elevator at the intermediate floors and the ground floor can be split into a polling process and a floor process. As it is to normally wait at the ground floor, it need not do any polling at the top floor.

A solution for the single elevator system along the lines indicated loses much of the simplicity and elegance of the solutions presented in the earlier two sections. This is perhaps inevitable for we are trying to represent a real life situation where local decisions must be made on the basis of global information. A solution with centralised, instead of distributed, control seems more suitable.

For a two elevator system, [J] suggests a solution in terms of "promises" by an elevator to service all requests between the current floor and, the top floor or ground floor depending on direction of travel. The complications of an efficient solution for a single elevator system deter us from attempting such a solution in CSP.

8. Implementation

Since most of our solutions have a large number of processes that do very little computing, it would be highly uneconomical to provide dedicated processors for each process in the solutions. Hence the issue of providing efficient implementations, on a conventional sequential processor, of these parallel solutions becomes important.

One possible approach is to use a parallel programming language such as OCCAM [In] implemented on a conventional sequential processor. In principle it is possible to implement our processes as OCCAM processes though an optimisation phase seems to be required to reduce the number of processes. In fact the bank and a part of the elevator examples have been implemented in OCCAM from the JSD solutions [F].

But a more general solution would be to use the theoretical framework of CSP to obtain an efficiently implementable version of the parallel one by algebraic transformations. The laws governing the various CSP operators [H2] can be used to do these algebraic transformations to reduce parallel systems to ones written using mutual recursion, which can be implemented efficiently on a conventional sequential processor. In this section, we pursue the algebraic transformation approach to derive a solution using mutual recursion starting from a highly parallel one. The example chosen is the first version of the Ruritanian Army problem (section 4.1).

We use the following two laws defined on the $\|$ operator for the algebraic transformations (Section 2.3.1, [H2])

If $x \in \alpha P - \alpha Q$, $y \in \alpha Q - \alpha P$ and $z \in \alpha P \cap \alpha Q$, then

$$\text{L1}\quad (x \rightarrow P) \parallel (z \rightarrow Q) = x \rightarrow (P \parallel (z \rightarrow Q))$$
$$\text{L2}\quad (z \rightarrow P) \parallel (z \rightarrow Q) = z \rightarrow (P \parallel Q)$$

Consider the distributed version of the solution given in section 4.1

```
Soldier ≙ enlist → (Life ‖ Rank)  and
Rank ≙ promote → promote → STOP
Life ≙ μX.(enrol → complete → X
          | promote → X)
     = (enrol → complete → Life | promote → Life)
```

For the two processes Life and Rank

$\{enrol, complete\} \subseteq \alpha Life - \alpha Rank$ and $promote \in \alpha Life \cap \alpha Rank$

Let $P_0 \triangleq Soldier$
$= (enlist \rightarrow (Life \parallel Rank))$
$= enlist \rightarrow P_1$
where $P_1 \triangleq Life \parallel Rank$

Expanding P_1, we have

$P_1 = (enrol \rightarrow complete \rightarrow Life \mid promote \rightarrow Life) \parallel (promote \rightarrow promote \rightarrow STOP)$ (definition)
$= enrol \rightarrow ((complete \rightarrow Life) \parallel (promote \rightarrow promote \rightarrow STOP)) \mid promote \rightarrow (Life \parallel (promote \rightarrow STOP))$ (L1,L2)
$= enrol \rightarrow complete \rightarrow (Life \parallel Rank) \mid promote \rightarrow (Life \parallel (promote \rightarrow STOP))$ (L1)
$= enrol \rightarrow complete \rightarrow P_1 \mid promote \rightarrow P_2$ (definition)

$P_2 \triangleq Life \parallel (promote \rightarrow STOP)$
$= (enrol \rightarrow complete \rightarrow Life \mid promote \rightarrow Life) \parallel (promote \rightarrow STOP)$
$= enrol \rightarrow ((complete \rightarrow Life) \parallel (promote \rightarrow STOP)) \mid promote \rightarrow (Life \parallel STOP_{\langle promote \rangle})$ (L1,L2)
$= enrol \rightarrow complete \rightarrow (Life \parallel (promote \rightarrow STOP)) \mid promote \rightarrow (Life \parallel STOP_{\langle promote \rangle})$ (L1)
$= enrol \rightarrow complete \rightarrow P_2 \mid promote \rightarrow P_3$ (definition)

$P_3 \triangleq Life \parallel STOP_{\langle promote \rangle}$
$= (enrol \rightarrow complete \rightarrow Life \mid promote \rightarrow Life) \parallel STOP_{\langle promote \rangle}$
$= enrol \rightarrow ((complete \rightarrow Life) \parallel STOP_{\langle promote \rangle})$ (L1)
$= enrol \rightarrow complete \rightarrow (Life \parallel STOP_{\langle promote \rangle})$ (L1)
$= enrol \rightarrow complete \rightarrow P_3$ (definition)

We have now obtained the following four equations starting from the parallel solution

$P_0 = enlist \rightarrow P_1$
$P_1 = enrol \rightarrow complete \rightarrow P_1 \mid promote \rightarrow P_2$
$P_2 = enrol \rightarrow complete \rightarrow P_2 \mid promote \rightarrow P_3$
$P_3 = enrol \rightarrow complete \rightarrow P_3$

Not surprisingly these four equations are identical to the mutual recursive version of the solution presented in section 4.1 with

$P_0 = Soldier$, $P_1 = Private$, $P_2 = Captein$ and $P_3 = General$

As a slightly more difficult exercise we derive an efficiently implementable solution of the final version of the Ruritanian Army (section 4.3), where the event death is included. The event death can be added to the first version of the Ruritanian Army problem as

```
MortalSoldier ≙ enlist → ((Life ‖ Rank)^(death → SKIP))
```

In addition to laws L1 and L2 we use the following law governing the ^ operator and prefixing (Section 5.4, [H2])

L3 $(x{:}B \rightarrow P(x))^Q = Q \;[]\; (x{:}B \rightarrow (P(x)^Q))$

We obtain five equations for MortalSoldier and we give below the derivation of one of these equations, P_1.

Let $D \triangleq death \rightarrow SKIP$ and
$P_0 \triangleq MortalSoldier = enlist \rightarrow P_1$

where $P_1 = (Life \;‖\; Rank)^D$

In the earlier derivation we have already shown that

```
Life ‖ Rank = (enrol → complete → (Life ‖ Renk)
                | promote → (Life ‖ promote → STOP))
```

Using the above we have

```
P1 = (enrol → complete → (Life ‖ Rank)
       | promote → (Life ‖ promote → STOP))^D
   = D [] (enrol → (complete → Life ‖ Rank)^D
             | promote → (Life ‖ promote → STOP)^D)        (L3)
   = D [] (enrol → (D [] complete → (Life ‖ Rank)^D)
             | promote → (Life ‖ promote → STOP)^D)        (L3)
   = D [] (enrol → (D [] complete → P1 | promote → P2)     (definition)
```

where P_2 is defined as (Life ‖ promote → STOP)^D. Processes P_2 and P_3 may be derived in a similar fashion. The five equations for MortelSoldier are

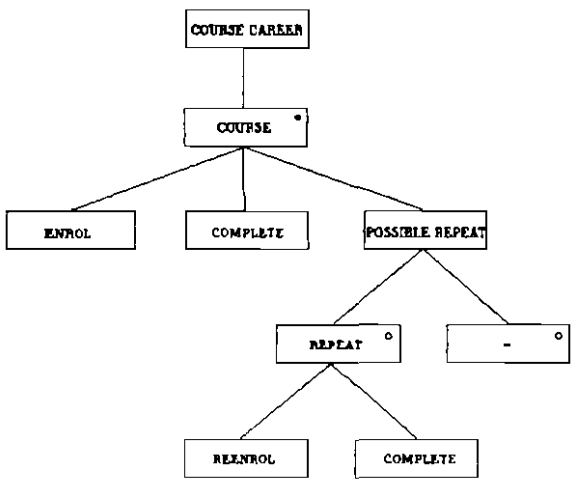
```
P0 = enlist → P1
P1 = D [] (enrol → (D [] complete → P1) | promote → P2)
P2 = D [] (enrol → (D [] complete → P2) | promote → P3)
P3 = D [] (enrol → (D [] complete → P3)
D = death → SKIP
```

Of course, these are very simple examples; yet even so, the derivation of a correct sequential program is a non-trivial calculation, for which some mechanical aid or check would be desirable. More substantial examples, perhaps involving arrays of processes, may present even greater difficulty in formal derivation of sequential programs from parallel ones. Jackson describes by example some practical techniques, but their theoretical counterparts are left for future research.

9. Comparison with JSD

Before we make a general comparison of CSP and JSD we outline the few changes we have made to the examples tackled in [J].

The bank example has no changes. Our second version of the Ruritanian Army (section 4.2) is slightly different from that of [J]. In [J] a soldier completes a course irrespective of whether his performance has been satisfactory or not. In the latter case, he re-enrols in the same course. Jackson's structure diagram for the soldier's course career is



The possible null action in the iteration COURSE results in the following equivalent CSP definition

```
μX.((enrol → complete → X)
    □ (enrol → complete → reenrol → complete → X))
```

However this leads to non-determinism; to avoid this, in our version, a soldier completes a course only if he is successful in it.

In the other three examples we have not introduced any changes to the problems except that our solution for the elevator problem is not complete. In most of the examples Jackson solves more variations of the problem and introduces more functions. The shorter length of this article does not permit us to tackle all his variations. Some of the other issues that we have not addressed include timing constraints, priority, etc.

The conciseness and expressive power of the CSP notation lead to shorter solutions which, it may be argued, are more easily understandable only to those sympathetic to mathematical notations. The majority of the target readership of Jackson's book may find his diagrams (structure diagrams) and English-like notation (structure text) more appealing. In view of this, there perhaps is a case to introduce some of the more useful CSP operators as new box types in structure diagrams, so that larger systems including parallelism can be described pictorially.

The level of abstraction employed by us in giving constructive specifications is somewhat higher than that used by Jackson. Consequently we have been able to ignore *levels* of processes (refer appendix) and different varieties of *process connections* (refer appendix). Both of these concerns become more relevant in the later stages of design and implementation.

Jackson introduces the notion of *marsupial entity* (refer appendix) to describe an entity that is derived from the structure of another entity with many instances of the marsupial existing in the sytem. The parallel and process naming operators together give a formal representation of this notion. This formalisation substantiates the relevance and importance of marsupial entities in system development.

The rich set of laws governing the CSP operators provides us a tool to formally derive efficiently implementable versions of our constructive specifications.

10. Further Work

One of the advantages of a specification is that it helps a designer to formulate and experiment with the design of his system. It becomes possible to make design decisions one at a time in a rational sequence starting with a simple structure and adding details to it. The benefits of such an approach are enhanced if the initial formulation remains unaltered and subsequent decisions just add to it. The solution of the Ruritanian Army problem is a good example of such an approach.

The other examples given here show that we have failed to meet this goal (particularly in the elevator problem). The use of trace descriptions, perhaps in a mixed style of specification [H3, O], might be of help in achieving this goal. The rewriting of constructive specifications for adding functions (as done in this paper) may be avoided by a suitable formalisation of the state vector connection in CSP. Perhaps the use of non-determinism to postpone decisions might lead to more elegant solutions, especially for the elevator problem.

Yet another problem with CSP is the intrusion of deadlock at an early stage in the specification. Further work should also be directed at techniques to avoid deadlock and establish its absence.

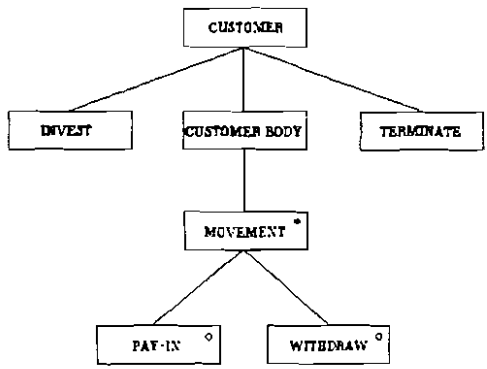
The extension of the suggested algebraic transformation approach, for deriving efficient implementations of distributed programs, to more complicated examples involving arrays of processes needs further study. Some form of mechanical aid may be needed.

APPENDIX

In this appendix we discuss the notational and terminological differences between CSP and JSD to aid readers familiar with only one of the two.

A system is developed in JSD in terms of *entities* specifying their *actions*. Entities are analogous to *processes* of CSP and actions are the *events* in the alphabet of a CSP process. For example, the JSD solution for the bank problem (section 3) would consider an entity type CUSTOMER with four actions INVEST, PAY-IN, WITHDRAW and TERMINATE.

Jackson uses *structure diagrams* and *structured text* to describe JSD entities while we give a process description using CSP operators. The entity CUSTOMER may be described by the following structure diagram



In the above tree diagram the branch nodes represent processes and the leaf nodes represent actions. Some of the nodes (called *boxes*) are marked by an asterisk or circle. A box is a *sequence* if all its children are unmarked with the left to right order of the child boxes indicating sequential composition. In the above diagram CUSTOMER is a sequence with three parts INVEST, CUSTOMER-BODY and TERMINATE. A box is an *iteration* if its child box is marked with an asterisk at the upper right corner; CUSTOMER-BODY is an iteration with zero or more occurrences of MOVEMENT. If circles are used instead of asterisks to mark the child boxes then the parent box is a *selection*. MOVEMENT is a selection, with one occurrence of either WITHDRAW or PAY-IN for every occurrence of it. A null action for a selection is indicated by

marking (as in the structure diagram of section 9) one of the child boxes with a dash. The above diagram may be compared with the equivalent CSP definition of process Customer

```
Customer ≙ invest → μX.(payin → X
                        | withdraw → X
                        | terminate → STOP)
```

Structured text is a textual notation for entity structures. It is a transcribed form of the structure diagram which is more convenient for inserting operations, and conditions for selections, iterations and is used in the later steps of JSD.

1. Process Connections

Jackson uses two types of connections between processes: the *data stream* connection and the *state vector* connection. We have found the synchronous communication of CSP adequate as a process connection for the example problems treated in this paper.

The data stream connection of JSD is simply a buffered communication channel between two processes. read and write statements are used to transmit and receive information from a data stream. The data stream connection can be specified whenever required in CSP notation by placing a buffer process between two communicating processes as shown in section 5 for inserting the postal service.

In the state vector connection of JSD, one process inspects the *state* of another process. The state information carried by a process includes the values of all local variables and the *text-pointer* (which is analogous to program counter). The inspected process does not participate in this form of communication. The initiative for the communication lies entirely with the inspecting process, and consequently neither process gets blocked on a state vector inspection. To avoid problems with consistency of the values obtained in such a process connection, Jackson imposes restrictions on the points at which the inspected process may update its state.

This form of process connection is somewhat like read-only store sharing and it is well known that sharing is not easy to model in CSP [H1, H2]. We do not attempt to accommodate the state vector connection within the notational framework of CSP. Again, due to the higher level of abstraction, we have not felt the need for this form of process connection in CSP for the examples treated in this paper; it is possible that

state-vector inspection would help in treating the more elaborate versions of the elevator problem, which we have omitted.

2. Level 0 and Level 1 Processes

JSD uses two (or more) levels of processes; real world processes which are abstract descriptions of the real world are at level 0; while the model processes which will eventually be run on a machine are at level 1 or higher. The level 0 processes are said to be external to the system boundary. The JSD model processes at level 1 invariably follow a structure which is broadly similar to their counterparts at level 0 but is more complicated. The primary concern in JSD regarding these two levels is the type of process connection used. The higher level of abstraction at which the CSP descriptions are given permit us to not use levels of processes.

3. Marsupial Entity

In Jackson's terminology a *marsupial entity* is one which must be created to express concurrency in the activities of another entity. In this sense it behaves like a marsupial animal "which spends the first part of its existence in its mother's pouch and later emerges to lead a life of its own." There does not seem to be any need to introduce marsupial processes initially, but they are usually created in the later phases of JSD.

The bank example can be used to illustrate the need for marsupial entities. To show that a customer may have many accounts, we may modify the structure of entity CUSTOMER to (we give below the equivalent CSP process definition instead of a structure diagram)

$$\mu Y.(invest \rightarrow \mu X.(payin \rightarrow X \mid withdraw \rightarrow X \mid terminate \rightarrow Y))$$

But this imposes an unrealistic constraint: a customer can open a second account only after terminating the first whereas in fact a customer can concurrently operate many accounts. A single Jackson structure diagram cannot show this concurrency. To reflect this concurrency it becomes necessary to introduce another entity ACCOUNT with a stucture equivalent to the CSP process

$$invest \rightarrow \mu X.(payin \rightarrow X \mid withdraw \rightarrow X \mid terminate \rightarrow STOP)$$

The structure of CUSTOMER is now modified by Jackson to (equivalent CSP process

definition given)

$$\mu X.(invest \rightarrow X \mid payin \rightarrow X \mid withdraw \rightarrow X \mid terminate \rightarrow X)$$

A customer may engage in any of the four actions. But for each account of a customer the ordering constraints are specified by the marsupial entity ACCOUNT. The relationship between the two structure diagrams for ACCOUNT and CUSTOMER is left informal in JSD.

We have already shown (section 3.2) how the parallel combinator and process naming operator may be used to describe a customer with many accounts in CSP without modifying the original process Customer or introducing a new process Account. These two operators provide a formalisation of the notion of marsupial entities without dispensing with it. In fact the CSP process Customer (section 3) is the same as the JSD marsupial entity ACCOUNT. The counterpart of the JSD parent entity CUSTOMER is RUN_A where $A = \alpha Customer$. Since

$$P \parallel RUN_A = P \qquad \text{where } A = \alpha P \qquad \text{(section 2.2.1 and 2.3.1, [H2])}$$

we need not explicitly represent the counterpart of the JSD parent entity.

References

[BHR] Brookes S., Hoare C.A.R. and Roscoe A.W., *A Theory of Communicating Sequential Processes*, JACM, July 1984.

[F] Feather A.H., *OCCAM as a Design Notation in JSD Method*, M.Sc. Dissertation, Oxford University Computing Laboratory, 1983.

[H1] Hoare C.A.R., *Notes on Communicating Sequential Processes*, PRG-33, Technical Monograph, Oxford University Computing Laboratory, 1983.

[H2] Hoare C.A.R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[H3] Hoare C.A.R., *Programs are Predicates*, Philosophical Transactions of the Royal Society, London, Vol. A 312, 1984.

[In] OCCAM, Inmos Ltd, Prentice-Hall International, 1985 (also David May, SIGPLAN Notices, April 1983).

[J] Jackson M., *System Development*, Prentice-Hall International, 1983.

[O] Olderog E.R., *Specifications Oriented Programing in TCSP*, to appear in Logics and Models for Verifications and Specification of Concurrent Systems, K.R.Apt (ed.), Springer Verlag.