Abstract:

Rigour, in the analysis of synchronisation mechanisms requires a firm mathematical milieu. This principle is applied to a study of the extended semaphore primitives (ESP's) of Agerwala. ESP's are provided with a formal semantics in terms of vector firing sequences (VFS's); objects for describing potential discrete concurrent behaviour, which may be manipulated in a clean algebraic manner. It is shown that any 'program' using ESP's as its only means of synchronization and which is in some sense 'bounded' has equivalent descriptions in the COSY formalism. COSY is a notation for the abstract specification of discrete systems which has a semantic in terms of VFS's. It is demonstrated that a bounded ESP program may be represented as a COSY program, both of which have identical sets of VFS's. The report concludes with a discussion concerning extensions of this construction to the unbounded case, which essentially speculates on a 'completion' construction for the COSY formalism.

A formal semantics for concurrent systems

By

M.W. Shields and P.E. Lauer


TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 132

April, 1979.

© 1979 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.
Rigour in the analysis of synchronization mechanisms requires a firm mathematical milieu. This principle is applied to a study of the extended semaphore primitives (ESP's) of Agerwala. ESP's are provided with a formal semantics in terms of vector firing sequences (VFS's); objects for describing potential discrete concurrent behaviour, which may be manipulated in a clean algebraic manner. It is shown that any program using ESP's as its only means of synchronization and which is in some sense 'bounded' has equivalent descriptions in the COSY formalism. COSY is a notation for the abstract specification of discrete systems which has a semantics in terms of VFS's. It is demonstrated that a bounded ESP program may be represented as a COSY program, both of which have identical sets of VFS's. The report concludes with a discussion concerning extensions of this construction to the unbounded case, which essentially speculates on a 'completion' construction for the COSY formalism.

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 the IBM Laboratory, Vienna.
1. Introduction

It is important, in the discussion of concurrent systems, to have a firm formal basis for their analysis, since in such cases intuition is even more inadequate and misleading than is the case for sequential systems. For this reason it is important to be able formally to define the 'meaning' of any synchronisation mechanism in order to be able to define systems properties involving such mechanisms and to analyse them properly. However, to reduce the great complexity arising from the interaction of processes in concurrent systems, it is important to abstract from irrelevant details of particular mechanisms. The COSY notation and its accompanying system theory, which is discussed in this paper, was designed to satisfy the requirements of formality and appropriate abstractness. Moreover, the accompanying system theory has been developed sufficiently to permit the application of formal results to discover deep (non-obvious) properties of concurrent systems such as absence of deadlock and starvation and degree of concurrency and distribution.

In this paper we look at one synchronisation mechanism, the extended semaphore primitives (ESPs) of Agarwala, and give a formal meaning for a class of programs involving these primitives. We then sketch out a concurrency preserving translation of such programs into the COSY notation. COSY is a formalism for describing and analysing synchronic properties of systems, those properties which have to do with the relationship between event occurrences. A program in the COSY notation is a collection of statements, essentially regular expressions, each of which describes, for a subset of the set of events associated with a system, how these are related sequentially. A collection of such statements relates elements of the whole set of events either sequentially or concurrently. With each such program is associated a set of n-tuples of strings, called vector firing sequences, which formally describe possible 'histories' of the system being specified, in terms of the component histories of the subsystems into which the system is decomposed. COSY has associated with it a formal theory which is concerned with the relationship between properties of descriptions, that is COSY programs, and the properties of their possible histories such as deadlock and starvation. The advantage of translation of ESP programs into COSY is that a formal theory for defining and analysing system properties is associated with the former.

Vector firing sequences are one of a number of possible ways of formally describing the set of behaviours or histories specified by a COSY program. The principal semantics of COSY are those given in [Lauer 75] and modified in [Lauer 73a], which map COSY programs to marked labelled transition nets [Petri 76]. In the papers cited we point out that there are standard ways in this semantics for associating
with a COSY program a set of either firing sequences, or of labelled causal nets which may be said to define the behaviour of the corresponding system. In [Lauer 75] a notion of behaviour based on labelled posets and in [Shields 78] a notion of behaviour based on firing sequences, in both cases defined directly in terms of COSY programs without the mediation of the net semantics, were given and used as the basis for the development of general theorems concerning the relationship between systems definition and system behaviour - mostly concerning deadlock problems. Vector firing sequences themselves may be considered as means for modelling concurrent behaviour which have the advantages both of the firing sequence and of the labelled poset model. They may be treated as strings, while labelled posets are clumsy to manipulate. At the same time they are formally equivalent to labelled causal nets and do represent concurrency. In fact, a set of vector firing sequences of a COSY program may be regarded as a trace language in the sense of Mazurkiewicz [Mazurkiewicz 77].

In the present paper, section 2 serves to give a brief introduction to the basic COSY notation. We explain how vector firing sequences may be considered as describing histories of a concurrent system. We then show how a basic COSY program determines a set of vector firing sequences, which may be considered as formally modelling the set of all possible histories of the system specified by the program. Section 3 deals with ESP programs in preparation for the translation to COSY programs in section 4. In order to show that our translation rules are 'correct', that is, preserve meaning, it is necessary to give the ESPs, which were introduced informally and by example in [Agerwala 77], a precise meaning. In section 3 we explicate this 'meaning' in terms of the notion of vector firing sequences referred to above. We may then formally define the translation to be correct iff, for a given ESP program E and its corresponding COSY program COSY(E), E and COSY(E) have the same set of vector firing sequences.

In section 4, we introduce this translation, using some of the macro notation of COSY, and sketch a proof of its correctness in the case in which the semaphores are bounded. In section 5 we indicate how the translation may be extended to deal with the unbounded case.

2. Introduction to the COSY notation and its vector firing sequence semantics

In this section we present the COSY (COncurrent SYstem) notation. COSY is a language whose terminal objects (programs) constitute abstract descriptions of systems in terms of their synchronic properties. The notation itself is a development of the path expressions of Campbell and Habermann [Campbell 74] and of the path-process notation of Lauer and Campbell [Lauer 75]. Essentially, the COSY notation adds generators to the path-process notation so that systems can be specified as path and process patterns (templates) from which instances of paths and processes can be generated in an orderly manner.
A system, from the COSY point of view, consists of a collection of resources and of sequential but nondeterministic processes. A resource is represented by a set of atomic actions (operations) together with a collection of statements expressing constraints on the order of activation of these operations (paths). A process is represented by an expression (process expression) which describes the pattern of usage of resources required by the process. Formally a process expression determines a collection of sequences of activations of operations. Distinct processes are notionally parallel; however, the paths determine a set of usages of resources of the system as a whole and thereby achieve a co-ordination of the processes.

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

\[
\begin{align*}
\text{<program> } &= \text{ begin } \text{<programbody>} \text{ end} \\
\text{<programbody> } &= \text{<path>}, \text{<process>}, \text{<path>}, \text{<process>} \text{<programbody>} \\
\text{<path> } &= \text{<sequence> end} \\
\text{<sequence> } &= \text{<element>}, \text{<sequence>}, \text{<element>} \\
\text{<element> } &= \text{<operation>}, \text{<element>}, \text{<sequence>} \\
\text{<process> } &= \text{ process } \text{<sequence> end}
\end{align*}
\]

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

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

\begin{verbatim}
begin
  process request_a1_a2; use_a1_a2; release_a1_a2 end
(1) process request_a2_a3; use_a2_a3; release_a2_a3 end
  path (request_a1_a2;release_a1_a2),(request_a2_a3;release_a2_a3) end
end
\end{verbatim}

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

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

\begin{verbatim}
(2) path push;(push;(push;pop)*;pop)*;pop end
\end{verbatim}

which defines the behaviour of a three-frame stack which is initially empty. The star binds more strongly than the comma (and hence the semicolon), whence the need for parentheses.
between two operations with the same name occurring in different processes. For example, in the program \begin{verbatim} begin process a end process a end end \end{verbatim}, one has two processes which may concurrently be activating a; there are two 'a's in both 'a-in-process-1' and 'a-in-process-2'. If we were to insert into this program \texttt{path a end} then the 'observer' associated with the path would have to see a sequence of 'a's in order that his constraint hold, that is, the path enforces mutual exclusion between occurrences of 'a-in-process-1' and 'a-in-process-2'. This suggests the following transformation, which makes the semantics of processes explicit and permits one to define \texttt{VFS(P)} for any program. Let \texttt{Pr = begin P1 \ldots Pm end}, where each \texttt{Pi} is either a path or a process. If \texttt{Pi} is a process, then replace every operation \texttt{a \in Ops(Pi)} by an operation \texttt{a\&i} ('a-in-Pi'). Do this for every process. Then we make the mutually excluding effect of paths on process operations with the same name explicit. Suppose \texttt{a} is an operation occurring in processes \texttt{P1, \ldots, Pn}; if \texttt{a} belongs to a path \texttt{Pi}, then replace \texttt{a} in \texttt{Pi} by the occurrence \texttt{a\&i}. Finally, replace each 'process' by 'path'. We shall denote the resulting program by \texttt{Path(Pr)}. \texttt{Path(Pr)} consists exclusively of paths, hence, we may thus define \texttt{VFS(Pr) = VFS(Path(Pr))}.

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

\begin{verbatim}
begin
(2) process rq; rl end process rq; rl end path rq|rl end
end
This translates as follows:
begin
(3) path rq\&1; rl\&1 end path rq\&2; rl\&2 end path rq\&1, rq\&2; rl\&1, rl\&2 end
end
\end{verbatim}

This has the following set of vector firing sequences:

\texttt{Pref(\{(rq\&1 rl\&1, e, rq\&1 rl\&1), (e, rq\&2 rl\&2, rq\&2 rl\&2)\}*).}

where 'Pref' is defined in analogy with the string case, that is for
\[ X \subseteq \text{Vops(P)}, \text{Pref}(X) = \{ x \in \text{Vops(P)}* | \exists y \in \text{Vops(P)}* : xy \in X \} \]

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

3. A vector firing sequence semantics for extended semaphore primitives

The extended semaphore primitives (ESP's) \texttt{pe} and \texttt{ve} are assumed indivisible and each operates on a set of semaphores which must be initialized to non-negative integer values [cf. Agerwala '77]
\[ \text{pe}(S_1, \ldots, S_k, \bar{S}_{k+1}, \ldots, \bar{S}_{k+j}) \]

\text{if for all } i, 1 \leq i \leq k, S_i > 0 \text{ and for all } j, 1 \leq j \leq S_{k+j} = 0

\text{then for all } i, 1 \leq i \leq S_i, i := S_i - 1

\text{else the process is blocked}

\[ \text{ve}(S_1, S_2, \ldots, S_k) \]

\text{for all } i, 1 \leq i \leq k, S_i := S_i + 1

ESPs are used, of course, to co-ordinate concurrent processes in order to satisfy some general desideratum associated with the system being designed. The desideratum may be of a synchronic nature, for example, that the system be free from deadlock. Formal verification of deadlock-freeness in a program using ESPs requires that such synchronic notions be precisely defined and hence it is necessary to have a formal notion of a possible history of an ESP program. No such notion was offered in [Agerwala 77] so we will have to do it ourselves. We use the vector firing sequence method of modelling behaviour as described in the last section.

Agerwala says nothing about the kind of programming language in which ESPs might be embedded or might have constructed around them. His examples however deal with situations in which ESPs are the only means of process co-ordination. We shall define, therefore, an ESP program to be a collection of cyclic, sequential processes without jumps or conditional statements, using ESPs as their only means of synchronisation. By an ESP program, therefore, we mean something of the form

\[ \text{semaphore } S_1, \ldots, S_n \text{ initial} (S_1, \ldots, S_n) = (M_1, \ldots, M_n). \]

\[ \text{loop } a_1^1; \ldots; a_r^1 \text{ end } \ldots \text{ loop } a_1^m; \ldots; a_r^m \text{ end} \]

which we shall call \( E \), where the \( a_j^i \) are either ESP operations or are in some sense local to the process containing them. Thus, if an operation occurs in two distinct processes of \( E \), then the two processes may be activating a concurrently, unless prevented from doing so by the semaphores. We are accordingly in the same situation as that concerning operations in distinct COSY processes. As in that case, we make a distinction between identically named operations in distinct processes. Our vector firing sequences \( x \in \text{VFS}(E) \), will thus contain operations of the form \( a_j^i \) as in the COSY case. Note that we are implicitly assuming the indivisibility of pe and ve operations by representing each instance of one of these operations in a process by a single symbol. Let us denote \( \text{loop } a_1^1; \ldots; a_r^1 \text{ end} \) by \( Q_1 \), and define \( \text{FS}(Q_1) \)

\[ \text{FS}(Q_1) = \text{Pref}(\{a_1^1; \ldots; a_r^1\}) \]

For the given program \( E \), we define \( \text{VFS}(E) \) and \( \text{eval}_z \text{VFS}(E) + Z^n \) as follows. (\( Z \) is the set of integers).

1) \( (\varepsilon, \ldots, \varepsilon) \) (n+1 times) = \( \varepsilon_0 \text{VFS}(E) \) and \( \text{eval}[[(\varepsilon_0)]]_i = M_i \) for each \( i \). (\( [z]_i \) denotes the \( i \)th co-ordinate of the \( n \)-tuple \( z \).)
2) Now suppose \( x \in \text{VFS}(E) \) and \( a \) is some \( a_i \) appearing in \( Q_1 \), such that \( \{ x \}_i^{a_i \in \text{VFS}(Q_1)} \).

a) If \( a \) is not an ESP then \( x \in \text{VFS}(E) \), where \( x \) is defined by
\[
\{ x \}_i = \{ x \}_j^{a_i} \text{ if } j = i, \{ x \}_j \text{ otherwise, and eval}_E(x) = \text{eval}_E(x).
\]

b) If \( a = \text{ve}(S_{i_1}, \ldots, S_{i_k}) \) and \( a \) belongs to \( Q_1 \), then \( x \in \text{VFS}(E) \), where
\[
\{ x \}_j = \{ x \}_j^{a_i} \text{ if } j = i \text{ or } j = m_i, h = 1, \ldots, k \text{ and } \{ x \}_j = \{ x \}_j \text{ otherwise, and we set}
\]
\[
\text{eval}_E(x)_j = \text{eval}_E(x)_j + 1 \text{ if } j \in \{ i_1, \ldots, i_k \} \text{ and } \text{eval}_E(x)_j = \text{eval}_E(x)_j \text{ otherwise.}
\]

\( \text{VFS}(E) \) contains only the \( n+m \)-tuples determined by (1) and (2).

Thus, if \( x \notin \text{VFS}(E) \), \( \{ x \}_i \text{ for } i \in \{ 1, \ldots, n \} \) is a history of the process \( Q_1 \) and \( \{ x \}_{i+k}^{a \in \text{ve}(S_{i_1}, \ldots, S_{i_k})} \) is a sequence of activations of pe and ve operations involving the semaphore \( S_i \). \( \{ x \}_j \) gives the value of the semaphore \( S_i \) after the history \( x \) has happened. (a), (b) and (c) reflect the manner in which ESPs determine how occurrences of the operations belonging to \( E \) may be partially ordered.

4. Proof of correctness of a concurrency preserving translation from ESP programs to COSY programs with bounded semaphores.

In this section we define a construction which takes a program of the type (4) and produces a program in COSY with the same 'behaviour'. In the process of doing so, we introduce some of the macro notation associated with COSY. This allows one to write long programs containing iteratively definable structure in a succinct manner.

Let us first consider the case of a semaphore \( S \) with test for zero initialised to \( M \) and capable of taking values from 0 to \( N+M-1 \). The semaphore will have a sequential history and may be described by a single path. This path will be of the form
\[
\text{path } I(\overline{N-M})p(0) \text{ end, where } I(\overline{N-K}) \text{ concerns that part of } S \text{'s history concerning}
\]
\[
\text{increments of } S \text{ within the range } M \text{ to } N \text{ and } D(N), \text{ that part of } S \text{'s history concerning}
\]
\[
\text{decrements in the range } M \text{ to } 0 \text{ and a test for zero, } p(\overline{S}). \text{ Let us see what } I(k)
\]
\[
\text{should be, for } k>0. \text{ Since we are concerned with increments, the first thing that may happen is an operation } v(S), \text{ after which comes a history of a semaphore which may perform up to } k-1 \text{ increments, after which comes a decrement, } p(S). \text{ It is thus sensible to define } I(k) = \langle v(S) p(1) I(k-1) p(S) \rangle^* \text{ with } I(1) = \langle v(S) p(S) \rangle^* \text{ and } D(0) = p(S)^*. \text{ The path describing the semaphore may thus be written:}
\]
\[
\text{path } (v(S); \ldots; (v(S);p(S)^*; \ldots); p(S)^*); (p(S); \ldots; (p(S);p(S)^*; v(S)^*); \ldots; v(S)^*); \text{end}
\]

\[
\text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M}
\]

\[
\text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M}
\]

\[
\text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M}
\]

\[
\text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M} \quad \text{N-M}
\]

8
The macro notation associated with COSY contains facilities for the definition of such iterative structures, specifically the replicator. One version of the replicator is \([p\leftarrow q\mid k,n,m]\) where \(k>0\), \(n>0\), \(m \neq 0\) and \(p\) and \(q\) are strings, which is expanded as follows:

\[
[p\leftarrow q\mid k,n,m] = \begin{cases} 
  p[p\leftarrow q\mid k+n,n,m] & \text{if } 0<k+m \leq n \\
  pq & \text{if } k+m < k+e \\
  s & \text{otherwise}
\end{cases}
\]

The "\(\leftarrow\)" in a replicator of the form \([p\leftarrow q\mid k,n,m]\) indicates that "\(\leftarrow\)" is a separator not a terminator and

\[
[p\leftarrow q\mid k,n,m] = \begin{cases} 
  p[p\leftarrow q\mid k+n,n,m] & \text{if } 0<k+m \leq n \\
  pq & \text{if } k+m < k+e \\
  s & \text{otherwise}
\end{cases}
\]

We shall not go into detail here – an extensive treatment of the COSY macro facilities may be found in [Lauer 75b] – but merely give macro definitions of the structures we need in the sequel. In terms of this notation, \(I(k)\) would be written as:

\[
[(v(S)\leftarrow 1,1,p(S))\mid 1,k,1]
\]

and the formal definition of the replicator ensures that \(I(0) = s\). \(D(k)\) would be written as:

\[
[(p(S)\mid 1,k,1) p(S) [v(S)\leftarrow 1,1,1,1]
\]

We may thus define a path \(P(S,M,N)\) describing a semaphore \(S\) with test for zero, initialized to \(N\) and bounded by \(N\) as follows. If \(N>M\), then \(P(S,M,N) = \text{path}\)

\(I(0),D(N)\) \(\text{end}\). If \(N = M\), then \(P(S,M,N) = \text{path}\ D(N)\) \(\text{end}\).

We now extend this argument to give a COSY description of a collection of extended semaphores in an ESP program \(E\) of the form (4). The COSY version of \(E\) will be a program of the form:

\[
\text{begin } Q_1 \ldots Q_m P_1 \ldots P_n \text{ end}
\]

where each \(Q_i\) is a process

\[
Q_i = \text{process } a_i^1 \ldots a_i^n \text{ end}
\]

and each \(P_i\) is a path corresponding to the semaphore \(S_i\).

Let us see what \(P_i\) ought to be. IN \(P(S_i,M_i,N_i)\), a \(p(S)\) operation has the effect of decreasing the value of \(S\) by 1. In the program \(E\), this may be effected by any operation \(p(S)\), \(i\), and wherever a \(p(S)\) is valid a \(p(S)\) is valid provided it is so for the other semaphores mentioned.

This suggests the following construction. For each \(S_i\) define \(N_i\) to be the maximal value taken by \(S_i\) over all possible histories of \(E\) and define:

- a) \(PS_i = PS_i^1, \ldots, PS_i^h\) where the \(PS_i^j\) are all the \(p(S)\) operations mentioned in \(E\).
- b) \(VS_i = VS_i^1, \ldots, VS_i^e\) where the \(VS_i^j\) are all the \(v(S)\) operations mentioned in \(E\).
c) \( \overline{P}_s = \overline{P}_s^1, \ldots, \overline{P}_s^j \) where the \( \overline{P}_s^j \) are all the pe\((\ldots, \overline{s}_1, \ldots) \) occurring in \( E \).

We assume that every semaphore is used somewhere in the program, and that thus all the orelments \( \overline{P}_s \) and VS\(_s \) are non-null.

In order to construct \( P\), first form \( P(\overline{s}_1, M_1, M_1) \). Next, replace each \( p(s_i) \) by \( \overline{P}_s \), each \( v(s_i) \) by \( VS_s \). If \( \overline{P}_s \) is not null, replace \( p(s_i) \) by it, otherwise delete "\( p(\overline{s}_1)\)" from the path. Call the COSY program derived from \( E \) by this procedure COSY(\( E \)).

We illustrate this construction by translating Agerwala's ESP solution to the second reader-writer problem:

```plaintext
semaphore A, R, M; initial(A, R, M) = (0, 0, 1)
(6) (reader) loop pc(M, A); vc(M, R); read; pe(R) end
(writer) loop ve(A); pc(M, R); write; ve(M); pe(A) end
```

We assume that there are \( r \) readers and \( w \) writers. It may be seen that the maximum value that can be attained by \( A, R \) and \( M \) is, respectively, \( w, r \) and 1. Applying the above construction, we obtain the following COSY program.

```plaintext
begin
  [proccnt pc(M, A); ve(M, R); read; pe(R) end (1) [1, r, 1]
  [proccnt ve(A); pc(M, R); write; ve(M); pe(A) end (1) [1, w, 1]
  path [(ve(A)); (1); (pc(A))]* [1, w, 1], pc(M, A)* end
  path [(vc(M, R)); (1); (pc(R))]* [1, r, 1], pc(M, R)* end
  path (pc(M, A), pc(M, R); ve(M, R); vc(H))]* end
end
```

The COSY version (7) of the ESP program (6) is slightly longer than (6). Note, however, that the synchronization properties implicit in (6) have been made explicit in (7).

An ESP program \( E \) is an object which defines synchronic relationships between its operations indirectly via a functional interpretation of the pe and ve operations contained in it. COSY(\( E \)) is an object which defines synchronic relationships between its operations directly; we shall now show that the pe and ve operations in COSY(\( E \)) may be given a functional interpretation. This, indeed, is central to the proof that the translation from \( E \) to COSY(\( E \)) is correct. For \( E \) as in (4), we define a function \( \text{eval}_{\text{COSY}(E)}: \text{VFS}(\text{COSY}(E)) \rightarrow \mathbb{Z}^n \) as follows:

1) \( \text{eval}_{\text{COSY}(E)}(\alpha)_i = M_i \) for each \( i \).

2) Suppose \( x \in \text{VFS}(\text{COSY}(E)) \) and \( a \) is some \( a^j_i \) appearing in a process \( Q_i \) and that \( x_{aki} = x \in \text{VFS}(\text{COSY}(E)) \)

   a) If \( a \) is not an ESP, then \( \text{eval}_{\text{COSY}(E)}(x) = \text{eval}_{\text{COSY}(E)}(x) \)
b) If \( a = ve(S_{i_1}, \ldots, S_{i_k}) \) then we set \([eval_{\text{COSY}}(E)(x)]_j = [eval_{\text{COSY}}(E)(x)]_j + 1 \) for \( j \in \{i_1, \ldots, i_k\} \) and \([eval_{\text{COSY}}(E)(y)]_j = [eval_{\text{COSY}}(E)(x)]_j \) otherwise.

c) If \( a = pe(S_{i_1}, \ldots, S_{i_k}, S_{i_{k+1}}, \ldots, S_{i_{k+1}}) \), then we set

\[
[eval_{\text{COSY}}(E)(x)]_j = [eval_{\text{COSY}}(E)(x)]_j - 1 \quad \text{for } j \in \{i_1, \ldots, i_k\} \text{ and } \]

\[
[eval_{\text{COSY}}(E)(x)]_j = [eval_{\text{COSY}}(E)(x)]_j \quad \text{otherwise}.
\]

It is important to note that \( eval_{\text{COSY}}(E) \) is well defined, which is not immediately apparent, since some of the elements of \( \text{Vops}(\text{COSY}(E)) \) commute. To see that it is well defined, consider \( x \in \text{VFS}(\text{COSY}(E)) \) and look at \([x]_{i_{k+1}} \), \( i \in \{1, \ldots, n\} \). This is the coordinate corresponding to the semaphore \( S_i \). If \( \text{nopes} \) is the number of operations \( pe(\ldots, S_{i}, \ldots) \) and \( \text{noves} \) is the number of operations \( ve(\ldots, S_{i}, \ldots) \) occurring in \([x]_{i_{k+1}} \), then it may be shown that \([eval_{\text{COSY}}(E)(x)]_1 = \text{nove} - \text{nopes} \).

We may now state

**Theorem**

With the above terminology \( \text{VFS}(E) = \text{VFS}(\text{COSY}(E)) \) and \( \text{eval}_E = \text{eval}_{\text{COSY}}(E) \)

**Proof (sketch)**

We have \( x \in \text{VFS}(E) \cap \text{VFS}(\text{COSY}(E)) \) and \( \text{eval}_E(x) = \text{eval}_{\text{COSY}}(E)(x) \) by definition. A comparison of the definition of \( \text{VFS}(E) \) with that of \( \text{eval}_{\text{COSY}}(E) \) shows that if \( x \in \text{VFS}(E) \cap \text{VFS}(\text{COSY}(E)) \) with \( \text{eval}_E(x) = \text{eval}_{\text{COSY}}(E)(x) \) then for all \( y \in \text{Vops}(\text{COSY}(E)), x, x \in \text{VFS}(E) \leftrightarrow x, x \in \text{VFS}(\text{COSY}(E)) \) and that \( \text{eval}_E(xa) = \text{eval}_{\text{COSY}}(E)(xa) \).

From these remarks the theorem follows by induction on the length of vector firing sequences.

Note that this result shows that the translation 'preserves concurrency'. It also permits one to use concepts, developed in terms of the COSY formalism, in connection with ESP programs. For example, we may now formally define an ESP program \( E \) to be deadlock-free iff \( \forall x \in \text{VFS}(E) \exists x \in \text{VFS}(E) : x, x \in \text{VFS}(E) \). A definition as succinct as possible without such a semantics and we have not only given bounded ESP programs such a semantics, but have shown that they may be reformulated in such a way as to be susceptible to treatment by the formal theory associated with COSY programs [Shields 78, Lauer 78a].

5. Extensions to unbounded semaphores

We have so far dealt only with the case in which the values that semaphores in an ESP program \( E \) may take are bounded, that is, the case in which the set \( \text{Val}(E) = \{ [eval_E(x)]_i \mid x \in \text{VFS}(E) \wedge i \in \{1, \ldots, n\} \} \) is finite. Programs \( E \) for which \( \text{Val}(E) \) is infinite (e.g., for which \( 0 \in \text{Val}(E) \) and \( i \in \text{Val}(E) \Rightarrow i + 1 \in \text{Val}(E) \) which implies of course that \( \text{Val}(E) \) is the set of natural numbers) are clearly unimplementable on anything other than a Turing machine and are therefore useless from a practical point of view. Indeed, an attempt to implement such a program could well lead to a systems
failure; the problem of boundedness is as important, in such a context, as the
problem of deadlock or starvation. Strangely enough, this subject is not discussed
in [Agerwala 71], instead the author talks about a property he calls ‘completeness’;
a mechanism is complete if it can simulate the action of an arbitrary Turing machine.
ESPs are complete, in this sense; that is, it is possible to write programs using this
mechanism which are incapable of being implemented. Such programs are precisely those
for which there is no possible translation into COSY, as it stands, which seems to
argue well for the notation, but out of theoretical interest, we here intimate how
the notation might be extended to deal with semaphores in the general, unbounded case.

To obtain a complete translation of a given ESP program, which may contain unbounded
semaphores, requires a real extension of the descriptive power of the COSY notation.
COSY may only describe finite systems. In order to describe infinite counters, one
would need to be able to write things like
(8) \text{path } P(\overline{s}), [(v(\overline{s}); [1]; p(s))]^* [1, \infty] \text{ and }

which ‘intuitively’ defines the behaviour of infinite counters with a test for zero;
it is the ‘path’ \( P(S,O,\infty) \).

Unfortunately, this is not something that could be generated by the production rules
PR nor could it be expanded using the definition of the expansion of a replicator
expression. It is not a path in a strict sense. It is a kind of least upper bound ‘lub’
of the ‘sequence’ \( P(S,O,n) \), \( n = 1, 2, \ldots \). To put it another way, the expression to
the right of the comma in (8) is a ‘fixed point’ of the function: \( F(s) = (v(s); s; p(s))^* \).
It is likely that the words ‘lub’, ‘sequence’ and ‘fixed point’ could be made formally
meaningful, say by defining an ordering relation on the finite paths and then introducing
a set \( \text{PATH} \) of equivalence classes of directed sets of finite paths, much as
in the construction of the reals from the rationals, hoping that the result would be
a domain with the finite paths as finite elements, in which fixed-point results hold.
The ordering relation should be such that \( PCQ \), \( P \) and \( Q \) finite, implies \( FS(P) \subseteq FS(Q) \),
as in the case of \( P(S,O,n) \) and \( P(S,O,m) \) with \( n \leq m \). If \( OPS \) denotes the set of operations
then we would have a function \( FS: \text{PATH} \times OPS^* \), where \( \text{PATH} \) is the set of all
finite paths. Regarding \( 2^\text{OPS} \) as being partially ordered with respect to inclusion,
one could then define for a directed set of paths \( \{P_1, P_2, \ldots\} = X \), \( FS(\cup X) = \text{lub } FS(P_i) \),
which, with the assumption that \( PCQ \Rightarrow FS(P) \subseteq FS(Q) \), would ensure that \( FS: \text{PATH} \times 2^\text{OPS} \)
agrees with \( FS \) on \( \text{PATH} \) and is continuous.

The purpose of these statements is to justify the following definition. Suppose \( n \)
is some integer. We define
\[
\text{Cyc}(P(S,n,\infty)) \text{ to be } \lim_{i \to \infty} \text{Cyc}(P(S,n,i))
\]

With this definition, we have given a ‘meaning’ to \( P(S,n,\infty) \). We may now carry out
the construction of a COSY program for every ESP program as in section 4, but starting
with paths \( P(S_1,m_1,\infty) \) as opposed to paths \( P(S_1,m_1,n_1) \). The justification for the
correctness of the translation in the finite case may be extended to deal with the
infinite case.

6. Summary

We have introduced the reader to the COSY notation and one of its accompanying semantic formalisms. Other semantic formalisms such as Petri Nets and firing sequences were introduced in [Lauer 75] and [Lauer 78a]. We have indicated how such a formalism can be used to formalise the meaning of a synchronisation mechanism such as the ESPs. The COSY notation is a way to program which makes the vector firing sequence semantics quite explicit whereas such a semantics is very implicit in programs involving specific synchronisation mechanisms. Hence, we indicated how a translation from programs involving mechanisms into COSY programs not only makes the concurrent semantics of the former explicit but immediately makes formal results applicable for a deep study of properties of programs using such mechanisms.

ACKNOWLEDGEMENTS

The research reported in this paper was supported by a grant from the Science Research Council of Great Britain. We would like to express our gratitude to Mrs. J. Armstrong for her patience and efficiency in preparing the manuscript.

REFERENCES


