Abstract

In this paper we illustrate the use of the COSY formalism [LTS79] for specifying, analysing and verifying highly parallel and distributed systems. We shall do this through a non-trivial example, the concurrent resource release mechanism which forms the central part of a novel, non-computational, concurrent and distributed solution to the problem of allocating reusable resources from a limited pool among a large number of concurrent users, the so-called COSY banker [LT80]. After a brief overview of the COSY approach, we formally define a behavioural semantics for COSY programs in terms of string vectors - the vector firing sequences - which generalise the well-known notion of firing sequence to permit the explicit representation of concurrency in an algebraic manner and which may be manipulated in the same manner as strings except in cases where strings are inappropriate for the modelling of concurrent behaviour. Behavioural properties may be formally defined in terms of vector firing sequences. In particular, an analysis of the vector firing sequences of a given program allows one to determine whether a system specified by the program possesses desirable properties, whether these be general such as absence of deadlock or starvation, or specific, that is relating to particular properties required of a particular mechanism. We shall mainly be concerned with the latter form of analysis in our investigation of the concurrent resource release mechanism. More precisely, we demonstrate: firstly, a full characterisation of the behaviours of the mechanism; secondly, the correctness of the mechanism with respect to its desired properties and a functional interpretation of the operations it involves and thirdly, as a consequence of these, its absence of partial system deadlock. Full reference to the copious work on other aspects of the notation are given in a conclusion.

In proceedings, 8th Symposium on Mathematical Foundations of Computer Science, August 31 - September 6, 1980, Rydzyna-Zamek, Poland. Springer Lecture Notes in Computer Science (to appear).

Verifying Concurrent System Specifications in COSY

By

M.W. Shields and P.E. Lauer

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 152
May, 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.
SHIELDS, Michael William

Verifying concurrent system specifications in COSY.
[By] M.W. Shields and P.E. Lauer

Newcastle upon Tyne: University of Newcastle upon Tyne,
Computing Laboratory, 1980.

(University of Newcastle upon Tyne, Computing Laboratory,

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

Suggested classmarks (primary classmark underlined)

Library of congress:
Dewey (17th) 001.64404 0016424
U. D. C. 681.322.06

Suggested keywords
CONCURRENT PROGRAM VERIFICATION
CONCURRENT SYSTEMS
COSY NOTATION
DEADLOCK DETECTION
DISTRIBUTED SYSTEMS
PATH EXPRESSIONS
RESOURCE ALLOCATION
VECTOR FIRING SEQUENCES

Abstract

In this paper we illustrate the use of the COSY formalism [LTS79] for specifying, analysing and verifying highly parallel and distributed systems. We shall do this through a non-trivial example, the concurrent resource release mechanism which forms the central part of a novel, non-computational, concurrent and distributed solution to the problem of allocating reusable resources from a limited pool among a large number of concurrent users, the so-called COSY banker [LTD80]. After a brief overview of the COSY approach, we formally define a behavioural semantics for COSY programs in terms of string vectors - the vector firing sequences - which generalise the well-known notion of firing sequence to permit the explicit representation of concurrency in an algebraic manner and which may be manipulated in the same manner as strings except in cases where strings are inappropriate for the modelling of concurrent behaviour. Behavioural properties may be formally defined in terms of vector firing sequences. In particular, an analysis of the vector firing sequences of a given program allows one to determine whether a system specified by the program possesses desirable properties, whether these be general such as absence of deadlock or starvation, or specific, that is relating to particular properties required of a particular mechanism. We shall mainly be concerned with the latter form of analysis in our investigation of the concurrent resource release mechanism. More precisely, we demonstrate: firstly, a full characterisation of the behaviours of the mechanism; secondly, the correctness of the mechanism with respect to its desired properties and a functional interpretation of the operations it involves and thirdly, as a consequence of these, its absence of partial system deadlock. Full reference to the copious work on other aspects of the notation are given in a conclusion.

About the author

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

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

In this paper we illustrate the use of the COSY formalism \cite{LT79} for specifying, analysing and verifying highly parallel and distributed systems. We shall do this through a non-trivial example, the concurrent resource release mechanism which forms the central part of a novel, non-computational, concurrent and distributed solution to the problem of allocating reusable resources from a limited pool among a large number of concurrent users, the so-called COSY banker \cite{LTD80}. After a brief overview of the COST approach, we formally define a behavioural semantics for COSY programs in terms of vectors of strings -- the vector firing sequences -- which generalise the well-known notion of firing sequence to permit the explicit representation of concurrency in an algebraic manner and which may be manipulated in the same manner as strings except in cases where strings are inappropriate for the modelling of concurrent behaviour. Behavioural properties may be formally defined in terms of vector firing sequences. In particular, an analysis of the vector firing sequences of a given program allows one to determine whether a system specified by the program possesses desirable properties, whether these be general such as absence of deadlock or starvation, or specific, that is relating to particular properties required of a particular mechanism. We shall mainly be concerned with the latter form of analysis in our investigation of the concurrent resource release mechanism. More precisely, we demonstrate: firstly, a full characterisation of the behaviour of the mechanism; secondly, the correctness of the mechanism with respect to its desired properties and a functional interpretation of the operations it involves and thirdly, as a consequence of these, its absence of partial system deadlock. Full references to the copious work on other aspects of the notation are given in a conclusion.

1. Introduction to COST

The COST approach leads us to regard a system as characterisable by the set of (notionally indivisible) operations (actions) it executes (performs) together with a collection of constraints which specify how executions of these operations are to be (partially) ordered. A COST program \( R \) thus determines a set, which we call \( \text{VFS}(R) \), of objects, actually vectors of strings, representing the possible histories of executions of its operations which obey its constraints. The formal theory associated with the notation is concerned with understanding the relationships between a specification object \( R \) and the behaviour it defines, \( \text{VFS}(R) \). Section 4 illustrates this.

The nature of the constraints is as follows. A constraint will prescribe that the elements of some particular subset of the set of operations of the system execute sequentially, that is, no more than one at a time, and only in certain orders. Thus, a constraint defines a set of strings composed of operations belonging to its corresponding subset. It is well known how to represent the set of histories of operation executions of a finite nondeterministic system by means of a regular expression. Briefly, COSY incorporates such regular expressions, called path expressions \cite{CH74}. A single constraint is thus expressed by using a single path expression \( F \); \( F \) constrains the operations it mentions to execute in some sequence. The set of all such sequences is called the set of firing sequences of \( P \) and denoted \( \text{FS}(P) \) (see section 2).

We may now see how a system may be described by a collection of paths. The operations of the system are those mentioned in at least one path in the collection. Roughly, a partial order of executions of these operations will be a permitted behaviour of the system if its restriction to the operations of any constituent path of the collection is a total order corresponding to a firing sequence of that path.

To give the reader some idea of the approach, consider the following example: \( P = \text{P1} \circ \text{P2}, \) where

\[
P_1 = \text{path a};b \text{ end} \quad \text{and} \quad P_2 = \text{path a};c \text{ end}.
\]
Here, the outermost path and end indicate looping forever, while the semicolon denotes sequentialization. The operations of \( P \) are given by \( \text{Ops}(P) = \{a, b, c\} \). Thus the sequences permitted by \( P_1 \) are given by

\[
P_S(P_1) = \{\varepsilon, a, ab, aba, abab, ababab, abababa, \ldots\}
\]

where \( \varepsilon \) denotes the null string. Similarly, we have

\[
P_S(P_2) = \{\varepsilon, a, ac, acac, acaca, acacaca, \ldots\}.
\]

Now, the behaviour "execution of a followed by the concurrent executions of \( b \) and \( c \)" is a behaviour of \( P \), since its restriction to \( P_1 \) is "execution of a followed by execution of \( b \)" or \( a \in P_S(P_1) \), and its restriction to \( P_2 \) is \( ac \in P_S(P_2) \). We remark, in view of what is to come later, that this behaviour may be completely represented by a vector composed of these restrictions, namely \( (a, b, c) \), which is in fact a member of \( VFS(P) \), the set of vector firing sequences of \( P \).

The COSY notation also incorporates generators (replicators) for economically defining regular expressions of arbitrary size and structure in terms of regular expression schemata. We shall see examples of replicators in section 3. Paths together with replicators constitute a subset of the full notation, further details of which may be found in [LTS79, LSB79].

We conclude this section with some remarks on the approach we are taking. 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 specifications of more concurrent and distributed systems. Hence COSY does not contain such constructs as assignment statements, conditional statements, block structures, or any specific synchronization primitives, such as semaphores or monitors. 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 [P76] and CSP [SH78]. There is some evidence that such a relearning process is worth while, since it has led to a number of programs which express interesting highly concurrent and distributed systems in an economical and informative way [LTS79, LD80].

2. Formal aspects of COSY: Syntax and Semantics

2.1 In this section we formally define the language of path expressions and give a semantics for this language by means of a mapping which associates with each path expression \( P \) a set \( VFS(P) \) consisting of vectors whose coordinates are strings made up of operation names belonging to \( P \). The elements of \( VFS(P) \) are interpreted as modelling possible discrete, asynchronous behaviours of a system satisfying the constraints defined in \( P \).

2.2 Individual (R*-path) Paths

An individual or \( R^* \)-path is a string derived from the non-terminal "path" by the following production rules

- \( \text{path} = \text{path} \text{sequence} \text{end} \)
- \( \text{sequence} = \{\text{orelement} \text{@} 1\}+\)
- \( \text{orelement} = \{\text{element} \text{@} 1\}+\)
- \( \text{element} = \text{operation}/\text{element}*/(\text{sequence}) \)

where non-underlined lower case words denote non-terminal symbols; the words "path" and "end", the comma, the semicolon, the star and the right and left parentheses are terminal symbols. The expression \( \{\text{nonterminal} \text{@} \text{1}\}^* \) 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 \( R^* \)-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, "seq" (respectively
"orel" "elem" "op"), denotes any string derivable from a non-terminal "sequence" (respectively "orelement" "element" and "operation").

\[
\begin{align*}
\text{Cyc} & \text{(path seq end)} = \text{Cyc}((\text{seq})) = \text{Cyc}(\text{seq}) \\
\text{Cyc}(\text{orel}_1; \ldots; \text{orel}_n) = \text{Cyc} (\text{orel}_1) \ldots \text{Cyc}(\text{orel}_n) \\
\text{Cyc}(\text{elem}_1; \ldots; \text{elem}_n) = \text{Cyc} (\text{elem}_1) \cup \ldots \cup \text{Cyc} (\text{elem}_n) \\
\text{Cyc}(\text{elem}) & = \text{Cyc}(\text{elem})^* \\
\text{Cyc}(\text{op}) & = \{\text{op}\}.
\end{align*}
\]

Here "." denotes string concatenation (except where it is used as ellipses or full stop), where if X,Y are sets of strings X,Y = \{x,y\}x\in X \land y\in Y."*" has its usual meaning.

To each R*-path P, we associate its set FS(P) of firing sequences.

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

where for any set X of strings, \text{Pref}(X) = \{x|x,y\in X, \text{ some } y\}.

We allow x or y, in the above, to be the null string \epsilon. Thus X\subseteq \text{Pref}(X) and \epsilon \in \text{Pref}(X).

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

2.3 General (GR*-+) paths

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

We now introduce and motivate the definition of VFS(P).

Suppose P = P_1\ldots P_n is a GR*-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 \text{Ops}(P_i); in other words, each x_i contains a record of those operations in \text{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 \text{Ops}(P_i).

Let us consider S as having run for a while and then having halted. It will have generated strings x_i \in \text{Ops}(P_i)^*. What can we say about these x_i? Well first, from the decideratum that the order of executions of operations must obey the constraints of all R*-paths in question, we must have (2) x_i \in \text{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 \bar{x}_i for the new value of x_i, we see that

\[
x_i^* = \begin{cases} 
  x_i.a & \text{if } a \in \text{Ops}(P_i) \\
  x_i & \text{otherwise}.
\end{cases}
\]

We can express the above observations more concisely by going to vectors of strings.

To backtrack slightly, let us consider a family of sets A_1,\ldots, A_n and the corresponding family of string sets A_1\times\cdots\times A_n. We may form the Cartesian product of the A_1\times\cdots\times A_n = \{(y_1,\ldots,y_n)|y_1 \in A_1\times\cdots\times A_n\} and define a concatenation operation on \bar{A}

\[
(x_1,\ldots,x_n) \cdot (y_1,\ldots,y_n) = (x_1,\ldots,y_n,\ldots).
\]

In particular our strings x_i of (3) may be made into a vector \bar{x} = (x_1,\ldots,x_n).

If we let \bar{a}_P = (a_1,\ldots,a_P), where

\[
a_i = \begin{cases} 
  a & \text{if } a \in \text{Ops}(P_i) \\
  \epsilon & \text{otherwise}
\end{cases},
\]

then we see that (3) may be expressed

\[
\bar{x}^* = \bar{x} \cdot \bar{a}_P.
\]

Here, we may say that a has executed at \bar{x} extending it to the history \bar{x}.
Let us denote by \( \text{VFS}(P) \), the set of all vectors \((x_1, \ldots, x_n)\) that might be produced by our system \( S \). Let us denote by \( \text{Vops}(P) \), the set of vectors \( \mathbf{a}_p, a \in \text{Ops}(P) \). We denote by \( \text{Vops}(P)* \) the closure of \( \text{Vops}(P) \) in \( \text{Ops}(P_1)* \times \ldots \times \text{Ops}(P_n)* \) with respect to vector concatenation. Note that \( \text{Vops}(P)* \) contains a null element \( \mathbf{e} = (e, \ldots, e) \). Now we have

\[
(5) \quad x \in \text{VFS}(P) \Leftrightarrow \text{Vops}(P)* \cap (\text{Ops}(P_1) \times \ldots \times \text{Ops}(P_n)*)
\]

\[
(6) \quad \text{Suppose } x \in \text{VFS}(P) \Rightarrow x \cdot \mathbf{a}_p \in \text{VFS}(P) \Leftrightarrow x \cdot \mathbf{a}_p \in \text{Vops}(P)* \cap (\text{Ops}(P_1) \times \ldots \times \text{Ops}(P_n)*)
\]

From (5) and (6), we conclude that

\[ \text{VFS}(P) = \text{Vops}(P)* \cap (\text{Ops}(P_1) \times \ldots \times \text{Ops}(P_n)*) \]

which shall serve as a formal definition for \( \text{VFS}(P) \).

Let us pause briefly and look at this object \( \text{VFS}(P) \). We know that we may write \( x = \mathbf{a}_1 \cdot \mathbf{a}_2 \cdot \ldots \cdot \mathbf{a}_n \), \( \mathbf{a}_i \in \text{Vops}(P) \).

In fact possibly \( x \) may possibly be written in several ways. Writing \([x]_i^1\) for the \( i \)th coordinate of a vector \( x \), let us consider a situation in which \( \forall i \in [1, \ldots, n] \) \[ a_{i-1} \cdot a_i \neq a_i \cdot a_{i-1} \].

In this case \( \mathbf{a}_1 \cdot \mathbf{a}_2 = \mathbf{a}_2 \cdot \mathbf{a}_1 \cdot \mathbf{a}_3 \), and \( x = \mathbf{a}_2 \cdot \mathbf{a}_3 \cdot \ldots \cdot \mathbf{a}_n \).

A glance at the definition of the vector operations \( \mathbf{a}_p \), shows that if \( a \neq b \) then

\[ \mathbf{a}_p \cdot \mathbf{b}_p = \mathbf{b}_p \cdot \mathbf{a}_p \Leftrightarrow \forall i \in [1, \ldots, n] \ a \in \text{Ops}(P_1) = b \in \text{Ops}(P_1) \]

that is no single path constrains the operations \( a \) and \( b \) to execute in any sequence. We conclude that the following interpretation may be made. Let \( x \in \text{VFS}(P) \) and let \( a, b \in \text{Ops}(P) \) with \( a \neq b \). Then \( x \cdot \mathbf{a}_p \cdot \mathbf{b}_p \in \text{VFS}(P) \) and \( \mathbf{a}_p \cdot \mathbf{b}_p = \mathbf{b}_p \cdot \mathbf{a}_p \) may be interpreted as follows: in the system state determined by \( x \), the operations \( a \) and \( b \) may execute concurrently.

Thus the elements \( \text{VFS}(P) \) model concurrent (or asynchronous) behaviors of a system obeying precisely the constraint \( P \).

Let us fix a GRM-path \( P = P_1 \ldots P_n \). We have already defined concatenation in \( \text{Vops}(P)* \). It is clear that \( \text{Vops}(P)* \) is a monoid with identity \( e \) with respect to concatenation.

If \( x, y \in \text{Vops}(P)* \), we define \( x \cdot y = [x \cdot y] \text{ops} \land y \text{ops} \cdot x \text{ops} \).

and \( x^0 = [e] \), \( x^n = x \cdot x^{n-1} \), \( x^* = x \cdot x^1 \cdot x^2 \cdot 1 \ldots \).

We may also define a relation \( \lessdot \) (vector prefix) on \( \text{Vops}(P)* \) by \( x \lessdot y \iff \exists x \in \text{Vops}(P)* \ x \cdot z = y \).

(Vops(P)*, <) is obviously a partially ordered set.

If \( x \subseteq \text{Vops}(P)* \), then we define Pref \( x \) = \( \{ x \in \text{Vops}(P)* | x \subseteq y \}, \text{ some } y \in \text{Ops}(P) \} \).

We observe that Pref(\( \text{VFS}(P) \)) = \( \text{VFS}(P) \); the beginning of a behavior is a behavior.

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

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

\[ I_a(x \cdot b_p) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} \]

2.4 Veriﬁcation of path programs

The semantic mapping \( P \rightarrow \text{VFS}(P) \) now permits us to speak formally of dynamic properties of a system speciﬁed by a path expression; properties of its set of possible behaviors that is of \( \text{VFS}(P) \). We may speak here of two types of properties, which we might call general and specific.
General properties are those which, as the name suggests, apply in general to pairs (P,VFS(P)) as abstract objects, where, roughly, one is not considering P in relation to any specific interpretation. Among such properties are freedom from deadlock and adequacy.

P is deadlock-free if and only if \( \forall x \in VFS(P) \exists a \in \text{Ops}(P) \; x \cdot a \cdot P \in VFS(P) \)

P is adequate if and only if \( \forall x \in VFS(P) \; \exists a \cdot \text{Ops}(P) \; \exists y \in \text{Ops}(P) \; x \cdot y \cdot a \cdot P \in VFS(P) \).

Adequacy is a property akin to absence of partial systems deadlock. Results have been obtained to assist the analysis of a path expression to detect their presence or absence [SL78, S79].

Specific properties, on the other hand, are to do with a path expression or class of path expressions as a description of some actual or projected mechanism. Thus, in the example in the next section, the operations are intended to denote actions of a mechanism on a pool of resources. The problem of verifying such a program is specific in that it applies only to that class and its interpretation and to no other.

Verification thus involves establishing that VFS(P) obeys some predicate which formally expresses the designer's intentions.

3. A resource releasing mechanism

3.1 The problem

We consider the following situation. We have a pool of N (reusable) resources \( R_1, \ldots, R_N \) (they could be pages or buffer frames or devices). We wish to describe/specify a mechanism which is such that

(a) free resources are made available in parallel
(b) resources which have been borrowed may be replaced in parallel
(c) (a) and (b) may proceed in parallel.

3.2 A solution

To each resource \( R_i \) are associated 3 operations; GET(1), signifying that \( R_i \) is secured by some user of the mechanism; SKIP(i), signifying that an available resource \( R_i \) has not been taken by any user of the mechanism and PUT(1), signifying that a borrowed \( R_i \) is replaced by a user of the mechanism.

After any period of activity of the mechanism (or to every vector firing sequence corresponding to such a period), the pool will be in any one of \( 2^N \) states, corresponding to the \( 2^N \) subsets of \( \{R_1, \ldots, R_N\} \). If \( C \subseteq \{R_1, \ldots, R_N\} \) then its corresponding states \( S(C) \) will be that in which the free resources are precisely those belonging to \( C \). A \( S(C) \) will be called a configuration (of free resources). We will represent each state \( S(C) \) by an integer \( c = c(C) \), \( 0 \leq c \leq 2^N-1 \) such that if \( \text{bin}(c) = a_0 \ldots a_{N-1} \) in its binary representation then \( a_i = 1 \iff R_i \notin C \). Thus represents the characteristic function of the set \( C \). To each integer \( 0 \leq c \leq 2^N-1 \) and integer \( 1 \leq i \leq N \) we define \( \text{bit}(c,i) \) to be the \( i \)-th bit in its binary representation i.e. \( \text{bin}(c) = \text{bit}(c,N) \ldots \text{bit}(c,1) \).

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

We now define operations \( \text{CONF}(c), \text{PGETB}(c), \text{PGETE}(c) \); \( \text{CONF}(c) \) signifying that a configuration of free frames \( C(c) \) has been detected; \( \text{PGETB}(c) \) signifying that the resources \( R_i \) belonging to \( C(c) \) are being made available and \( \text{PGETE}(c) \), signifying the end of a block of acquisitions in parallel of the resources in \( C(c) \).

The mechanism proceeds in cycles or blocks of activity; each cycle begins with a \( \text{CONF} \) test followed by the execution of the corresponding \( \text{PGET} \), which releases the appropriate free resources, to be either taken (via \( \text{GET} \) operations) or ignored (\( \text{SKIP} \)). The cycle concludes with a corresponding \( \text{PGETE} \).

Let us begin with the configuration test and parallel releases. We have

\[
(3.1) \quad \text{path} \left( \text{CONF}(0); \text{PGETB}(0); \text{PGETE}(0) \right),
\left( \text{CONF}(1); \text{PGETB}(1); \text{PGETE}(1) \right),
\left( \text{CONF}(2^N-1); \text{PGETB}(2^N-1); \text{PGETE}(2^N-1) \right) \text{ end}
\]
The comma entails exclusive choice between the cycles; configuration test - parallel release begin - parallel release end.

Next we look at the parallel GET, SKIP mechanism. The solution requires that GET\(r\) or SKIP\(r\) may only execute if the precedingly executed operation was a parallel release begin corresponding to a configuration \(c\) in which \(R_r\) is free, that is, such that \(\text{bit}(c,r) = 1\). Further, the GET or SKIP must be immediately followed by the corresponding release end. The following path, one for each resource \(R_r\), is intended to express this:

\[
(3.2r) \text{path } \text{GET}(c_1), \ldots, \text{GET}(c_m); \text{GET}(r), \text{SKIP}(r);
\]

\[
\text{PATH}(c_1), \ldots, \text{PATH}(c_m) \text{ end}
\]

where \([c_1, \ldots, c_m] = \{ c \mid \text{bit}(c,r) = 1 \}\).

Finally we add paths which express the manner in which configurations are modified by GETs and PUTs (SKIPS have no effect on configurations, of course). We sketch the idea behind the path. Suppose \(R_r\) is available, then until \(R_r\) is taken (i.e. before execution of \(\text{GET}(r)\)) the system should be able to make any number of configuration tests showing \(R_r\) to be available i.e.

\[
(3.3r) \text{("configuration tests showing } R_r \text{ available")}^*; \text{GET}(r)
\]
or

\[
(3.4r) (\text{CONF}(c_1), \ldots, \text{CONF}(c_m))^*; \text{GET}(r)
\]

where \(c_1, \ldots, c_m\) are as in (3.2r).

Of course no further \(\text{GET}(r)\) should be executed until \(\text{PUT}(r)\) has been executed. Before the execution of a \(\text{PUT}(r)\), the system should be able to make any number of configuration tests showing \(R_r\) to be taken, or

\[
(3.5r) (\text{CONF}(c_1'), \ldots, \text{CONF}(c_n'))^*; \text{PUT}(r)
\]

where \([c_1', \ldots, c_n'] = \{ c \mid \text{bit}(c,r) = 0 \}\).

Since \(\text{GET}\)'s and \(\text{PUT}\)'s should, of course, strictly alternate, and since initially all resources are available, our path for \(R_r\) may be obtained by sequentially combining 3.4r and 3.5r using a semicolon, as:

\[
(3.6r) \text{path } (\text{CONF}(c_1'), \ldots, \text{CONF}(c_m'))^*; \text{GET}(r);
\]

\[
(\text{CONF}(c_1'), \ldots, \text{CONF}(c_n'))^*; \text{PUT}(r) \text{ end}
\]

The complete solution is obtained by combining the above paths

\[
\]

We shall give the program in a concise form using replicators in the next subsection.

3.3 Some replicator notation

As we remarked in section 1, the full COSY notation contains facilities for the economical representation of paths of arbitrary size and structure, the replicator. We have not the space to go into great detail here, but we shall introduce enough of the replicator notation to give a concise statement of the program constructed in the previous subsection. Further details may be found in [LTS79, LTD59].

First, we introduce the collectivisor. This is a statement declaring an array of subscripted operations. For example, the statement

\[
\text{array GET, PUT, SKIP (1: N)};
\]
declares \(3N\) operations, \(\text{GET}(1), \text{GET}(2), \ldots, \text{GET}(N), \ldots, \text{PUT}(N), \ldots, \text{SKIP}(N)\).

Next, we introduce one form of the replicator. Suppose we have declared a collectivisor, array \(A(j: k)\). Suppose we also have a predicate \(P_r\) defined on the integers. Define

\[
\{i_1, \ldots, i_m \mid i \in [j, j+1, \ldots, k], P_r(i)\}.
\]
If "&" denotes one of the separators "," or ":", then the replicator expression
\[
[A(i) \& \begin{array}{l}
\vdots \\
1, \ldots\ , n, \ldots, 1
\end{array}
]
\]
expands to \( A(i_1) \& A(i_2) \& \ldots \& A(i_m) \).

Thus, for (3.2r), we require a predicate \( F(c,r) \) for each resource \( R_i \); \( F(c,r) \) is true if \( R_i \) is available in configuration \( c \), or formally \( F(c,r) \equiv (\text{bit}(c,r) = 1) \).

We also require a collective array \( \text{PGETB}, \text{PGETE}(0:2^n-1) \). Given this the path may be written
(3.7) \text{path } [\text{PGETB}(c)@, c \mid F(c,r)] \text{; GET(r), SKIP(r); [PGETE(c)@, c \mid F(c,r)] end.}

The replicator is not used only for individual collective names. On the same principle, one can replicate whole subexpressions and even paths. For example (3.1) may be expressed:

\text{path } [(\text{CONF}(c) ; \text{PGETB}(c) ; \text{PGETE}(c))@, c \mid \text{true}] \text{ end}

where \text{true} denotes the predicate that is identically true. In practice, we would not bother to write the predicate in such a case. Finally, as an example of a replicator applied to an entire path, consider

(3.8) \text{path } [\text{PGETB}(c)@, c \mid F(c,r)] \text{; GET(r), SKIP(r); [PGETE(c)@, c \mid F(c,r)] end } [r:1,N,1]

which defines \( N \) paths in each of which \( r \) is replaced by an integer from 1 to \( N \).

Hopefully, this somewhat informal exposition of the replicator notation is sufficient to make understandable the following version of the concurrent resource releasing mechanism:

\begin{verbatim}
program GET, PUT, SKIP (1:N);
array CONF, PGETB, PGETE (0:2^N-1);
predicate F(c,r) \equiv (\text{bit}(c,r) = 1);
(3.9) path [(CONF(c) ; PGETB(c) ; PGETE(c))@, c ] end
[PGETB(c)@, c \mid F(c,r)] ; GET(r), SKIP(r);
[PGETE(c)@, c \mid F(c,r)] end [r:1,N,1]

[CONF(c)@, c \mid U(c,r)] *; GET(r);
[(CONF(c)@, c \mid U(c,r)] *; PUT(r) end [r:1,N,1]
\end{verbatim}

In the next section we shall characterise the behaviour and properties of this system.

4. Analysis of the resource releasing program

In constructing the program (3.9) we made certain statements in justification of the various constructions introduced. For example, we assumed, in building it, that the CONF test was actually meaningful as an indication of the instantaneous configuration of free resources. Making such assumptions is part of the business of writing programs. That one has made them does not guarantee their accuracy, however.

In this section we are going to sketch a formal proof that (3.9) does what it is supposed to do, and without deadlocking. We call (3.9) \text{RHM}(N). One of our tasks will be to characterise the possible behaviours of \text{RHM}(N), that is, the set \text{VPS}(\text{RHM}(N)). Proofs have been omitted from consideration of space. They may be found in [SL88].

To begin with, let us express \text{RHM}(N) as a concatenation of its individual paths
\( \text{RHM}(N) = \text{SEL}(N)\{\text{PG}(1) \ldots \text{PG}(N)\}\{\text{RP}(1) \ldots \text{RP}(N) \} \) where we are using the mnemonic \text{SEL} (select configuration), \text{PG}(i) (i\text{th parallel get}) and \text{RP}(i) (i\text{th resource pool}}
We shall find it useful to write out the Cycle sets of these individual paths, which are as follows

\[
\begin{align*}
\text{Cyc}(\text{SEL}(N)) &= \{\text{CONF}(c).\text{GET}(c).\text{PUT}(c) \mid c \in [0, \ldots, 2^{N-1}]\} \\
\text{Cyc}(\text{PG}(r)) &= \{\text{GET}(c).\text{PUT}(c) \mid c \in [0, \ldots, 2^{N-1}]\} \\
\text{Cyc}(\text{RC}(r)) &= \{\text{CONF}(c).\text{GET}(c).\text{PUT}(c) \mid c \in [0, \ldots, 2^{N-1}]\} \\
\text{Cyc}(\text{PF}(r)) &= \{\text{CONF}(c).\text{GET}(c).\text{PUT}(c) \mid c \in [0, \ldots, 2^{N-1}]\}
\end{align*}
\]

where \(P(c, r)\) is the predicate defined in (3.7).

\[
\text{Cyc}(\text{RF}(r)) = \{\text{CONF}(c).\text{GET}(c).\text{PUT}(c) \mid c \in [0, \ldots, 2^{N-1}]\}
\]

In section 3, we explained that the system was intended to operate in a sequence of 'blocks' of activity, beginning with a configuration test \(\text{CONF}(c)\), for some \(c\). We shall state this in our first lemma, in which we begin the characterisation of \(\text{VFS}(\text{RRM}(N))\). We now define the blocks in question. First redefining \(C\) from section 3, if \(c \in [0, \ldots, 2^{N-1}]\), then \(C(c) = \{r \in [1, \ldots, N] \mid P(c, r)\}; C(c)\) indicates the set of resources in a given configuration \(c\). Likewise we define \(E(c)\) to be the corresponding set of empty resources \(E(c) = \{1, \ldots, N\} \setminus C(c)\).

If \(\text{RRM}(N)\) is working properly, it should be the case that in configuration \(c\), the only \(\text{GET}(i)\)'s and \(\text{SKIP}(i)\)'s that may execute must satisfy \(i \in C(c)\) and the only \(\text{PUT}(i)\)'s that may execute must satisfy either \(i \in E(c)\) or \(\text{GET}(i)\) must have executed. A block should therefore be of the form

\[
x(c, y, y) = \text{CONF}(c).\text{GET}(c).y(c, y, y).\text{GET}(c)
\]

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

\[
\text{GET}(r_1), \ldots, \text{GET}(r_k), \text{SKIP}(r_{k+1}), \ldots, \text{PUT}(r_m), \ldots, \text{PUT}(r_n)
\]

where \([r_1, \ldots, r_k] = X \subseteq C(c), [r_{k+1}, \ldots, r_m] = C(c) \setminus X\) and \([r_{m+1}, \ldots, r_n] = Y \subseteq E(c) \setminus X\).

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

Finally, we let

\[
\text{Block}(N) = \{x(c, X, Y) \mid X \subseteq 2^{N-1} \setminus X \subseteq C(c) \wedge Y \subseteq E(c) \setminus X\}
\]

**Lemma 1**

\[
\text{VFS}(\text{RRM}(N)) \subseteq \text{Pref}(\text{Block}(N)^+).
\]

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

\[
\text{PB}(N) = \text{VFS}(\text{RRM}(N)) \cap \text{Block}(N)^+.
\]

Suppose \(x \in \text{VFS}(\text{RRM}(N))\) with \(x \neq \emptyset\), then \(x = x'.\text{CONF}(c).y, x' \in \text{PB}(N)\) and \(\text{CONF}(c).y \in \text{Pref}(\text{Block}(N))\). We may define sets \(X = \{r \mid \text{GET}(r) \text{ is in } y\}, Y = \{r \mid \text{PUT}(r) \text{ is in } y\}\) and \(Z = \{r \mid \text{SKIP}(r) \text{ is in } y\}\). It may be shown that \(\text{CONF}(c).y \leq x(c, x, y)\). In fact, more than this is true.

**Lemma 2**

(1) With the above notation, for any set \(X^*\), \(X \subseteq X \subseteq C(c) \setminus Z\), and for any set \(Y^*, Y \subseteq Y \subseteq E(c) \setminus U\), we have \(x \leq x'.x(c, x^*, y^*) \in \text{PB}(N)\) and in particular

(2) If \(x \in \text{CONF}(c) \in \text{VFS}(\text{RRM}(N))\), then \(X \subseteq C(c) \setminus Z \subseteq E(c) \setminus U\). \(x.x(c, x, y) \in \text{PB}(N)\) whenever

(3) \(\text{VFS}(\text{RRM}(N)) = \text{Pref}(\text{PB}(N))\).

We must now characterise \(\text{PB}(N)\). We know that its elements are of the form \(x(c_1, x_1, y_1), \ldots, x(c_1, x_1, y_1)\), and we know how the \(X_1\) and \(Y_1\) are constrained by the \(c_1\). Hopefully, \(c_{i+1}\) will be determined by the previous configuration \(c_i\) and what has been done to it (by \(X_i\) and \(Y_i\)). With this in mind, we define, for every \(x(c, X, Y)\) in \(\text{Block}(N)\):
Next(c,X,Y) = c', where \( O(c') = \{ O(c) - x \} \cup Y \).

Clearly Next(c,X,Y) \( \in \{ 0, \ldots, 2^N - 1 \} \).

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

**Lemma 3**

Suppose \( x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9 \in \text{FB}(N) \). Let \( a \in \text{Vops}(RRM(N)) \), then \( x \cdot x(c,X,Y), a \in \text{VFS}(RRM(N)) \Leftrightarrow a = \text{CONF}(\text{Next}(c,X,Y)) \).

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

**Proposition 1**

1. \( x \in \text{FB}(N) = \exists 1, c_1, \ldots, c_l, x_1, \ldots, X_1, Y_1, \ldots, X_l \)
   
   (a) \( x = x(c_1, x_1, Y_1), \ldots, x(c_l, x_l, Y_l) \)

   (b) \( \forall i \in \{ 1, \ldots, l \}: X_i \subseteq O(c_i) \land Y_i \subseteq B(c_i) \cup X_i \)

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

   (d) \( c_l = 2^N - 1 \)

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

Now that we have fully characterised the behaviour of RRM(N), we may ask ourselves whether it does what it's supposed to. We define a function \( f: \text{VFS}(RRM(N)) \rightarrow Z^N \), where \( f(x) \) will be designed to represent the content of the resource pool after \( x \) has happened: for \( x \in \text{VFS}(RRM(N)) \) define \( f(x) = g(x) + 1 \) where \( 1 = (1, \ldots, 1) \) and

\[
[f(x)]_i = \begin{cases} 
I_{\text{PUT}(i)}(x) - I_{\text{GET}(i)}(x), & i \in \{ 1, \ldots, N \}, \\
0, & \text{otherwise}.
\end{cases}
\]

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

**Lemma 4**

1. \( f(c) = 2^N - 1 \)

2. \( \forall x,y \in \text{Vops}(RRM(N)) ^* : g(x,y) = g(x) + g(y) \)

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

**Lemma 5**

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

From this, we may deduce:

**Proposition 2**

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

From proposition 3, we now have the proof of correctness of \( \text{RRM}(N) \), as follows.

1. If \( x \) is a history of the system, then a configuration test applied at this point correctly identifies the configuration [Proposition 2].

2. GET's can only execute on free resources and PUT's can only execute on borrowed resources [From Proposition 4, the definition of \( x(c,X,Y) \) and Proposition 2].

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

We conclude this section with a sketch of a proof of adequacy. Adequacy, which implies freedom from deadlock, follows from the next lemma.
Lemma 6

(1) \( \forall z \in [0,1]^N \forall c \in [0, \ldots ,2^N-1] \exists x \in C(c) \exists y \in E(c): z = g(x, y, c) + v(c) \)

(2) \( \forall a \in \text{Vopc}(\text{RRM}(N)) \exists x \in \text{Block} (N): I_a (x) = 1 \).

Proposition 3

For all \( N > 0 \) RRM(\( N \)) is adequate.

We have now established that RRM(\( N \)) performs as required.

We remark that the program RRM(\( N \)) is actually part of a rather larger program specifying a highly concurrent and distributed COSY solution to the Banker's Problem [Dijkstra 68], given in [LTD80]. This larger program contains, as well as RRM(\( N \)), which acts as a 'kernel' of it, the specification of access to the kernel by customers, a specific resource-to-customer allocation mechanism, various devices, based on counters, which ensure fairness and absence of starvation and a specification of customer structure. A proof of the correctness of the banker program, in a style similar to the above, has been sketched.

Conclusion

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

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

We have not here presented the full COSY notation as presently developed; for example we have avoided the topic of processes, distributors or multiply nested distributors. Details of these language features may be found in [LT879].

There also exist a number of results concerning the problem of deducing the adequacy properties of a system defined by a path from the syntactic structure of that path. Some of these may be found in [ST79].

Finally, there has always been a strong connection between the COSY notation and Net Theory. The original formal semantics for the basic COSY notation was given in terms of a mapping from programs to nets [LC75] and this was developed further in [LSB79] and [LSB79a]. A relationship between paths and nets on both the system (path to net) and process (vector firing sequence to causal net) levels is given in [ST79].

Acknowledgements

The research reported in this paper has been encouraged and supported by the Science Research Council of Great Britain. Some of the examples in this paper were developed in collaboration with P.R. Torrigiani and R. Devillers [LT880]. Thanks to Mrs. Joan Armstrong for her excellent typing on such short notice.

Bibliography


