A COSY BANKER

SPECIFICATION OF HIGHLY PARALLEL AND DISTRIBUTED RESOURCE MANAGEMENT

P.E. Lauer(1), P.R. Torrigiani(2), R. Devillers(3)

Abstract: A highly parallel strategy is described for the well known banker's problem. Efficiency is reached by distributing the tasks to be performed between a set of parallel activities and by avoiding whenever possible the time consuming conflict situations and choice problems, which otherwise could slow down the parallel processes.

The solution is described in the COSY notation, a macro notation designed to express long, parametric path programs in an easy, compact and structured way.

-
- (1) Computing Laboratory, University of Newcastle upon Tyne, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
 - (2) Institut für Informationssystemforschung, Gesellschaft für Mathematik und Datenverarbeitung MBH Bonn, West Germany.
 - (3) Université Libre de Bruxelles, Brussels, Belgium.

1. INTRODUCTION: A NON COMPUTATIONAL BANKER

The banker's problem and its connection with the allocation of non shareable, non preemptive and reusable resources in an operating system are well known features in computer science ([1,2]).

We will here consider a banker with a single currency, say dollars. He is supposed to give loans to customers of his bank, which may apply for them as many times as they please, requesting in each application as maximum the whole capital of the banker, and committing themselves to give back the loan and not to apply for new loans before they gave back the old one. Of course, the banker wants to avoid any deadlock or starvation situations and to spread his money as much as possible.

In [5,6,7], two of the authors devised the following non computational strategy for this problem:

the banker builds in his bank office as many windows as he has money and he lets the people queue at a window depending on the amount of money they ask for. Cyclically, he looks at how much money he may loan and opens the windows corresponding to loans not greater than this amount. At each open window, he serves a single customer, if any, by giving him a dollar and by sending him to the next window, i.e. that corresponding to a loan one less, and then he closes the window. Moreover, in order to avoid starvation, he counts the customers applying for a new loan and blocks them when a certain counter amount is reached, until he gets all his money back at which time he resets the counter.

This strategy is more flexible than the "one slot" allocation or pre-reservation strategy ([10]). It is suboptimal w.r.t. the safety test ([1,2]) or the promotion test ([3]) solutions, in the sense that it may delay some safe requests, but it has the advantages of being essentially non-computational and of presenting an inherent parallelism.

A first specification of this strategy ([5,6]) was written in the COSY (COncurrent SYstem) notation ([4,7]), a macro notation designed to express long, parametric path programs ([8,9]) in an easy, compact and structured way. In fact, the development of the COSY notation was strongly connected with the elaboration of the banker's program, and some other related problems.

The present paper is devoted to an improved strategy, expressed in an extended COSY notation. The level of parallelism is increased by distributing the tasks between a set of cooperating parallel actors or resources; time consuming conflicts and choice problems are avoided whenever possible. In order to express the solution in a straight, easy and readable way, we have added some interesting features to the early COSY notation.

2. INTRODUCTION TO THE COSY NOTATION AND ITS VECTOR FIRING SEQUENCE SEMANTICS

In this section we present the COSY (COncurrent SYstem) notation. COSY is a language whose terminal objects (programs) constitute abstract descriptions of systems in terms of their synchronic properties. The notation itself is a development of the path expressions of Campbell and Habermann [18] and of the path-process notation of Lauer and Campbell [8]. Essentially, the COSY notation adds generators to the path-process notation so that systems can be specified as path and process patterns (templates) from which instances of paths and processes can be generated in an orderly manner [7,19].

A system, from the COSY point of view, consists of a collection of resources and of sequential but nondeterministic processes. A resource is represented by a set of atomic actions (operations) together with a collection of statements expressing constraints on the order of activation of these operations (paths). A process is represented by an expression (process expression) which describes the pattern of usage of resources required by the process. Formally a process expression determines a collection of sequences of activations of operations. Distinct processes are notionally parallel; however, the paths determine a set of usages of resources of the system as a whole and thereby achieve a co-ordination of the processes.

2.1 THE BASIC NOTATION

A program in the basic notation is a string derived from the following BNF-type production rules.

```

<program> = begin <programbody> end
<programbody> = <path>|<process>|<path><programbody>|<process><programbody>
<path> = path <sequence> end
(PR) <process> = process <sequence> end
<sequence> = <orelement>|<orelement>;<sequence>
<orelement> = <element>|<element>,<orelement>
<element> = <operation>|<element>*|(<sequence>)

```

where the nonterminals are included between < and > and we assume a set of terminals called operations disjoint from the set {begin, end, , , ; , path, process, *, (,)}.

The following is an example of a program in this notation.

```

begin
  process request_a1_a2; use_a1_a2; release_a1_a2 end
(1) process request_a2_a3; use_a2_a3; release_a2_a3 end
    path (request_a1_a2;release_a1_a2),(request_a2_a3;release_a2_a3) end
end

```

Intuitively, this program describes a pair of sequential processes progressing through cycles of requests, usages and releases of pairs of resources, the a_i . Semicolon may be thought of as specifying sequentialization. The path effects mutual exclusion of requests and releases of the two sets of resources since comma denotes exclusive choice. It binds more strongly than the semicolon, hence the need for parentheses. Paths and processes are cyclic.

To illustrate the use of the star, which denotes iteration zero or more times, we give the following program fragment:

(2) path(push;(push;(push;pop)*;pop)*;pop)* end ¹.

which defines the behaviour of a three-frame stack which is initially empty. The star binds more strongly than the comma (and hence the semicolon), whence the need for parentheses.

The 'formal meaning' of a basic program will be obtained here by associating with each program P , an object called $VFS(P)$, the vector firing sequences of P , a set of n -tuples whose coordinates are strings of operations of P . The notion of vector firing sequences is due to Mike Shields and a detailed formal development of this notion can be found in [20].

The intuitive basis of the VFS semantics is the notion of a system composed of a finite number n of (sequential) subsystems, each of which can only 'observe' that part of a behaviour of the system which consists of sequences of occurrences of events belonging to the subsystem in question. If E_i is the set of events belonging to the i -th subsystem, then the set of events E belonging to the whole system is $E_1 \cup \dots \cup E_n$. If $a \in E$, then let $\underline{a}$ denote the n -tuple $(a_1, \dots, a_n)$, where $a_i = a$ if $a \in E_i$ and $a_i = \epsilon$ (the null string) otherwise. $\underline{a}$ may be considered to represent the event:

"for all i for which $a \in E_i$, subsystem i observes a occur".

Let $V = \{\underline{a} \mid a \in E\}$. Let V^* be the smallest set such that $\underline{\epsilon} = (\epsilon, \dots, \epsilon) \in V^*$, $V \subseteq V^*$ and that if $(x_1, \dots, x_n), (y_1, \dots, y_n) \in V^*$, then $(x_1, \dots, x_n)(y_1, \dots, y_n) = (x_1 y_1, \dots, x_n y_n) \in V^*$. V^* is clearly a semigroup with identity $\underline{\epsilon}$.

If $\underline{x} \in V^*$ then $\underline{x}$ can be regarded as a behaviour in which for each i , the i -th subsystem observes the sequence $[\underline{x}]_i$, the i -th coordinate of $\underline{x}$. If $a, b \in E$ and $\underline{ab} \neq \underline{ba}$, then for some i $a, b \in E_i$. Every i -th subsystem with $a, b \in E_i$ will in fact observe in $\underline{ab}$ an occurrence of a precede an occurrence of b . In this sense, we may interpret the fact that a and b both belong to some E_i as meaning that ordering of occurrences of a and b in behaviours is objective; any pair of subsystems capable

Note 1. Since all paths are cyclic we will omit the outermost "*" and parentheses to write path ... end instead of path(...)* end

of observing both a and b will always agree on the order in which they have occurred. In contrast, if $\underline{ab} = \underline{ba}$, then for no i will $a, b \in E_i$ be the case. Orderings of occurrences of a and b are not objective. a and b are concurrent. Note that no subsystem may actually 'observe concurrency', but that the concurrency of two event occurrences in a behaviour $\underline{x}$ may be deduced from the set $\{[\underline{x}]_i \mid i \in \{1, \dots, n\}\}$ of observations of $\underline{x}$ by the subsystems of the system. Concurrency is 'social'.

Suppose, now, that we define a system by associating with each E_i a constraint regarding permitted sequences of occurrences of the events in E_i . We may think of E_i as corresponding to some system component whose well-functioning is defined by this constraint and which it is the i -th subsystems duty to observe. The constraint determines a subset $B_i \subseteq E_i^*$, B_i is the set of permitted behaviours of the i -th subsystem. The set of behaviours of a system obeying precisely these constraints will thus be $(B_1 \times \dots \times B_n) \cap V^*$. This is the set of behaviours $\underline{x}$ of the system such that the view $[\underline{x}]_i$ of $\underline{x}$ of the i -th subsystem is consistent both with its own associated constraint and with the views $[\underline{x}]_j$ of all the other subsystems.

It should be pointed out that there is an intended conceptual difference between the VFS semantics and descriptions of behaviours such as firing sequences, in that vector firing sequences are not to be thought of as being generated step by step according to some 'firing rule'. As abstract mathematical objects, they are, indeed, obtained by a sequence of concatenations, but a particular history thus obtained is not in general generated by a sequence of events.

We may now see how such n -tuples of strings, which we call vector firing sequences may be used to describe histories of systems specified by COSY programs. Each path or process may be considered to be 'observing' a sequence of activations of operations mentioned in it; paths and processes themselves define constraints on orderings of occurrences - by the use of the separators comma, semicolon and star.

Let us first consider the set of permitted orderings of occurrences determined by a single path or process, its set B_i , in the terminology of the preceding discussion. To do this, consider a string of terminal symbols e generated from one of the non-terminals $\langle \text{element} \rangle$, $\langle \text{orelement} \rangle$ or $\langle \text{sequence} \rangle$ by the rules (PR). We define a set $\text{Cyc}(e)$ as follows:

- a) If a is an operation, then $\text{Cyc}(a) = \{a\}$
- b) If e is a string of type element, then $\text{Cyc}(e^*) = \text{Cyc}(e)^*$
- c) If e is a string of type element and e' is a string of type orelement then $\text{Cyc}(e, e') = \text{Cyc}(e) \cup \text{Cyc}(e')$

- d) If e is a string of type orelement and e' is a string of type sequence then
 $Cyc(e;e') = Cyc(e)Cyc(e') = \{xx' | x \in Cyc(e) \wedge x' \in Cyc(e')\}$ where juxtaposition denotes concatenation of strings as usual.
- e) If e is a string of type sequence then
 $Cyc(\underline{process} \ e \ \underline{end}) = Cyc(\underline{path} \ e \ \underline{end}) = Cyc((e)) = Cyc(e)$

Intuitively, each path or process loops through a number of cycles in using its component operations. The history of a given path or process P denoted by $FS(P)$ may thus be defined to be $Pref(Cyc(P)^*)$, where for a string set X , $Pref(X)$ is defined to be the set $\{x | \exists y: xy \in X\}$.

If we now have a program $P = \underline{begin} \ P_1 \dots P_n \ \underline{end}$, where each P_i is a single path, then each P_i 'observes' operations from $Ops(P_i)$, which is defined to be the set of operations mentioned in P_i . Exactly as in the preceding discussion, we form $Ops(P) = \bigcup Ops(P_i)$ and vector operations $\underline{a}_P = (a_1, \dots, a_n)$, where $a_i = a$ if $a \in Ops(P_i)$ and $a = \epsilon$ otherwise. We let $Vops(P) = \{\underline{a}_P | a \in Ops(P)\}$ and construct $Vops(P)^*$ as above. We may now define the set of histories of P , $VFS(P)$, to be

$$VFS(P) = (FS(P_1) \times \dots \times FS(P_n)) \cap Vops(P)^*$$

We cannot apply this semantics directly to programs involving paths and processes. This is because the process semantics of [8] entails an implicit distinction between two operations with the same name occurring in different processes. For example, in the program begin process a end process a end end, one has two processes which may concurrently be activating a ; there are two ' a 's; an ' a -in-process-1' and an ' a -in-process-2'. If we were to insert into this program path a end then the 'observer' associated with the path would have to see a sequence of ' a 's in order that his constraint hold, that is, the path enforces mutual exclusion between occurrences of ' a -in-process-1' and ' a -in-process-2'. This suggests the following transformation, which makes the semantics of processes explicit and permits one to define $VFS(P)$ for any program. Let $Pr = \underline{begin} \ P_1 \dots P_m \ \underline{end}$, where each P_i is either a path or a process. If P_i is a process, then replace every operation $a \in Ops(P_i)$ by an operation $a \& i$ (' a -in- P_i '). Do this for every process. Then we make the mutually excluding effect of paths on process operations with the same name explicit. Suppose a is an operation occurring in processes $P_{i_1}, \dots, P_{i_n}$, if a belongs to a path P_i , then replace a in P_i by the orelement $a \& i_1, \dots, a \& i_n$. Finally, replace each 'process' by 'path'. We shall denote the resulting program by $Path(Pr)$. $Path(Pr)$ consists exclusively of paths, we may thus define $VFS(Pr) = VFS(Path(Pr))$.

We illustrate this construction by the following example. ' rq ' and ' rl ' stand for 'request resource' and 'release resource' respectively.

begin

- (2) process rq; rl end process rq; rl end path rq;rl end

This translates as follows:

begin

- (3) path rq&1; rl&1 end path rq&2; rl&2 end path rq&1,rq&2; rl&1, rl&2 end

This has the following set of vector firing sequences:

$\text{Pref}(\{(rq\&1\ rl\&1, e, rq\&1\ rl\&1), (e, rq\&2\ rl\&2, rq\&2\ rl\&2)\}^*)$.

where 'Pref' is defined in analogy with the string case, that is for

$X \subseteq \text{Vops}(P)^*$, $\text{Pref}(X) = \{\underline{x} \in \text{Vops}(P)^* \mid \underline{y} \in \text{Vops}(P)^*: \underline{xy} \in X\}$.

For example, $(rq\&1, rq\&2\ rl\&2, rq\&2\ rl\&2\ rq\&1) = \underline{rq\&2\ rl\&2\ rq\&1}$ is a history of this system. Note that each process coordinate consists of a sequence of alternating requests and releases and that only one process may be active at any one time, for if not, we may have, say rq&1 and rq&2 concurrently active. But this is not possible, since rq&1 rq&2 = $(rq\&1, rq\&2, rq\&1\ rq\&2) \neq (rq\&1, rq\&2, rq\&2\ rq\&1) = \underline{rq\&2\ rq\&1}$.

2.2 STANDARD COSY NOTATION

The COSY notation is a macro facility designed to furnish an easy, compact and structured, way of writing complex parametric path programs. It is based on the introduction of collective names, i.e. on the ability to give the same name to a whole set of similar operations, the individual operations being distinguished by the use of indices. Collective names will be written in capital letters and the standard operation names will be written in small characters; indices will be integer values or integer expressions.

In the standard COSY notation, one only considers rectangular shaped collective names, with indices starting from 1, so that they are declared by array-like collectivisors.

For instance, the collectivisor: array GET,PUT(m,n)

defines a collection of $m \times n$ indexed operations GET(i,j), with $1 \leq i \leq m$ and $1 \leq j \leq n$, and a similar collection of PUT(i,j).

An iterative copy operator, called the replicator, then allows to generate a succession of similar patterns:

[pattern(i) i [from , to , step]

where pattern(i) is a pattern (possibly) depending on the index i, generates the string

pattern(from) pattern(from+step) pattern(from+2xstep)...pattern(from+kxstep)

with $(\text{from} + k \times \text{step} - \text{to}) \times \text{step} \leq 0$ and $(\text{from} + (k+1) \times \text{step} - \text{to}) \times \text{step} > 0$.

The step parameter may be negative, the number of generated copies may be null and the replicators may be imbricated.

For instance

`[path[GET(i,j), [j] |1,n-1,1]GET(i,n); [PUT(i,j), [j] |1,n-1,1]PUT(i,n) end [i] |1,m,1]`
 defines a set of m frames initially full, for each of them, the corresponding information may be removed by n distinct get operations, and deposited by n distinct put operations.

A special character, @, allows one to drop the part of the pattern which follows it in the last generated copy. The same paths as above may thus be generated by

`[path[GET(i,j)@, [j] |1,n,1]; [PUT(i,j)@, [j] |1,n,1] end [i] |1,m,1]`

It is also possible to generate the last copies of the pattern inside the first ones. The next copies are in fact inserted in the place indicated by the position of the index notation.

Consequently, the replicator

`path PUT(1,1)[; (PUT(1,j)[j]; GET(1,j))* |2,n,1]; GET(1,1) end`

defines a stack-like resource of capacity n, initially empty, where PUT(1,j) is the operation depositing an information in position j in the stack, and GET(1,j) is the one removing this information.

A last facility of the notation, called the distributor, allows the distribution of a connector ", " or ";" between the various copies automatically generated for all the permissible values of a certain index in a pattern. This index is determined by a (first) empty index position.

For instance, the distributor

`, (GET(i,))`

will be equivalent to the replicator

`[GET(i,j)@, [j] |1,n,1]`

and the set of frame-like resources may be rewritten:

`[path, (GET(i,)); (PUT(i,)) end [i] |1,m,1].`

2.3 EXTENDED COSY NOTATION

In some circumstances, a clear expression of a path program may need the generation of a succession, or an imbrication, of similar patterns for which the indices progress in a more complex way than simple arithmetic progressions with constant steps. In order to cope with such situations, we have introduced an extended form of the replicating operator, the test replicator, which uses a predicate to select the exact indices of the sequence.

For instance,

`[pattern(i) [i P(i)] | from,to,step]`

will generate the concatenation of the pattern(i), for the indices i such that

i = from+kstep, with k=0,1,2,...,
 (i-to)+step ≤ 0
 and predicate P(i) is true.

The same extension may be applied to the forms with @ and with insertions.

Thus, [SIEVE(i)@; **i i is prime** | 2,15,1]

will generate the string

SIEVE(2) ; SIEVE(3) ; SIEVE(5) ; SIEVE(7) ; SIEVE(11) ; SIEVE(13)

It can be seen that the standard replicators may be viewed as test replicators with a tautological predicate $P(i) \equiv \text{true}$.

It may also happen that we need indexed collective names which do not correspond to rectangular shaped arrays. This problem may be solved by allowing the use of replicators, or test replicators, in the collectivisor declaration to specify the exact set of admissible indices, or equivalently the exact set of admissible indexed names. We shall use this last form, by introducing a new declarator, operations, which specifies a set of admissible names.

For instance,

operations[[GRSK(i,j)@, **i** | 1,j,1]@, **j** | 1,n,1]

will specify that the collective name GRSK only covers the operation names

GRSK(1,1),GRSK(1,2),GRSK(2,2),GRSK(1,3),GRSK(2,3),GRSK(3,3),...GRSK(n,n)

The collectivisor array GET(m,n)

may then be considered as an abridged form for

operations [[GET(i,j)@, **i** | 1,m,1]@, **j** | 1,n,1]

The same declaration may also be used to specify beforehand the non collective operation names which will occur in the program:

operations resb,rese

The distributor operator will of course reflect these (possibly) non rectangular shapes: only the indices corresponding to admissible operation names have to be automatically generated.

Thus, the distributor

; (GRSK(i,))

would here mean

GRSK(i,i) ; GRSK(i,i+1) ; ... ; GRSK(i,n)

i.e. [GRSK(i,j) @; **j** | i,n,1]

or [GRSK(i,j) @; **j i ≤ j** | 1,n,1]

In order to avoid starvation situations, it may also be useful to introduce a fixed priority feature, i.e. to impose that some conflicts are always solved in favour of some specific operations. The authors [5,6,7] have already used for this purpose an implicit priority for the operations occurring in process expressions over the operations occurring only in paths.

In other papers [11, 12, 13, 14, 15, 16, 17], specific connectors "<" and ">" are used as alternate forms of the comma to indicate an exclusive choice with a priority rule for the solution of the (possible) conflicts:

skip < get

thus means that "skip" and "get" are exclusive and that, if they are both enabled, "get" is always chosen.

In order to be more flexible, to cope with the conflicts induced by the star as well as with the conflicts due to the comma and to ease coherence checks on the priority constraints, we shall here use a separate specification of the priority rule in the declaration part of the program:

priority skip < get

will for instance specify that if the operations "skip" and "get" are both enabled and in conflict, whatever origin this conflict has, "get" is always chosen. We shall of course allow the use of replicators in a priority specification.

3. THE HIGHLY PARALLEL BANKER

We shall now introduce and describe the various resources and processes we have used to get a highly parallel specification of our banker's strategy.

3.1 THE BANK DIRECTOR

As explained previously, the banker will cyclically examine the state of his safe. If he has some money he will open a set of windows and if he has all his money he will first reset the counters in order to allow new customers to apply for a loan.

The money will be modelled by a free buffer [5,7], each dollar corresponding to a single frame. Initially, each frame is full, meaning that all the money is in the bank. A unit may be borrowed by a GET operation: the frame is then empty. It is paid back by a PUT operation: the frame is full again. The frames will be numbered from 1 to n , from right to left, where n is the capacity of the bank.

In order to avoid conflicts between customers when possible, it will be useful to know not only the current amount of the safe but also the state, loaned or not, of each dollar, i.e. the state, empty or full, of each frame. Such a configuration will be represented by an integer c , ranging from 0 to $2^n - 1$, whose binary representation reflects the state of the various frames. More precisely, if we number the bits of a binary representation from 1 to n , from right to left, and if $\text{bit}(b, c)$ is bit b of c , $\text{bit}(f, c)$ will be 1 if frame f is full in the considered configuration, 0 otherwise. The banker will then check the current configuration with a set of operations $\text{CONF}(c)$, where c is ranging from 1 to $2^n - 1$, since the empty configuration does not interest him.

Instead of himself opening the appropriate windows sequentially, as in [5,7], the banker will use the services of a set of clerks, working concurrently. Their service will be activated by a $\text{PGETB}(c)$ operation and the completion of the job

will be controlled by a PGETE(c) operation, where PGETB and PGETE stand for parallel-get-begin and parallel-get-end, respectively.

Similarly, when the configuration is full, the counters will reset themselves in parallel, from an activation operation, resb, until a completion operation, rese, where resb and rese stand for reset-begin and reset-end, respectively.

The banker will not collect the money back himself. This will be done independently and in parallel. It may then happen that the safe is increased while the banker is controlling a money distribution: this will not affect him until his next cycle.

The banker path is shown in (4.1). The kernel of the bank, namely, the banker, the vault and the clerks are subjected to a very careful formal analysis in [22] and proved to be strongly deadlock free and correct.

3.2 THE VAULT

The vault of the bank will be represented by the n frames of the free buffer. As the distribution of money, and paying it back, may be done in parallel to, or by, the various customer processes p , at the various windows w , for the various frames f , we shall use 3-dimensional GET(p,w,f) and PUT(p,w,f) operations. We shall see later however that a frame f will never be associated with a window w with a higher number: the cases where $w \leq f$ have only to be considered. This is taken into account in the declarations (4.0) for these operations and the application of distributors to them will automatically reflect this shape.

As the GET and PUT operations regulate the configuration changes, they are associated to the CONF operations.

The vault paths are then shown in (4.2).

3.3 THE DOORS

Access to the windows will be made through various doors. In front of each window, there will be a set of m doors, one for each customer. When a customer wants to go to a window, he will first knock at his corresponding door by a 2-dimensional RQ(p,w) operation, where RQ stands for request (a dollar at window w for process p). Each door will cyclically look if there is a request pending for it and then wait until the corresponding customer has got a frame. If there is no pending request, this will be reflected by a 2-dimensional RQSK(p,w) operation, where RQSK stands for request-skip.

The door paths are shown in (4.3).

3.4 THE FAIR DOORMEN

At each window, a single customer may be served at a time. A doorman will then be used to select a customer among the ones, if any, who knocked at their doors, i.e. who expected a RW operation, thus invalidating the corresponding request-skip

operation. In order to avoid the starvation of a door by the other ones, the doorman will inspect them circularly: he will thus select the first knocking customer he finds after the one previously served and wait until this customer is served, i.e. gets the requested frame. During this period, the other knocking customers, if any, will consequently be delayed.

The doorman paths are shown in (4.4).

3.5 THE COUNTERS

The allowance counters, which will sometimes temporarily block some customers applying for a new loan, may be attached to the whole system, to each customer, to each window or to each customer-window pair. In order to avoid unnecessary conflicts between customers and to get the simplest program, we shall here associate a counter with each customer.

Each counter will have a capacity 1 and will be modelled by a free buffer of 1 frames initially void. The frames are filled by 2-dimensional CNT(p,k) operations, which will be performed by the customers before each loan transaction. They are emptied by 2-dimensional RSET(p,k) operations, activated by the resetting mechanism, which is controlled by the resb-rese pair. A 2-dimensional LEAVE(p,k) operation is also introduced in order to allow the resetting of an already empty frame.

In order to avoid unnecessary and possibly time consuming choice problems for the customers when they count themselves, we shall also use sequentializing paths, which will impose a cyclical order in the way each customer accesses his counting frames.

The paths for the counters, for the resetting mechanism and for the sequentialisers are shown in (4.5).

3.6 THE CLERKS

The clerks are in charge of effectively giving the money at the various windows. Their job is controlled by the PGETB-PGETE operations. In order to achieve the highest level of parallelism, one has to avoid competitions between paths and/or processes whenever possible. This was obviously not possible for the doorman but, as far as the clerks are concerned, we are simply faced with a so-called "Argentinian party problem". An A.P.P. is a problem where a set of consumers apply for certain resources, one knows that there are enough resources to satisfy everyone and one wants the distribution to be made without conflict. To solve the problem one may specify beforehand a strategy which says which resource goes to which consumer, if any. Here we know that the number of windows opened from the right is equal to the number of full frames observed by the banker. Hence one may simply associate the window w with the w 'th full frame from the right. We shall

then associate a clerk with each window w and with each frame $f \geq w$. He will only be activated if frame f is the w 'th full frame from the right. If a customer has been selected by the doorman for this window, the clerk will grant his request and give him frame f . Otherwise he performs a 2-dimensional $\text{GRSK}(w, f)$ operation, where GRSK stands for grant-skip.

The paths for the clerks are shown in (4.6).

3.7 THE FAIR REIMBURSERS

When a customer p reaches the point where he wants to give back the amount $\underline{a}$ of his loan to the bank, instead of doing it himself sequentially, he will simply activate, by a 2-dimensional operation $\text{PPUTB}(p, \underline{a})$ a set of $\underline{a}$ reimbursers who will concurrently give back one dollar, i.e. fill one frame, each. Each reimbursers may be seen as associated with a customer and with a window where the customer received a frame. He may then be activated by a $\text{PPUTB}(p, \underline{a})$ operation where $\underline{a} \geq$ window number.

We will also impose that the reimbursements are "fair", i.e. that reimbursers (p, w) gives back exactly the dollar that customer p got at window w during the preceding loan. We will thus associate the PUT operations with their corresponding GET , and prevent the customer from getting a new dollar at a window, during a new loan, if he got precedently at that window. This fair behaviour will also avoid an unnecessary competition between the reimbursers for refilling the empty frames. The paths for the reimbursers and for the fair reimbursements are shown in (4.7).

3.8 THE CUSTOMERS

Each customer is modelled by a process which will cyclically count himself (CNT) for a new loan, choose a loan amount $\underline{a}$, go through a sequence of windows as requested by the strategy (knocking at a door, RQ , and waiting until he gets a frame, GET) and finally give back his whole loan (PPUTB).

The customers will thus ask for their dollars one at a time and will not give them back until they have received their whole loan. This constraint may be overridden however as we have shown in [15, 16].

The processes for the customers are shown in (4.6)

3.9 PRIORITY CONSTRAINTS

The various operations of the program have been introduced above. The declarations for the collective names are shown in (4.0).

In order to obtain a fair behaviour and to avoid starvation situations, we will have to add some priority constraints.

In a general manner, the skip-like operations will have a lower priority than their conflicting operations, since they are only introduced to reflect the situations where these conflicting operations are disabled. Thus: $RQSK < RQ$ (to avoid that knocking at a door is indefinitely delayed by the watch round of the corresponding doorman), $LEAVE < CNT$ (to avoid that counting a customer is indefinitely delayed by the resetting mechanism, i.e. by the bank director's supervision) and $GRSK < GET$ (to avoid that a clerk always skips the money distribution). Note that the rule does not concern the pairs $RQSK-GET$ and $LEAVE-RSET$, since the examination of (4.3) and (4.5) shows that they are never in conflict.

It is also necessary to impose that $CONF < PUT$, in order to avoid that the examination of the configuration by the bank director indefinitely delays the reimbursements. It is not necessary to impose the same condition on $CONF$ and GET , since GET is controlled by a $PGETB-PGETE$ pair, i.e. by the bank director himself, so that they are never in conflict.

These priority constraints are also shown in (4.0)

3.10 GENERALIZATION AND FORMAL ANALYSIS OF THE HIGHLY PARALLEL BANKER

In [17] Devillers and Lauer formulated some initial ideas for obtaining highly parallel and distributed Banker's with multiple currencies. In [22] Shields and Lauer subjected some of the techniques involved in these bankers to a formal analysis and they were formally proven to be correct using the formal theory for the COSY notation developed in [19, 20, 21].

Acknowledgements

Credit is due to Mike Shields for contributing to the formulation of Section 2.1 of this paper. The research reported in this paper has been encouraged and supported by the Science Research Council of Great Britain, the visits to the Computing Laboratory of the University of Newcastle upon Tyne by two of the authors were sponsored as follows: The visit of P.R. Torrigiani was partly supported by the Gesellschaft für Mathematik und Datenverarbeitung, Bonn, and the visit of R. Devillers was supported by the Fonds National de la Recherche Scientifique of Belgium.

4. THE PROGRAMbegin

```

(4.0) array CONF, PGETB, PGETE( $2^n-1$ ); array RQ, RQSK, PPUTB(m,n);
      array CNT, LEAVE, RSET(m,1);
      operations [[GRSK(w,f), [GET(p,w,f), PUT(p,w,f),  $\boxed{p}$  | 1,m,1]
                   $\boxed{f}$  | w,n,1]  $\boxed{w}$  | 1,n,1]resb,rese;
      priority [[RQSK(p,w)<RQ(p,w),  $\boxed{p}$  | 1,m,1]  $\boxed{w}$  | 1,n,1]
                [[LEAVE(p,k)<CNT(p,k),  $\boxed{p}$  | 1,m,1]  $\boxed{k}$  | 1,1,1]
                [[GRSK(w,f)<((GET(,w,f))),  $\boxed{f}$  | w,n,1]  $\boxed{w}$  | 1,n,1]
                [(CONF(c)@,  $\boxed{c}$  | bit(f,c)=0 | 1,  $2^{n-2}$ , 1)]<((PUT(,f)))@,  $\boxed{f}$  | 1,n,1];
(4.1) path[(CONF(c);PGETB(c);PGETE(c)),  $\boxed{c}$  | 1,  $2^{n-2}$ , 1]
      (CONF( $2^n-1$ );resb;rese;PGETB( $2^n-1$ );PGETE( $2^n-1$ ))end
(4.2) [path[(CONF(c)@,  $\boxed{c}$  | bit(f,c)=1 | 1,  $2^{n-1}$ , 1)]*, ((GET(,f)));
      (CONF(c)@,  $\boxed{c}$  | bit(f,c)=0 | 1,  $2^{n-2}$ , 1)]*, ((PUT(,f)))end  $\boxed{f}$  | 1,n,1]
(4.3) [[path RQSK(p,w), (RQ(p,w);, (GET(p,w)))end  $\boxed{p}$  | 1,m,1]  $\boxed{w}$  | 1,n,1]
(4.4) [path RQSK(p,w),, (GET(p,w))@,  $\boxed{p}$  | 1,m,1]end  $\boxed{w}$  | 1,n,1]
(4.5) [[path LEAVE(p,k), (CNT(p,k);RSET(p,k))end
      path resb;LEAVE(p,k),RSET(p,k);rese end  $\boxed{k}$  | 1,1,1]
      path; (CNT(p,))end  $\boxed{p}$  | 1,m,1]
(4.6) [[path[PGETB(c)@,  $\boxed{c}$  | bit(f,c)=1  $\wedge$   $\sum_{k < f}$  bit(k,c)=w-1 | 1,  $2^{n-1}$ , 1];GRSK(w,f),
      , (GET(,w,f))];
      [PGETE(c)@,  $\boxed{c}$  | bit(f,c)=1  $\wedge$   $\sum_{k < f}$  bit(k,c)=w-1 | 1,  $2^{n-1}$ , 1]end  $\boxed{f}$  | w,n,1]
       $\boxed{w}$  | 1,n,1]
(4.7) [[path PPUTB(p,a)@,  $\boxed{a}$  | w,n,1];, (PUT(p,w))end  $\boxed{p}$  | 1,m,1]  $\boxed{w}$  | 1,n,1]
      [[path, ((GET(p,w);PUT(p,w,))end  $\boxed{w}$  | 1,n,1]  $\boxed{p}$  | 1,m,1]
(4.8) [process, (CNT(p,));[(RQ(p,w);, (GET(p,w,))];  $\boxed{w}$  | a,1,-1]
      PPUTB(p,a)@,  $\boxed{a}$  | 1,n,1]end  $\boxed{p}$  | 1,m,1]

```

endREFERENCES

- [1] - DIJKSTRA, E.W. - Co-operating Sequential Processes, in Programming Languages, ed. Genuys, Academic Press, N.Y. (1968).
- [2] - HABERMANN, A.N. - Prevention of System Deadlocks, CACM, vol. 12, No. 7, (1969).
- [3] - HABERMANN, A.N. - A New Approach to Avoidance of System Deadlocks, RAIRO, ed. Dunod, vol. 9, No. B-3 (1975).
- [4] - TORRIGIANI, P.R. and LAUER, P.E. - An Object Oriented Notation for Path Expressions, AICA Annual Conference, Pisa, vol. 3, pp 349-371 (1977).

- [5] LAUER, P.E., and TORRIGIANI, P.R. - Toward a System Specification Language Based on Paths and Processes; TR120, University of Newcastle upon Tyne. Computing Laboratory (1978).
- [6] TORRIGIANI, P.R. - Synchronic Aspects of Data Types; Construction of a Non-Algorithmic Solution of the Banker's Problem, ECI 78, Information System Methodology, Lecture Notes in Computer Science, Springer-Verlag (1978).
- [7] LAUER, P.E., TORRIGIANI, P.R., SHIELDS, M.W. - COSY - a System Specification Language Based on Paths and Processes. Acta Informatica 12, 109-158 (1979).
- [8] LAUER, P.E., and CAMPBELL, R. - Formal Semantics for a Class of High-Level Primitives for Coordinating Concurrent Processes, Acta Informatica, vol. 5, pp. 247-332 (1975).
- [9] CAMPBELL R. - Path Expressions: a Technique for Specifying Process Synchronization, Ph.D. Thesis, University of Newcastle upon Tyne (1976).
- [10] HAVENDER J.W. - Avoiding Deadlock in Multitasking Systems, IBM System Journal, vol. 7, no. 2, pp. 74-84 (1968).
- [11] HABERMANN A.N. - Path Expressions, Technical Report, Carnegie-Mellon University, Pittsburgh, PA USA (1975).
- [12] DEVILLERS R. - Non Starving Solutions for the Dining Philosophers Problem, ASM/30, University of Newcastle upon Tyne, Computing Laboratory (1977).
- [13] DEVILLERS R. and LAUER P.E. - Some Solutions for the Reader/Writer Problem, ASM/31, University of Newcastle upon Tyne, Computing Laboratory (1977).
- [14] DEVILLERS R. and LAUER P.E. - A General Mechanism for the Local Control of Starvation: Application to the Dining Philosophers and to the Reader/Writer Problem, ASM/32, University of Newcastle upon Tyne, Computing Laboratory (1977).
- [15] LAUER P.E., TORRIGIANI P.R. and DEVILLERS R. - The Hyperfast Banker, ASM/55, University of Newcastle upon Tyne, Computing Laboratory (1979).
- [16] DEVILLERS R. - A simplified Hyperfast Banker, ASM/56, University of Newcastle upon Tyne, Computing Laboratory (1979).
- [17] DEVILLERS R. - On the Banker with Several Currencies, ASM/57, University of Newcastle upon Tyne, Computing Laboratory (1979).
- [18] CAMPBELL, R.H., and HABERMANN, A.N. - The Specification of Process Synchronization by Path Expressions. Lecture Notes in Computer Science (Ed. G. Goos and J. Hartmanis) pp. 89-102, V16 Springer Verlag, 1974.
- [19] LAUER, P.E., SHIELDS, M.W., BEST, E. - The design and certification of asynchronous systems of processes. Proc. of EEC Advanced Course on Abstract Software Specification, Lyngby, Jan. 22 - Feb. 2, 1979. Lecture Notes in Computer Science (to appear)

- [20] SHIELDS, M.W., - Adequate Path Expressions. Proc. Symp. on the Semantics of Concurrent Computation, Evian - Les - Bains, July 2-4, 1979.
Springer Lecture Notes in Computer Science Vol. 70, 1979.
- [21] LAUER P.E., SHIELDS, M.W., BEST, E. - Formal Theory of the Basic COSY Notation. The Computing Laboratory, University of Newcastle upon Tyne, Tech. Rep. Series No. 143, Nov. 1979.
- [22] SHIELDS, M.W., LAUER, P.E. - Programming and Verifying Concurrent Systems In COSY. The Computing Laboratory, University of Newcastle upon Tyne, Asynchronous System Memo 68, 31 January, 1980.