Abstract

This report is intended as a fairly relaxed introduction to COSY, an environment for the design and analysis of concurrent and distributed systems. We begin by introducing the basic and macro notation via a sequence of simple programming problems. We then give a formal semantics for the notation based on vector firing sequences and use it to demonstrate, for a given problem, how one may formally verify a COSY specification. System COSY is then introduced, involving class-like constructs which enable the expression of modularity and hierarchy. The report concludes with a section relating COSY to other design and analysis environments, specifically Petri Nets and the regular expression based formalisms of Riddle and Shaw.

Presented at the Symposium on Software Engineering Environments, Lahnstein, West Germany, June 16–20, 1980.

COSY: An environment for development and analysis of concurrent and distributed systems

By

P.E. Lauer and M.W. Shields

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 153
June, 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.
This report is intended as a fairly relaxed introduction to COSY, an environment for the design and analysis of concurrent and distributed systems. We begin by introducing the basic and macro notation via a sequence of simple programming problems. We then give a formal semantics for the notation based on vector firing sequences and use it to demonstrate, for a given problem, how one may formally verify a COSY specification. System COSY is then introduced, involving class-like constructs which enable the expression of modularity and hierarchy. The report concludes with a section relating COSY to other design and analysis environments, specifically Petri Nets and the regular expression based formalisms of Riddle and Shaw.

About the author
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.

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.
1. **Introduction: The COSY methodology**

Our intention in this paper is to present, in a gentle and didactic manner, an environment for the development, analysis and implementation of complex concurrent and distributed software systems, which goes under the name COSY (COSY is a kind of "second order acronym" for Concurrent Systems).

In the process of developing COSY, we gradually came to formulate what we now consider to be some important requirements that should be possessed by a design methodology for concurrent and distributed systems. Firstly we considered that it was important to encourage the designer to rid himself, as much as possible, of those preconceptions he might have developed as a result of his experience in sequential programming, where it is natural to think of systems as sequential, centralised and synchronous. Indeed, we felt that it would not suit our purpose to base our approach on any kind of modification of a sequential programming language, since in such cases, the designer is obliged to make his understanding of his problem conform to the available programming language constructs. This is undesirable in the sense that the solution of his problem becomes confused with the implementation of the solution, that the structure of the solution may be hidden behind a mass of implementation details which are not in themselves an essential part of it. Another language might serve just as well. This is particularly important if one wishes, as we do, to concentrate on the specification of synchronisation properties of systems rather than the problem-oriented functions these systems perform. To base the development (rather than the implementation) of a solution to a systems organisation problem on some specific language construct (such as semaphores or monitors) would be rather like basing the business of solving a problem in linear algebra (as opposed to implementing the solution) on FORTRAN as opposed to the appropriate mathematics. In view of this, we chose to specify synchronisation properties in terms of the general notion of permitted behaviours in a linguistic or black box manner. A system will be associated with a grammatical type object whose "language" models a set of permitted or required behaviours.

Our system design methodology should therefore be built around a notation which should abstract from non-synchronisation semantics of conventional programming constructs and specific synchronisation mechanisms.

However, it should be possible to relate the notation both to any other well defined programming notation and to any other major theory of concurrent and distributed systems.

The ideal candidate for such a notation was the path notation, due to Campbell and Habermann [CH74], which was designed so that one could state the proper coordination of concurrent processes as the permissible order of execution of operations on shared system objects as part of the object definition. The idea behind the Campbell-Habermann path concept was put into a more abstract form in the generalised path expressions of Lauer and Campbell [LC75]. Here we have, as required, an abstract notation of a linguistic kind. Indeed a path expression is essentially a collection of regular grammars, represented as regular...
expressions. Just as a single regular expression determines a set of strings, each of which may be considered as a labelled total order modelling a sequence of executions of the operations which label it, so may a collection of regular expressions determine a set of vectors of strings, each of which may be considered as a labelled partial order, modelling a non sequential behaviour of operation executions.

The path notation thus satisfies our requirements for a notation for abstractly describing concurrent systems. It also satisfies our requirement that our notation should be easily relatable to other major theories of concurrent systems. The relationship between regular expressions and state machines is well known. There is a relationship between path expressions and Petri-nets which generalises this relationship. In fact every path expression defines a safe net which decomposes into a set of state machines corresponding to the regular expressions from which the path expression is composed. The path expression and corresponding net may each be shown to define exactly the same set of asynchronous behaviours [LC75, S79]. We shall outline this relationship later.

An additional requirement is that the methodology should contain facilities for the verification of systems designed in its notation. The semantics in terms of nets, or equivalently, the semantics which regards a path expression as generating a language of vectors, provide a mathematical milieu for the formal definition of systems properties and for the analysis of a path expression to see whether the system it defines possesses such properties. The choice of regular expressions, rather than some more general structure with the power of Turing machines means, in addition, that all such properties are in principle decidable.

So far, then, we have a notation for describing concurrent systems in an abstract manner, clearly related to Net Theory and possessed of a clean behavioural semantics which provides a firm mathematical foundation for verifying behavioural properties of systems. The further evolution of the COSY methodology was conditioned by other desiderata that we considered essential in a software design environment.

The first is a matter of convenience for the programmer and a facility for generalisation; it is required that regular expressions of arbitrary size and complexity should be capable of being defined in a succinct and illuminating manner, particularly when the regular expression in question has the kind of regularity of structure which might be expressed using iteration. For this, the replicator notation, which contains facilities for the iterative definition of regular expressions involving indexed operation names, was introduced.

It also seemed that the notation should contain facilities for the expression of hierarchy and modularity, in the sense that it contains features which allow the programmer to introduce levels of abstractness into his design and to practice information hiding and which support other techniques of structured programming. Furthermore it should facilitate the design of distributed systems consisting of subsystems which are capable of proper concurrent behaviour without the presupposition of a centralised or global system state or single clock. For these purposes, the notation has been equipped with a class-like construct, for which we use the term system.

Finally, we need to be able to systematically relate COSY system specifications with implementations in any programming notation. The system construction also contains means for associating specifications with implementations.

The rest of the paper is structured as follows; in the next section the basic notation is introduced slowly and largely by means of examples. Section three is concerned with verification. Here the behavioural semantics is introduced and its use is illustrated by means of an example, a simple spooling system with interrupt mechanism. Section four deals with the system notation, its use in expressing distribution and hierarchy and how to relate specifications with implementations.
Section five deals briefly with the relationship to nets. There is a short conclusion.

2. Introduction to COSY programming in "Basic" COSY.

In this section we introduce the COSY notation \cite{LTS79} 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 \cite{B76} and Communicating Sequential Processes \cite{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 \cite{LTS79}.

Our approach to the study of systems leads us to regard a system as characterisable 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 synchronic 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 \(c\) 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 relationships 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 constraints embodying such a relationship in the form of a grammar. The strings belonging to the language of the grammar represent the strict sequences 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 \([LSB79]\) and \([LTS79]\).

2.1 Sequential system examples

By a sequential system we will mean a system of operations \(o_1, \ldots, o_n\) 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 \(o_1, \ldots, o_n\) such that some of the \(o_i, o_j\) 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 \(o_1, \ldots, o_n\) such that some of the \(o_i, o_j\), 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 or 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. '*' 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 \(USE(1), USE(2), USE(3)\), 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:

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

From \((N3)\) we see that the regular expression

\(USE(1), USE(2), USE(3)\)

means

"exactly one of \(USE(1)\) or \(USE(2)\) or \(USE(3)\) may be executed"

and the whole path \((P1)\) with its implicit outermost Kleene star means
repeatedly, exactly one of USE(1) or USE(2) or 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 3.
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

Pref(Cyc(P))° = Cyc(P)°*Pref(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 (P) to
stand for the path it labels to write:

Cyc(P1) = {USE(1), USE(2), USE(3)}

and

FS(P1) = {USE(1), USE(2), 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 E1.j replaces all conditions E1.i, i<j. 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) path USE(1); USE(2); USE(3) end

for which

Cyc(P2) = {USE(1), USE(2), USE(3)}

and

FS(P2) = {USE(1), USE(2), USE(3)}°°{e, USE(1), USE(1), 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) \[ \text{path USE(1);USE(2),USE(3) end} \]

for which

\[ \text{FS(P3) = \{USE(1),USE(2),USE(1),USE(3)\}$\epsilon$\{\epsilon,USE(1)\}.} \]

P3 indicates that $\epsilon$ binds stronger than $\epsilon$'. 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 $\epsilon$' 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. [79]). 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

\[ \text{FS}_x(P) = \{y \mid x \ast y \in \text{FS}(P)\} \]

In general it is not true that cycles always return a system to its initial state; (the simplest example to hand is

\[ \text{path (a; b'); (c; d')end} \]

but it may be asserted in general that in cases where we do not have

$x \in \text{Cyc}(P), x \in \text{Cyc}(P)$ and $x \in \text{FS}(P)$, (what we have called pathological cases) then it is true that if $x \in \text{Cyc}(P)$ then $\text{FS}(P) = \text{FS}_x(P)$, that is, $S(x) = S(\epsilon)$, that is $x$ returns us to our initial state.

We see that we may identify states, $S(x)$, with 'continuations' $\text{FS}_x(P)$ 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

$x \in \text{FS}(P)$ then $x = y \ast z$, where $y \in \text{Cyc}(P)$ and $z \in \text{Pref}(\text{Cyc}(P))$. In non-pathological cases

\[ \text{FS}_x(P) = \text{FS}_y(P) \]

or

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

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(i) 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(i) may execute in any state of the system, since any USE(i) 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(P1) = {USE(1), USE(2), USE(3)} and hence FSx(P1) = FS(P1)

for each x ∈ FS(P1). 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 CONF(1), CONF(2), CONF(3) which can be used to test in which of 3 possible configurations the system is, and E1.2 is replaced by:

E4.2 USE(i) may be executed only if the system is in configuration i.

This can be written as:

(P4) path (CONF(1);USE(1)),(CONF(2);USE(2)),(CONF(3);USE(3))end

for which

FS(P4) = {CONF(1).USE(1),CONF(2).USE(2),CONF(3).USE(3)}

Examination of FS(P4) indicates that USE(i) does not determine a state, i.e. USE(i) ∈ Pref(Cyc(P4)) whereas CONF(i) does determine a state for 1≤i≤3.

Examination of Cyc(P4) indicates that CONF(i).USE(i) does bring the system to its initial state for every 1≤i≤3. From these observations it follows that it is impossible:

(i) for USE(i) to be executed before CONF(i) has been executed immediately prior to it, and

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

To be more precise:

USE(i) may be executed now if and only if ...,CONF(i) is the current history of execution.
Hence, 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 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:

E5.3 execution of 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 USE(i) operations has a direct 'feedback' on the conditions which control their execution. A solution could be written as:

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

for which

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

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

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

At this point it is illuminating to note that we can specify the sequential system E5 by using two paths in combination (that is, concatenated). Hence, we can combine the sequential subsystem E4 with a path of the form

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

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

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

(P6)

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) \[FS(P4) \times FS(P6) =\]

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

\[\text{CONF}(1),\text{CONF}(2),\text{CONF}(3))] \times ([\text{CONF}(1).\text{CONF}(2).\text{CONF}(3)]* \]

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 FS(P4)×FS(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), conprecetable

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 Cyc(P6) then a corresponding firing sequence is Conf(1),Conf(2),Conf(3)\inCyc(P4), where Cyc(P4) indicates the concatenation of three copies of Cyc(P4). Note that the
corresponding firing sequence is a concatenation of the three distinct elements of \( \text{Cyc}(P4) \). In fact these two cyclic histories, when combined into a vector, act as a kind of 'generalised cycle', generating all behaviours of the combined path \( P7 \).

Now note that

\[ \text{CONF}(1).\text{USE}(1).\text{CONF}(2).\text{USE}(2).\text{CONF}(3).\text{USE}(3) \in \text{Cyc}(P5) \]

and that it is the only element of \( \text{Cyc}(P5) \). Note that path \( P4 \) and \( P5 \) share exactly the same operations. If we now consider the vector corresponding to \( P1 \) and compare the history in the coordinate corresponding to the path \( P4 \) with the history of \( P5 \) 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 \( (P7) \), the cyclic history of the combined system is just the cyclic history of the subsystem \( (P4) \). Thus, from the standpoint of this observer the behaviour of the system corresponding to \( P7 \) is the same as that of the system corresponding to \( P5 \).

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 \( x \in \text{Cyc}(P_1) \cdot \text{Cyc}(P_1)^{-1} \), that we can find \( y \in \text{Cyc}(P_2) \cdot \text{Cyc}(P_2)^{-1} \) such that \( x \) and \( y \) are congrueeble. Indeed for paths in which 't' 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 any \( P_1 \cdot P_2 \) has 'generalised cycles' such as are possessed by \( P7 \).

2.2 Concurrent system examples

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

**Example 6**

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

\begin{itemize}
  \item E6.1 Some operations may be executed concurrently.
  \item E6.3 Execution of \( \text{TR}(i) \) leads from configuration \( i \) to configuration \( i+1 \) if \( i \leq 3 \) and 1 otherwise.
  \item E6.4 Execution of \( \text{TR}(1) \) and \( \text{USE}(1) \) should be concurrent (i.e. not necessarily interleaved)
\end{itemize}

This can be written by combining \( P4 \) and \( P8 \)

\[
(P8) \quad \text{path} \ \text{CONF}(1);\text{TR}(1);\text{CONF}(2);\text{TR}(2);\text{CONF}(3);\text{TR}(3) \end{path}
\]

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

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

\[
(P2) \quad \text{CONF}(1).\text{TR}(1).\text{CONF}(2).\text{TR}(2).\text{CONF}(3).\text{TR}(3) \in \text{Cyc}(P8)
\]

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

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

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

both agree with \( (P2) \) on the order of the operations they have in common. Note also that \( (P4) \) belongs to \( \text{Cyc}(P4)^2 \). The vector

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

acts thus rather like a 'generalised cycle' and we can write out the set of vector histories of \( (P9) \) as
\[
\left\{ \begin{array}{c}
\text{CONF}(1), \ldots, \text{USE}(3) \\
\text{CONF}(1), \ldots, \text{TR}(3)
\end{array} \right\}^* \left\{ \begin{array}{c}
\epsilon \\
\text{CONF}(1), \text{CONF}(1), \text{CONF}(1)
\end{array} \right\},
\]
\[
\left\{ \begin{array}{c}
\text{CONF}(1), \text{TR}(1) \\
\text{CONF}(1), \text{TR}(1)
\end{array} \right\} \text{, USE}(1), \text{CONF}(1), \ldots, \text{CONF}(3)
\]

where '*' denotes coordinatwise concatenation.

The reader should observe the manner in which \( P8 \) constrains the execution of operations belonging to \( P4 \). The set of firing sequences of \( P4 \) is, as we have seen,

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

whereas the set of firing sequences which are congrueable to some firing sequence in \( P8 \) are

\[
CFS_{P8}(P4) = \{ \text{CONF}(1), \text{USE}(1), \text{CONF}(2), \text{USE}(2), \text{CONF}(3), \text{USE}(3) \}^*.
\]

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

a proper subset of \( FS(P4) \). \( CFS_{P8}(P4) \) is the set of firing sequences congrueable with others in \( P9 \), or more precisely the '4' projection of the congrueable 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 \( CFS_{P8}(P4) \) and that no executions described in \( FS(P4) - CFS_{P8}(P4) \) are permitted.

Now note that every time \( \text{CONF}(i) \) is executed and, hence, coincidentally extended to both the histories of \( P4 \) and \( P8 \), both \( \text{USE}(i) \) and \( \text{TR}(i) \) must be executed before \( \text{CONF}(i+1) \) modulo 3 can be executed, i.e. appended coincidentally to both histories. However, nothing is prescribed concerning the ordering of executions of \( \text{USE}(i) \) and \( \text{TR}(i) \) relative to each other, hence, they may be executed (appended to the respective histories) concurrently.

**Example 7**

As example 4 with regard to the operations \( \text{USE}(i) \) and \( \text{CONF}(i) \). However

**E7.1** \( \text{USE}(i), \text{USE}(j), i \neq j \) may be executed concurrently, and

**E7.2** \( \text{USE}(i) \) may be executed only if the system is in configuration \( j \).

This means \( \text{USE}(i) \) for all \( 1 \leq i \leq j \) may execute concurrently in configuration \( j \). It also means that \( j \) \( \text{USE}(i) \)'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 \( \text{parallel}_\text{begin} \), denoted 'PB(i)', and \( \text{parallel}_\text{end} \), denoted 'PE(i)', which are used to relate the \( \text{CONF}(i) \) to the \( \text{USE}(i) \) in the manner required by Example 7. Now a solution can be written as

(P10)

(P10.1) \( \text{path} \{ \text{CONF}(1); \text{PB}(1); \text{PE}(1) \}, \{ \text{CONF}(2); \text{PB}(2); \text{PE}(2) \} \),

\( \text{path} \{ \text{CONF}(3); \text{PB}(3); \text{PE}(3) \} \),

(P10.2) \( \text{path} \text{PB}(1), \text{PB}(2), \text{PB}(3); \text{USE}(1), \text{PE}(1), \text{PE}(2), \text{PE}(3) \),

(P10.3) \( \text{path} \text{PB}(2), \text{PB}(3), \text{USE}(2), \text{PE}(2), \text{PE}(3) \),

(P10.4) \( \text{path} \text{PB}(3), \text{USE}(3), \text{PE}(3) \).

for which the respective sets of firing sequences are
\( \text{FS}(P10.1) = [\text{CONF}(1).PB(1).PE(1), \text{CONF}(2).PB(2).PE(2), \text{CONF}(3).PB(3).PE(3)]^* \)
\[ \{e, \text{CONF}(1), \text{CONF}(1).PB(1), \text{CONF}(2), ..., \text{CONF}(3).PB(3) \} \]

\( \text{FS}(P10.2) = \)
\[ \{\text{PB}(i).\text{USE}(j).\text{PE}(i)|1 \leq i, j \leq 3\}^* \]
\[ \{\text{PB}(i)|1 \leq i \leq 3\} \{e, \text{USE}(1)\} \]

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

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

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 \( PE( ) \) 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 \( \leq i \leq j \) will be executed concurrently in configuration j, since PB(j) and PE(j) occur around every USE(i), 1 \( \leq i \).

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: Replicators, Collectivisors and Distributors

In the present section we introduce some facilities for obtaining economical representations of paths of arbitrary size and structure.

2.3.1 Replicators

For example, if we consider

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

and generalize it to n operations \( \text{USE}(i) \) by writing

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

we see that one needs to use \textit{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) \( \text{path } (\text{CONF}(1); \text{USE}(1)), ..., (\text{CONF}(n); \text{USE}(n)) \text{ end} \).

When one uses such ellipses extensively and the patterns become longer and the nestings 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 \textit{replicator}, whose concise definition will require us to introduce some more syntactic formalism. First some grammar:

(BS) \textit{basicsymbol} = some finite set of basic symbols not including "@".

(I) \textit{index} = some (possibly infinite) set of symbols distinct from basic symbols.

(IE) \textit{indexexpression} = integer expression involving only indices and integer constants.

(P) \textit{pattern} = \{basicsymbol/index\}*#/replicator

(R) \textit{replicator} = \{pattern[@[i,?]/] [1]pattern[@[i,?]/] [1]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) \textbf{path} [\texttt{USE(i)}@, [i | 1,3,1] end]

and

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

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

\textbf{path} [\texttt{CONF(i);USE(i)}@, [i | 1,3,1] end]

\textbf{path} [\texttt{CONF(i);USE(i)}@, [i | 2,3,1] end]

\textbf{path} [\texttt{CONF(i);USE(i)}], [\texttt{CONF(i);USE(i)}] [i | 3,3,1] end

\textbf{path} [\texttt{CONF(i);USE(i)}], [\texttt{CONF(i);USE(i)}], [\texttt{CONF(i);USE(i)}] end

For the statement of the general expansion rule let "p" and "q" be patterns "i" be an index, \( n \) 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:

(Rule 1) \[ p \emptyset q \mid n, m, k = \]

\textbf{Df}

\{ \begin{align*}
& \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)[p \emptyset q [n+k, m, k] \text{ Substitute } (q, i, n) \text{ otherwise}; \text{ where}
\end{align*}

"\emptyset" denotes the empty string.

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

(Rule 2) \[ \texttt{[p}@[\texttt{q}@@[n,m,k] = \]

\textbf{Df}

\{ \begin{align*}
& \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)[p \emptyset q [n+k, m, k] \text{ Substitute } (q, i, n) \text{ otherwise}.}
\end{align*}

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

(P11)

(P11.1) \textbf{path} [\texttt{CONF(i);PB(i);PE(i)}@, [i | 1,n,1] end]

(P11.2) \textbf{path} [\texttt{PB(i)}@, [i | 1,n,1] \texttt{USE(i);PE(i)}@, [i | 1,n,1 end] [i | 1,n,1] end]

2.3.2 Collection

We will require that indexed names such as \texttt{CONF(i)}, \texttt{PB(i)}, \texttt{PE(i)} and \texttt{USE(i)} be declared in a \textit{collection} by writing:

\[ \texttt{array CONF}, PB, PE, USE(1:n) \]
which expands to give \(4n\) operations

\[
\text{CONF}(1), \text{CONF}(2), \ldots, \text{CONF}(n), \ldots, \text{PB}(n), \ldots, \text{PE}(n), \ldots, \text{USE}(n).
\]

2.3.3 Distributors
Replicators with Patterns involving indexed names of the form

\[
[\text{pattern} @ \text{separator} [\text{index} | 1, n, 1]]
\]

for example the patterns in (P1.2), (P4.3) and (P11.1) above, occur so frequently that it has been distinguished as a special feature of the notation, the distributor. If we have a collectivisor as in (C1) we can abbreviate (P11.1) to

\[
(\text{path}, \text{CONF}; \text{PB}; \text{PE}) \quad \text{end}
\]

using a special case of the general definition of the distributor of the form

\[
((\text{CONF}; \text{PB}; \text{PE})) = [((\text{CONF}(i); \text{PB}(i); \text{PE}(i)))_{\mathbb{I}} | 1, n, 1]
\]

Def

Similarly, since we can use any combination of separators ",", and ":", provided the corresponding dimensions of all collective names occurring in the pattern of the replicator are the same, we could write for instance

\[
(\text{path}, (A(i)) \quad \text{end} \quad \text{which}
\]

expands to

\[
(\text{path} \quad A(1,1); A(2,1), A(1,2); A(2,2)) \quad \text{end}
\]

provided one has declared

\[
(\text{array} \quad A(1,2,1,2)
\]

previously.

For a more complete definition of nested distributors for multidimensional arrays of operations we refer the reader to the relevant sections of [LTS79, LBS79].

2.4 Processes and their conversion to paths
Suppose we wanted to express the fact that there are \(m\) processes in the system described by the paths in P11 each capable of using any of the USE(i) operations. Then we would simply add to P11

\[
[\text{process} \quad \ldots; (\text{USE})_{\mathbb{I}} \quad \ldots \text{end} | 1, m, 1].
\]

To see precisely what the addition of these processes means consider the following rule for converting a program involving paths and processes into an equivalent one involving only paths.

Given

\[
R = \text{program} \; P_1 \ldots P_n \; Q_1 \ldots Q_m \; \text{endprogram}
\]

where the \(P_i\) are paths and the \(Q_j\) are processes, then \(\text{Path}(R)\) denotes the result of converting \(R\) into a program involving only paths according to the following rule

(Rule 3) 1. Replace every occurrence of operation 'a' in process \(Q_j\) by 'a\&j';

2. Replace every occurrence of operation 'a' in process \(Q_j\) by an element of the form

\[
a_{i_1}, \ldots, a_{i_e}
\]

where \([i_1, \ldots, i_e]\) is the set of indices of all the processes involving 'a';

3. Replace all occurrences of 'process' by 'path'.

13
Note that the rule assumes that one and the same operation name occurring in two processes indicates that the processes may execute them concurrently unless the operation additionally occurs in at least one path.

For example

\[(P16)\text{ program } process \text{ deposit end process deposit end endprogram}\]

translates to

\[(P17)\text{ program path deposit&1 end path deposit&2 end endprogram.}\]

Now \(V_{ops}(P17) = V_{ops}(\text{Path}(P16)) = [(\text{deposit&1}, e),(e, \text{deposit&2})]\)

whence concatenation commutes, that is, they are concurrent.

Finally, \(\text{Path(program } (P11)(P15) \text{ endprogram)}\) gives rise to

\[\text{program}\]
\[\text{path, (CONF;FB;PE) end}\]
\[\text{[path(FB(j))@, } [1, n, 1] \text{; USE(i)&k @, } [1, n, 1] \text{; PE(j)} @, [1, n, 1] \text{ end}\]
\[\text{(P18)}\]

endprogram

3. Construction and Analysis of an Example Program: A Spooler with Interrupts

3.1 The nature of analysis in COSY

In this section we give some idea of the tools and techniques which have been developed for the analysis of COSY programs.

Given that a COSY program describes a system by specifying constraints on the execution of its operations, we may see that properties of interest are behaviour-al in nature. The most general form of the questions asked by a COSY programmer, about his program, "does it do what it's supposed to do?", thus becomes a question concerning the behaviour determined by the program, and the business of answering such questions thus involves an examination of the set of all such behaviours.

Our first task here, then, is to construct a formal model of behaviour. The construction of path programs from path-process programs given in 2.4 means that it is sufficient to do this for path expressions only. We shall show how to associate each path expression \(P\) with a set \(V_{FS}(P)\), the set of vector firing sequences of \(P\), whose elements model possible behaviours in which operations named in \(P\) execute according to the constraints enshrined in \(P\).

Verification questions may now be expressed formally in terms of the set \(V_{FS}(P)\).

Such verification questions fall into two classes which we might call questions on general properties and questions on specific properties.

General properties are those which apply to any COSY program, properties such as absence of deadlock or starvation which may be defined in terms of uninterpreted operations. We remark that the analysis of the spooler with interrupt which concludes this section is concerned mostly with the detection of general properties, especially absence of partial deadlock.

Specific properties, on the other hand, involve the interpretation of a given or projected COSY program as describing an actual real or projected system. For example, a version of the parallel release program \((P10)\) was used in a parallel resource allocation mechanism described and verified in [SL80]. Here it was important to show, not only absence of partial deadlock but also that the behaviour defined by the mechanism satisfied the requirements of the design problem, for example, that the configuration tests of \((P10)\) accurately detected the availability or non-availability of resources. The correct functioning of the interrupt mechanism in the spooler example of this section is a specific
property.

Our analytic techniques mainly involve the deduction of information about \( VFS(P) \) from the structure of \( P \). Apart from the special case of paths without non-deterministic choice, the \( GE_0 \)-paths discussed in theorem 1 below, these techniques depend on knowing that a modification of a path \( P \) giving a path \( S(P) \) is reflected by a transformation of the set \( VFS(P) \) into a set \( S(VFS(P)) \) equal to \( VFS(S(P)) \). Knowing \( VFS(P) \), we may thereby know \( VFS(S(P)) \). For example \( S \) may involve adding sequentialisation, as in theorem 2 or deleting certain operations, as in theorem 3 or replacing operations in a well-behaved manner by more complex subexpressions, as in theorem 4.

We illustrate these techniques in the spooler program. Beginning with a \( GE_0 \)-path whose behaviour may be derived using theorem 1, we construct the spooler by gradual modification. At each stage we apply an appropriate theorem to deduce the behaviour of the modified program. When the example is finally constructed, we have also constructed the set of its behaviours, from which the correctness of the program may be deduced.

3.2 Formal Behavioural Semantics of path programs

Let us consider a path \( P = P_1 \ldots P_n \) where each \( P_i \) is an individual path. (In the sequel, whenever we write "Let \( P = P_1 \ldots P_n \)" we shall assume each \( P_i \) to be an individual path). As has been explained intuitively in section 2, each individual path \( P_i \) describes a constraint on the order of executions of the operations it names, namely that in any behaviour of \( P \) as a whole, the operations belonging to \( P_i \) must have executed in sequence and any such sequence must be a firing sequence of \( P_i \). The only other "rule" is that when two individual paths mention an operation they are talking about the same operation.

These two observations lead us to consider that, firstly, a behaviour defines a set of firing sequences, one for each path, and that, secondly these sequences must agree as to the number and ordering of the particular operations. There are no more rules; we may therefore represent any behaviour by a set of firing sequences with the property that no two disagree on number and ordering. We shall, in fact, represent a behaviour by vectors, where coordinate \( i \) corresponds to individual path \( P_i \). We explain this vector construction in subsection 3.3.

First, however, we must formally define the concepts mentioned in section 2. We begin with individual paths.

3.2.1 Individual \( (\mathbb{R}_\mathcal{P}) \) Paths

An individual or \( \mathbb{R}_\mathcal{P} \)-path is a string derived from the non-terminal 'path' by the following production rules

\[
\text{path} \rightarrow \text{sequence} \quad \text{and} \\
\text{sequence} \rightarrow \{\text{orelement} \oplus\}^+ \\
\text{orelement} \rightarrow \{\text{element} \ominus, \odot\}^+ \\
\text{element} \rightarrow \text{operation/element}*/(\text{sequence})
\]

where non-underlined lower case words denote non-terminal symbols; the words 'path' and 'and', the comma, the semicolon, the star and the right and left parentheses are terminal symbols. The expression \( \{\text{nonterminal} \oplus\}^+ \) indicates expressions of the form 'nonterminal' or 'nonterminal...&nonterminal' and '/' indicates alternative substrings. Finally, the non-terminal 'operation' may be replaced by any suitable operation name, usually an ALGOL-like identifier.

With each \( \mathbb{R}_\mathcal{P} \)-path \( P \), we associate its set of operations, \( \text{Ops}(P) \), and its set of cycles, \( \text{Cyc}(P) \). In the definition of \( \text{Cyc}(P) \) that follows, \( \text{seq} \) (respectively \( \text{orel}, \text{elem}, \text{op} \) ), denotes any string derivable from a non-terminal 'sequence' (respectively; 'orelement', 'element' and 'operation').
Cyc(path seq end) = Cyc((seq)) = Cyc(seq)
Cyc(orel_1 \ldots orel_n) = Cyc(orel_1) \ldots Cyc(orel_n)
Cyc(elem_1 \ldots elem_n) = Cyc(elem_1) U \ldots U Cyc(elem_n)
Cyc(elem^*) = Cyc(elem)^*
Cyc(op) = \{op\}.

Here '^' denotes string concatenation, where if X,Y are sets of strings X.Y = \{xy | x \in X \land y \in Y\}.

To each H^*-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 \in X, some y\}.

We allow x or y, in the above, to be the null string \epsilon. Thus 
X^I Pref(X) and \epsilon \in Pref(X).

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

3.2.2 General (GR^w) Paths

A general, or GR^w-path is a string of the form P = P_1 \ldots P_n, where P_i is an 
H^*-path, for each i.

With each GR^w-path, P = P_1 \ldots P_n, we associate a set of operations,
Ops(P) = Ops(P_1)U \ldots 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 \ldots P_n is a GR^w-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, \ldots, x_n. Initially, all of them are null (x_i = \epsilon, each i). Whenever
some operation a executes, x_i is reset to x_i.a if a \in Ops(P_i); in other words, each
x_i contains a record of those 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, b \in Ops(P_i).

Let us consider S as having run for a while and then having halted. It will have
generated strings x_1 \in Ops(P_1)^*,. What can we say about these x_1? Well first, from
the desideratum that the order of executions of operations must obey the
constraints of all H^*-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 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 set of string vectors A_1^* \cdots A_n^* = \{(y_1, \ldots, y_n) | y_i \in A_i^*\} and
define a concatenation operation by

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

In particular our strings x_i of (2) may be made into a vector \vec{x} = (x_1, \ldots, x_n),
\vec{x} \in FS(P_1) \times \cdots \times FS(P_n) \subseteq Ops(P_1)^* \times \cdots \times Ops(P_n)^*
If we let \( a_p = (a_1, \ldots, a_n) \), where
\[
a_i = \begin{cases} a & \text{if } a \in \text{Ops}(P) \\ \varepsilon & \text{otherwise} \end{cases}
\]
then we see that (3) may be expressed
\[
(4) \quad X' = X \cdot a_p^*.
\]
Let us denote by VFS(P), the set of all vectors \((x_1, \ldots, x_n)\) that might be produced by our system \(S\). Let us denote by Vops(P), the "set of vectors \(a_p^*, a \in \text{Ops}(P)\). We denote by Vops(P)* the closure of Vops(P) in Ops(P)* \(x \ast x\) Ops(P)* with respect to vector concatenation. Note that Vops(P)* contains a null element \(\varepsilon = (\varepsilon, \ldots, \varepsilon)\). Now we have
\[
(5) \quad \varepsilon \in \text{Vops}(P) \ast \cap (\text{FS}(P_1) \ast \ldots \ast \text{FS}(P_n) \ast \cap \text{VFS}(P))
\]
and a \(\in \text{Ops}(P)\), then \(X \cdot a_p \in \text{VFS}(P)\) if and only if \(X \cdot a_p \in (\text{FS}(P_1) \ast \ldots \ast \text{FS}(P_n) \ast \setminus \text{VFS}(P))\).
Thus VFS(P) = Vops(P)* \cap (FS(P_1) \ast \ldots \ast FS(P_n) \ast).

We take this as the definition of VFS(P).

Let us pause briefly and look at this object VFS(P). By our construction, we see that every element, \(X\), 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
\[
X = a_1 \ast a_2 \ast \ldots \ast a_n, \quad a_i \in \text{Vops}(P).
\]
In fact \(X\) 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 for each \(i\), \([a]_i \neq \varepsilon\) implies \([a_2]_i = \varepsilon\).

In this case \(a_1 \ast a_2 = a_2 \ast \varepsilon\) and \(X = a_2 \ast a_1 \ast \varepsilon \ast \ldots \ast a_n\).

A glance at the definition of the vector operations \(a_p\), shows that if \(a \neq b\) then \(a_p \cdot b_p = b_p \cdot a_p\) if and only if, for every, \(a \in \text{Ops}(P_1)\) implies \(b \neq \text{Ops}(P_1)\).

We conclude that the following interpretation may be made. Let \(X \in \text{VFS}(P)\) and let \(a_b \in \text{Ops}(P)\) with \(a \neq b\). Then \(X \ast a_p \in \text{VFS}(P)\) and \(a_p \cdot a_p = b_p \cdot a_p\) may be interpreted as follows: that in the system state determined by \(X\), the operations \(a_p\) and \(b_p\) may execute concurrently.

Let us therefore fix a GF* - path \(P = P_1 \ast \ldots \ast P_n\). 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 \(X, Y \in \text{Vops}(P)*\), we define
\[
X \cdot Y = \{x \cdot y \mid x \in X \land y \in Y\}.
\]
and
\[
x^0 = \{\varepsilon\}, x^n = x \ast x \ast \ldots \ast x = X^0 \cup X^1 \cup X^2 \cup \ldots.
\]

We may also define a relation \(\leq\) (vector prefix) on Vops(P)* by
\[
x \leq y \text{ if and only if there exists } z \in \text{Vops}(P)* : x \ast z = y.
\]
We also define \(x < y\) to mean \(x \leq y\) and \(x \neq y\). \(x \leq y\) may be understood to mean that \(x\) is an initial part of \(y\).

Using this formalism, we may define the important notion of adequacy. A path expression \(P\) is adequate if and only if
for all \( x \in VPS(P) \), for all \( a \in Ops(P) \), there exists \( \gamma \in Vops(P)^* \) such that \( x \cdot \gamma \cdot a \in VPS(P) \).

Adequacy is a kind of absence of partial system deadlock.

### 3.3 An example system: A simple spooling system with interrupts

As promised in 3.1, we illustrate the formal side of COSY by constructing and analyzing a program describing a system consisting of a cardreader and lineprinter communicating across a ring buffer. We also provide the system with an interrupt handler. What this does is to register an error message from somewhere, brings the whole system to a halt and then restarts it again. We might picture the situation as one in which both the registering of an error and the restarting of the system result from an intervention from the system’s environment, say a human operator.

We begin with the ring buffer itself. We shall suppose that it has \( n \) frames and that we are not concerned with the actual content of the buffer at any time.

Given this, the only operations associated with it are deposits and removes into its various frames. We have thus \( 2n \) operations, \( \text{DEPOSIT}(1), \ldots, \text{DEPOSIT}(n), \text{REMOVE}(1), \ldots, \text{REMOVE}(n) \), which we may declare in a collector:

\[
(C) \quad \text{array DEPOSIT}, \text{REMOVE}(1\text{\text{-}n})
\]

Assuming that all the frames are initially empty, the desired pattern of behaviour of an individual frame must be a sequence of alternating deposits and removes beginning with a deposit. This constraint is expressed succinctly by the following path:

\[
(P19) \quad \text{path} \quad \text{DEPOSIT}(1); \text{REMOVE}(i) \quad \text{end}
\]
for each frame \( i \). To impose a ring discipline on deposits and removes, we simply add paths:

\[
(P20) \quad \text{path} \quad \text{DEPOSIT}(1); \text{DEPOSIT}(2); \ldots; \text{DEPOSIT}(n) \quad \text{end}
\]
and similarly for removes or, using the distributor described in the last section:

\[
(P21) \quad \text{path} \quad ;(\text{DEPOSIT}) \quad \text{end}
\]

The ring buffer program is thus:

\[
(P22) \quad \text{array DEPOSIT}, \text{REMOVE}(1\text{\text{-}n})
\]

\[
\text{[path } \text{DEPOSIT}(i); \text{REMOVE}(i) \quad \text{end} \quad \text{[1, } n, 1 \text{]}
\]

\[
\text{path } ;(\text{DEPOSIT}) \quad \text{end} \quad \text{path } ;(\text{REMOVE}) \quad \text{end}
\]

We might ask what kind of behaviour this program specifies. Observe that the only connective used in it is the semicolon. For such programs, the set of all possible behaviours has, in fact, been classified, as we see from the following theorem.

**Theorem 1: Behavioural characterisation of connected GE\( _2 \)-paths**

Let \( P = P_1 \ldots P_n \), where each \( P_i \) is of the form \( \text{path } a_1^i ; \ldots ; a_m^i \quad \text{end} \) and no operation occurs more than once in any given path (paths such as \( P \) are called GE\( _2 \)-paths). Suppose also that the paths are connected that is we cannot partition the \( P_i \) into two sets such that no path in the first set has an operation in common with any path in the second set. Let \( x \) be the vector:

\[
(a_1^1, \ldots, a_m^1, \ldots, a_1^n, \ldots, a_m^n)
\]

then we have two possibilities:

1. \( x \in Vops(P)^* \). In this case \( P \) is adequate and \( VFS(P) = \text{Pref}(x^*) \).
2. \( x \notin Vops(P)^* \). In this case \( P \) is not adequate and there exists a unique \( \gamma \leq x \) such that \( \gamma \in Vops(P)^* \) and \( VFS(P) = \text{Pref}^*(\gamma) \).

We remark that this theorem is a special case of a more general result which deals with paths in which the restriction on multiple occurrences of operations and the
restriction to connected paths have both been dropped. [579, theorem 3.3].

The reader may readily check that the expansion for any given \( n \) of P22, which we shall call \( \text{RB}(n) \) is a connected \( \mathcal{G}_0 \)-path. The \( x \) in question is a vector

\[
\mathbf{rb}(n) = \begin{pmatrix}
\text{DEPOSIT}(1), \text{REMOVE}(1) \\
\vdots \\
\text{DEPOSIT}(n), \text{REMOVE}(n) \\
\text{DEPOSIT}(1), \text{DEPOSIT}(2), \ldots, \text{DEPOSIT}(n) \\
\text{REMOVE}(1), \text{REMOVE}(2), \ldots, \text{REMOVE}(n)
\end{pmatrix}
\]

The reader may also verify that this vector may be expressed as a product of vector operations

\[
\mathbf{rb}(n) = \text{DEPOSIT}(1), \text{DEPOSIT}(2), \ldots, \text{DEPOSIT}(n), \text{REMOVE}(1), \text{REMOVE}(2), \ldots, \text{REMOVE}(n).
\]

It follows that \( \mathbf{rb}(n) \) does belong to \( \text{Ops}(\text{RB}(n))^\ast \) for each \( n \). Theorem 1 allows us to conclude that \( \text{RB}(n) \) is adequate, for any \( n \), and that \( \text{VPS}(\text{RB}(n)) = \text{Pref}(\mathbf{x}(n)^\ast) \).

Anything we want to know about the behaviours determined by \( \text{RB}(n) \) may be deduced from this characterization. For example, we might ask about possible concurrency in a typical behaviour. An examination of the set \( \text{Ops}(\text{RB}(n)) \) shows that two distinct vector operations commute if and only if one of them is a \( \text{DEPOSIT}(i) \) and the other is a \( \text{REMOVE}(i) \), with \( i \neq j \). Applying this observation to the behaviour \( \mathbf{rb}(n) \) itself, we can see, for example, that after the execution of \( \text{DEPOSIT}(1), \text{DEPOSIT}(2) \) and \( \text{REMOVE}(1) \) execute concurrently.

Let us now go a step further and add some processes to P22. They will be very simple; one of them will do nothing but repeatedly perform arbitrary deposits while the other will do nothing but repeatedly perform arbitrary removes. Since these two processes are disjoint, they may be represented by paths. Thus, the depositing process will be

\[
(P23) \quad \text{path DEPOSIT}(1), \text{DEPOSIT}(2), \ldots, \text{DEPOSIT}(n) \text{ end}
\]

or using distributors

\[
(P24) \quad \text{path ,(DEPOSIT) end}.
\]

(P22) accordingly becomes

\[
\begin{array}{l}
\text{array DEPOSIT, REMOVE}(1 \text{in}); \\
[\text{path DEPOSIT}(i) ; \text{REMOVE}(i) \text{ end } ]_{i=1,n,1} \\
\text{path ,(DEPOSIT) end} \\
\text{path ,(REMOVE) end} \end{array}
\]

\[
\begin{array}{l}
\text{path ,(DEPOSIT) end} \\
\text{path ,(REMOVE) end}
\end{array}
\]

We shall call this new program \( \text{PRB}(n) \). Note that it is just \( \text{RB}(n) \text{DR} \), where \( D \) and \( R \) are the deposit and remove processes respectively. The behaviours of this new program may be deduced from those of the old one using the next theorem.

Let us take any path \( P \) and two sets (of operations) \( A \) and \( B \); we suppose that \( A \subseteq \text{Ops}(P) \) and \( B \subseteq \text{Ops}(P) \) = \( \emptyset \). We shall suppose \( A = \{a_1, \ldots, a_k\} \) and \( B = \{b_1, \ldots, b_m\} \) (and either set could be empty).

Let \( P_{A,B} = \text{path } a_1, \ldots, a_k, b_1, \ldots, b_m \text{ end} \). We are going to compare behaviours of \( P \) with those of \( P_{A,B} \) - the path you get by adding \( P_{A,B} \) to \( P \). Let us denote \( \text{PP}_{A,B} \) by \( P' \) for simplicity. Let us take \( x \in \text{Ops}(P') \); we construct a set of vectors \( S_{A,B}(x) \subseteq \text{Ops}(P') \) as follows - essentially we are just "sloting" elements of \( B \) into interstices in \( x \); suppose \( x = x_1 \ldots x_t \), with \( x \in \text{Ops}(P) \). Define \( S_{A,B}(x) \text{ to }
be the set $\left\{ x_0, x_1, \ldots, x_{r-1}, x_r, x_{r+1}, \ldots, x_p, x_p' \mid x \in \left\{ x_1, \ldots, x_p \right\} \right\}$. 

Theorem 2: Side condition excision/addition

With the above terminology

1. $\text{VFS}(P') = S_{A,B}(\text{VFS}(P))$

2. $P$ is adequate if and only if $P'$ is adequate

Let's apply this theorem to PRB(n) -- we have to do it twice, in fact. Since, in the terminology of the theorem, $B = \emptyset$, our set $S_{A,B}(x)$ for a given $x$ is just a singleton $\{x'\}$, where all we have done is to add a couple of dimensions to $x$. In fact, in this case nothing has changed; the addition of these two paths has not introduced any additional sequentialization into RB(n). Certainly adequacy is preserved. Indeed, in some formal sense, VFS(P) and VFS(P') are isomorphic. We have gained nothing new by this transformation in terms of behaviours.

We do, however, have two new paths, and we may now begin to transform them. Let us make our deposit process into a cardreader process by forcing deposits to be subsequent on some operation "readcard" and, similarly, make our remove process into a lineprinter process by forcing remove to be interleaved with some operation "printline". The new form of the deposit process is accordingly

(P26) path readcard; DEPOSIT(1), ..., DEPOSIT(n) end

or using the distributor

(P27) path readcard; (DEPOSIT) end

and similarly for the remove process. P25 therefore becomes

array DEPOSIT, REMOVE(1:n);

[[path DEPOSIT(i); REMOVE(i) end[i][1, n, 1]]

path ; (DEPOSIT) end

(P28) path ; (REMOVE) end

path readcard; , (DEPOSIT) end

path , (REMOVE); printline end

Now, what have we done here? We have added two operations each of which appears only in one path. Furthermore they have been placed there in a rather "regular" way; the firing sequences of the cardreader process, for example, are exactly those of the deposit process except that now there is a "readcard" between every two successive deposits. In fact, it should not be very hard to see that the only difference between elements of VFS(PR}(n)) and the vector firing sequences of the new program -- which we shall call SP(n) -- is that the latter have interleaved "readcard"s and "printline"s in, respectively, their penultimate and ultimate coordinates.

Of course, we can generalise this situation, and our next theorem will do this, but first let us try to see what is special about the way in which "readcard" and "printline" appear in their paths. The most important thing about it is that they do not influence any non-deterministic choice; "hiding" the operation "readcard", say, doesn't hide any significant activity in the program. If, on the other hand, "readcard" appeared in a subexpression of the form "..., (readcard; ...)", then "hiding" it would hide some significant activity, namely the nondeterministic choice which would commit the system to progress through the section of program following "readcard". This sort of thing can cause all sorts of problems, so we shall exclude it via the following definition; an individual path $P$ will be said to be proper for some operation "a" if and only if "a" only appears in $P$ in substrings of any of the following form

path a; or ;a; or ;a) or ;;a end

20
The reader will see immediately that the cardreader process is proper for "readcard" - which appears in a substring of the form "path readcard:" and that the lineprinter process is proper for "printline".

We assume for simplicity that Ops(P) ≠ {a}.

If P is proper for a, then define P/a to be the path obtained from P by deleting all "a"s and retaining superfluous semicolons. Thus path readcard, (DEPOSIT) end/readcard = path (DEPOSIT) end. If x ∈ FS(P), we define x/a to be the string obtained from x by deleting every "a" from x. It is easy to see that in this case we will have x/a ∈ FS(P/a).

It may be shown that if P contains no repeated operation except possibly "a" itself, then for every cycle c ∈ Cyc(P/a) there exist a unique cycle c' ∈ Cyc(P) such that c'|a = c.

Thus if P is P26 then every cycle c = DEPOSIT(1) gives rise to a unique cycle c' = readcard,DEPOSIT(1). Let us define S_c(c) = c'. Further, if x ∈ Pref(Cyc(P/a)) then there is a unique "longest" y ∈ Pref(Cyc(P)) such that y|a = x. Again, we are getting y from x by "sloting in" as many "a"s as possible without going over the edge of a cycle. We call this unique y, S_a(x).

If, finally, x ∈ FS(P/a), that is x = c_1* ... *c_m* v with c_i ∈ Cyc(P_i) and v ∈ Pref(Cyc(P_1)), then we may define

S_a(x) = S_a(c_1). ... . S_a(c_m) . S_a(v).

(Note that S_a(x) ∈ Cyc(P)*Pref(Cyc(P)) and that Cyc(P)*.Pref(Cyc(P)) = FS(P).)

Now let us tackle general paths. P = P_1... P_n will be said to be proper for "a" if "a" appears in only one path, P_i say, which is itself proper for it and contains no repeated operations apart from "a", and we define P/a = P_1... P_{i-1} P_i/a P_{i+1}... P_n. Again, we are interested in constructing, from elements x ∈ VFS(P/a), elements of VFS(P) by "sloting in" "a"s. Define for

x ∈ VFS(P/a)

S_a(x) = ([x]_1, ... , [x]_{i-1}, S_a([x]_i), [x]_{i+1}, ... , [x]_n).

Theorem 3: Proper excision
Suppose P is proper for a, then

(1) VFS(P) = Pref(S_a(VFS(P/a)))
(2) if VFS(P/a) = Pref([x]), some x ∈ VFS(P/a) then VFS(P) = Pref(S_a([x])*)
(3) P is adequate if and only if P/a is adequate.

Let us apply this to SP(n) P28 which is proper for both "readcard" and "printline". Note that (SP(n)/readcard)/printline = PRB(n), which, from theorem 2, we know to be adequate. (3) of Theorem 3 tells us that therefore SP(n) is adequate. We also know that

VFS(PRBN) = Pref(PRN)* where

PRBN = DEPOSIT(1), REMOVE(1)
   :  :  :  :
   DEPOSIT(n), REMOVE(n)
   DEPOSIT(1), ... , DEPOSIT(n)
   REMOVE(1), ... , REMOVE(n)
   DEPOSIT(1), ... , DEPOSIT(n)
   REMOVE(1), ... , REMOVE(n)

from which, by 2 of Theorem 3, we obtain VFS(SP(n)) = Pref(S_a(PRN))* and

21
\[ S_e(p_E(n)) = \begin{cases} 
\text{DEPOSIT}(1), \text{REMOVE}(1) \\
\quad \ast \quad \ast \\
\text{DEPOSIT}(n), \text{REMOVE}(n) \\
\text{DEPOSIT}(1), \ast \quad \ast \quad \text{DEPOSIT}(n) \\
\text{REMOVE}(1), \ast \quad \ast \quad \text{REMOVE}(n) \\
\text{readcard}, \text{DEPOSIT}(1), \ast \quad \ast \quad \text{DEPOSIT}(n) \\
\text{REMOVE}(1), \text{printline}, \ast \quad \ast \quad \text{REMOVE}(n), \text{printline} 
\end{cases} \]

or writing it as a product of vector operations

\[ S_e(p_E(n)) = \text{readcard}, \text{DEPOSIT}(1), \ast \quad \ast \quad \text{DEPOSIT}(n), \text{printline} \]

We shall call this vector \( sp(n) \).

We now add an interrupt mechanism to \( SP(n) \). This consists of a process which operates as follows: at any time it may receive a message from some unspecified source, after which it halts spooling, performs some unspecified action, then allows spooling to be resumed and returns to its initial state. Again this process contains no operation in common with any other process in the system and may thus be regarded as a path. It will have the form

\[ \text{path getmessage; stopystem; startsystem end} \]

Now, as soon as "getmessage" executes, no other operation should be able to execute until "startsystem" executes. This means that, for every operation, op, in \( SP(n) \), executions of op and of the uninterrupted sequence "getmessage; stopystem; stopystem; startsystem" should be strictly interleaved. We may obtain this effect by adjoining paths

\[ \text{path op, (getmessage; stopystem; startsystem) end} \]

for each operation "op" in \( SP(n) \).

The new program becomes:

\[ \begin{array}{l}
\text{array DEPOSIT, REMOVE(1n)} \\
\text{[path DEPOSIT(i); REMOVE(i) end [1, n, 1]} \\
\text{path ;(DEPOSIT) end path ;(REMOVE) end} \\
\text{path readcard, ;(DEPOSIT) end} \\
\text{path ;(REMOVE); printline end} \\
\text{path DEPOSIT(i), (getmessage; stopystem; startsystem) end} \\
\text{path REMOVE(i), (getmessage; stopystem; startsystem) end [1, n, 1]} \\
\text{path readcard, (getmessage; stopystem; startsystem) end} \\
\text{path printline, (getmessage; stopystem; startsystem) end} \\
\text{path getmessage; stopystem; startsystem end}
\end{array} \]

(P29)

We can analyze P29, which we shall call ISP(n), by breaking its construction into two parts. Since the sequence "getmessage; stopystem; startsystem" is intended to be atomic, we may provisionally regard it as a single operation "int" and add paths

\[ \text{path op, int end} \]

for each operation op in \( SP(n) \). We are providing \( SP(n) \) with a means of interrupting any operation by int. The intermediate path, which we shall call ISPI(n) is

\[ \begin{array}{l}
\text{spooler path SP(n)} \\
\text{[path DEPOSIT(i), int end [1, n, 1]} \\
\text{[path REMOVE(i), int end [1, n, 1]} \\
\text{path readcard, int end} \\
\text{path printline, int end}
\end{array} \]

(P30)

We can deduce the behavior of ISPI(n) from that of \( SP(n) \) using Theorem 2. Initially one is adding a path
path DEPOSIT(1), int and

where "DEPOSIT(1)" belongs to the program and "int" does not. From then on,
both operations in the adjoined path belong to the modified program. The reader
will easily be able to calculate the new set of behaviours. It will be
Pref(ispi(n)*) where

ispi(n) = X(1).X(2) ... .X(n).Y(1).Y(2) ... .Y(n)

where

\[ X(i) = \{ \text{int}_{\text{readcard}}, \text{DEPOSIT}(i) | m \in Z^+ \} \]
\[ Y(i) = \{ \text{int}_{\text{REMOVE}(i)}, \text{printing} | m \in Z^+ \} \]

Note that ispi(n) is a set rather than a single vector as in all previous cases
since a certain amount of non-determinacy, reflecting the unpredictability of the
environment that is introduced into the program. Notice also that no concurrency
has been suppressed by this addition.

We now add on the interrupt process in a shortened form — a very shortened form —
by adjoining path int end to ISPI(n). The behaviour of this new path is exactly
the same as that of ISPI(n) — except that the vectors have an extra dimension
containing nothing but strings of "int"s. Our next step is to replace "int"
everywhere by "(getmessage;stostop;stostart;stostarty)" giving a program ISPI(n). The
effect of this on behaviour is indicated by the next theorem.

What we are doing here is to replace an operation everywhere in the program by a
subexpression. In the terminology of the following, we are replacing an
operation v-int by a subexpression subexpr = (getmessage;stostop;stostarty).
This induces a corresponding transformation of VFS(ISPI(n)) into a set

\[ \text{Pref}(S_v, \text{subexpr}(\text{ISPI}(n))) = \text{VFS}(\text{ISP}(n)) \]

with this, we finally arrive both at
our final program and at its behaviour.

Let P = P_1 ... P_n and suppose v \in Ops(P). Let subexpr be any subexpression which
involves operations, none of which belong to P. Let S_{subexpr} be the set

\[ \{ w | w \in \text{Ops}(\text{subexpr}) \} \]

where \( w = (w_1, ..., w_n) \) and

\[ w_i = w \text{ if } v \in \text{Ops}(P_i) \]
\[ = w \text{ otherwise} \]

Define S_{v, subexpr}(P) to be the path obtained from P by replacing "v" everywhere
in P by "(subexpr)"

If \( x \in \text{VFS}(P) \), define S_{v, subexpr}(x) to be the set of vectors obtained from \( x \) by
replacing every \( v \) in \( x \) by some element of S_{v, subexpr}.

**Theorem 4: Constrained Substitutions**

With the above terminology:

1. \( \text{VFS}(S_v, \text{subexpr}(P)) = \text{Pref}(S_v, \text{subexpr}(\text{VFS}(P))) \)

2. P is adequate if and only if \( S_v, \text{subexpr}(P) \) is adequate.

We remark that this theorem is a very very special case of a much more general
theorem which may be found in [ST9, Theorem 4.6]

Applying theorem 4 to our example — "v" being "int" and "subexpr" being
"getmessage;stostop;stostarty" — we can derive a form for \( \text{VFS}(\text{ISP}(n)) \).

\[ \text{VFS}(\text{ISP}(n)) = \text{Pref}(x(1), ..., x(n), \hat{y}(1), ..., \hat{y}(n)) \]

23
where

\[ \hat{X}(i) = \{(\text{getmassage}, \text{stopystem}, \text{startystem})^{mi}, \text{readcard}, \text{DEPOSIT}(i) | \text{mi} \in \mathbb{Z}^+ \} \]

\[ \hat{Y}(i) = \{(\text{getmassage}, \text{stopystem}, \text{startystem})^{ni}, \text{REMOV}(i), \text{printline} | \text{ni} \in \mathbb{Z}^+ \} \]

We also know that adequacy has been conserved at each stage in the construction of ISP(n). The form of VFS(ISP(n)) shows that there is a correct alternation of deposits and removes. In fact the correctness of the buffer and the card-reader and lineprinter process may be deduced directly from the paths themselves. The important thing is that the paths when put together do not produce any inconsistency, that is, give rise to deadlocks.

We also claim that the interrupt handler performs its functions correctly. We cannot do this formally without reference to the notion of maximal behaviours and infinite vector firing sequences, which are beyond the scope of this paper.

Speaking very roughly, however, we may say the following:

1. The interrupt mechanism works if it is not possible to have a behaviour in which a "getmessage" is never followed by a "stopystem".

2. This occurs if either the system deadlocks at some point (and we know it doesn't) or if there is an ascending chain of behaviours \( X_1 < X_2 < X_3 < \ldots \) such that \[ [X_1]_{j+3n} = [X_2]_{j+3n} = \ldots \]

   where \( [X_j]_{j+3n} \) is the 2 coordinate relating to the interrupt process, with "getmessage" as the last operation in \([X_j]_{j+3n}\), that is to say, it is possible to take a behaviour \( X_1 \), in which the interrupt process is halted after executing "getmessage", and extend it indefinitely without executing any other operation in the interrupt process.

We can show, from the form of VFS(ISP(n)), that such a chain is impossible, from which it follows that the interrupt mechanism works as required.

4. Introduction to programming in "system" COSY

In this section we will assume that the reader has overcome his initial difficulty in writing and reading basic COSY programs, that is, programs involving the constructs of section 2, paths, processes, replicators, collectivisors and distributors. Similarly we will assume that after section 2 and section 3 the reader has some idea how to use the vector firing sequence formalism for analysing what a basic COSY program "means" and whether what it means satisfies a specification which might have lead to the construction of the basic COSY program. Furthermore we will assume that we are writing for experienced programmers familiar with at least one general programming construct for abstraction and instantiation like the SIMULA class [cf. DHT72].

Therefore we will not go into such stepwise construction and detailed analysis of programs and their meanings as in the two previous sections but rely on the completed programs themselves to make our points for us. So this section will mostly consist of illustrative examples to allow us to make a number of fairly concise points about programming in system COSY which we briefly sketch below.

4.1 System COSY

If we use "program" and "endprogram" as program parentheses a basic COSY program essentially had the form:

```
program
    ...  
endprogram
```

here one can write any basic COSY, that is, paths and processes involving, replicators, collectivisors, distributors

endprogram
One way to introduce structure into programs was to use macro generators which would replicate a number of paths and processes from given path and process patterns involving references to collections of operations. But these generators did not allow one to express explicit hierarchic structures. Since we use a very special kind of class notion and we are mainly concerned with classes defining systems we use the term "system" rather than "class" in our programs [cf. LSB79]. Hence in system COSY one can now write programs of the general form:

```
program
  system systemname (formal parameters)
  instantiations of other systems
  operation
    definitions of operations associated
    with the system called "systemname"
  endoperation
  any basic COSY
endsystem
  further system definitions
  any basic COSY
endprogram
```

Processes may enter instances of a system only by means of the operations associated with that system in its definition. Hence, the operation definitions define the boundary between what is externally and what is internally "visible" of the system in question.

Note that this feature of the system COSY notation allows us to define, for example, what the synchronization properties of a particular monitor construct are, make this definition the body of a system definition and from then on program as if we had this type of monitor as a high level primitive. But if the particular monitor concept had unfortunate consequences in practice we would not be irrevocably committed to this notion of monitor as we would be if we really had to program in a language having such a monitor as a primitive feature. Our design vehicle allows us to easily redefine our notion of monitor suitably by changing the synchronization properties which are specified for it in the corresponding system definition. This is one reason why we have also called our design vehicle a tool for software specification [cf. LSB79].

Finally, in system COSY it is possible to systematically relate COSY style system specifications with implementations in any programming notation in a manner similar to Alphard [WLS76]. Hence, one can write system definitions of the form:

```
system systemname
  specification (formal parameters of specification part)
  here one can write system specifications in
  the abstract system COSY design notation
endspecification
implementation (formal parameters for implementation part)
  here one can write an implementation of the system
  using specific synchronization primitives like
  semaphores, fork and join, monitors, etc.
endimplementation
endsystem
```

25
Similarly the program body part of a program involving such "implemented" systems will have a specification and an implementation part.

This feature of the notation also allows different implementations of the same abstract strategy to be systematically compared within the system COSY notation. In the rest of this section we illustrate the use of all these features by a number of examples which will be briefly and informally discussed. Of course we will allow intuitive specifications of systems to be written as comments in COSY programs.

4.2 Relating specification to implementation in system COSY

This intuitive specification of the desired synchronization properties of an initially empty buffer frame in terms of its operations can be stated as:

\begin{verbatim}
system frame
  operation deposit remove endoperation
  comment: No two operations may execute concurrently and every
            removal of a message from the frame must strictly alternate
            with a previous deposit of a message in the frame
endsystem
\end{verbatim}

A possible implementation of such a frame using a notion of abstract type, with associated data structures and procedures and binary semaphores as the only synchronization mechanism could be:

\begin{verbatim}
type frame
  message f  semaphore ss0=0 ss1=1
procedure
  (T1) deposit (accepts message m) = (P(ss1);f:=m;V(ss0))
  remove (returns message m) = (P(ss0);m:=f;V(ss1))
endprocedure
endtype
\end{verbatim}

The corresponding way of writing this implementation in system COSY is:

\begin{verbatim}
system frame
  implementation
    message f  semaphore ss0=0 ss1=1
procedure
  (T2) deposit (accepts message m) = (P(ss1);f:=m;V(ss0))
  remove (returns message m) = (P(ss0);m:=f;V(ss1))
endprocedure
endimplementation
endsystem
\end{verbatim}

where all that has happened is that the keyword "type" has been replaced by the keyword "implementation" in the abstract type parentheses, and the type name is attached to the enclosing opening system parentheses.

The abstract specification of the behaviour specified in (S1) using system COSY could be:
system frame
  specification
    (S3) operation deposit remove endoperation
    path deposit; remove end
  endspecification
endsystem

(S3) and (S2) could be combined in an obvious way to yield a single system
definition involving both specification and implementation parts.

Before going on with further examples we would like to sketch how one could show
that the implementation in terms of semaphores (S2) can be considered to impose
the same sequential behaviour on deposits into and removals from the frame as ex-pressed in the more abstract specification (S3). It can be shown that if we replace
"r:=m" and "m:=r" in (S2) by the names "deposit" and "remove" respectively, and
call the resulting system definition (S2) then

FS(S3) = \{deposit, remove \}
FS(S2) = \{P(s1), deposit, V(ss0), P(ss0), remove, V(ss1) \}
                  \{ε, P(ss1), P(ss1), deposit, ..., remove \}

and the following theorem holds:
Theorem projB3(FS(S2)) = FS(S3)

where projB is a function which deletes from any operation sequence those
operations not occurring in the system definition S.

Before we leave this section on implementation we want to point out that in our
opinion if one uses an ordinary programming language together with some added
synchronization mechanism, nothing prevents one from expressing certain synchro-nization properties of the system under consideration operationally by means of
ordinary programming constructs rather than by the synchronization mechanism
explicitly designed for this. Hence one tends to make explicit only certain parts of the synchronization structure of the system and to leave other such
structure implicit in a very algorithmic and operational form. To see what we
mean consider the following type definition for a ring buffer. We assume the
type frame defined as in (T2):

type ring-buffer (n1:integer)
  array (frame) B(0:n1-1)
  semaphore s2 = s3 = 1 integer front = back = 0
procedure
  deposit(accept message m) = (P(s2); front := (front+1) mod n1;
               B(front), deposit(m); V(ss2))
  remove RETURNS message m = (P(s3); back := (back+1) mod n1;
               B(back), remove(m); V(s3))
endprocedure
endtype

The synchronization specification of a ring buffer requires that both deposits
to and removals from the collection of buffer frames must be strictly and
cyclically ordered respectively. Though the use of assignment statements and
the indexed names in the procedure bodies enforces this discipline the latter is
not expressed by means of semaphores, the synchronization primitive proper.
Using only this primitive and the parallel begin–end construct one could write:

```
type ring-buffer (n1:integer)
array (frame) B(1:n1)
semaphore s11 = s21 = 1
s12 = ... = s1n1 = s22 = ... = s2n1 = 0

procedure
  deposit(accept message m) =
  parbegin P(s11);B(1).deposit;V(s12)||
  P(s12);B(2).deposit;V(s13)||...||
  P(s1n1);B(n1).deposit;V(s1n1)parend

remove:returning message m) =
parbegin P(s21);B(1).remove;V(s22)||
P(s22);B(2).remove;V(s23)||
P(s2n1);B(n1).remove;V(s2n1)parend
```

endtype

Of course semaphores are not very convenient to express this kind of cyclic ordering. We leave it to the reader to convince himself that (S4) below specifies the same behaviour as (T3) that is

```
proj_S4(FS(T3)) = FS(S4).
```

```
system ring-buffer (n1:integer)
array (frame) B (1:n1)
operation deposit = (,B.deposit))
(S4) remove = (,B.remove))
endoperation
path:(B.deposit)end path: (B.remove)end
```
endsystem

Finally we note that grafting path notation onto some ordinary programming language as done by Campbell and Habermann in their original paper introducing path expressions causes them to express the ring synchronization structure not by means of a path expression [cf. CH74 p.14]. In his later paper on path expressions Habermann [H75] still does the same in the case of certain synchronization structures. Our development of COSY from our path–processes notation [LC75] explicitly prevents such uncontrolled mixing of styles of description of synchronization structure.

4.3 Hierarchic and distributed systems in COSY

In this last subsection we will first develop a hierarchic system of error handlers for a ring buffer and then a spooling system with ten ring buffers each with its own independent (distributed) error handling capability.

4.3.1 Instantiation of system definitions

To lead the reader from an understood program to an equivalent but different program we reformulate (P13.1) from section 3 in terms of system definitions and illustrate what it would stepwise expand to upon instantiation.
program
system frame (s1:sequence)
  path deposit; remove end
  path deposit, (s1) end
  path remove, (s1) end
  operation deposit remove endoperation
endsystem
system ring-buffer (s2:sequence | n1:integer)
  array (frame(s2)) B (1:n1)
  path; B.deposit end path; B.remove end
  operation deposit = (, (B.deposit))
  remove = (, (B.remove)).
(S5) endoperation
endsystem
system spooler (s3:sequence | n2:integer)
  ring-buffer (s3) n2) r
  process readcard; r.deposit end
  process r.remove; printline end
  path readcard, (s3) end path printline, (s3) end
endsystem
system error-handler
spooler (error; start | 50) s
  process error; stop; fix; start end
endsystem
error-handler eh
endprogram

Expansion of the instantiation of the error-handler "eh" in the program body would lead to the replacement of the instantiation and the error-handler system definition by:

(S6) process eh.error; eh.stop; eh.fix; eh.start end
  spooler (eh.error; eh.start | 50) eh.s

Expansion of the instantiation of the spooler "eh.s" would lead to the deletion of the spooler system definition and the replacement of the instantiation by:

(S7a) process eh.s.readcard; eh.s.r.deposit end
(S7b) process eh.s.r.remove; eh.s.printline end
(S7c) path eh.s.readcard, (eh.error; eh.start) end
(S7b) path eh.s.printline, (eh.error; eh.start) end
(S7c) ring-buffer (eh.error; eh.start | 50) eh.s.r

Expansion of the ring buffer "r" would replace (S7c) by:
array (frame(ch.error;ch.start)) eh.s.r.B(1:50)

(S6) path ;(eh.s.r.B.deposit) end
    path ;(eh.s.r.B.remove) end

and (S7a) by:

process eh.s.readcard ;(eh.s.r.B.deposit) end
process (eh.s.r.B.remove); eh.s.println end

Finally expansion of the array instantiation and remaining distributors in (S8) and (S9) results in:

process eh.error; eh.stop; eh.fix; eh.start end
process eh.s.readcard; eh.s.r.B(1).deposit, ..., eh.s.r.B(50).deposit end
process eh.s.r.B(1).remove, ..., eh.s.r.B(50).remove; eh.s.println end
path eh.s.readcard, (eh.error; eh.start) end
path eh.s.println, (eh.error; eh.start) end
path eh.s.r.B(1).deposit, ..., eh.s.r.B(50).deposit end

(S10) path eh.s.r.B(1).remove; ..., eh.s.r.B(50).remove end
    path eh.s.r.B(1).deposit; eh.s.r.B(1).remove end
    ***
    path eh.s.r.B(50).deposit; eh.s.r.B(50).remove end
    path eh.s.r.B(1).deposit, (eh.error; eh.start) end
    ***
    path eh.s.r.B(50).remove, (eh.error; eh.start) end

which is the whole of (S5) expanded. Of course one mostly designs systems without thinking about expanded programs.

4.3.2 Ring buffer with hierarchical error handlers

We assume the system definitions of frame and ring-buffer from (S5). Next we define two error handlers, an ultimate error handler and an intermediate error handler as follows:

system ultimate-error-handler

operation error stop fix start endoperation

(S11) process error; stop; fix; start end
endsystem

system intermediate-error-handler (s3:sequence)

operation error stop fix start endoperation

process error; stop; fix; start end
path error, (s3) end
path stop, (s3) end
path fix, (s3) end
path start, (s3) end
endsystem

Finally we define a system involving a ring-buffer with an associated hierarchy of two error handlers as follows:
system ring-with-hierarchic-error-handlers (n2:integer)  
ultimate-error-handler uch  
intermediate-error-handler (ueh.error;uch.start) ieh  
ring-buffer (ieh.error;ieh.start)n2 r  
(S13) operation deposit = r.deposit  
remove = r.remove  
endoperation  
endsystem

Now one can write spooling processes in a program including definitions (S11) - (S13) and the two definitions from (S5) thus:

program  
definitions (S11) - (S13) and those of ring-buffer and frame  
ring-with-hierarchic-error-handlers(50) rwheh  
process readcard; rwheh.deposit end  
process rwheh.remove;printline end  
endprogram

Of course the readcard and printline operations of the spooling processes are not themselves subject to the error handlers in contradistinction to what was the case in (S5).

4.3.3 Spooling system with distributed error handling

To obtain a ten ring-buffer spooling system with distributed error handling capability associated with each ring-buffer all we have to do is change the program body of (S14) to read:

program  
system definitions as (S14)  
array (ring-with-hierarchic-error-handlers(50))RWHEN(1:10)  
(S15) array EXECUTE (1:8)  
process readcard;RWHEN(1).deposit end  
[process RWHEN(1).remove;EXECUTE(1);RWHEN(1+1).deposit end [1] [1,9,1]  
process RWHEN(10).remove;printline end  
endprogram
5. COSY in context: relationship to other formalisms

In this section we relate COSY to other theories or notations, particularly Petri Nets [P60] and other notations which use regular expressions, the event expressions of Riddle [R72] and the flow expressions of Shaw [S78]. We shall also discuss the differences between COSY and the path expressions of Campbell and Habermann [CH74].

5.1 The relationship between COSY and Net Theory

In this section, we discuss formal relationships between COSY programs and transition nets. Such relationships have been an important feature of the notation ever since its inception as a specification language, (as opposed to the use of paths as a synchronisation primitive in [CH74]), as presented in [LC75].

This latter paper used Petri nets as a formal semantics. A construction was defined which associated any path-process program with a marked, labelled transition net, which was intended to express its 'meaning'. Formal notions, such as adequacy, could then be defined in terms of formal concepts in Net Theory, such as liveness. This semantic mapping has been the basis, either directly or indirectly, of all subsequent work on the formal analysis of the notation. It's significance in the development of our thinking about COSY cannot be overstressed. (It should also be pointed out that the paper appeared at a time in which Net Theory had not yet begun to enjoy its current popularity as a system theory.)

The net semantics of [LC75] have since been modified, but the central idea remains the same. Each individual path or process, being essentially a regular expression, is associated, in the classical sense, with a labelled state machine. Putting paths and processes together into a program corresponds to a composition of their associated state machines. The distinction between paths and processes is expressed formally in the nature of the composition in each case.

The current net semantics are based on a composition rule which takes two marked, labelled nets, N₁ and N₂, and delivers a marked labelled net N₁ ⊕ N₂ constructed from N₁ and N₂ by the identification of transitions with the same label.

For those not familiar with Petri-nets, we give the following brief introduction. It must be stressed that the following account is naive and incomplete. More precisely, we are explaining the "token game" on labelled nets, the small part of the theory we need for the following explanation of the semantics.

A marked, labelled, net (hereafter, net for brevity) may be pictured as being composed of labelled boxes ("transitions") and circles ("places") connected by arrows in such a way that no two boxes or circles are connected by arrows.

![A net](image)

Some circles contain dots ("tokens"). The interpretation is that the transitions represent compound events named by their labels, the circles represent compound conditions and the presence of a token in a circle indicates the holding, in some sense, of the condition. An event may occur (a transition may "fire") if and only if all its input places contain tokens (are "marked"). The firing of a transition transforms state holdings in the following manner; one token is
removed from each input place and one token is put into each output place.
Thus, in (Nt1), the transition labelled "a" may fire giving rise to a net with a new marking.

The movement of tokens through a net as a consequence of transition firings may be considered as modelling the dynamic activity of a system.

We remark that Net Theory contains the notion of occurrence net, an object modelling concurrent behaviour, which corresponds to the notion of vector firing sequence.

We may now give the construction of $N_1 \oplus N_2$ from nets $N_1$ and $N_2$, illustrating it with an example.

1) The set of places of $N_1 \oplus N_2$ is the union of the sets of places of $N_1$ and $N_2$ with inherited marking.

2) Suppose $t$ is a transition in either $N_1$ or $N_2$ such that no transition in the other net is labelled with the label of $t$ then $N_1 \oplus N_2$ contains a transition $t'$ with the same label as $t$, whose input and output places are the same as those of $t$ (recall 1).

3) Suppose $t_1,t_2$ are transitions of $N_1,N_2$ respectively with the same label, then $N_1 \oplus N_2$ contains a transition $(t_1,t_2)$ with the same label as $t_1$ and $t_2$ and whose set of input (respectively output) places is the union of the sets of input (respectively output) places of $t_1$ and $t_2$.

"$\oplus$" may be shown to be commutative and associative. If $P = P_1 \ldots P_m$ and $N_i$ is the marked labelled state machine associated with $P_i$, then the net associated with $P$ is defined to be $N_1 \oplus \ldots \oplus N_m$.

We illustrate this construction with an example.

Let us take two paths:

```plaintext
path a;b;a end
path a,c;d end
```
These give two marked, labelled, state machines

(NT3)

(NT4)

which, when combined, give

(NT5)

path a;b;a end
path a;c;d end
5.2 Specification and description based on regular expressions

COSY is only one of a number of software specification tools that have been developed from regular expressions, among which are the event expressions of Riddle [R72], and the associated DREAM system design package, the flow expressions of Shaw [Sh78] not to mention the original path notation of Campbell and Habermann [CH74] and Campbell’s extension of path expressions incorporated into Pascal as a synchronization primitive [CK79]. The three former are compared with each other and with COSY in a comprehensive paper by Alan Shaw [Sh79].

Time and space do not permit us to discuss the differences between all four in depth. However we will try briefly to describe and compare these approaches and indicate their differences.

An event expression is a regular expression enhanced by a number of operators, a shuffle operator and a dagger, together with synchronization symbol pairs, whose job it is to restrict shufflings or interleavings in accordance with a synchronization rule. To quote Shaw [Sh79] "... the effect of the concurrent execution of two components ... is the same as that obtained by interleaving or shuffling the execution history of [then]". In COSY, a sequentialization of a behaviour in which two processes have executed independently would be just such a shuffle. Synchronization symbols constrain shuffles in the way paths constrain the concurrent execution of processes (see section 2.4). The dagger expresses unbounded concurrent availability and corresponds to Campbell and Habermann’s ‘‘[‘’]’s [CH74].

Event expressions were designed to give a semantics for systems consisting of intercommunicating processes described using a “Program process modelling scheme” (PPMS). Each PPMS determines an event expression, whose associated set of strings gives finally a behavioural semantics for the PPMS. We see here that in Riddle’s approach Event expressions do not occupy the “system level” slot, but are intermediate. They themselves define languages of strings.

Riddle has since built upon the above formalism a Design Realization Evaluation and Modelling (DREAM) system [RBBW77] elsewhere in this volume, supported by computer. Riddle’s aim in the development of DREAM are remarkably similar to our own in the development of COSY (see section 1). For example, DREAM is intended for that part of the design process which precedes any specific implementation, intends abstraction and, by use of classes, attempts to achieve modularity and flexibility of design.

Shaw’s flow expressions differ from Riddle’s event expressions in the following way. Shuffle and dagger are present, again intended to model independence and unbounded concurrency, and again, Shaw introduces additional symbols intended to constrain independence. One class of symbols enforces a kind of atomicity on expressions, the second class correspond to binary semaphores. As in Riddle’s case, a flow expression determines a language of strings, where concurrency is intended to be expressed by arbitrary interleaving.

The path expressions of Campbell and Habermann were intended to be used in data objects to express constraints on usage of external procedures by processes. We see here in embryo the “object oriented” approach to system organisation which is typical of COSY, as opposed to the “process oriented” approach seemingly adopted by Riddle and Shaw. It seems clear, however, that originally paths were not introduced as part of a design notation, rather as part of a programming language. Campbell has continued in this vein. His open path expressions have been incorporated into the programming language PASCAL [CK79].

As to the descriptive power of these languages, comparison is rendered difficult by the absence, in both events expressions and flow expressions, of a behavioural semantics which makes a distinction between nondeterminism and concurrency. Presumably, this is because of the still-current confusion in the minds (and models!) of many workers between concurrency and arbitrary interleaving, a confusion unhappily present in the otherwise admirable report of Shaw [Sh79]. Stated succinctly, the position of the aficionados of interleaving semantics

35
seems to be as follows. Firstly, they assume that an interleaving model of behaviour is adequate for the description of concurrency. Feeling confident about this, they do not feel the need to construct or refer to any model of concurrent behaviour. From this it follows that it is not possible for them to conceive of actually justifying their original assumption, since a justification would involve a formal comparison between an interleaving model and one in which non-determinacy and concurrency were not confused. As a consequence, they fail to discover cases in which their original assumption may be shown to be invalid; their attitude is self-justifying.

Possessed with a formal model of concurrent behaviour, we are happily in a position to detect such problems. This can be expressed rather nicely in the sentence in justification of the VPS semantics, that "they may be manipulated as strings except precisely in those cases in which strings are not suitable for describing concurrent behaviour".

That such cases exist (and they do) makes it necessary to view all claims in support of interleaving models with some suspicion.

6. Conclusion

While we have not been able to present here the full scope of the COSY design environment, we hope that we have given a clear idea of its principal aspects. Firstly, our various desiderata, concerning abstraction and the desire to concern ourselves only with the synchronization aspect of systems, have led us to base our design tool around a programming notation. Section two was designed to give a gentle introduction to this basic notation. It was written with the aim of showing how one might program in the notation.

A second, important, desideratum is that a specification in the notation should mean something precise. While section two was designed to help someone express design intentions in the language, section three shows how a program, once written, can be shown to determine a set of behaviours, which may then be formally tested against the programmer's intention and for general properties such as absence of deadlock.

In section four we attempt to show how naturally the basic notation fits in with standard notions of hierarchical design, introducing classlike constructs by which hierarchy and modularity may be expressed.

Section five briefly compares COSY with other system design formalisms.

Further examples of the use of the COSY tool are referred to in the bibliography.

Acknowledgements

The work reported in this paper was supported by a grant from the Science Research Council of Great Britain. The authors would like to thank Mrs. Joan Armstrong for her patience and efficiency in preparing this typescript.

Bibliography


