Abstract

This report is an introduction to concurrent system specification and analysis using the COSY (COncurrent SYstem) notation. In the first section we attempt, by heavy use of examples, to show how the formalism may be used to develop formal specifications of systems in terms of their desired behaviours. In the remainder of the report, we introduce the behavioural semantics of COSY, based on vectors of strings, and demonstrate, by an example, how a formal specification may be verified, both in terms of the intentions of the designer and in terms of general properties, such as deadlock.

Programming and verifying concurrent systems in COSY

By

M.W. Shields and P.E. Lauer

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 155
September, 1980

©1980 University of Newcastle upon Tyne. Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
SHIELDS, Michael William Enduan
Programming and verifying concurrent systems in COSY.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1980.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 155).

Added entries
LAUER, Peter Ernst
UNIVERSITY OF NEWCASTLE UPON TYNE. Computing Laboratory.

Suggested classmarks (primary classmark underlined)
Library of congress:
Dewey (17th)
U. D. C.
001.64404  001.6424
681.322.06

Suggested keywords
ADEQUACY
BANKER'S ALGORITHM
CONCURRENT SYSTEMS
COSY NOTATION
DISTRIBUTED SYSTEMS
DEADLOCK ANALYSIS
PATH EXPRESSIONS
PREDICATE REPLICATOR NOTATION
RESOURCE ALLOCATION
VECTOR FIRING SEQUENCES
VERIFICATION

Abstract
This report is an introduction to concurrent system specification and
analysis using the COSY (Concurrent System) notation. In the first section
we attempt, by heavy use of examples, to show how the formalism may be used
to develop formal specifications of systems in terms of their desired
behaviours. In the remainder of the report, we introduce the behavioural
semantics of COSY, based on vectors of strings, and demonstrate, by an example,
how a formal specification may be verified, both in terms of the intentions
of the designer and in terms of general properties, such as deadlock.

About the author
Dr. M.W. Shields has been a research associate in the Computing Laboratory since
December 1975. Prior to that he was a research student at Leeds University.

Dr. P.E. Lauer has been a lecturer in the Computing Laboratory of the University
of Newcastle upon Tyne since 1972, before then he was with IBM Laboratory, Vienna.
1. Introduction

The purpose of this paper is to illustrate the use of the COSY notation [LTS79] in the design, description and analysis of asynchronous systems.

In the next section, we indicate the difficulties that even programmers proficient in programming in sequential languages may experience when trying to conceptualize non-sequential programs. We argue the necessity for a formalism which aids such conceptualization, pointing out that the activity may involve a change of orientation in one's thinking.

The first section attempts to give the reader an intuitive understanding of the uses to which our formal notions may be put to serve as a guide to programming in the CONcurrent SYstems notation COSY. The section also indicates some of the ways in which we have found such formal notions useful in understanding what the programs one has written mean. To give the reader some facility we analyse a series of ten problems and discuss, semi-formally, the meanings of their COSY solutions.

The following section then formally introduces an important subset of the COSY notation, the language of path expressions. This language is equipped with a formal semantics, which maps each path expression to a set of vectors of strings, its set of vector firing sequences, which model its potential (asynchronous) behaviour. This allows us formally to define properties of systems behaviour. The activity of verification is then briefly discussed.

Sections 4 and 5 bring the preceding sections together. Section 4 contains the development of a COSY solution to a specification problem involving a mechanism for adequately granting concurrent access to reusable resources. We rely on the reader having acquired an intuitive understanding of the notation in section 2. Section 5 uses the formalism of section 3 to produce a verification of the solution.

The paper concludes with a short summary.
2. Introduction to COSY programming.

In this section we introduce the COSY notation [LT579] by way of a series of progressively more complex and general examples. In this manner we lead the reader through the thoughts leading to the construction of COSY programs, as opposed to asking him to understand complex programs that have already been written. The notation was developed with a view to encouraging the programmer to break free from the over centralizing and over sequentializing tendencies of conventional programming notations and to arrive at more concurrent and distributed systems. Hence COSY does not contain such constructs as assignment statements, conditional statements, block structure etc. This means a programmer who knows how to write a centralized and sequential program which is a solution for some general problem will have to relearn how to program the solution from the standpoint of obtaining maximal concurrency and distribution of control.

To a certain extent this means he has to unlearn certain programming skills and acquire a new set of programming skills before he will be able to program with the same proficiency as in the sequential case. Similar remarks also apply in the case of programming by means of Guarded Commands [H76] and Communicating Sequential Processes [H78]. There is some evidence that such a relearning process is worth while since it has lead to a number of programs which express interesting highly concurrent and distributed systems in an economical and informative way [LT579].

Our approach to the study of systems leads us to regard a system as characterizable by the set of notionally indivisible actions it performs - what we shall call its set of operations - together with a collection of constraints which specify how the occurrences of these actions or as we shall say the executions of these operations are to be related. In other words we are considering systems from the point of view of their synchronous properties.

To understand how we formally specify constraints, the following might be useful; any two operations may execute concurrently unless otherwise stated. [By 'concurrent' we do not mean 'simultaneous' or 'in any order']. By 'otherwise stated', we mean that some relationship, to be understood as a sequential constraint is defined between occurrences of the operations. For example, in a system whose operations are a, b, c, we may specify that a executes before or after b and that a executes before or after c. a and b, a and c are related, but b and c are not. In such a system b and c would execute concurrently.
To put it succinctly, a system is specified by defining, for each one of
certain subsets of its set of operations, a sequential constraint relating all
operations of that subset.

If one now considers such a system 'running', that is the operations of
the system executing, then the executions will obey these sequential constraints
and no others.

Let us now clarify what we mean by 'related'. Intuitively, the relation-
ships of which we are now speaking are always of the 'before/after' kind; in
any history of the activity of the system, executions of operations which are
mutually related will be in strict sequence, whatever else is happening in the
system. (However much concurrency there is in the Universe, you know
that the events, your own birth and your taking of an examination are in a
strictly sequential order).

We thus define or specify or describe a relationship between occurrences
in such a way that the relationship determines possible sequences of occurrences.
An obvious way of doing this would be to express a constraints embodying such
a relationship in the form of a grammar. The strings belonging to the language
of the grammar represent the strict sequence of occurrences mentioned in the
previous paragraph.

It is well known how to represent the set of histories of operation
executions of a nondeterministic sequential system, consisting of a definite
and finite number of operations, by means of a regular expression. Briefly,
the COSY notation incorporates such regular expressions (called path expressions
[CH74]), and in a program a path expression is used for specifying the sequential
constraint relating all the operations of the subset of operations of the program
mentioned in the path. However, the COSY notation also incorporates generators
(replicators) for economically defining regular expressions of arbitrary size
and structure in terms of regular expression schemata. Furthermore, these
generators allow the generation of a set of regular expressions from such
regular expression schemata in such a way that the patterns in which regular
expressions belonging to the set share operations can be made explicit and
mathematically described. These features will now be introduced and explained
by means of examples. Further details about these and other features of the
notation can be found in [LBB79] and [LBB79].

Another way would be to use labelled state machines; the connection between
grammars and state machines in this context is embodied in a net semantics for
basic COSY [LBB79].

3.
2.1 Sequential system example:

By a sequential system we will mean a system of operations \( \text{op}_1, \ldots, \text{op}_n \) no two of which may be executed concurrently, that is, a system whose histories of execution are sequences (total orders) of operation executions.

By a concurrent system we will mean a system of operations \( \text{op}_1, \ldots, \text{op}_n \) such that some of the \( \text{op}_i, \text{op}_j \) for \( i \neq j \) may be executed concurrently, that is, a system whose histories of execution are partial orders of operation executions.

Another way to say this is: a concurrent system is a system of operations \( \text{op}_1, \ldots, \text{op}_n \) such that some of the \( \text{op}_i, \text{op}_j \) for \( i \neq j \) do not belong to one and the same sequential subsystem of the concurrent system.

We consider all sequential systems of operations to be cyclic in the sense that constituent operations may be executed repeatedly subject to such constraints as the sequentialization of two operations or an arbitrary choice of one of two operations. Hence, the corresponding regular expression called a path will have the general form:

\[(N1) \quad \text{path}(\ldots)\text{end} \]

where 'path' and 'end' are parentheses around regular expressions, since in general programs are collections of regular expressions. \( \ast \) stands for the Kleene star and it and the outermost parentheses will be omitted in the sequel. Given operations \( a \) and \( b \):

\[(N2) \quad 'a;b' \text{ stands for 'a and b may only be executed strictly in the order written'} \]

and

\[(N3) \quad 'a,b' \text{ stands for 'exactly one of a or b may be executed'} \]

Example 1

Given three operations \( \text{USE}(1), \text{USE}(2), \text{USE}(3) \), combine them to a single system such that:

E1.1 no two operations may be executed at any one instant; and
E1.2 they may be executed in any order.

This can be written as

\[(P1) \quad \text{path USE}(1), \text{USE}(2), \text{USE}(3)\text{ end}. \]

From (N3) we see that the regular expression

\[\text{USE}(1), \text{USE}(2), \text{USE}(3).\]

means

'exactly one of \( \text{USE}(1) \) or \( \text{USE}(2) \) or \( \text{USE}(3) \) may be executed'

and the whole path \( (P1) \) with its implicit outermost Kleene star means

'repeatedly, exactly one of \( \text{USE}(1) \) or \( \text{USE}(2) \) or \( \text{USE}(3) \) may be executed'

which can be argued to express the same as requirement E1.1 and E1.2. In addition to the type of explanation of meaning given above it is often useful
to have some compact way of characterizing the set of histories or execution sequences of a single path. This is particularly so when one is concerned with the problem of ensuring that the path one has written actually specifies the histories one intended to specify when one wrote it as we shall see in section 5. Since all paths will be cyclic it is convenient to base the notion of execution sequence of a path on the notion of the set of cycles of a path, that is, the set of execution sequences which in non-pathological cases return the path to its initial 'state', namely the state which can be thought of as corresponding to the empty history denoted by 'e'.

In section 3 we will precisely indicate how to obtain the set of cycles of a given path but at the present we will merely exhibit them where useful. The set of possible execution sequences of operations of a given path is then defined as the set of all prefixes of multiples of the set of cycles of the path. The set of execution sequences of a path P has traditionally been called the set of firing sequences of P and is denoted by FS(P). The set of cycles of P is denoted by Cyc(P). So we write FS(P) = Pref(Cyc(P)) where Pref yields the set of all prefixes of the elements of a given set of sequences. The star '*' has its usual meaning. Because for a single path P

\[ \text{Pref}(\text{Cyc}(P)) = \text{Cyc}(P)^* \text{Pref}(\text{Cyc}(P)) \]

we make use of the right hand form when it is illuminating. '.' indicates (elementwise) concatenation of (sets of) sequences. We will also use (P1) to stand for the path it labels to write:

\[ \text{Cyc}(P1) = \{\text{USE}(1), \text{USE}(2), \text{USE}(3)\} \]

and

\[ \text{FS}(P1) = \{\text{USE}(1), \text{USE}(2), \text{USE}(3)\}^* \]

and the regular set in the latter definition is a very precise formulation of E1.1 and E1.2. Now we are in a position to proceed more rapidly and precisely with the development of our example.

**Example 2**

The same as Example 1 but replace E1.2 by:

**E2.2** they must be executed in the fixed order of increasing indices.

**Note**

In numbering conditions in our examples (e.g. E1.2), we use the convention that a condition Ei,j replaces all conditions Ei',j', i'<i. Thus in example 2, E2.2 replaces E1.2 and the example is defined by E1.1 and E2.2).

This can be written as

\[ (P2) \text{path USE}(1); \text{USE}(2); \text{USE}(3) \text{ end} \]

for which

\[ \text{Cyc}(P2) = \{\text{USE}(1); \text{USE}(2); \text{USE}(3)\} \]

and
$\text{FS}(P_2) = \{\text{USE}(1)\text{USE}(2)\text{USE}(3)\} \ast [\epsilon, \text{USE}(1), \text{USE}(1)\text{USE}(2)]$.

**Example 3**

The same as Example 1 but replace E1.2 by:

E3.2 USE(1) must strictly alternate with either
USE(2) or USE(3) but not both.

this can be written as

(P3) path USE(1); USE(2), USE(3) end

for which

$\text{FS}(P_3) = \{\text{USE}(1)\text{USE}(2), \text{USE}(1)\text{USE}(3)\} \ast [\epsilon, \text{USE}(1)]$.

P3 illustrates that ',' binds stronger than ';'. Considering P1 we see that the
order of execution of operations is completely arbitrary, whereas in P2 it is
completely fixed and in P3, which lies somewhere between them, we have removed
some but not all of the arbitrary ordering of operation executions.

Suppose next that we do not want the order of execution of operations
to be as arbitrary as in Example 1 nor as inflexible as Example 2, but rather
to depend on the possible configurations of the system.

We need to be more precise about what we mean by a configuration of the
system. Recall that we used the notion of 'state' earlier when we said that
'the state which can be thought of as corresponding to the empty history
denoted by 'ε' is called the initial state'. Furthermore, we said that for
cyclic paths it is convenient to base the notion of firing sequence on the
notion of cycles, that is firing sequences which in non-pathological cases
return the system to its initial state.

The notion of state may be formalised as follows (c.f [S79]). We
postulate:

S1 Every firing sequence $x \in \text{FS}(P)$ sends the system into a specific
state (which we may call $S(x)$).

S2 Every state uniquely determines a set of possible behaviours
starting in that state. Thus, in the initial state, $S(\epsilon)$, the
set of possible behaviours starting in that state is $\text{FS}(P)$.

The uniqueness part of the definition entails that if for any firing
sequence $x$, the set of behaviours starting in that state is also $\text{FS}(P)$, then
$S(x) = S(\epsilon)$, that is, that $x$ returns the system to its initial state.

More formally, we consider the set of behaviours starting in state $S(x)$,
which is precisely

$\text{FS}_x(P) = \{y \mid x, y \in \text{FS}(P)\}$

In general it is not true that cycles always return a system to its
initial state; (the simplest example to hand is
path (a;b*)c; (c;id*)end)

but it may be asserted in general that in cases where we do not have
$x \in \text{Cyc}(P), x, a \in \text{Cyc}(P)$ and $a \in \text{FS}(P)$, (what we have called pathological cases)
then it is true that if \( x \in \text{Cyc}(P) \) then \( FS(P) = FS_x(P) \), that is, \( S(x) = S(c) \), that is \( x \) returns us to our initial state.

We see that we may identify states, \( S(x) \), with 'continuations' \( FS_x(P) \) and that (from S2) two firing sequences send the system to the same state if and only if they have the same set of continuations. We can say more about these states. If \( x \in FS(P) \) then \( x = y.z \), where \( y \in \text{Cyc}(P)^* \) and \( x \notin \text{Pref}((\text{Cyc}(P)) \); in non-pathological cases

\[
FS_x(P) = FS_y(P)
\]

or

\[
S(x) = S(y)
\]

Thus in such cases, all states of the system may be represented by elements of \( \text{Pref}((\text{Cyc}(P)) \), although not necessarily uniquely. We remark that all our examples are non-pathological.

Finally we may say roughly that a configuration is a state we are 'interested in'.

If we reconsider the firing sequences of the paths P1–P3 in light of the above statements we find that:

(P1) characterises a system with only one state the initial state denoted by 'e'.

(P2) characterises a system with three states denoted by 'e', 'USE(1)' and 'USE(1).USE(2)'.

(P3) characterises a system with two states denoted by 'e' and 'USE(1)'.

Note that in all these paths the execution of USE(1) may be interpreted as a coincident test of whether the system is in the appropriate state or configuration since it is only in states \( e \) (respectively USE(1), USE(1).USE(2)) that the operation USE(1) (respectively USE(2), USE(3)) may execute. For example, according to P1 any of the USE(1) may execute in any state of the system, since any USE(1) may be executed in the initial state, and every subsequent state since every subsequent state is identical to the initial state. In terms of the preceding discussion \( FS(P_1) = \{USE(1), USE(2), USE(3)\}^* \) and hence

\[
FS_x(P_1) = FS(P_1)
\]

for each \( x \in FS(P_1) \). Hence in P1 no execution of an operation causes a transition of the system from one state to another. In such a case the notions of test and transition collapse completely. All that is being tested is that no other operation is being executed in the same instant.

In the case of P2 however, an execution of e.g. USE(2) is a state transition from state USE(1) to a different state USE(1).USE(2), which involves an implicit test of whether the system is in state USE(1) the same is true of P3 where e.g. the execution of USE(2) or USE(3) is a state transition from the state USE(1) to the state e.
Note however that even though we can talk about states or configurations with the help of the notion of history in this way we cannot refer to them explicitly in paths.

In order to do the latter we will need to introduce some additional operations which can be interpreted as tests as to whether the system is in a particular configuration.

**Example 4**

As Example 1 except that there are three more mutually exclusive operations called \( \text{CONF}(1) \), \( \text{CONF}(2) \), \( \text{CONF}(3) \) which can be used to test in which of 3 possible configurations the system is, and E1.2 is replaced by:

\[ \text{USE}(i) \] may be executed only if the system is in configuration \( i \).

This can be written as:

\[ \text{FS}(P4) = \{ \text{CONF}(1) \text{USE}(1), \text{CONF}(2) \text{USE}(2), \text{CONF}(3) \text{USE}(3) \}^* \{ \text{CONF}(1), \text{CONF}(2), \text{CONF}(3) \} \]

Examination of \( \text{FS}(P4) \) indicates that \( \text{USE}(i) \) does not determine a state, i.e. \( \text{USE}(i) \notin \text{Pref} (\text{Cyc}(P4)) \) whereas \( \text{CONF}(i) \) does determine a state for \( 1 \leq i \leq 3 \). Examination of \( \text{Cyc}(P4) \) indicates that \( \text{CONF}(1) \text{USE}(i) \) does bring the system to its initial state for every \( 1 \leq i \leq 3 \). From these observations it follows that it is impossible:

(i) for \( \text{USE}(i) \) to be executed before \( \text{CONF}(i) \) has been executed immediately prior to it; and

(ii) for any operation except \( \text{USE}(i) \) to be executed after \( \text{CONF}(i) \) has been executed immediately prior to it.

To be more precise:

\( \text{USE}(i) \) may be executed now if and only if \( x, \text{CONF}(i) \) is the current history of execution, some \( x \).

Hence, \( \text{USE}(i) \) can only be executed when the system is in configuration \( i \) as E4.2 requires.

Note that anytime the system is in the implicit configuration corresponding to the initial state any one arbitrary \( \text{CONF}(i) \) could successfully execute.

Our interpretation of this fact is that there is no logical relation between named configurations except that the system can only be in one configuration at a time.

**Example 5**

As Example 4 except that we add:

\[ \text{E5.3} \] execution of \( \text{USE}(i) \) leads from configuration \( i \) to configuration

\( i+1 \) if \( i<3 \) or \( 1 \) if \( i=3 \).

The problem statement envisages that the execution of \( \text{USE}(i) \) operations
has a direct 'feedback' on the conditions which control their execution. A solution could be written as:

(P5) \[ \text{path CONF}(1); \text{USE}(1); \text{CONF}(2); \text{USE}(2); \text{CONF}(3); \text{USE}(3) \text{ end} \]

for which

\[ F_5(P5) = \{ \text{CONF}(1) \text{USE}(1) \text{CONF}(2) \text{USE}(2) \text{CONF}(3) \text{USE}(3) \}^* \]

\[ \{ \epsilon, \text{CONF}(1), \text{CONF}(1) \text{USE}(1), \text{CONF}(1) \text{USE}(1) \text{CONF}(2), \ldots, \text{CONF}(1) \text{USE}(1) \text{CONF}(2) \text{USE}(2) \text{CONF}(3) \}. \]

At this point it is illuminating to note that we can specify the sequential system \( E_5 \) by using two paths in combination (that is, concatenated). Hence, we can combine the sequential subsystem \( P_4 \) with a path of the form

(P6) \[ \text{path CONF}(1); \text{CONF}(2); \text{CONF}(3) \text{ end} \]

to obtain a generalised path (regular expression) of the form:

(P7) \[ \text{path CONF}(1); \text{CONF}(2); \text{CONF}(3) \text{ end} \]

In general, the executions of operations occurring in a collection of individual paths, which have been concatenated, must obey the ordering constraints of all the individual paths in which they occur.

We can visualise a history of such a collection of paths as consisting of a vector of histories of their component, individual paths. Thus, the histories of (P7) are contained in the set

(P1) \[ F_5(P4) \times F_5(P6) = \]

\[ \{ \text{CONF}(1) \text{USE}(1) \text{CONF}(2) \text{USE}(2) \text{CONF}(3) \text{USE}(3) \}^* \{ \epsilon, \text{CONF}(1), \text{CONF}(2), \text{CONF}(3) \}^* \]

\[ \{ \epsilon \text{CONF}(1), \text{CONF}(1) \text{CONF}(2) \}. \]

However, the principle that executions must obey all the constraints of the appropriate paths entails that an operation in the combined system may only be executed if it may be coincidentally executed in all the sequential subsystems in which it occurs.

This means that we only allow as histories of (P7) those vectors of \( F_5(P4) \times F_5(P6) \) whose coordinate firing sequences agree on the ordering and numbers of the operations they share. We call such pairs of firing sequences (in general such tuples of firing sequences), congresable.

For example, consider the firing sequence;

\[ \text{CONF}(1), \text{CONF}(2), \text{CONF}(3) \in \text{Cyc}(P6) \]

which is also the only element of \( \text{Cyc}(P6) \) then the corresponding firing sequence is \( \text{CONF}(1) \text{USE}(1) \text{CONF}(2) \text{USE}(2) \text{CONF}(3) \text{USE}(3) \in \text{Cyc}(P4)^3 \) where \( \text{Cyc}(P4)^3 \) indicates the concatenation of three elements of \( \text{Cyc}(P4) \). Note that the corresponding firing sequence is a concatenation of the three distinct elements of \( \text{Cyc}(P4) \). These facts imply that these two cyclic histories, or multiples of them, are the only cyclic histories which may be combined to
form the vector of histories corresponding to the combined path \( P_7 \).

Now note that

\[ \text{CONF1} \cup \text{USE1} \cup \text{CONF2} \cup \text{USE2} \cup \text{CONF3} \cup \text{USE3} \in \text{Cyc}(P_5) \]

and that it is the only element of \( \text{Cyc}(P_5) \). Note that path \( P_4 \) and \( P_5 \) share exactly the same operations. If we now consider the vector corresponding to \( P_1 \) and compare the history in the coordinate corresponding to the path \( P_4 \) with the history of \( P_5 \) we discover that they are identical. Hence, from the standpoint of an observer capable of perceiving executions of all the operations of the combined system \( \langle P_7 \rangle \), the cyclic history of the combined system is just the cyclic history of the subsystem \( \langle P_4 \rangle \). Thus, from the standpoint of this observer the behaviour of the system corresponding to \( P_7 \) is the same as that of the system corresponding to \( P_5 \).

It should be pointed out that it is not in general true, for a pair of individual paths \( P_1 \) and \( P_2 \) and for \( x \in \text{Cyc}(P_1) \), \( \text{Cyc}(P_1)^* \), that we can find \( y \in \text{Cyc}(P_2) \), \( \text{Cyc}(P_2)^* \) such that \( x \) and \( y \) are congruence. Indeed for paths in which \( ; \) is the only connective, the existence of such a pair \( x, y \) is equivalent to absence of deadlock [S79]. Nor is it true, in general, that \( P_1 \), \( P_2 \) has 'cycles' such as are possessed by \( P_4 \), \( P_6 \).
2.2 Concurrent system examples

We begin this subsection with a simple modification of example 4 to illustrate the possibility of some concurrent execution of operations:

Example 6

As example 4 except that there are three more mutually exclusive operations called TR(1), TR(2), TR(3) which cause transitions from one configuration to another but we add:

P6.1 Some operations may be executed concurrently.

P6.3 execution of TR(i) leads from configuration i to configuration i+1 if i<3 and 1 otherwise.

P6.4 execution of TR(i) and USE(i) should be concurrent (i.e. not necessarily interleaved)

This can be written by combining P4 and (P6)

(P6) \[ \text{path \ CONF(1); TR(1); CONF(2); TR(2); CONF(3); TR(3) end} \]

to form

(P9) \[ \text{path (CONF(1); USE(1)), (CONF(2); USE(2)), (CONF(3); USE(3)) end} \]

As in the case of (P7), we may take

(P2) \[ \text{CONF(1), TR(1), CONF(2), TR(2), CONF(3), TR(3) \in Cyc(P6)} \]

and ask what firing sequences are congruent with it; we find the following

(P3) \[ \text{CONF(1), USE(1), CONF(2), USE(2), CONF(3)} \]

and

(P4) \[ \text{CONF(1), USE(1), CONF(2), USE(2), CONF(3), USE(3)} \]

both agree with (P2) on the order of the operations they have in common. Note also that (P4) belongs to Cyc(P4)^3. The vector

(P5) \[ \text{(CONF(1), USE(1), CONF(2), USE(2), CONF(3), USE(3), CONF(1), TR(1), CONF(2), TR(2), CONF(3), TR(3))} \]

acts thus rather like a 'generalised cycle' and we can write out the set of vector histories of (P9) as

\[ \{ (\text{CONF(1), USE(3)}, \ldots, \text{USE(3)}), (\text{CONF(1)}, \ldots, \text{CONF(1)}), (\text{CONF(1), USE(1)}) \}

\[ \{ (\text{CONF(1), USE(1), TR(1)}, \ldots, \text{CONF(1), USE(1), TR(1), \ldots, CONF(3)}) \} \]

where ' ,' denotes coordinatewise concatenation.

The reader should observe the manner in which (P6) constrains the execution of operations belonging to (P4). The set of firing sequences
of (P4) is, as we have seen,
\[ FS(P4) = \{ CONF(1), USE(1), CONF(2), USE(2), CONF(3), USE(3) \}^* . \]
\[ \{ \varepsilon, CONF(1), CONF(2), CONF(3) \} . \]
whereas the set of firing sequences which are congeerable to some firing sequence in (P8) are
\[ CPS_{P9}(P4) = \{ CONF(1), USE(1), CONF(2), USE(2), CONF(3), USE(3) \}^* . \]
\[ \{ \varepsilon, CONF(1), CONF(2), USE(1), ... , CONF(1), USE(1), ... , CONF(3) \} . \]
a proper subset of \( FS(P4) \). CPS_{P9}(P4) is the set of firing sequences congeerable with others in P9, or more precisely the 'P4' projection of the congeerable relation. It is important to note that when reasoning about a system such as P9, the executions of operations in P4 are those described in CPS_{P9}(P4) and that no executions described in \( FS(P4) - CPS_{P9}(P4) \) are permitted.

Now note that every time CONF(i) is executed and, hence, coincidently appended to both the histories of P4 and P6, both USE(i) and Th(i) must be executed before CONF (i+1) (mod 3) can be executed, i.e. appended coincidently to both histories. However, nothing is prescribed concerning the ordering of executions of USE(i) and Th(i) relative to each other, hence, they may be executed (appended to the respective histories) concurrently, i.e. not only in some arbitrary order but even coincidently.

**Example 7**

As example 4 with regard to the operations USE(i) and CONF(i). However
\[ USE(i), USE(j), i \neq j \] may be executed concurrently, and
\[ USE(1) \] may be executed only if the system is in configuration 3a. This means \( USE(i) \) for all \( 1 \leq i \leq j \) may execute concurrently in configuration j. It also means that j \( USE(i) \) 's will execute concurrently in configuration j.

So our system is picking different numbers of operations to be executed concurrently depending on the configuration of the system.

We assume that there are pairs of operations parallel begin, denoted 'PB(i)', and parallel end, denoted 'PE(i)', which are used to relate the CONF(i) to the USE(i) in the manner required by Example 7. Now a solution can be written as
\[ \text{(P10)} \]
\[ \text{(P10.1)} \text{ path(CONF(1);PB(1);PE(1)), (CONF(2);PB(2);PE(2)), (CONF(3);PB(3);PE(3)) end} \]
\[ \text{(P10.2)} \text{ path PB(1),PB(2),PB(3);USE(1);PE(1),PE(2),PE(3) end} \]
\[ \text{(P10.3)} \text{ path PB(2),PB(3);USE(2);PE(2),PE(3) end} \]
\[ \text{(P10.4)} \text{ path PB(3);USE(3);PE(3) end} \]
for which the respective sets of firing sequences are

12.
(P3) \( \text{FS}(P10.1) = \{\text{CONF}(1)\text{PB}(1)\text{PE}(1), \text{CONF}(2)\text{PB}(2)\text{PE}(2), \text{CONF}(3)\text{PB}(3)\text{PE}(3)\}^*\)  
\[\{e, \text{CONF}(1), \text{CONF}(2)\text{PB}(1), \text{CONF}(3), \ldots, \text{CONF}(3)\text{PB}(3}\]\n
(P4) \( \text{FS}(P10.2) = \{\text{PB}(i), \text{USE}(1), \text{PE}(j)\mid 1 \leq i, j \leq 3\}^* \{\text{PB}(i)\mid 1 \leq i \leq 3\}, \{e, \text{USE}(1)\}\)

(P5) \( \text{FS}(P10.3) = \{\text{PB}(i), \text{USE}(2), \text{PE}(j)\mid 2 \leq i, j \leq 3\}^* \{\text{PB}(i)\mid 2 \leq i \leq 3\}, \{e, \text{USE}(2)\}\)

(P6) \( \text{FS}(P10.4) = \{\text{PB}(3)\text{USE}(3)\text{PE}(3)\}^* \{e, \text{PB}(3), \text{PB}(3)\text{USE}(3)\}\)

Examination of the firing sequences of P10.1 indicates that:

(i) the system can be in exactly one of the three configurations at any one time; and

(ii) in any configuration exactly one of the pairs of parallel operations PB( ) and PB( ) may be executed, namely that pair which corresponds to the configuration.

Examination of the firing sequences of P10.2 - P10.4 indicates that:

(iii) all USE(i) for 1 \leq i \leq 3 will executed concurrently in configuration j, since PB(j) and PE(j) occur around every USE(i), i \neq j.

We will leave further analysis of P10 in the style of the foregoing discussion to the interested reader, who may want to reconsider the examples of the present section after having read the formal exposition in Section 3.

In the next section 2.3 we will be reconsidering the example paths from the standpoint of obtaining a more economical way of writing collections of paths of arbitrary size and structure.

2.3 A Notational Interlude: The Replicators

In the present section we introduce some facilities for obtaining economical representations of paths of arbitrary size and structure. For example, if we consider

(P1) \[\text{path USE}(1), \text{USE}(2), \text{USE}(3) \text{ end}\]

and generalize it to n operations USE(i) by writing

(P1.1) \[\text{path USE}(1), \text{USE}(2), \ldots, \text{USE}(n) \text{ end}\]

we see that one needs to use ellipses. Similarly for

(P4) \[\text{path } (\text{CONF}(1); \text{USE}(1)), (\text{CONF}(2); \text{USE}(2)), (\text{CONF}(3); \text{USE}(3)) \text{ end}\]

the generalization would be something like

(P4.1) \[\text{path } (\text{CONF}(1); \text{USE}(1)), \ldots, (\text{CONF}(n); \text{USE}(n)) \text{ end}\].

13.
When one uses such ellipses extensively and the patterns become longer and the notations more complex it soon becomes impossible to be sure that one has chosen to include the right symbol patterns and ellipses to ensure an unambiguous characterization of the general pattern intended. So it would be a great advantage to be able to replace any general pattern involving repeated subpatterns and ellipses by a generator capable of expanding to the general pattern by means of repeated concatenation of possibly modified copies of the subpatterns.

In COSY we have such a generator called the replicator whose concise definition will require us to introduce some more syntactic formalism. First some grammar:

\[(BS) \text{ basicsymbol} = \text{some finite set of basic symbols not including "@".} \]
\[(I) \text{ index} = \text{some (possibly infinite) set of symbols distinct from basic symbols} \]
\[(IE) \text{ indexexpression} = \text{integer expression involving only indices and integer constants} \]
\[(P) \text{ pattern} = [\text{basicsymbol/index恬]@/replicator} \]
\[(R) \text{ replicator} = [\text{pattern[@pattern[@pattern/]}\text{indexexpression, indexexpression, indexexpression]} \]

Examples of replicators involving patterns formed from the basic symbols of the COSY notation as introduced in P1 and P4 are respectively:

\[(P1.2) \text{ path } [\text{USE(1)@, [1|1,3,1]} \text{ end and} \]
\[(P4.2) \text{ path } [\text{CONF(i);USE(i)@, [1|1,3,1]} \text{ end and} \]

Before we state the general expansion rule for the replicator we stepwise expand P4.2

\[\text{path } [(\text{CONF(1);USE(1)})@, [1|1,3,1]} \text{ end and} \]
\[\text{path } (\text{CONF(1);USE(1)})[1, (\text{CONF(1);USE(1)})[2,3,1]} \text{ end} \]
\[\text{path } (\text{CONF(1);USE(1)}), (\text{CONF(2);USE(2)})[1, (\text{CONF(1);USE(1)})[3,3,1]} \text{ end} \]
\[\text{path } (\text{CONF(1);USE(1)}), (\text{CONF(2);USE(2)}), (\text{CONF(3);USE(3)}) \text{ end} \]

For the statement of the general expansion rule let "p" and "q" be patterns "i" be an index n, m and k be indexexpressions. Furthermore, if "Substitute (p,i,k)" indicates the result of substituting index expression "k" for all occurrences of index "i" throughout P then:

\[(\text{Rule 1) } [p[1] q | n,m,k] = \]
\[ \begin{cases} 
  p & \text{if } (n \leq m \text{ and } k \geq 0) \text{ or } (n < m \text{ and } k < 0) \text{ or } k = 0 \\
  \text{Substitute } (p,i,n)[p[1] q | n+k,m,k] \text{ Substitute } (q,i,n) \text{ otherwise} 
\end{cases} \]

where "" denotes the empty string.

The occurrences of "@" in a replicator of the form \[p@\text{@@}[k,n,m],\]
where "r" and "s" are patterns not involving indices, indicates that "r" and "s" are separators not terminators and it expands according to:

(Rule 2) \[ [p\@{1}q\@{s}|n,m,k] = \]
\[
\begin{cases} 
\epsilon & \text{if } (n>m \text{ and } k>0) \text{ or } (n<m \text{ and } k<0) \text{ or } k=0 \\
\text{Substitute } (p,i,n)[r\@{1}q\@{s}|n+k,m,k] \text{ Substitute } (q,i,n) & \text{otherwise.}
\end{cases}
\]

The reader should test his understanding of the meaning of the replicator by studying the following generalized statement of P10:

(P11)

(P11.1) \[ \text{path} \left[ (\text{CONF}(i);\text{PB}(i);\text{PE}(i))\@{3} | 1,n,1 \right] \text{ end} \]

(P11.2) \[ \text{path}\@{PB}(i)\@{3} | i,n,1] ; \text{USB}(i);[\text{PE}(i)\@{3} | i,n,1] \text{ end } [1|1,n,1] \].

3.1 In the previous section, we attempted to give the reader a feel for the notation from the point of both reading it and writing it. We did this by explaining the 'meaning' of a COSY specification as a set of rules which together say how a system, considered merely as an entity concerned with the execution of a given set of named, uninterpreted actions, called operations, "evolves in time" as its operations are executed, and by using the idea of such a 'meaning' in explaining the design of our example programs.

Such an intuitive understanding seems necessary if the notation is to be used at all, since it will probably be in such terms that the designed system will originally be conceived. However, the notation is not intended merely to provide a terminology for the expression of solutions to design problems, but also to provide a milieu in which these solutions may be analysed, formally, in order that it may be determined whether some 'solution' is indeed a solution. As we have said elsewhere [L37579] concurrent systems are so complex that it is not sufficient to have merely an intuitive understanding of the possible behaviour of a system obeying the design specification. We need to be able formally to verify the design.

In this section we formally define the language of path expressions and give a semantics for this language by means of a mapping which associates with each path expression P a set WFS(P) consisting of vectors whose coordinates are strings made up of operation names belonging to P. The elements of WFS(P) are interpreted as modelling possible discrete, asynchronous behaviours of a system satisfying the constraints defined in P. The business of formally verifying a path expression, may thus be seen to be the activity of determining, by an examination of P, whether WFS(P) possesses certain desirable properties. Such a verification is given in section 5.

3.2 Individual (R^*)-Paths

An individual or R^*-path is a string derived from the non-terminal 'path' by the following production rules:

- path = path sequence end
- sequence = {orelement @_1}+
- orelement = {element @_2,}+
- element = operation/element*(sequence)

where non-underlined lower case words denote non-terminal symbols; the words 'path' and 'end', the comma, the semicolon, the star and the right and left parenthesis are terminal symbols. The expression {nonterminal @_2}^* indicates
expressions of the form 'nonterminal' or 'nonterminal1...nonterminaln', and 'I' indicates alternative substrings. Finally, the non-terminal 'operation' may be replaced by any suitable operation name, usually an ALGOL-like identifier.

With each R^k-path P, we associate its set of operations, Ops(P), and its set of cycles, Cyc(P). In the definition of Cyc(P) that follows, seq (respectively; orel, elem, op), denotes any string derivable from a non-terminal 'sequence' (respectively; 'orelement', 'element' and 'operation').

Cyc(path seq end) = Cyc((seq)) = Cyc(seq)
Cyc(orel1,...;oreln) = Cyc(orel1) ... • Cyc(oreln)
Cyc(elem1,...;elemn) = Cyc(elem1) ... U Cyc(elemn)
Cyc(op*) = Cyc(elem)*
Cyc(op) = {op}.

Here '.' denotes string concatenation, where if X,Y are sets of strings
X.Y = {x.y | x ∈ X ∧ y ∈ Y}. '*' has its usual meaning.

To each R^k-path P, we associate its set FS(P) of firing sequences.
FS(P) = Pref(Cyc(P)*), where for any set X of strings,
Pref(X) = {x | x, y ∈ X, some y}.

We allow X or Y, in the above, to be the null string ε. Thus
ε ε Pref(X) and ε ε Pref(X).

Recall that FS(P) denotes the set of a sequences of operation executions permitted by P.

3.3 General (GR^k-) paths

A general, or GR^k-path is a string of the form P = P_1...P_n, where P_i is an R^k-path, for each i. In future, when we write 'P = P_1...P_n', a GR^k-path, the P_i will be understood to be R^k-paths.

With each GR^k-path, P = P_1...P_n, we associate a set of operations,
Ops(P) = Ops(P_1)U...U Ops(P_n), and a set VFS(P), its set of permitted histories. We now introduce and motivate the definition of VFS(P).

Suppose P = P_1...P_n is a GR^k-path. Let us consider a period of activity of a system S obeying the constraint P. Let us suppose that we have a set of string variables x_1,..., x_n. Initially, all of them are null (x_i = ε, each i). Whenever some operation a executes, x_i is reset to x_i.a if a ∈ Ops(P_i); in other words, each x_i contains a record of these operations in Ops(P_i) which have executed, written in order of execution. Note that this action on the x_i's is well defined, since:

(1) If a and b execute concurrently, then the system contains no constraints relating to the order of execution of a and b, whence, a fortiori, there is no i such that a_i,b ∈ Ops(P_i).
Let us consider $S$ as having run for a while and then having halted. It will have generated strings $x_i \in \text{Ops}(P_i)^*$. What can we say about these $x_i$? Well first, from the desideratum that the order of executions of operations must obey the constraints of all $\Pi^n$-paths in question, we must have

(2) $x_i \in FS(P_i)$ each $i$.

Next, consider what happens if we restart $S$; suppose it executes exactly one operation, $a$, and then halts again. Writing $x_i'$ for the new value of $x_i$, we see that

(3) $x_i' = \begin{cases} x_i + a & \text{if } a \in \text{Ops}(P_i) \\ x_i & \text{otherwise.} \end{cases}$

We can express the above observations more concisely by going to vectors of strings. To backtrack slightly, let us consider a family of sets $A_1, \ldots, A_n$ and the corresponding family of string sets $A_1^*, \ldots, A_n^*$. We may form the Cartesian product of the $A_i^* \times A_i^* = \{ y_1, \ldots, y_n \} | y_i \in A_i^* \}$

and define a concatenation operation on $A$ by

$$(x_1, \ldots, x_n) \times (y_1, \ldots, y_n) = (x_1 y_1, \ldots, x_n y_n).$$

In particular our strings $x_i$ of (i) may be made into a vector $x = (x_1, \ldots, x_n)$, $x \in FS(P_1) \times \ldots \times FS(P_n) \subset \text{Ops}(P_1)^* \times \ldots \times \text{Ops}(P_n)^*$

If we let $x_P = (a_1, \ldots, a_n)$, where

$a_i = \begin{cases} a & \text{if } a \in \text{Ops}(P_i) \\ e & \text{otherwise,} \end{cases}$

then we see that (3) may be expressed

(4) $x' = x \cdot x_P$.

Let us denote by $\text{VFS}(P)$, the set of all vectors $(x_1, \ldots, x_n)$ that might be produced by our system $S$. Let us denote by $\text{Vops}(P)$, the set of vectors $x_P$, $a \in \text{Ops}(P)$. We denote by $\text{Vops}(P)^*$ the closure of $\text{Vops}(P)$ in $\text{Ops}(P_1)^* \times \ldots \times \text{Ops}(P_n)^*$ with respect to vector concatenation. Note that $\text{Vops}(P)^*$ contains a null element $e = (e, \ldots, e)$. Now we have

(5) $e \in \text{Vops}(P)^* \cap (FS(P_1) \times \ldots \times FS(P_n)) \cap \text{VFS}(P)$

(6) if $x \in \text{Vops}(P)^* \cap (FS(P_1) \times \ldots \times FS(P_n)) \cap \text{VFS}(P)$ and $a \in \text{Ops}(P)$

then $x \cdot x_P \in \text{Vops}(P)^* \cap (FS(P_1) \times \ldots \times FS(P_n)) \iff x \cdot x_P \in \text{VFS}(P)$. Thus $\text{VFS}(P) = \text{Vops}(P)^* \cap (FS(P_1) \times \ldots \times FS(P_n))$.

This is the definition we employ.
Let us pause briefly and look at this object $VFS(P)$. By our construction, we see that every element, $x_i$, of it, represents everything that has happened in some possible period of activity of $S$ in the order in which it has happened. We know that we may write

$$x = a_1 \cdot a_2 \cdots a_m \cdot a_i \in VOps(P)$$

In fact possibly $x$ may possibly be written in several ways. Writing $[x]_i$ for the $i$th coordinate of a vector $x$, let us consider a situation in which

$$\forall i \in \{1, \ldots, n\}: [a_i]_i \neq [a_j]_i \Rightarrow [a_i] = [a_0] = \varepsilon.$$

In this case $a_1 \cdot a_2 = a_2 \cdot a_1$ and $x = a_1 \cdot a_2 \cdots a_m$.

A glance at the definition of the vector operations $a_i \cdot b$ shows that if $a \neq b$ then

$$a_p \cdot b_p = b_p \cdot a_p \Leftrightarrow \forall i \in \{1, \ldots, n\}: a \in Ops(P) \Leftrightarrow b \neq Ops(P)$$

that is that no single path constrains the operations $a$ and $b$ to execute in any sequence. We conclude that the following interpretation may be made.

Let $x \in VFS(P)$ and let $a_i \in Ops(P)$ with $a_i \neq b_i$. Then $x \cdot a_i \cdot x \cdot b_p \in VFS(P)$ and $a_p \cdot b_p = b_p \cdot a_p$ may be interpreted as follows: that in the system state determined by $x$, the operations $a$ and $b$ may execute concurrently.

Thus the elements $VFS(P)$ model concurrent (or asynchronous) behaviours of a system obeying precisely the constraint $P$.

Just as string languages are subsets of sets $X^*$, ($X$ some set of terminal symbols) and may be used to model sequential behaviour — as we do with $FS(P)$, $P$ an $FS$-path. — so also may we use subsets of sets like $VOps(P)^*$ to model asynchronous behaviour. The sets $VFS(P)$ are examples of such asynchronous languages, which seems to be related to the class of general asynchronous languages in the way the class of regular languages is related to the class of general string languages. We have no time to go into this here, but we shall discuss some of the operations and relations belonging to elements and subsets of $VOps(P)^*$ as we shall need them in what follows.

Let us therefore fix a GR^k-path $P = P_1 \cdots P_n$. We have already defined concatenation in $VOps(P)^*$. It is clear that $VOps(P)^*$ is a monoid with identity $\varepsilon$ with respect to concatenation. If $x, y \in VOps(P)^*$, we define

$$x \cdot y = [x \cdot y]_{x \in X^* \wedge y \in X^*}.$$

and

$$x^0 = [\varepsilon]$$

$$x^n = x \cdot x^{n-1}$$

$$x^* = x^0 \cup x^1 \cup x^2 \cup \cdots.$$

We may also define a relation $\leq$ (vector prefix) on $VOps(P)^*$ by

$$x \leq y \Leftrightarrow \exists z \in VOps(P)^*: x \cdot z = y$$

($VOps(P)^*, \leq$) is obviously a partially ordered set.
If $X \subseteq \text{Vops}(P)^*$, then we define $\text{Pref}(X) = \{x \in \text{Vops}(P)^* \mid \exists y \in X, \text{some } y \leq x\}$. We observe that $\text{Pref}(\text{VFS}(P)) = \text{VFS}(P)$; the beginning of a behaviour is a behaviour.

Finally, we shall find it convenient to define, for $x \in \text{Vops}(P)^*$ and $a \in \text{Ops}(P)$, the expression $I_a(x)$, which denotes the number of occurrences of $a$ in $x$. Formally, if $b \in \text{Ops}(P)$ and $\exists y \in \text{Vops}(P)^*$

\[
I_a(x \cdot y) = I_a(x) + I_a(b)
\]

\[
I_a(b_0) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise.} \end{cases}
\]
3.4 Verification of path programs

The semantic mapping $P \rightarrow \mathbf{VFS}(P)$ now permits us to formally speak of
dynamic properties of a system specified by a path expression; properties of
its set of possible behaviours that is of $\mathbf{VFS}(P)$. We may speak here of two
types of properties, which we might call general and specific.

General properties are those which, as the name suggests, apply in
general to pairs $(P, \mathbf{VFS}(P))$ as abstract objects, where, roughly, one is not
considering $P$ in relation to any specific interpretation. Among such properties
are freedom from deadlock and adequacy.

- $P$ is deadlock-free if and only if $\forall x \in \mathbf{VFS}(P) \exists a \in \mathbf{Ops}(P) : \exists \alpha \in \mathbf{VFS}(P)$
- $P$ is adequate if and only if $\forall x \in \mathbf{VFS}(P) \forall a \in \mathbf{Ops}(P) : \forall y \in \mathbf{Vops}(P)^*$

Adequacy is a property akin to absence of partial systems deadlock.

We remark that these properties have been extensively studied and results
obtained to assist the analysis of a path expression to detect their presence
or absence [SI78, S79].

Specific properties, on the other hand, are to do with a path expression
or class of path expressions as a description of some actual or projected
mechanism. Here the operation are intended to signify real events. Thus,
in the example in the next section, the operations are intended to denote
actions of a mechanism on a pool of resources. The problem of verifying such
a program is specific in that it applies only to that class and to no other.
For example, in [SI79] the verification of a class of programs relied on the
fact that some of the operations involved were to behave 'exactly like' $P$ and
$V$ operations on extended semaphores.

Verification thus involves establishing that $\mathbf{VFS}(P)$ obeys some predicate
which formally expresses the designer's intentions. We do not deal with the
example of section 4 as formally as this, though it seems likely that such a
formal approach could be developed. The paper of Masurkiewicz [Mas79],
which uses our vector firing sequence model of asynchronous behaviour, is of
interest here in that it presents a formalism in which, conceivably, such an
analysis could be carried out formally.

We do not therefore lay out any general principles for the analysis of
specific properties, but remark that in all the cases we have looked at so
far, the analysis has involved the following

1. We construct a domain, $D$, (for example, the set of all $n$-tuples of
   integers) consisting notionally of all the possible states the inter-
   preted system could be in.
(2) We construct an interpretive function \( f: \mathcal{P}(\mathcal{P}) \rightarrow \mathcal{P} \) where \( f(x) \) is intended to express the state of the interpreted system consequent on a history \( x \).

(3) We interpret some or all of the operations of \( P \) as operators on the domain \( D \).

Verification, in these terms, involves the following kind of activity; establishing that the execution of an operation \( op \), given that the system is in state \( f(x) \), is permitted

(1) if and only if the path expression permits it

(2) if and only if the informal design requirement allows it to execute in state \( f(x) \) and that the state transformation which interprets \( op \) carries the system from state \( f(x) \) to state \( f(x, op) \).

From such an analysis, one may deduce inductively that interpreted executions permitted by the control structure explicit in the path expression under consideration transform the system state in a manner consistent with the specification desideratum.

We remark, in conclusion, that the equivalence aimed for in (1) and (2) above may be established by arguments involving coordinates of vector firing sequences, which are firing sequences of the corresponding paths and whose structure is therefore explicit. This remark is illustrated by the analysis in section 5.
4. **A resource releasing mechanism**

1. **The problem**

   We consider the following situation. We have a pool of \( N \) (reusable) resources \( R_1, \ldots, R_N \) (they could be pages or buffer frames or devices). We wish to describe/specify a mechanism which is such that
   (a) free resources are made available in parallel
   (b) resources which have been borrowed may be replaced in parallel
   (c) (a) and (b) may proceed in parallel.

2. **A Solution**

   In constructing our solution we shall make heavy use of structures developed and explained in section 2.

   To each resource \( R_i \) is associated 3 operations GET\((i)\), signifying that \( R_i \) is secured by some user of the mechanism
   SKIP\((i)\), signifying that an available resource \( R_i \) has not been taken by any user of the mechanism
   PUT\((i)\), signifying that a borrowed \( R_i \) is replaced by a user of the mechanism.

   In a sense, the purpose of the desired mechanism is to ensure correct partial orderings of executions of the operations GET\((i)\), PUT\((i)\) and SKIP\((i)\).

   To do this we make use of the configuration idea and the parallel select idea introduced in section 2.

   After any period of activity of the mechanism (or to every vector firing sequence corresponding to such a period), the pool will be in any one of \( 2^N \) states, corresponding to the \( 2^N \) subsets of \( \{ R_1, \ldots, R_N \} \). If \( C \subseteq \{ R_1, \ldots, R_N \} \) then its corresponding state \( S(C) \) will be that in which the free resources are the operations belonging to \( C \). \( e \) will be called a configuration (of free resources). We will represent each state \( S(C) \) by an integer \( c = c(C) \), \( 0 \leq c < 2^N \) such that if \( \text{bin}(c) = a_1 \ldots a_i \) is its binary representation then \( a_i = 1 \) if \( R_i \in C \). \( c \) thus represents the characteristic function of the set \( C \). To each integer \( 0 \leq c < 2^N \) and integer \( 1 \leq s \leq N \) we define \( \text{bit}(c, r) \) to be the \( r \)th bit in its binary representation i.e.

   \[ \text{bin}(c) = \text{bit}(c, 0) \ldots \text{bit}(c, N). \]

   Let \( C(c) = \{ R_i \mid \text{bit}(c, i) = 1 \} \). Clearly \( c(C(c)) = c \).

   We now define operations \( \text{CONF}(c), \text{PGETB}(c), \text{PGETE}(c) \); \( \text{CONF}(c) \) signifying that a configuration of free frames \( C(c) \) has been detected \( \text{PGETB}(c) \), signifying that the resources \( R_i \) belonging to \( C(c) \) are being made available. \( \text{PGETE}(c) \), signifying the end of a block of acquisitions in parallel of the resources in \( C(c) \).
The mechanism proceeds in cycles or blocks of activity; each cycle begins
with a CONF test followed by the execution of the corresponding GET, which
releases the appropriate free resources, to be either taken (via GET
operations) or ignored (SKIP). The cycle concludes with a corresponding FGETE.

Let us begin with the CONF tests and releases. We have

(4.1) \[ \text{path} \left( \text{CONF}(c) ; \text{FGETE}(c) ; \text{FGETE}(c) \right) @ [2]
0, 2^{N-1}, 1] \text{ end} \]

establishing the cycle.

Next, we introduce the parallel GET, SKIP mechanism. As in the simple
example of section 2, the form of a path for resource \( R_{i} \) is

(4.2) \[ \text{path} \left( \text{FGETE}(c_{1}) , ... , \text{FGETE}(c_{m}) ; \text{GET}(r) , \text{SKIP}(r) ; \text{FGETE}(c_{1}) , ... , \text{FGETE}(c_{m}) \right) \text{ end} \]

where \( c_{1}, ... , c_{m} \) represents those configurations to which \( R_{i} \) belongs.

To express (4.1) without elipses, we shall make the following tentative
extension to the replicator notation of section 2.

Suppose \( P \) is a predicate defined on the integers and suppose \( A \) is a collective
name defined in collectivisor expression array \( A(1:u) \);

Let \( \{ i \in \mathbb{Z} | P(i) \} \cap \{ i_{1}, i_{2}, ... , i_{m} \} = \{ i_{1} , i_{2}, ... , i_{m} \} \) where \( i_{1} < i_{2} < ... < i_{m} \), then if \& is

one of the separators \( \lor \) or \( \land \), we define the expansion of

\[ [A(i) @ P(i)] \]

to be \( A(i_{1}) \lor A(i_{2}) \land ... \land A(i_{m}) \).

When a predicate is used in this way in a program, we give an explicit
definition of it, as we shall illustrate shortly.

In the case of the parallel gets, we clearly require a predicate \( P_{r}(c) \) for
each resource \( R_{r} \).

We have for (4.2) \[ \text{predicate} \ P(c,r) \equiv \text{bit}(c,r) = 1 \]

\[ \text{path} \left( \text{FGETE}(c) @ [P(c,r)] \right) ; \text{GET}(r) , \text{SKIP}(r) ; \]

(4.3) \[ \text{FGETE}(c) @ [P(c,r)] \text{ end [1,N,1]} \]

Note that, in the light of our informal explanation, we are interpreting

\( P(c,r) \) as a set of predicates \( P(c,1), ... , P(c,N) \).

(4.1) and (4.3) together show configuration tests CONF(c) initiating parallel
GETs and SKIPS. In the actual situation, of course, a block of GETs, not to
mention PUTs, will modify the configuration. We thus need to introduce a third
collection of paths describing this 'feedback' effect as a property of the
resource pool.

We remark:

1) An operation \( \text{GET}(r) \) may only execute if some operation \( \text{CONF}(c) \), with \( R_{r} \in C(c) \),
exectutes prior to it;

2) An operation \( \text{PUT}(r) \) may only execute if some operation \( \text{CONF}(c) \), with \( R_{r} \notin C(c) \)
may execute prior to it.

3) \( \text{GET}(r) \) and \( \text{PUT}(r) \) must alternate beginning with a GET.
The operations CONF(c) with $R_c \in G(c)$ are, of course, the operations
\{CONF(c) | F(c,r)\}. For (2), we define a new predicate
\[ Q(c,r) = \text{bit}(c,r) = 0 \]
The 'feedback' or resource pool paths are thus.
\[(4.4) \quad \text{path}([\text{CONF}(c)@, [c|F(c,r)]]^*; \text{GET}(r); \]
\[ ([\text{CONF}(c)@, [c|Q(c,r)]]^*; \text{PUT}(r) \text{ end } 1,1,1] \]
Note that SKIP does not appear in 4.4; after all, why should it? It has no
effect on the configurations of free resources.
Our solution is now:
begin
array CONF, PGETB, PGETE (0:2^N-1);
array GET, PUT, SKIP (1:N);
predicate P(c,r) = \text{bit}(c,r) = 1;
predicate Q(c,r) = \text{bit}(c,r) = 0;
path \[ ([\text{CONF}(c); \text{PGETB}(c); \text{PGETE}(c)]@, [2] 0,2^N-1,1] \text{ end }\]
path \[ \text{PGETB}(c)@, [c|F(c,r)] 1,1,1 \]
\[ \text{PGETE}(c)@, [c|Q(c,r)] 1,1,1 \]
\[ ([\text{CONF}(c)@, [c|F(c,r)]]^*; \text{GET}(r); \]
\[ ([\text{CONF}(c)@, [c|Q(c,r)]]^*; \text{PUT}(r) \text{ end } 1,1,1] \]
end
We shall examine (4.5) in the next section, using the formal tools mentioned
in section 3.
5. Analysis of the resource releasing program

In constructing the program (4.5) we made certain statements in justification of the various constructions introduced. For example, we assumed, in building (4.4), that the CONF test was actually meaningful as an indication of the instantaneous configuration of free resources. Making such assumptions is part of the business of writing programs. That one has made them does not guarantee their accuracy, however.

In this section we are going to sketch a formal proof that (4.5) does what it is supposed to do, and without deadlocking. We call (4.5) RR(N).

One of our tasks will be to characterize the possible behaviors of RR(N), that is, the set VRF(RR(N)).

To begin with, let us express RR(N) as a concatenation of its individual paths RR(N) = SEL(N) PG(1)...PG(N)HP(1)...HP(N) where we are using the mnemonic names SEL (select configuration), PG(i) (i-th parallel get) and HP(i) (i-th resource pool path) for the corresponding paths in 4.5 written in that order.

We shall find it useful to write out the Cycle sets of these individual paths, which are as follows

\[
\text{Cyc}(\text{SEL}(N)) = \{\text{CONF}(0), \text{PG}(0), \text{GET}(0), \ldots, \text{CONF}(2^{N}-1), \text{GET}(2^{N}-1), \text{PG}(2^{N}-1)\}
\]

\[
\text{Cyc}(\text{PG}(i)) = \{\text{GET}(c), \text{GET}(r), \text{GET}(c'), \text{AP}(c,r)\}
\]

where \(P(c,r)\) is the predicate defined in (4.3).

\[
\text{Cyc}(\text{HP}(i)) = \bigcup \{\text{CONF}(c), \text{GEN}(c), \text{CONF}(c'), \text{PUT}(c')\}
\]

P(c,r)A
Q(c',r)

In section 4, we explained that the system was intended to operate in a sequence of 'blocks' of activity, beginning with a configuration test CONF(c), some c. We shall prove this in our first lemmas, in which we begin the characterization of VRF(\text{RR}(N)). We now define the blocks in question. First redefine from section 4 if \(c \in \{0, \ldots, 2^{N}-1\}\), then \(C(c) = \{x \in [1, \ldots, N] | \text{bit}(c,r) = 1\}\); C(c) is the set of indices of free resources in a given configuration c. Likewise we define E(c) to be the corresponding set of indices of empty resources E(c) = \([1, \ldots, N] - C(c)\).

If \text{RR}(N) is working properly, it should be the case that in configuration c, the only GET(i)'s and SKIP(i)'s that may execute must satisfy i \in C(c) and the only PUT(i)'s that may execute must satisfy i \in E(c). A block should therefore be of the form
\[ x(c, X, Y) = \text{CONF}(c) \cdot \text{PGETB}(c) \cdot \text{y}(c, X, Y) \cdot \text{PGETE}(c) \]

where \( y(c, X, Y) \) is of the form

\[ \text{GET}(r_1) \cdot \ldots \cdot \text{GET}(r_i) \cdot \text{SKIP}(r_{i+1}) \cdot \ldots \cdot \text{SKIP}(r_m) \cdot \text{PUT}(r_{m+1}) \cdot \ldots \cdot \text{PUT}(r_n) \]

where \( \{r_1, \ldots, r_i\} \subseteq C(c), \)
\( \{r_{i+1}, \ldots, r_m\} \subseteq c(c) - X \) and
\( \{r_{m+1}, \ldots, r_n\} \subseteq E(c) \cup X \)

Here all operations denote their corresponding vectors; underlining and subscripts have been omitted for the sake of typographical convenience.

Finally, we let

\[ \text{Block}(N) = \{ x(c, X, Y) \mid \exists c \subseteq C(c - 1) \land X \subseteq c(c) \land y \subseteq E(c) \cup X \} \]

Lemma 1

\[ \text{VFS}(\text{RRM}(N)) \subseteq \text{Pref}(\text{Block}(N)*) \]

Proof

Let \( x \in \text{VFS}(\text{RRM}(N)) \). The proof is in a number of steps.

1. We may write
\[ x = [y_0] \cdot \text{CONF}(c_1), y_1 \cdot \text{CONF}(c_2), \ldots, \text{CONF}(c_i), y_i \]
where no \( y_i \) contains any \( \text{CONF}(c) \), for each \( i \) and \( i \geq 0 \).

2. From the path \( \text{SEL}(N) \), we see that \( [y_0] \) contains no \( \text{PGETB} \) or \( \text{PGETE} \) and hence, by the paths \( \text{PG}(i) \), no \( \text{GET}(r) \) or \( \text{SKIP}(r) \). The paths \( \text{RF}(i) \) show that \( y_0 \) contains no \( \text{PUT}(r) \). Thus \( y_0 = \xi \) and if \( x \) contains no \( \text{CONF}(c) \) then \( x = \xi \). Suppose \( x \neq \xi \).

3. The paths \( \text{PG}(i) \) show that for \( 1 \leq i \leq 1 \), \( y_1 = u_i \cdot \text{PGETB}(c_i), v_i \cdot \text{PGETE}(c_i), w_i \)
where \( u_i, v_i \cdot w_i \) consists entirely of \( \text{GET}(r) \), \( \text{PUT}(r) \) and \( \text{SKIP}(r) \).

4. The paths \( \{\text{PG}(i)\} \) show that \( u_i, v_i \cdot w_i \) consist entirely of \( \text{PUT}(r) \). Since no \( \text{PUT} \) belongs to a path containing a \( \text{PGETB} \) or \( \text{PGETE} \), we have \( y_1 = \text{PGETB}(c_i)u_i + v_i \cdot \text{PGETE}(c_i) \).
Let \( y_1' = u_i + v_i \cdot w_i \).

5. Let \( x_1 = \{x_i \mid \text{GET}(r) \text{ is in } y_1'\} \)
\[ z_i = \{z_i \mid \text{SKIP}(r) \text{ is in } y_1'\} \]
\[ x_1 \setminus z_i = \emptyset \]
\( y_i \) is a trifle more involved. From the paths \( \text{RF}(i) \), we see that either

(a) \[ [y_1']_{i+1 \cdot j} = \text{GET}(r), \text{PUT}(r) \text{ or} \]

(b) \[ [y_1']_{i+1 \cdot j} = \text{PUT}(r), \]

where \( r \in E(c) \).

In case (b) \( Q(c_i, r) \) holds, ie \( r \in E(c) \) and \( \text{PUT}(r) \) commutes with all \( \text{GET}(r) \) and \( \text{SKIP}(r) \) in \( y_i' \). In case (a) \( r \notin X_i \) and \( \text{PUT}(r) \) commutes with all \( \text{GET}(r) \) and \( \text{SKIP}(r) \) except \( \text{GET}(r) \). These observations entail that
\[ y_1' = y(c_i, x, y_1) \in \text{Block}(N). \]

27.
Finally, we leave it to the reader to check that
\[ \text{CONF}(c_1) \cdot y_1 \leq x(c_1, x_1, y_1). \]
This completes the proof.

In order to continue and complete our characterisation, we shall find it convenient to draw attention to a special set of vector firing sequences of \( \text{RRM}(N) \), those consisting of a full set of blocks:
\[ \text{FB}(N) = \text{VFS}(\text{RRM}(N)) \cap \text{Block} (N). \]

Suppose \( x \in \text{VFS}(\text{RRM}(N)) \), then \( x = x', \text{CONF}(c) \cdot y, x' \in \text{FB}(N) \) and \( \text{CONF}(c) \cdot y \in \text{Pref}(\text{Block}(N)). \)

As in (5) of Lemma 1, we may define sets \( X, Y, Z \) associated with \( y \). In (6) of lemma 1, we said that \( \text{CONF}(c) \cdot y \leq x(c, X, Y) \). In fact, more that this is true.

**Lemma 2**
1. With the above notation, for any set \( X' \), \( X' \subseteq \text{CONF}(c) \cap Z \), and for any set \( Y', T \subseteq \text{CONF}(c) \cup X \), we have \( x = x' \cdot x(c, c', x' \cdot x(c, X, Y) \in \text{FB}(N) \)
and in particular
2. If \( x \cdot \text{CONF}(c) \in \text{VFS}(\text{RRM}(N)) \), then \( \forall c \subseteq \text{CONF}(c) \forall x \subseteq X \cdot x \cdot x(c, X, Y) \in \text{FB}(N) \)
whence
3. \( \text{VFS}(\text{RRM}(N)) = \text{Pref}(\text{FB}(N)). \)

We must now characterise \( \text{FB}(N) \). We know that its elements are of the form \( x(c_1, x_1, y_1) \).

and we know how the \( x_1 \) and \( y_1 \) are constrained by the \( c_1 \). Hopefully, \( c_1 \) will be determined by the previous configuration \( c \) and what has been done to it (by \( X \) and \( Y \)). With this in mind, we define, for every \( x(c, x, y) \) in \( \text{Block} (N) \)
\[ \text{Next}(c, x, y) = c', \quad \text{where} \]
\[ c(c') = (c(c) \cup y) - x. \]
Clearly \( \text{Next}(c, x, y) \subseteq \{0, \ldots, 2^n - 1\} \).

Our next lemma shows the 'feed back' effects of GET's and PUT's on configuration tests.

**Lemma 3**
Suppose \( x, x \cdot x(c, x, y) \in \text{FB}(N) \). Let \( a \in \text{Vops} (\text{RRM}(N)) \), then \( x \cdot x(c, x, y) \cdot a \in \text{VFS}(\text{RRM}(N)) \)
\[ a = \text{CONF}(\text{Next}(c, x, y)). \]

We omit the proof of this lemma, which uses an argument by coordinates, as in lemma 1.

From lemmas 1, 2 and 3 and induction, we have finally:

**Proposition 1**
1. \( \bar{x} \in \text{FB}(N) \Rightarrow \exists! c_1, \ldots, c_1, X_1, \ldots, X_1, Y_1, \ldots, Y_1: \)
   \( x = x(c_1, x_1, y_1) \cdot x(c_1, X_1, y_1) \)
2. \( x \in \text{FB}(N) \Rightarrow \exists! c_1, \ldots, c_1, X_1, \ldots, X_1, Y_1, \ldots, Y_1: \)
   \( x = x(c_1, x_1, y_1) \cdot x(c_1, X_1, y_1) \)
3. \( x \in \text{FB}(N) \Rightarrow \exists! c_1, \ldots, c_1, X_1, \ldots, X_1, Y_1, \ldots, Y_1: \)
   \( x = x(c_1, x_1, y_1) \cdot x(c_1, X_1, y_1) \)
(b) \( \forall i \in \{1, \ldots, l\}: X_i \subseteq c_i \land X_i \subseteq \mathcal{F}(c_i) \cup X_i \)

(c) \( \forall i \in \{1, \ldots, l-1\}: c_{i+1} = \text{Next}\(c_i, X_i, Y_i\)\)

(a) \( c_1 = 2^{N-1}\).

(2) \( \text{VFS}(\text{RRM}(N)) = \text{Pref}(\text{FB}(N))\).

Now that we have characterised the behaviour of RRM(N), we may ask ourselves whether it does what it's supposed to. We define a function

\[ f: \text{VFS}(\text{RRM}(N)) \rightarrow 2^N, \]

where \( f(x) \) will be designed to represent the content of the resource pool after \( x \) has happened: for \( x \in \text{VFS}(\text{RRM}(N)) \), \( i \in \{1, \ldots, N\} \), define \( f(x) = g(x) + 1 \) where \( 1 = (1, \ldots, 1) \) and

\[ [g(x)]_i = \mathcal{I}_{\text{PUT}(i)}(x) - \mathcal{I}_{\text{GET}(i)}(x). \]

Consideration of \( \text{Gyc}(\text{RR}(I)) \), \( i \in \{1, \ldots, N\} \), shows that \( [f(x)]_i \in \{0, 1\} \) for each \( x \in \text{VFS}(\text{RRM}(N)) \). Clearly \( [f(x)]_i = 1 \) if and only if \( R_i \) is free. \( f(x) \) thus represents the configuration of free resources consequent on history \( x \) of RRM(N). Our program will thus be doing its job correctly if the configuration tests \( \text{CONF}(c) \) test \( f(x) \) correctly. We shall now show that they do. First a simple lemma.

**Lemma 4**

(1) \( f(e) = 2^N - 1 \)

(2) \( \forall x, y \in \text{VFS}(\text{RRM}(N)) \cdot f(x, y) = g(x) + g(y) \)

We now tie the \( \text{CONF}(c) \)'s and \( f(x) \)'s together. First, for integers \( c \), define \( v(c) = (\text{bit}(c, 1), \text{bit}(c, 2), \ldots, \text{bit}(c, N)) \).

**Lemma 5**

Let \( x(c, X, Y) \in \text{Block}(N) \), then \( v(\text{Next}(c, X, Y)) = v(c) + g(x(c, X, Y)) \).

From this, we may deduce

**Proposition 2**

Let \( x \in \text{FB}(N) \) then

\( x \cdot \text{CONF}(c) \in \text{VFS}(\text{RRM}(N)) \Rightarrow v(c) = f(x) \).

**Proof**

By proposition 1, \( x = x(c_1, X_1, Y_1) \ldots x(c_l, X_l, Y_l) \) with the appropriate conditions on the \( c_i, X_i \) and \( Y_i \). Using lemmas 4 and 5 we see

\[ g(x) = \sum_{i=1}^{l} g(x(c_i, X_i, Y_i)) \]

\[ = \sum (v(c_{i+1}) - v(c_i)) = v(c_{i+1}) - v(c_i), \]

where \( c_{i+1} = \text{Next}(x(c_i, X_i, Y_i)) \). But \( c_1 = 2^{N-1} \), whence \( g(x) = \text{Next}(x(c_1, X_1, Y_1)) - 1 \) or \( f(x) = \text{Next}(c_1, X_1, Y_1) \). Lemma 3 completes the proof.

From proposition 3, we now have the proof of correctness of RRM(N), as follows.
(1) If \( x \) is a history of the system, then a configuration test applied at this point correctly identifies the configuration of [Proposition 2].

(2) GET's can only execute on free resources and PUT's can only execute on borrowed resources [from Proposition 1, the definition of \( x(c, X, Y) \) and Proposition 2].

(3) In any block all GET's, PUT's, and SKIP's execute in parallel. [because the corresponding vector operations commute] apart from those GET(1) and PUT(1) that occur in the same block.

We conclude this section with a proof of adequacy. Freedom from deadlock follows more or less immediately from lemmas 2 and 3. Adequacy, which implies freedom from deadlock, follows from the next lemma.

Lemma 6

(1) \( \forall x \in \{0, 1\}^N \forall c \in \{0, \ldots, 2^N - 1\} \exists X \subseteq (c) \exists X \subseteq (c): x = g(x(c, X, Y)) + v(c) \)

(2) \( \forall \alpha \in \text{Vops}(\text{HRM}(N)) \exists X \subseteq \text{Block} (N): I_{\alpha}(x) = 1. \)

Proof (Sketch)

(1) Let \( \chi = [1][v-v(c)]_1 = -1 \) and \( \psi = [1][v-v(c)]_1 = 1 \)

(2) If \( \alpha \in \{\text{CONF}(c), \text{GET}(c), \text{GET}(c)\} \), let \( x = x(c, \psi, \psi) \)

If \( \alpha = \text{GET}(x) \), let \( x = x(2^{N-1}, \{\text{GET}(x)\}, \psi) \).

If \( \alpha = \text{SKIP}(x) \), let \( x = x(2^{N-1}, \psi, \psi) \).

If \( \alpha = \text{PUT}(x) \), let \( x = x(\psi, \psi, \{\text{PUT}(x)\}) \).

Proposition 3

For all \( N \geq 0 \) \( \text{HRM}(N) \) is adequate.

Proof

Let \( x \in \text{VFS}(\text{HRM}(N)) \), \( \alpha \in \text{Vops}(\text{HRM}(N)) \). We must find \( y \in \text{Vops}(\text{HRM}(N))^* \):

\[ x.y \in \text{VFS}(\text{HRM}(N)) \] \( I_{\alpha}(y) = 1 \).

By lemma 2 we may assume \( x \in \text{FB}(N) \). By lemma 6 (2), \( I_{\alpha}(x(c, X, Y)) = 1 \), some \( x(c, X, Y) \in \text{Block}(N) \). Let \( c \in \{0, \ldots, 2^N - 1\} \) be the unique integer such that \( v(c') = f(x) \), and let \( X \subseteq c(c') \), \( Y \subseteq c(c') \) such that \( v(c') + g(x(c', X, Y')) = v(c) \). (Lemma 6(1)).

Now \( x \cdot \text{CONF}(c') \in \text{VFS}(\text{HRM}(N)) \) [Proposition 2] so

\[ x.x(x(c', X, Y') \in \text{VFS}(\text{HRM}(N)) \] [lemma 2] and \( f(x, x(x(c', X, Y'))) = f(x) + g(x(c', X, Y')) = v(c') \)

Thus \( x.x(x(c', X, Y') \cdot \text{CONF}(c) \in \text{VFS}(\text{HRM}(N)) \) [proposition 2]

[proposition 2]. whence \( x.x(c', X, Y') \cdot x(c, X, Y) \in \text{VFS}(\text{HRM}(N)) \) [lemma 2]

Let \( y = x(c', X, Y') \cdot x(c, X, Y) \) then \( x.y \in \text{VFS}(\text{HRM}(N)) \) and \( I_{\alpha}(y) = 1 \), by construction.
Hence RRM(N) is adequate.

We have now established that RRM(N) is adequate (and hence deadlock free) and performs as required.

We remark that the program RRM(N) is actually part of a rather larger program specifying a highly concurrent and distributed COSY solution to the Banker's Problem [Dij68], given in [LMGO]. This larger program contains, as well as RRM(N), which acts as a 'kernel' of it, the specification of access to the kernel by customers, a specific resource-to-customer allocation mechanism, various devices, based on counters, which ensure fairness and absence of starvation and a specification of customer structure. A proof of the correctness of the banker program, in a style similar to the above, has been sketched.
Conclusion

We have attempted to give the reader an idea of the potentials of COSY, both as a precise means for expressing a solution to an asynchronous design problem and also as a formalism in which such solutions may be analysed to any desired depth of rigour.

This paper does not cover the full range of our present understanding of the notation, nor has it spoken of the relationships between the work presented here and related work in the field.

We have not here presented the full COSY notation as presently developed; for example we have avoided the topic of processes, distributors or multiply nested distributors. Details of these language features may be found in [LTS79].

There also exist a number of results concerning the problem of deducing the adequacy properties of a system defined by a path from the syntactic structure of that path. Some of these may be found in [S79].

Finally, there has always been a strong connection between the COSY notation and Net Theory. The original formal semantics for the basic COSY notation was given in terms of a mapping from programs to nets [LC75] and this was developed further in [LS879] and [LS879a]. A relationship between paths and nets on both the system (path to net) and process (vector firing sequence to causal net) levels is given in [S79].

Acknowledgements

The research reported in this paper has been encouraged and supported by the Science Research Council of Great Britian. Some of the examples in this paper were developed in collaboration with P.R. Torrigiani and R. Devillers [LTD80]. Thanks to Mrs. Diane Pelley for her excellent typing on such short notice.
Bibliography


