Abstract:

The notion of system dossier is introduced as a framework for organising information about a system in a way which unifies several complementary but differing viewpoints supported by formalisms and automated procedures with the help of which one is gathering and analysing this information. The main part of the paper consists of three dossiers, the first is a dossier on the dossier concept itself, the second a dossier on the COSY notation and formal semantics for specifying and analysing concurrent and distributed systems, and the last is a dossier on a classical synchronisation problem called the Smoker's problem.

Computer System Dossiers

By

P.E. Lauer

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick

Number 185
June, 1983

© 1983 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.
The notion of system dossier is introduced as a framework for organising information about a system in a way which unifies several complementary but differing viewpoints supported by formalisms and automated procedures with the help of which one is gathering and analysing this information. The main part of the paper consists of three dossiers, the first is a dossier on the dossier concept itself, the second a dossier on the COSY notation and formal semantics for specifying and analysing concurrent and distributed systems, and the last is a dossier on a classical synchronisation problem called the Smoker's problem.
## CONTENTS

1 INTRODUCTION .................................................. 1

2 THE DOSSIER ABOUT THE DOSSIER CONCEPT .................. 4

3 THE DOSSIER ABOUT BASIC COSY SYSTEMS ...................... 12

4 A COSY SMOKERS DOSSIER .................................... 29

5 ACKNOWLEDGEMENTS .......................................... 49

6 BIBLIOGRAPHY .................................................. 50
1 INTRODUCTION

Concurrent systems are more difficult to specify and analyse than sequential ones, because they require the conceptualisation not only of their sequential subsystems, but also of the complex interactions between them. It follows that the programmer's intuition is not enough, being unreliable in cases of high complexity. Here solution of the problem of verification of correct behaviour of the design becomes crucial, and a satisfactory conceptual apparatus for rigorous verification becomes essential.

In such an apparatus, reduction of complexity, by abstracting away from all irrelevant detail specific to some implementation of a concurrent and distributed system strategy, is desirable. But it must be easy to obtain implementations from the abstract specification of a strategy, given enough information about the synchronisation mechanisms of the concrete system on which it is to be implemented. We therefore need a notion of "system" sufficiently abstract to allow analysis only of those aspects of systems arising from their concurrency and yet capable of being readily translated into practical terms.

We have developed such a conceptual apparatus called COSY (from concurrent systems) to a great level of sophistication. The results we have obtained are outlined below and are treated in detail in the references. Analysis based on the COSY formalism often involves extensive mechanical transformations of programs and their corresponding semantical objects, a laborious process which inhibits thorough exploration of more than one or two alternative designs. Judicious use of the computer for performing these mechanical tasks would thus greatly extend the practical utility of the method.

Some of the uses one can make of the COSY formalism are:

1. Specification: To specify the synchronisation aspects of systems of congreable (cooperating) concurrent behaviours.

2. Abstraction: To abstract to the synchronisation aspects of programs (Implementation). That is, rather than expressing "how" a set of concurrent behaviours is enforced by means of the synchronisational elements of a program, one expresses "what" the set of synchronised concurrent behaviours is.

3. Implementation: To translate a specification of synchronised concurrent behaviours into a program involving synchronisation primitives in such a way as to enforce the behaviour specified when the program is executed. That is, we have the inverse of abstraction, rather than expressing "what" the set of synchronised concurrent behaviours is, we are to express "how" a set of synchronised concurrent behaviours is enforced by the synchronisational elements of a program of some implemented programming language.

4. Verification:
   a. Implementation (program) satisfies specification.
b. Specification is implemented by program.

5. **Transformation**: To translate one specification into another which is more or less concurrent and or distributed but defines the same interleaved or pseudo concurrent behaviours.

6. **Comparison**: To compare two different programs implementing some synchronisation strategy by comparing their corresponding abstract specifications in COSY. Comparisons may be made with regard to their degree of concurrency and distribution. However, the abstractions must preserve concurrency and distribution.

7. **Evaluation**: To evaluate programming languages with respect to their ability to support concurrency and distribution. Hence, specification1 may be transformed to a more concurrent and or distributed specification2, but it may not be possible to implement it by means of a program2 which has the same degree of concurrency and distribution.

8. **Realisation**: To obtain a means of "executing" specifications by either:
   a. direct implementation of specification in VLSI; or
   b. implementation in the sense of point 3 above; or
   c. combining the specification grammar with the grammar for a more conventional algorithmic programming language.

Full system COSY notation including the distinction between specification and implementation was developed to support the processes of abstraction and implementation (programming, coding, translating). **Verification** requires that there exist formal methods for determining the "meaning" of both the specification part and the implementation part. If there is a formal semantics associated with the implementation part then its meaning can be determined and must be shown to determine the same behaviours in some sense as the specification. We have developed a suite of computer programs which constitute a system design and analysis environment for basic COSY called BCS. This environment is intended to reduce the mechanical labour of manipulating COSY specifications of example systems during the various uses mentioned above.

Given the BCS environment it is possible to produce a lot of information about a system formulated in COSY and it becomes necessary to devise a means for organising this information in a way which facilitates the analysis of the system formulated and allows ready comparison between different system designs. For this purpose, we introduced the notion of a **system dossier** as a framework for organising one's information about a system. This notion is particularly helpful if one has several complementary but differing viewpoints supported by formalisms and automated procedures with the help of which one is gathering and analysing this information.

In the process of developing the dossier notion it became apparent that it would be presumptuous if we fixed the possible viewpoints
constituting a dossier a priori. Hence, the dossier notation was introduced as an extendable notation. In this notation it is possible to introduce new viewpoints provided each viewpoint is adequately defined particularly with respect to its interface with other viewpoints which have already been defined.

Furthermore, since the dossier is a rather general notion it should not only be applicable to systems developed in COSY but should also serve as a means for describing the COSY formalism itself, the BCS environment and even the dossier concept itself.

Finally, we envisage the dossier notation as eventually being implemented and if it were then "compiled" this would mean, for example:

1. certain communication paths are established between, for example, the user and maintainer or designer of the system;

2. interconnections between suite components are made which allow, for instance:

   a. output of one component to become input to another component;

   b. systems descriptions written in a combination of notations from different components, e.g. Algo68 programs and COSY specifications, to be interpreted using the semantics of the components involved. It might even be possible to run a program written, e.g. in a combination of Algo68 and Concurrent Pascal, by linking the respective compilers in appropriate ways.

Hence, the dossier concept is intended to combine aspects of a programming language construct, an operating system construct, a communication system construct and even a text processing construct.

Our present formulation of the concept is only tentative and works tolerably well for our purposes in this paper, but we hope that a more definitive concept will emerge eventually from an interaction of interested colleagues in the field. Hence, we invite the reader to communicate to us any criticisms and suggestions.

Our purpose in the present paper will be to introduce the dossier concept and give it an informal meaning by applying it to the dossier concept in section 2, to the COSY formalism and BCS environment in section 3, and finally to a familiar synchronisation problem and its solution in section 4.
A dossier is an accumulation of records, reports, miscellaneous pertinent data, and documents bearing on a single subject of study or investigation.

A suite is a series or group of things forming a unit or constituting a complement or collection.

With regard to computing systems (hardware, software and human components) the dossier concept is intended as a means for:

a. Organising
   1) information about the system;
   2) communication paths between the designers, developers, maintainers and users of the system;

b. Integrating the computational facilities, their documentation, the communication system and the general text processing facilities.

It is hoped that such a reorganization and integration will significantly increase:

a. the overall effectiveness of the system and the individuals involved with it both as users and service personnel;

b. the effectiveness of communication between users and maintainers of a system;

c. the responsiveness of the system to queries, complaints and needs of the users;

d. the control system personnel have over how they want users to inform them about the behaviour of the system as a result of their use;

e. the control managers and project leaders have over the form and emphasis of research into and development of systems.

By a suite in the narrower sense we shall mean a complementary set of viewpoints supported by software, hardware and personnel who are responsible for the system and its documentation.
syntax
DN1: <dossier>=dossier dossier_name
        <viewpoint>+
        enddossier

DN2: <viewpoint>=<dossier> | <description> | <specification> | <syntax> | <program> | <definition> | <semantics> | <verification> | <relation> | <intermediary> | <model>

DN3: <description>=
        description descriptionname
        enddescription

DN4: <specification>=
        specification specificationname
        endspecification

DN5: <syntax>=
        syntax syntaxname
        endsyntax

DN6: <program>=
        program programme name
        endprogram

DN7: <definition>=
        definition definitionname
        enddefinition

DN8: <semantics>=
        semantics semanticsname
        endsemantics

DN9: <verification>=
        verification verification name
        endverification

DN9: <relation>=
        relation relationname
        endrelation

DN10: <intermediary>=
        intermediary intermediaryname
        endintermediary

DN11: <model>=
        model modelname
        endmodel

endsyntax

5
semantics
Note: For each of the viewpoints we indicate:

1. rules of composition of dossier text;
   a. manual
   b. automatic

2. suite component.

General form of a COSY dossier

dossier pure_COSY_dossier

description descriptionname
   Here one writes intuitive descriptions of the
   system that is to be specified, implemented,
   verified, etc.
enddescription

specification specificationname
   Here one can write an abstract behavioural system
   specification in the Pure COSY notation
endspecification

definition definitionname
   Here one can write a definition of the concurrent behaviour
   corresponding to the specification or to the implementation
   using the Vector Firing Semantics notation [LS81a and b].
   This part may also contain certain kinds of verification as
   for example a proof that the behaviours displayed
   are indeed complete in the sense that no possible
   behaviour has been omitted.
enddefinition

verification verificationname
   Here we can write rigorous proofs or exhaustive
   simulations using the Vector Firing Sequence Model [LS81a and b]
   verifying that the system specified is deadlock
   or starvation free, or that two implementations
   implement the same specification, or that two
   specifications are inequivalent with respect to
   degree of concurrency and distribution though they
   can both be considered to "compute" the same
   relation from input to output, etc.
endverification

enddossier
dossier applied_COSY_dossier

description
descriptionname
description
enddescription

specification
specificationname
endspecification

definition
definitionname
enddefinition

dossier application_to_program

description
descriptionname
enddescription

specification
implementation
Here one can write an implementation of the abstract
specification using specific synchronisation primitives
like semaphores, fork and join, monitors, etc.,
in the shared memory approach; or communication
primitives like the input and output commands
in Communicating Sequential Processes [Ho80], in the
non-shared memory approach.
endspecification

definition
definitionname
enddefinition

enddossier

verification
verificationname
endverification

enddossier
Examples of general dossier patterns

```
dossier    dossier1

description descriptionname
enddescription

syntax syntaxname
endsyntax

semantics semanticsname
endsemantics

program programname
endprogram

enddossier

verification verificationname
endverification

enddossier
```

Dossier pattern dossier1 has the form of the basic COSY system dossier of section 3. The abstract COSY notation is described, its syntax given and its semantics defined. Dossier1.1 will then be a complete dossier on the BCS environment mentioned in the introduction. The verification part will demonstrate that the BCS environment indeed implements the semantics defined in the semantics part of the basic COSY system dossier.
Dossier pattern dossier2 has the form of a pure COSY dossier, which means it is not related to any programming language involving synchronisation primitives. In fact dossier2.1 could be a problematic system specification and dossier2.2 a specification of a solution of the problems of the first system. The verification part would then contain a proof that the system of dossier2.2 is indeed a solution of the problems of the system of dossier2.1.
dossier
dossier1

description descriptionname
enddescription

specification specificationname
endspecification

definition definitionname
enddefinition

dossier
dossier1.1

description descriptionname
enddescription

specification specificationname
endspecification

definition definitionname
enddefinition

enddossier

verification verificationname
endverification

enddossier

dossier
dossier2

description descriptionname
enddescription

specification specificationname
endspecification

definition definitionname
enddefinition

dossier
dossier2.1

description descriptionname
enddescription

specification specificationname
endspecification

definition definitionname
enddefinition

enddossier

verification verificationname
Dossier pattern dossier3 has the form of the smokers dossier in the third chapter of this paper and it is an example of an applied COSY dossier. Again, as in the previous pattern, dossier3.1 is a problematic system and dossier3.2 is a corresponding unproblematic system. The outermost verification would be a demonstration that the latter solves the problems of the former. Now, however, the sub dossiers 3.1.1 and 3.2.1 are statements of the respective systems in a programming language involving binary semaphores as synchronisation primitives, assignment statements, loops, compound statements and arrays as data structures. The respective inner verification parts would show that the respective specifications are correct abstractions of the behaviours enforced by the semaphores in the respective programs.

We will not give any formal semantics for our COSY dossier but only indicate the above patterns in the present chapter. The two subsequent chapters contain fragments of applied COSY dossiers and they will serve as implicit definitions of the semantics intended.
A basic COSY system is a collection of cyclic, sequential and non-deterministic subsystems consisting of elementary events (also called actions, operations, procedures, etc.). Each event is capable of giving rise to a number (possibly zero) of occurrences of that event during any period of (discrete) behaviour of the system.

The basis (also called set of operations, or alphabet) of a system is the set of all events considered to constitute the system.

A trace (also called firing sequence) of a sequential subsystem is any, possibly empty, sequence of event occurrences which constitutes the history of an actual or potential sequential behaviour of the subsystem from some initial event occurrence, and to some final event occurrence in the finite case.

A sequential subsystem is completely determined by the set of all its traces. From the definition of trace it follows that for any sequential system:

1. the empty trace is a trace of the system, and
2. every prefix of a trace is a trace.

The behaviour of a sequential subsystem is usually defined in terms of a set of successful initial traces (also called cycles previously) by stating that all traces of the system are all possible prefixes of multiples of successful initial traces. In the case of a cyclic sequential system we identify the notion of successful initial trace with the notion of a trace whose corresponding sequence of event occurrences returns the system to the point which preceded the first occurrence of any of that system's constituent events, or briefly, in the sequential and centralized case, what would be called its initial state.

A vector (also called vector firing sequences) of congreable traces of a concurrent system consisting of n sequential subsystems is an n-place vector whose i-th component is a trace of the i-th subsystem for all i in the range 1 to n, and in which the traces of all subsystems agree about the number and order of occurrences of events they share. In other words, a vector of traces of a concurrent system is any vector of (finite, possibly empty) sequences of event occurrences which constitutes the history of some actual or potential concurrent behaviour of the system from some (possibly concurrent) event occurrences, and in the finite case, up to some (possibly
concurrent) event occurrences. The $i$-th subhistory of the vector of histories is a history of an actual or potential sequential behaviour of the $i$-th sequential subsystem from some event occurrence, and to some event occurrence in the finite case. But in addition, if several subsystems involve the same event then their respective traces have to agree about the number and order of occurrences of the event in which they coincide. Hence we talk about vectors of congruable traces.

A concurrent system is completely determined by the set of its vectors of congruable traces. Again it follows from the definition of vector of traces that:

1. the vector of empty traces is a concurrent history of the system
2. every prefix (generalized to vectors of traces) of a vector of traces of the system is a vector of traces of the system.

More generally, our semantic model divides a view of a concurrent system into three aspects:

1. the individual view of the sequential subsystems independent of their combination into a single concurrent system, i.e. the traces of the subsystems considered independently of whether they share events;
2. the common view of the combined concurrent system which all sequential subsystems must share if they are to constitute a single system, independent of their view of their own sequential history, i.e. the vectors of "traces" which agree on the number and order of occurrences of events subsystems share disregarding the order prescribed by the traces of the individual subsystems;
3. the combined view of the whole system in which both views (1) and (2) coincide.

In the application of the COSY formalism, (notation and semantic model), one will want to be able to start from any one of these views and obtain any of the other views from them while developing ones conceptualisation of the system. Furthermore, one will want to derive specifications to correspond to systems which have been designed by means of the semantic model alone.

enddescription

taxonomy

A basic COSY specification is a string derived from the production rules given below. The following meta-language conventions have been used in the syntax rules: The symbols
"=". "(". ")". "/". "+". "+@" are used as meta-symbols. The symbol "=" denotes replacement of its left hand side by the string on its right hand side. The braces "{ }" are used to group items together, "/" indicates alternate productions, "{item}+" indicates production of "item" zero or more times, "{item}+" production of "item" one or more times. The notation

{item1 @ item2}+

is used as a shorthand for

item1 {item2 item1}*

In the syntax rules for basic COSY specifications "item2" may be one of the terminal symbols ";" and ",". Non-underlined lower case words, except single lower case letters and digits, are non-terminal symbols, and all other symbols like ";","", "+", "+@", "+", underlined lower case words and single lower case letters and digits are terminal symbols. We shall additionally use the following convention: in right parts of production rules the concatenation of terminals and non-terminals has precedence over alternation. Thus A B/C means either A B or C. When necessary we use "{ }" to override the normal precedence. Thus A {B/C} means either A B or A C.

The syntax of a basic COSY specification is given by the following rules:

BN1. basicspecification = specification body endspecification
BN2. body = {path/process}+
BN3. path = path sequence end
BN4. process = process sequence end
BN5. sequence = {orelement @;}+
BN6. orelement = {starelement @,+}
BN7. starelement = element/element*
BN8. element = operation/(sequence)
BN9. operation = simple-op/subscr-op
BN10. simple-op = letter(letter/digit/ )*
BN11. subscr-op = simple-op({integer @}+)
BN12. letter = a/b/.../Z/A/B/.../Z
BN13. digit = 0/1/.../9
BN14. integer = {digit}+

endsyntax

semantics

In the regular expressions produced by the non-terminal "sequence" the symbols ";" and "," denote sequentialization and arbitrary choice respectively; the symbol "*" is the Kleene star, which denotes zero or more repetitions.

All the regular expressions in paths and processes are considered to be cyclic in the sense that constituent operations may be executed repeatedly subject to the constraints of sequentialization and arbitrary choice. For
this reason the outermost star and parentheses are always omitted, their presence being implicit. The semantics of a basic path $P$ are given in terms of its set of firing sequences denoted by $\text{FS}(P)$. The infinite set $\text{FS}(P)$ is constructed from a set consisting of the cycles of $P$ by the function "Cyc". The function "Cyc" applies to syntactic components of basic paths, that is to say substrings produced by non-terminals. Syntactic components of paths are denoted by syntactic variables. A path $P$ is represented by

$$\text{path SEQ end}$$

where $\text{SEQ}$ denotes a sequence, which is represented by

$$\text{OREL}_1; \ldots; \text{OREL}_n$$

where $\text{OREL}_i$ for $i=1, \ldots, n$ denote orelements. An orelement is represented by

$$\text{STAREL}_1, \ldots, \text{STAREL}_n$$

where $\text{STAREL}_i$ for $i=1, \ldots, n$ denote starelements. A starelement is represented by

$$\text{ELEM}^* \text{ or ELEM}$$

where $\text{ELEM}$ denotes an element which is represented by

$$(\text{SEQ})$$

when it is produced by the second option of the syntax rule for element BNS, or by

$$\text{OP}$$

when produced by the first option. The function "Cyc" is defined as follows:

$$\text{Cyc}(e) = \begin{cases} \text{SEQ} & \text{when } e \text{ is a path} \\ \text{OREL}_1; \ldots; \text{OREL}_n & \text{when } e \text{ is an orelement sequence} \\ \text{STAREL}_1, \ldots, \text{STAREL}_n & \text{when } e \text{ is a starelement sequence} \\ \text{ELEM}^* \text{ or ELEM} & \text{when } e \text{ is an element} \\ \text{SEQ} & \text{when } e \text{ is a sequence} \\ \text{OP} & \text{when } e \text{ is an operation} \end{cases}$$

In the above definition of "Cyc" the symbol "U" denotes the set-union operator and the symbol "*" the concatenation of sets of strings operator. The operation

$$X * Y$$
where $X, Y$ of sets of strings is defined as:

$$X \cup Y = \{x, y | x \in X, y \in Y\}$$

where "." denotes string concatenation and "," element of a set.

In the definition of "Cyc" a starred set $X^*$ indicates the set obtained by concatenation of zero or more times of elements of the set $X$. Formally $X^*$ is defined by

$$X^* = X^0 \cup X^1 \cup X^2 \cup \ldots$$

where $X$ is a set of strings and $X^i$ is defined recursively by

$$X^i = X^{i-1} \cdot X$$

where "," denotes the empty string.

From the set Cyc($P$) we construct the set of firing sequences of $P$ denoted by FS($P$) as follows:

$$FS(P) = \text{Pref}(\text{Cyc}(P)^*)$$

where Pref($X$) is defined as

$$\text{Pref}(X) = \{x | x.y \in X, \text{ for some } y\}$$

where $X$ is a set of strings.

The set FS($P$) is the set of sequences of operation executions permitted by the path $P$. As already mentioned, to model the non-sequential behaviour of a basic specification $R$ consisting of paths $P_1, \ldots, P_n$ partial orders of occurrences of operations will be constructed which are represented by vectors of strings. An $n$-vector $x$

$$x = (x_1, \ldots, x_n)$$

is a possible behaviour of $R$ if each $x_i$ for $1 \leq i \leq n$ is a possible firing sequence of $P_i$ for $i=1, \ldots, n$ and furthermore, if the $x_i$'s agree on the number and the order of occurrences of operations they share.

To formally define the set of possible behaviours or histories of $R$, vectors of strings are introduced together with a composition operation on them. Let $S_1, \ldots, S_n$ be a family of sets of strings and let

$$X_{S_1^*} \times X \ldots X S_n^* = \{ (s_1, \ldots, s_n) | \text{for all } i, s_i \in S_i^* \}$$

$$i=1$$

where "$X" denotes the cross product operator. If the vectors $x$ and $y$ belong to the above set then their composition $x * y$ is
defined as
\[ x \sim y = (x_1, \ldots, x_n) \sim (y_1, \ldots, y_n) = (x_1 y_1, \ldots, x_n y_n) \]
where "\sim" denotes the vector concatenation operation.

To each specification \( R \) consisting exclusively of paths \( R = P_1 \ldots P_n \)
we associate its set of operations \( \text{Ops}(R) \) defined by
\[ \text{Ops}(R) = \text{Ops}(P_1) \cup \ldots \cup \text{Ops}(P_n) \]
and its set of vector operations \( \text{Vops}(R) \) defined as follows:
For each operation "a" in \( R \) we construct an \( n \)-vector \( a \). The \( i \)th component of this vector for \( 1 \leq i \leq n \) denoted by \( [a]_i \) is given by
\[
[a]_i = \begin{cases} 
  a & \text{if } a \in \text{Ops}(P_i) \\
  e & \text{otherwise}
\end{cases}
\]
where "e" denotes the null string. The set of vector operations of \( R \), \( \text{Vops}(R) \) is then defined as
\[ \text{Vops}(R) = \{ a | a \in \text{Ops}(R) \} \]

Let us define \( \text{Vops}(R)^* \) to be the submonoid of
\[ \prod_{i=1}^{n} \text{Ops}(P_i)^* \]
by \( \text{Vops}(R) \) and \( e = (e, \ldots, e) \) under the vector composition operation. The set of all possible behaviours or histories of \( R \), the vector firing sequences of \( R \), denoted by \( \text{VFS}(R) \) is defined by:
\[ \text{VFS}(R) = ( \prod_{i=1}^{n} \text{FS}(P_i) ) \cap \text{Vops}(R)^* \]
The set
\[ \prod_{i=1}^{n} \text{FS}(P_i) \]
in the definition of \( \text{VFS}(R) \) guarantees that each string component of a history \( x \in \text{VFS}(R) \) is a firing sequence of the corresponding path, i.e. it represents the individual view, and the set \( \text{Vops}(R)^* \) guarantees that all these firing sequences agree on the number and order of activations of the operations they share, i.e. it represents the collective view. \( \text{VFS}(R) \) of
course represents the combined view.

By the construction of VFS(R) each of its elements \( x \) represents everything that has happened in some possible period of activity of \( R \). We may write \( x \) as a composition of vector operations \( a_1, \ldots, a_m \) of Vops(R) as in (V1)

\[(V1) \quad x = a_1^{\ldots} a_m^{\ldots} \]

Consequently, for every \( x = (x_1, \ldots, x_n) \in \text{Vops}(\mathbb{R})^* \), the symbol \([x]_i \) denotes the string \( x_i \), i.e., \([x]_i = x_i \), for \( i = 1, \ldots, n \). If for some operations "a_k" and "a_l" for \( 1 \leq k, l \leq m \) and \( k \neq l \), \([a_k]_i = e \) implies \([a_l]_i = e \) for \( i = 1, \ldots, n \) then the composition \( a_k e a_l \) is the same as \( a_l e a_k \). Such operations are said to be independent and we write \( \text{ind}(a_k, a_l) \). If furthermore \( l = k + 1 \) that is \( a_k \) and \( a_l \) are neighbouring vectors in (V1), as in (V2)

\[(V2) \quad x = a_1^{\ldots} a_k^{\ldots} a_l^{\ldots} a_m^{\ldots} \]

then \( x \) may also be written as (V3)

\[(V3) \quad x = a_1^{\ldots} a_l^{\ldots} a_k^{\ldots} a_m^{\ldots} \]

The commutativity of vector operations in a vector firing sequence is interpreted to mean that the operations corresponding to these vector operations may execute concurrently. We say that two operations "a" and "b" are concurrent at a history \( x \) and we write

\[
\begin{align*}
\text{a co b at x}
\end{align*}
\]

if \( \text{ind}(a, b) \) and \( x^a, x^b \in \text{VFS}(R) \). This definition implies that only independent operations may execute concurrently. However, independent operations may not always execute concurrently or may never execute concurrently at all.

For the construction of the vector firing sequences of a basic specification \( R \), the following sets need to be constructed directly from \( R \):

1. the cycle sets of all paths in \( R \), and
2. the set of the vector operations in \( R \), Vops(R).

In general, a basic specification \( R \) is a string of the form

\[
R = P_1 \ldots P_n \ Q_1 \ldots Q_m
\]

where \( P_j \) for \( j = 1, \ldots, n \) and \( Q_i \) for \( i = 1, \ldots, m \) denote paths and processes respectively. Although paths and processes may be intermixed in a basic specification, in the above expressions for convenience, we assumed that all paths are collected before processes.
The semantics of a basic specification involving processes is given by means of the vector firing sequences of an equivalent basic specification \( R' \) involving just paths. The conversion of \( R \) into \( R' \) is denoted by \( \text{Path}(R) \) and is obtained by the following rule: (Path Conversion Rule)

1. For every \( a \in \text{Ops}(R) \) construct a set
   \[ \text{Ia} = \{ i | a \in \text{Ops}(Q_i) \text{ for } 1 \leq i \leq m \} \]
   and, if the cardinality of the set \( \text{Ia} \) denoted by \( |\text{Ia}| \) is greater than zero, say \( l = |\text{Ia}| > 0 \) then
   - replace the operation "a" in each path it occurs by the element
     \[ (a; \text{Ia}, \ldots, a; \text{Ia}) \]
     where \( i_k \in \text{Ia} \) for \( k = 1, \ldots, l \)
   - replace the operation "a" in processes \( Q_{i_k} \) by \( a; \text{Ia} \) for all \( i_k \in \text{Ia} \).

2. Replace all occurrences of "process" by "path".

Then the semantics of \( R \) are given by means of \( \text{VFS}(\text{Path}(R)) \) and are obtained as defined in the previous section. As we have mentioned, a basic COSY specification describes a system by specifying partial orders on the execution of its operations and therefore, the only properties of interest are behavioural in nature.

The formal model of behaviour, the vector firing sequences of path-specifications permit us to speak formally of dynamic properties of a system specified by a path-specification \( R \). All properties of \( R \) may be expressed in terms of its corresponding vector firing sequences \( \text{VFS}(R) \). Such properties fall into two classes, the general and the specific properties.

The general properties are those which apply to any specification, properties such as absence of deadlock or starvation, which may be defined in terms of uninterpreted operations. We say that a path-specification \( R \) is deadlock-free if and only if
\[ \forall x \in \text{VFS}(R) \exists a \in \text{Ops}(R): x^{-a} \in \text{VFS}(R) \]
that is, if and only if every history \( x \) may be continued. We say that a specification \( R \) is adequate if and only if
\[ \forall x \in \text{VFS}(R) \forall a \in \text{Ops}(R) \exists y \in \text{Vops}(R)^*: x^{-y}a \in \text{VFS}(R) \]
that is, if and only if every history of \( R \) may be continued enabling eventually every operation in \( R \). Adequacy is a property akin to absence of partial system deadlock (see also [579 and B82]).
The relation \( \rightarrow^* \text{Vops}(R) \times \text{Vops}(R) \), defined as:
\[
x \rightarrow^* y : \iff x \in \text{VFS}(R) \land
\begin{align*}
& (\forall i \in [1, \ldots, n]) \ (\forall j \in [1, \ldots, k]) \ (V_i \neq j \to \text{ind}(a_i, a_j)) \\
& (\forall i \in [1, \ldots, n]) \ (\forall j \in [1, \ldots, k]) \ (V_i = j \to \text{ind}(a_i, a_j))
\end{align*}
\]
is called the concurrent reachability in one step [LJ82].

It can be proved that \( \text{VFS}(R) = \{ x : x \rightarrow^* x \} \). For every \( x \in \text{VFS}(R) \), let \( \text{enabled}(x) \) be the following family of action names:
\[
\text{enabled}(x) = \{ a_1, \ldots, a_k : \{ a_1, \ldots, a_k \} \subseteq \text{Ops}(R) \land
\begin{align*}
x & \rightarrow^* x. a_1. \ldots. a_k
\end{align*}
\]
and let \( \text{maxenabled}(x) \) be the family of all maximal elements contained in \( \text{enabled}(x) \), i.e.
\[
\text{maxenabled}(x) = \{ A : \forall \text{enabled}(x) : (A \in \text{enabled}(x) \land A \neq B) \rightarrow A = B \}.
\]
And finally, let \( \rightarrow^* \text{Vops}(R) \times \text{Vops}(R) \) be the relation given by:
\[
x \rightarrow^* y : \iff (\forall i \in [1, \ldots, n]) \ (\forall j \in [1, \ldots, k]) \ (a_i \in \text{maxenabled}(x) \land a_j = x. a_1. \ldots. a_k)
\]
The relation \( \rightarrow^* \) is called the maximally concurrent reachability in one step. For more details on the last notions the reader is referred to [LJ82].

The specific properties involve the interpretation of a COSY specification as a description of an actual system. The operations of a COSY specification are interpreted as actions of a system and the behaviour of the specification as the behaviour of the system.

Considerable work has been done concerning the verification of general properties of specifications and in particular relating to adequacy [SL78, S79, LS80] and a number of general theorems have been obtained [S79]. For simple comma-free path specifications there is a complete characterization for adequacy. Other theorems have been obtained which permit certain specification transformations which preserve adequacy.

As far as specific properties of specifications are concerned, various specifications have been shown to satisfy some design requirements. The most involved of these is the parallel resource releasing mechanism [SL80a, SL80b and LS81b].

description

The basic COSY system BCS is a program which emphasises the analysis and simulated execution of concurrent systems. It is coded in SIMULA, and is of modular design to localise as much as
possible those parts which are unavoidably specific to the host operating system. From the abstract viewpoint, BCS is nothing but the implementation of the relation $\rightarrow$. It is just a device which produces the relation $\rightarrow$ for a given path program $\mathbf{R}$ (see [LJ82]). The program has the following principal functional procedures, or processors:

a. Compile, which inputs a source specification written in the basic COSY syntax, producing an object file which is used as input by other processors. The object file includes the original text, and also contains the parsing tree in a compact form, and tables of operation names with cross-references to paths and processes.

b. Recompile also produces an object file, but instead of a source input, it uses an object file as input, usually after editing, and so provides a recycling capability which makes the process of refining a COSY specification much more efficient than the conventional approach of editing source code.

c. Fire generates the firing sequences (cyclic histories) of individual paths and processes under interactive user control.

d. Simulate is the most powerful of the processors, being a mechanism for stepping a simulation of the concurrent system, described in COSY notation, through all its permitted histories.

endsyntax

Here we put the syntax of the BCS command language.

endsyntax

semantics

BCS has been written in SIMULA 67, and runs on the Newcastle IBM 370 computer under the Michigan Terminal System (MTS). By December 1981, as we then reported in more detail [LH81], BCS had been developed to the stage where three principal functions were in useful working order:

Compilation BCS will accept a source specification, written in the basic COSY notation, which describes a distributed system comprising an arbitrary number of sequential, cyclic, and nondeterministic processes, and from that source construct a parse-tree representation of the distributed system.

The tree structure is then written out in a compact object language to a file, from which the tree may be readily reconstructed by the other functions of BCS.

Generation of Firing Cycles The Firing Sequence Generator (FSG) function of BCS operates on the individual paths or processes of a compiled COSY specification, without regard to the synchronisation restraints imposed by the coincidence of operations in different paths. The FSG is mainly useful for checking the semantics of more
complicated paths which are otherwise difficult to read, such as the path

\[
\text{path } 5p, (2p; (2p; lp), (lp; 2p, (lp; lp)), (lp; (2p; 2p, (lp; lp)), (lp; (2p; lp), (lp; 2p, (lp; lp)))) \text{ end}
\]

which represents the ways in which coins (halfpennies excluded) can be loaded into a machine to a total of 5p. The path may be shown by the FSM to have the nine firing cycles

1) 5p
2) 2p.2p.lp
3) 2p.lp.2p
4) 2p.lp.lp.lp
5) lp.2p.2p
6) lp.2p.lp.lp
7) lp.lp.2p.lp
8) lp.lp.lp.2p
9) lp.lp.lp.lp.lp

Indeed such an output suggests that the original path be replaced by the more readable

\[
\text{path } 5p, (2p; 2p; lp), (2p; lp; 2p), (2p; lp; lp), (1p; 2p; 2p), (1p; 2p; lp; lp), (lp; 1p; 2p; lp), (lp; 1p; lp; 2p), (lp; 1p; lp; lp; lp) \text{ end}
\]

but such a representation is not without disadvantages. There are now nine distinct operations from which only one may be chosen to begin a cycle of operations permitted by the path, but they are no longer uniquely labelled, e.g. choosing to fire the operation labelled "lp" does not define which of the five distinct cycles beginning with a "lp" has in fact been chosen, even in the context of the history of operations fired prior to the choice. The first representation does not lead to such ambiguity.

**Simulation** The Simulation function of BCS provides what is effectively a COSY machine, a mechanism for the simulated execution of a system of distributed sequential processes.

The mechanism realises the Vector Firing Sequence semantics of a COSY specification thus:

- For each sequential subsystem (path and processes) there is a pointer to the last operation fired (a null pointer if no operation has yet fired).
- There is a variable set of operations which are enabled, that is, permitted to fire by the constraints implied by all the path expressions comprising the COSY specification. (A path or process which contains no operation in the enabled set is blocked, and when the enabled set is empty, the simulated system is totally deadlocked. Starvation arises when certain operations fail to become enabled).
- From the enabled set of operations (while that set is not empty) some may be selected to "fire". A firing operation leaves the enabled set (taking with it all those operations which are subject to mutual exclusion with it), and becomes the "last operation fired" in each of the sequential subsystems in which it occurs. All the paths and processes containing that fired operation now define new (local) sets of potentially enabled operations, and those operations now permitted to fire by all the paths citing them join the enabled set.

- Each operation fired is stored (effectively as a vector operation), in a sequential history list.

Driving the simulation of a COSY specification may be done, at present, in two ways.

a. The current set of enabled operations is displayed as a list of operation names at the terminal, from which the user may select one or more (depending on the mutual exclusion relations among the operations). After the selected operations have been "fired" and recorded, the new enabled set is displayed, and the manual selection made again.

b. The user may choose a number of simulation steps to be run automatically. At each step, one operation is selected at random from the current enabled set. Recording of vector firing sequences is done as when in the manual selection mode.

Playback of the simulated history, which may be accumulated after arbitrary alternation between the two modes of driving, can be in two (so far) styles.

a. The "packed" style shows the history of each sequential path as a simple string of operation names, and does not emphasise the synchronisation of operations occurring in several paths at once.

b. The "padded" or "vector" style of replay appears as a concatenation of column vectors. Such a display is generally more bulky than the packed display but provides a very nice illustration of the meaning of concurrency of operations, as the following example shows.

Consider the following COSY specification, which describes a five-frame buffer subject to first-in-first-out discipline. The first five paths relate to the frames individually, which may alternately deposit and remove, and the last two paths enforce the cyclic and sequential discipline on deposits and removes respectively.
Using the simulator in manual selection mode, one could step the model through five deposits, followed by five removes, leading to a history which can be written as a sequence of vector operations:

(H1) \[ \overline{d1,d2,d3,d4,d5} - \overline{r1,r2,r3,r4,r5} \]

which in the packed style would replay thus:

(H2)

\[
\begin{align*}
P1 & : d1.r1 \\
P2 & : d2.r2 \\
P3 & : d3.r3 \\
P4 & : d4.r4 \\
P5 & : d5.r5 \\
P6 & : d1.d2.d3.d4.d5 \\
P7 & : r1.r2.r3.r4.r5
\end{align*}
\]

But in the vector style the history appears as:

(H3)

\[
\begin{align*}
P1 & : \overline{d1..r1..r1..} \\
P2 & : \overline{d2..r2..r2..} \\
P3 & : \overline{d3..r3..r3..} \\
P4 & : \overline{d4..r4..r4..} \\
P5 & : \overline{d5..r5..r5..} \\
P6 & : \overline{d1.d2.d3.d4.d5} \\
P7 & : \overline{r1.r2.r2.r4.r5}
\end{align*}
\]

and now the meaning of a history of vector operations becomes visible. Where two vector operations can move horizontally in the display without collision, that is when they have no non-empty components in the same row, the vectors commute and the corresponding operations are concurrent. We can see, for instance, that \( r1 \) thus commutes with \( d2,d3,d4 \) and \( d5 \) (but not with \( d1,r2,r3,r4, \) and \( r5 \)). Because of this commutativity, the vector history (H1) does not imply that the operation \( r1 \) occurred after the operations \( d2,d3,d4 \) and \( d5 \).

The simulator can be used to discover deadlocks in a distributed system, and a prolonged run under random selection can also reveal starvations, but as yet it cannot be used (in the general case) to prove automatically that a system is deadlock free. However we expect the simulator mechanism to be the basis of more refined analytic procedures in future work, as described in [LH81]. Indeed, as it stands, the simulator can be used as an aid to manual
implementation of the proof rules on the (restricted) class of specifications for which such rules exist.

Modelling, Analysing and Developing Specifications. Examples above and and in the more detailed memo on BCS [Lj81] illustrate how the semantic routines of BCS may be used to:

a. clarify one's understanding of a specification leading to a possible reformulation which is more readable;

b. reliably obtain particular histories of system behaviour, either by terminal dialogue or automatically, and display them in various forms;

c. semantically correct a specification for which one has obtained a history by (2), for example by transforming the specification to exclude a deadlock (cf. [LBS77] p. 324;

d. verify correctness of a specification by

1) using a theorem (cf. [L81] p. 11) which says that if there exists a generalised cyclic history in the combined view of the system then it involves no partial system deadlock.

2) using a theorem (cf. [SL78] p. 22) which says when a specification may be reduced to a simpler one with the same deadlock properties by removing parts of the specification which could not contribute to deadlock. If a given specification reduces to the empty specification, the original specification allows no partial system deadlock.

e. analyse and develop the system from any of the three viewpoints, individual, collective, and combined.

f. analyse the concurrency of the system which can be made explicit by using the vector operation representation of a history (see above).

Obviously these approaches may all be combined in the analysis and development of a system. For example, one can use the above method (b) to reduce a specification not to the empty specification, which may not be possible, but to a collection of disjoint sub-specifications each of which can be verified by method (a).

endsemantics

verification

Problem of maximally concurrent simulation.

By maximally concurrent simulation, we mean a simulation where we always perform concurrently as many actions as possible. It turns out that if this kind of simulation is enough to describe the complete system behaviour, then the whole verification procedure is easier and less labour-consuming.
Let us start with the following lemma:

**Lemma**: \( n, m \) independent actions and concurrent at some point:

\[ n, m = m, n \]

**Proof**: depends on argument based on:

i) their independence, i.e. firing of one cannot disable the other,

ii) neither can make use of any choices which might arise at an invisible intermediate point.

This lemma tells us that it suffices to request independent and simultaneously enabled actions be done in one step by writing them in one order to obtain all traces obtainable by writing them in some other order.

Unfortunately, the following fact is also true:

**Fact**: Doing two independent and simultaneously enabled actions in one step, that is, without making use of choices that might arise after one or the other of the actions has been completed, is not the same as doing them in either order but sequentially, that is, in two steps.

**Proof**: the specification below and the accompanying trace generated by taking all possible choices at least once demonstrate the non-equivalence.

```
specification
   P1: path a,c,d end
   P2: path b,c,d end
endspecification
```

\[
\text{Enabled} : \begin{align*}
& l = a \rightarrow 4 = b \\
& = 1, 4 \\
& \text{Enabled} : 3 = d \\
& = 3 \\
\end{align*}
\]

\[
\text{Enabled} : \begin{align*}
& l = a \rightarrow 4 = b \\
& = 1 \\
& \text{Enabled} : 4 = b \\
& = 4 \\
& \text{Enabled} : 3 = d \\
& = 3 \\
\end{align*}
\]

\[
\text{Enabled} : \begin{align*}
& l = a \rightarrow 4 = b \\
& = 4 \\
& \text{Enabled} : * l = a \rightarrow * 2 = c \\
& = 1 \\
& \text{Enabled} : 3 = d \\
& = 3 \\
\end{align*}
\]
Enabled: \( l = a \quad 4 = b \)
\( = 4 \)
Enabled: \( * l = a \quad * 2 = c \)
\( = 2 \)
Enabled: \( 4 = b \)
\( = 4 \)
Enabled: \( 3 = d \)
\( = 3 \)

\[ P_1 : a \quad .d.a. \quad .d. \quad .a.d. \quad .c \quad .d \]
\[ P_2 : \quad .b.d. \quad .b.d.b. \quad .d.b.c.b.d \]

The problem sketched above is precisely studied in [LJ82]. Let us recall some basic results. For every specification \( R = P_1 \ldots P_n \) (if \( R \) contains processes we consider \( \text{Path}(R) \)), let:

\[
\text{VMFS}(R) = \{ x : e \rightarrow^{*} x \}.
\]

\( \text{VMFS}(R) \) is simply the set of all vectors of firing sequences produced by maximally concurrent performances. \( R \) is said to be completely characterised by maximally concurrent simulation if and only if

\[
\text{VFS}(R) = \text{Pref}(\text{VMFS}(R)),
\]

where here \( \text{Pref}(X) \) is defined as:

\[
\text{Pref}(X) = \{ x : x.y \in X \text{ for some } y \}
\]

The fact stated above shows that there is an \( R \) such that \( \text{VFS}(R) \neq \text{Pref}(\text{VMFS}(R)) \).

Let \( \text{pre} \), \( \text{exc} \), \( \text{con} \) \( \text{Ops}(R) \times \text{Ops}(R) \) be the following relations:

\[
(a, b) \in \text{pre} : \Longleftrightarrow (4i)(\exists x \in \text{FS}(P_i)) x.a.b \in \text{FS}(P_i),
\]

\[
(a, b) \in \text{exc} : \Longleftrightarrow (4i)(\exists x \in \text{FS}(P_i)) x.a \in \text{FS}(P_i) \land x.b \in \text{FS}(P_i),
\]

\[
(a, b) \in \text{con} : \Longleftrightarrow \text{ind}(a, b) \land (\exists x \in \text{VFS}(R)) x.a.b \in \text{VFS}(R).
\]

Let \( \text{PDT}(R) \) \( \text{Ops}(R) \times \text{Ops}(R) \times \text{Ops}(r) \) be the relation defined as follows:

\[
(a, b, c) \in \text{PDT}(R) : \Longleftrightarrow (a, b) \in \text{pre} \cup (b, c) \in \text{exc} \cup (a, c) \in \text{con}.
\]

The relation \( \text{PDT}(R) \) is called potentially dangerous triples. In the case of the example considered in the proof of the Fact above we have:

\[
(b, c, a) \in \text{PDT}(R).
\]

In [LJ82] the following theorem is proved:

If \( \text{PDT}(R) = \{ \} \) then \( \text{VFS}(R) = \text{Pref}(\text{VMFS}(R)) \),

where \( \{ \} \) denotes the empty set.
In other words, if there are no potentially dangerous triples then maximally concurrent simulation alone is enough to describe the complete system behaviour.

endverification

program

Here goes a copy of the SIMULA program BCS#24

endprogram

doodnoier
doodnoier
dossier Smokers

description smoking_analogy

Three smokers are sitting at a table. One of them has tobacco, another has cigarette papers, and the third one has matches; each one has a different ingredient required to roll, light and smoke a cigarette but he may not give any ingredient to another. On the table in front of them, two of the three ingredients will be placed. No new set of ingredients will be placed on the table until some smoker has successfully smoked. Smokers can only pick up ingredients that are on the table sequentially, one after the other. Smokers cannot return ingredients to the table if they discover that the third required ingredient for them to smoke has not been placed on the table. Hence, once the wrong smoker picks up an ingredient which another smoker could successfully use, no smoker can successfully smoke and the smokers are deadlocked.

description

specification semaphore_smoking_analogy

P1: path tobaccoV; tobaccoP end
P2: path paperV; paperP end
P3: path matchV; matchP end
P4: path smokertV; smokertP end
P5: path smokersV; smokersP end
P6: path smokermV; smokermP end
P7: path sP; sV end
Q1: process sP; paperV; matchV end
Q2: process sP; tobaccoV; paperV end
Q3: process sP; matchV; tobaccoV end
Q4: process smokertP; sV end
Q5: process smokersP; sV end
Q6: process smokermP; sV end
Q7: process paperP; matchP; smokertV end
Q8: process tobaccoP; paperP; smokersV end
Q9: process matchP; tobaccoP; smokersP end

description

endspecification
**Definition: Semaphore Smoking Analogy**

The vector operations of the problematic smokers

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
</tr>
</thead>
</table>

**Note:** The table represents vector operations where each vector is either `P` or `Q`, and the operations involve `paperV`, `tobaccoV`, `smokeV`, etc. Each row corresponds to a different vector operation, with `V` indicating a vector and `P` or `Q` indicating the type of vector operation.
The Periodic Vector Firing Sequences of the Problematic Smokers

problem dialogue pdl:

Enabled : *19=sp&1 *20=sp&2 *21=sp&3
   =19
Enabled : 5=paperV&1
   =5
Enabled : 7=paperP&7 9=matchV&1
   =7,9
Enabled : *11=matchP&7 *12=matchP&9
   =11
Enabled : 13=smokertV&7
   =13
Enabled : 14=smokertP&4
   =14
Enabled : 22=sV&4
   =22
Enabled : *19=sp&1 *20=sp&2 *21=sp&3

problem vector firing sequence pvfs1 "packed":

P1 : 
P2 : paperV&1.paperP&7
P3 : matchV&1.matchP&7
P4 : smokertV&7.smokertP&4
P5 : 
P6 : 
P7 : sp&1.sV&4
Q1 : sp&1.paperV&1.matchV&1
Q2 : 
Q3 : 
Q4 : smokertP&4.sV&4
Q5 : 
Q6 : 
Q7 : paperP&7.matchP&7.smokertV&7
Q8 : 
Q9 : 

problem vector firing sequence pvfs1 "padded":

P1 : 
P2 : .paperV.paperP. 
P3 : .matchV.matchP. 
P4 : .smokertV.smokertP. 
P5 : 
P6 : 
P7 : sp. 
Q1 : sp.paperV. .matchV. .sV
Q2 : 
Q3 : 
Q4 : .matchP.smokertV. 
Q5 : .smokertP.sV
Q6 : 
Q7 : .paperP. 
Q8 : 
Q9 : 

31
problem vector firing sequence pvfs1 "as is":

P1 : . . . . . . . . 
P2 : .paperV.paperP. . . . . . 
P3 : . . matchV.matchP. . . . . 
P4 : . . . . . . smokertV.smokertP. . 
P5 : . . . . . . . . 
P6 : . . . . . . . . 
P7 : sP. . . . . . . . sV. 
Q1 : sP.paperV.matchV. . . . . 
Q2 : . . . . . . . . 
Q3 : . . . . . . . . 
Q4 : . . . . . . . . smokertP.sV. 
Q5 : . . . . . . . . 
Q6 : . . . . . . . . 
Q7 : . . . . . . paperP.matchP.smokertV. . 
Q8 : . . . . . . . . 
Q9 : . . . . . . . . 

problem dialogue pd2:

Enabled : *19=sP&1 *20=sP&2 *21=sP&3 =20
Enabled : 1=tobaccoV&2 =1
Enabled : 3=tobaccoP&8 6=paperV&2 =3,6
Enabled : *7=paperP&7 *8=paperP&8 =8
Enabled : 17=smokermV&8 =17
Enabled : 18=smokermP&6 =18
Enabled : 24=sV&6 =24
Enabled : *19=sP&1 *20=sP&2 *21=sP&3

problem vector firing sequence pvfs2 "padded":

P1 : .tobaccoV.tobaccoP. . . . . . . 
P2 : . . . . . . paperV.paperP. . . 
P3 : . . . . . . . . 
P4 : . . . . . . . . 
P5 : . . . . . . . . 
P6 : . . . . . . . . smokermV.smokermP. 
P7 : sP. . . . . . . . sV 
Q1 : . . . . . . . . 
Q2 : sP.tobaccoV. .paperV. . . . . 
Q3 : . . . . . . . . 
Q4 : . . . . . . . . 
Q5 : . . . . . . . . 
Q6 : . . . . . . . . smokermP.sV 
Q7 : . . . . . . . . 
Q8 : . . . . . . . . tobaccoP. paperP.smokermV. 
Q9 : . . . . . . . . 
problem dialogue pd3:

Enabled : *19=sP&1 *20=sP&2 *21=sP&3
=21
Enabled : 10=matchV&3
=10
Enabled : 2=tobaccoV&3 12=matchP&9
=2,12
Enabled : *3=tobaccoP&8 *4=tobaccoP&9
=4
Enabled : 15=smokerP&9
=15
Enabled : 16=smokerP&5
=16
Enabled : 23=sV&5
=23
Enabled : *19=sP&1 *20=sP&2 *21=sP&3

problem vector firing sequence pvfs3:

P1 : . tobaccoV. tobaccoP. . .
P2 : . . . . .
P3 : . matchV. matchP. . .
P4 : . . . . .
P5 : . . . . smokerP. smokerP.
P6 : . . . . .
P7 : sP. . . . .
Q1 : . . . . .
Q2 : . . . . .
Q3 : sP.matchV.tobaccoV. . . .
Q4 : . . . . .
Q5 : . . . . smokerP.sV
Q6 : . . . . .
Q7 : . . . . .
Q8 : . . . . .
Q9 : . . . matchP.tobaccoP.smokerP.

The non-periodic vector firing sequences of the problematic smokers

problem dialogue pd4:

Enabled : *19=sP&1 *20=sP&2 *21=sP&3
=19
Enabled : 5=paperV&1
=5
Enabled : 7=paperP&7 9=matchV&1
=7,9
Enabled : *11=matchP&7 *12=matchP&9
=12
System deadlocked!
problem vector firing sequence pvfs4 "packed":

P1 : <R>
P4 : <R>
P5 : <R>
P6 : <R>
P7 : sP&1.<R>
Q1 : sP&1.paperV&1.matchV&1.<R>
Q2 : <R>
Q3 : <R>
Q4 : <R>
Q5 : <R>
Q6 : <R>
Q7 : paperP&7.<R>
Q8 : <R>
Q9 : matchP&9.<R>

problem vector firing sequence pvfs4 "padded":

P1 : . . . . . . . . . <R>
P2 : .paperV.paperP. . . . <R>
P3 : . . matchV.matchP.<R>
P4 : . . . . . . . . . <R>
P5 : . . . . . . . . . <R>
P6 : . . . . . . . . . <R>
P7 : sP. . . . . . . . . <R>
Q1 : sP.paperV. .matchV. .<R>
Q2 : . . . . . . . . . <R>
Q3 : . . . . . . . . . <R>
Q4 : . . . . . . . . . <R>
Q5 : . . . . . . . . . <R>
Q6 : . . . . . . . . . <R>
Q7 : . . paperP. . . . . <R>
Q8 : . . . . . . . . . <R>
Q9 : . . . . . . . . . <R>

problem dialogue pd5:

Enabled : *19=sP&1 *20=sP&2 *21=sP&3
=20
Enabled : 1=tobaccoV&2
=1
Enabled : 3=tobaccoP&8 6=paperV&2
=3,6
Enabled : *7=paperP&7 *8=paperP&8
=7
System deadlocked!
problem vector firing sequence pvfs5:

P1 : .tobaccoV.tobaccoP. . . <R>
P2 : . . . paperV.paperP. <R>
P3 : . . . . <R>
P4 : . . . . <R>
P5 : . . . . <R>
P6 : . . . . <R>
P7 : sP. . . . <R>
Q1 : . . . . <R>
Q2 : sP.tobaccoV. . paperV. . <R>
Q3 : . . . . <R>
Q4 : . . . . <R>
Q5 : . . . . <R>
Q6 : . . . . <R>
Q7 : . . . . . paperP. <R>
Q8 : . . . . . tobaccoP. <R>
Q9 : . . . . . <R>

problem dialogue pd6

Enabled : *19=sP&1 *20=sP&2 *21=sP&3
=21
Enabled : 10=matchV&3
=10
Enabled : 2=tobaccoV&3 12=matchP&9
=2,12
Enabled : *3=tobaccoP&8 *4=tobaccoP&9
=3

System deadlocked!

problem vector firing sequence pvfs6:

P1 : . . . tobaccoV. . . tobaccoP. <R>
P2 : . . . . . . . <R>
P3 : . . . matchV. . . matchP. . <R>
P4 : . . . . . . . <R>
P5 : . . . . . . . <R>
P6 : . . . . . . . <R>
P7 : sP. . . . . . . <R>
Q1 : . . . . . . . <R>
Q2 : . . . . . . . <R>
Q3 : sP.matchV.tobaccoV. . . . . . <R>
Q4 : . . . . . . . <R>
Q5 : . . . . . . . <R>
Q6 : . . . . . . . <R>
Q7 : . . . . . . . <R>
Q8 : . . . . . . . tobaccoP. <R>
Q9 : . . . . . . . matchP. <R>
In the two definitions of the smokers problem given above we have exhibited six vector firing sequences and we have implicitly claimed that these six vector firing sequences completely define all possible behaviours of the system of problematic smokers. We now prove that our implicit assumption is justified. One can easily prove by inspection that the interesting part of the relations con and exc are the following:

![Diagram of relations con and exc]

The relation pre is much more complicated but it is not necessary to consider the whole relation pre. Let us consider

\{a: (a, paperP7) \in pre or (paperP7, a) \in pre\}

and denote it by $\text{pre(paperP7)}$. It is very easy to show by inspection (see the form of the process Q7), that:

$\text{pre(paperP7)} = \text{smokerV&7 paperP7 matchP7}$

Now we have:

\[(\text{paperP7, matchP7}) \in \text{pre} \land (\text{matchP7, matchP9}) \in \text{exc but (paperP7, MatchP9)} \notin \text{con};\]

and

\[(\text{smokerV&7, paperP7}) \in \text{pre} \land (\text{paperP7, paperP9}) \in \text{exc but (smokerV&7, paperP9)} \notin \text{con};\]

thus there is no potentially dangerous triple containing the action paperP7.
In a similar way we may prove that the remaining actions from the domain of con, i.e., matchV&1, tobaccoP&8, paperV&2, matchP&9, tobaccoV&3, do not belong to any potentially dangerous triples. But this means that there is no potentially dangerous triple so by the theorem we obtain that the maximally concurrent simulation given above completely characterises the whole behaviour of the problematic smokers.

enddefinition

dossier Patil's problem

description Patil's problem statement

To inform the smokers about the ingredients which are placed on the table, three binary semaphores representing tobacco, paper and matches are provided. On placing the ingredients on the table the corresponding semaphore is Incremented by performing a V[] operation. On the smoker's side semaphores smokert, smokerp and smokerm are used to signal that a cigarette has been smoked. The smoker who completes smoking a cigarette performs the operation V[] on the corresponding semaphore. [Pat71]

enddescription
<table>
<thead>
<tr>
<th>Specification</th>
<th>Patil's problematic smokers</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(tobacco)</td>
<td>R(paper)</td>
</tr>
<tr>
<td>rt: (P(s))</td>
<td>rp: (P(s))</td>
</tr>
<tr>
<td>V(\text{paper})</td>
<td>V(\text{tobacco})</td>
</tr>
<tr>
<td>V(\text{match})</td>
<td>V(\text{match})</td>
</tr>
<tr>
<td>goto (rt)</td>
<td>goto (rp)</td>
</tr>
<tr>
<td>(bt: (P(\text{smokert})))</td>
<td>(bp: (P(\text{smokerp})))</td>
</tr>
<tr>
<td>V(\text{s})</td>
<td>V(\text{s})</td>
</tr>
<tr>
<td>goto (bt)</td>
<td>goto (bp)</td>
</tr>
<tr>
<td>(\text{smokert})</td>
<td>(\text{smokerp})</td>
</tr>
<tr>
<td>(\text{smokerm})</td>
<td>(\text{smokerm})</td>
</tr>
<tr>
<td>at: (P(\text{paper}))</td>
<td>ap: (P(\text{tobacco}))</td>
</tr>
<tr>
<td>(\text{P(match)})</td>
<td>(\text{P(match)})</td>
</tr>
<tr>
<td>(\text{V(smokert)})</td>
<td>(\text{V(smokerp)})</td>
</tr>
<tr>
<td>goto (at)</td>
<td>goto (ap)</td>
</tr>
</tbody>
</table>

**-- initially: \(s=1\), \(\text{tobacco}=\text{paper}=\text{match}=\text{smokert}=\text{smokerp}=\text{smokerm}=0\)**

**stands for some operation performed by the process**

**endspecification**

**definition**

A binary semaphore is a variable whose value can only be 0 or 1. Instruction \(P[s]\) decrements the value of the semaphore \(s\) by 1 and if the value of the semaphore is 0, the execution of the instruction is held up until it becomes 1. This provides a way to allow a process to wait until other processes catch up. Furthermore, if the value of the semaphore is 1 and several processes try to execute \(P[s]\) then only one of the processes is permitted to complete execution of \(P[s]\) and others are held waiting for the value of the semaphore to become 1. The instruction \(V[s]\) just increments the value of the semaphore by 1.

Go to statements to simple lables are used to indicate cyclicity of the processes. All the processes are cyclic.

**enddefinition**

**enddossier**

**verification**

abstraction correctness

Here we need to verify that the specification semaphore smoking analogy in some more abstract sense is equivalent to the specification Patil's problematic smokers. In the paper [LS81a] we have shown how to do this formally, but here we will only state that:

(1) A binary semaphore \(s=1\) initially, an exclusion semaphore, can be represented by path \(sP; sV\) end.
(2) A binary semaphore \(s=0\) initially, a sequencing semaphore, can be represented by path \(sV; sP\) end.
(3) Cyclic processes consisting of sequences of semaphore instructions can be directly modelled by COSY processes.
Note: There is a slight permutation of the sequential actions of the smoker with paper "smokerp". It seemed to us that two of the smokers, namely, smokerP and smokerM should not both start with the same initial action, since this makes the possible interactions between the three processes unnecessarily unsymmetric. So though the original deadlock discussed above is no longer possible, it is represented by three distinct deadlocks in the transformed specification.

end verification
end dossier
dossier smokers solution
description description name

Three processes, the pushers, are added to the problematic smokers which pick up the ingredients from the table and they pass the ingredients to the appropriate smoker and then signal him to smoke.

description

specification Janicki Parnas solution
P1:path tobaccoV;tobaccoP end
P2:path paperV;paperP end
P3:path matchV;matchP end
P4:path smokerV;smokerP end
P5:path smokerpV;smokerpP end
P6:path smokerM;smokerM end
P7:path sP;sV end
P8:path mutexP;mutexV end
P9:path VS(1);PS(1) end
P10:path VS(2);PS(2) end
P11:path VS(3);PS(3) end
P12:path VS(4);PS(4) end
P13:path VS(5);PS(5) end
P14:path VS(6);PS(6) end
P15:path (VS(1);VS(3),VS(5),(VS(2);VS(3),VS(6)), (VS(4);VS(5),VS(6)) end
Q1:process sP;paperV;matchV end
Q2:process sP;tobaccoV;paperV end
Q3:process sP;matchV;tobaccoV end
Q4:process smokerP;sV end
Q5:process smokerpP;sV end
Q6:process smokerM;sV end
Q7:process tobaccoP;mutexP;VS(1),VS(3),VS(5);mutexV end
Q8:process paperP;mutexP;VS(2),VS(3),VS(6);mutexV end
Q9:process matchP;mutexP;VS(4),VS(5),VS(6);mutexV end
Q10:process PS(6);smokerV end
Q11:process PS(3);smokerM end
Q12:process PS(5);smokerpV end
Q13:process PS(1) end
Q14:process PS(2) end
Q15:process PS(4) end
endspecification
specification  Campbell Lauer solution
P1: path tobaccoV; tobaccoP end
P2: path paperV; paperP end
P3: path matchV; matchP end
P4: path smokertV; smokertP end
P5: path smokerP; smokerP end
P6: path smokermV; smokermP end
P7: path sP; sV end
P8: path (tobaccoP; ((paperP; smokermV), (matchP; smokerP))),
    (paperP; ((tobaccoP; smokermV), (matchP; smokertV))),
    (matchP; ((tobaccoP; smokerP), (paperP; smokertV))) end
Q1: process sP; paperV; matchV end
Q2: process sP; tobaccoV; paperV end
Q3: process sP; matchV; tobaccoV end
Q4: process smokertP; sV end
Q5: process smokerP; sV end
Q6: process smokermP; sV end
Q7: process smokertV end
Q8: process smokermV end
Q9: process smokerP end
Q10: process tobaccoP end
Q11: process paperP end
Q12: process matchP end
endspecification

specification  minimal change solution
P1: path tobaccoV; tobaccoP end
P2: path paperV; paperP end
P3: path matchV; matchP end
P4: path smokertV; smokertP end
P5: path smokerP; smokerP end
P6: path smokermV; smokermP end
P7: path sP; sV end
P8: path (tobaccoV; ((paperV; smokermB), (matchV; smokerB))),
    (paperV; ((tobaccoV; smokermB), (matchV; smokertB))),
    (matchV; ((tobaccoV; smokerB), (paperV; smokertB))) end
Q1: process sP; paperV; matchV end
Q2: process sP; tobaccoV; paperV end
Q3: process sP; matchV; tobaccoV end
Q4: process smokertP; sV end
Q5: process smokerP; sV end
Q6: process smokermP; sV end
Q7: process smokertB; paperP; matchP; smokertV end
Q8: process smokermB; tobaccoP; paperP; smokermV end
Q9: process smokerB; matchP; tobaccoP; smokerV end
endspecification
specification Janicki_Patil_solution
path tobaccoV;tobaccoP end
path paperV;paperP end
path matchV;matchP end
path smokertV;smokertP end
path smokerP;smokerP end
path smokerV;smokerV end
path sP;sV end
path VS(1);PS(1) end
path VS(2);PS(2) end
path VS(3);PS(3) end
path VS(4);PS(4) end
path VS(5);PS(5) end
path VS(6);PS(6) end
path (xEQO;yGTO;zGTO), (yEQO;xGTO;zGTO), (zEQO;xGTO;yGTO) end
process sP;paperV;matchV end
process sP;tobaccoV;paperV end
process sP;matchV;tobaccoV end
process smokertP;sV end
process smokerP;sV end
process smokerV;sV end
process PS(3);PS(1);xGTO,(xEQO;VS(5);VS(6);smokertV) end
process PS(2);PS(5);yGTO,(yEQO;VS(3);VS(4);smokerP) end
process PS(4);PS(6);zGTO,(zEQO;VS(1);VS(2);smokerV) end
process tobaccoP;VS(5);VS(6) end
process paperP;VS(3);VS(4) end
process matchP;VS(1);VS(2) end
endspecification

definition Campbell_Lauer_solution

<table>
<thead>
<tr>
<th>The Vector</th>
<th>Operations of the Adequate Smokers</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 :</td>
<td></td>
</tr>
<tr>
<td>P2 :</td>
<td>paperV,paperV,paperP</td>
</tr>
<tr>
<td>P3 :</td>
<td></td>
</tr>
<tr>
<td>P4 :</td>
<td></td>
</tr>
<tr>
<td>P5 :</td>
<td></td>
</tr>
<tr>
<td>P6 :</td>
<td></td>
</tr>
<tr>
<td>P7 : sP.sP.sP.sV.sV.sV.</td>
<td></td>
</tr>
<tr>
<td>P8 :</td>
<td>paperP</td>
</tr>
<tr>
<td>Q1 : sP.</td>
<td>paperV</td>
</tr>
<tr>
<td>Q2 : sP.</td>
<td>paperV</td>
</tr>
<tr>
<td>Q3 : sP.</td>
<td>paperV</td>
</tr>
<tr>
<td>Q4 : sV.</td>
<td>paperV</td>
</tr>
<tr>
<td>Q5 : sV.</td>
<td></td>
</tr>
<tr>
<td>Q6 : sV.</td>
<td></td>
</tr>
<tr>
<td>Q7 :</td>
<td></td>
</tr>
<tr>
<td>Q8 :</td>
<td></td>
</tr>
<tr>
<td>Q9 :</td>
<td></td>
</tr>
<tr>
<td>Q10 :</td>
<td></td>
</tr>
<tr>
<td>Q11 :</td>
<td>paperP</td>
</tr>
<tr>
<td>Q12 :</td>
<td></td>
</tr>
</tbody>
</table>

41
<table>
<thead>
<tr>
<th></th>
<th>tobaccoV</th>
<th>tobaccoV</th>
<th>tobaccoP</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>matchV</td>
<td>matchV</td>
<td>matchP</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td></td>
<td></td>
<td>tobaccoP</td>
</tr>
<tr>
<td>Q1</td>
<td>matchV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td></td>
<td>tobaccoV</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td></td>
<td></td>
<td>tobaccoV</td>
</tr>
<tr>
<td>Q4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q10</td>
<td></td>
<td></td>
<td>tobaccoP</td>
</tr>
<tr>
<td>Q11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Periodic Vector Firing Sequences of the Adequate Smokers

Here also there are no potentially dangerous triples so we may restrict our considerations to maximally concurrent simulations alone.

solution dialogue sdl

Enabled : *16=sP&1 *17=sP&2 *18=sP&3
=16
Enabled : 4=paperV&1
=4
Enabled : 6=paperP&11 7=matchV&1
=6,7
Enabled : 9=matchP&12
=9
Enabled : 10=smokertV&7
=10
Enabled : 11=smokertP&4
=11
Enabled : 19=sV&4
=19
Enabled : *16=sP&1 *17=sP&2 *18=sP&3

solution vector firing sequence svfsl:

P1 : 

P2 : .paperV.paperP.

P3 : .matchV.matchP.

P4 : .smokertV.smokertP.

P5 : 

P6 : 

P7 : sP.

P8 : .paperP. .matchP.smokertV.

Q1 : sP.paperV. .matchV.

Q2 : 

Q3 : 

Q4 : .smokertP.sV

Q5 : 

Q6 : 

Q7 : .smokertV.

Q8 : 

Q9 : 

Q10 : 

Q11 : .paperP.

Q12 : .matchP.
solution dialogue sd2:

Enabled : *16=sP&1 *17=sP&2 *18=sP&3
=17
Enabled : 1=tobaccoV&2
=1
Enabled : 3=tobaccoP&10 5=paperV&2
=3,5
Enabled : 6=paperP&11
=6
Enabled : 14=smokermV&8
=14
Enabled : 15=smokermP&6
=15
Enabled : 21=sV&6
=21
Enabled : *16=sP&1 *17=sP&2 *18=sP&3

solution vector firing sequence svfs2:

P1 : .tobaccoP.tobaccoP.
P2 : . paperV.paperP.
P3 : .
P4 : .
P5 : .
P6 : . . smokermV.smokermP.
P7 : sP.
P8 : . . . .
Q1 : .
Q2 : sP.tobaccoV.
Q3 : . paperV.
Q4 : .
Q5 : .
Q6 : . smokermP.sV
Q7 : .
Q8 : . . smokermV.
Q9 : .
Q10 : . tobaccoP.
Q11 : . paperP.
Q12 : .

44
solution dialogue sd3:
   Enabled : *16=sP61 *17=sP62 *18=sP63
   =18
   Enabled : 8=matchV&3
   =8
   Enabled : 2=tobaccoV&3 9=matchP&12
   =2,9
   Enabled : 3=tobaccoP&10
   =3
   Enabled : 12=smokerP&9
   =12
   Enabled : 13=smokerP&5
   =13
   Enabled : 20=sV&5
   =20
   Enabled : *16=sP61 *17=sP62 *18=sP63

solution vector firing sequence svfs3:

P1 : .tobaccoV. .tobaccoP. .
P2 : .
P3 : .matchV. .matchP. .
P4 : .
P5 : . . . . . .smokerP&.smokerP&.
P6 : .
P7 : sP. . . . . . . . . sV
P8 : . . . . . .matchP&.tobaccoP&.smokerP&.
P9 : .
Q1 : .
Q2 : .
Q3 : sP.matchV&.tobaccoV. .
Q4 : .
Q5 : . . . . . . . . . . . .smokerP&.sV
Q6 : .
Q7 : .
Q8 : .
Q9 : . . . . . . . . . . . .smokerP&.
Q10 : . . . . . . . . . . . .tobaccoP.
Q11 : .
Q12 : . . . . . .matchP. .

By the theorem we have that these are the only vector firing sequences one need consider since all others can be obtained as prefixes of multiples of these three.

enddefinition

dossier Parnas_unproblematic_smokers

description Parnas_description

Simulation of a six branch case statement using a semaphore array, assignment and simple arithmetic operations yield a solution.

enddescription
**specification** Parnas_program

<table>
<thead>
<tr>
<th>RtoC</th>
<th>Rp</th>
<th>Rmatch</th>
<th>the agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>rt: P(s)</td>
<td>rp: P(s)</td>
<td>rm: P(s)</td>
<td></td>
</tr>
<tr>
<td>V(paper)</td>
<td>V(toC)</td>
<td>V(match)</td>
<td></td>
</tr>
<tr>
<td>V(match)</td>
<td>goto rt</td>
<td>goto rp</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>bp</th>
<th>bm</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt: P(smoker)</td>
<td>bp: P(smokerp)</td>
<td>bm: P(smokerm)</td>
</tr>
<tr>
<td>V(s)</td>
<td>goto bt</td>
<td>goto bp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>ap</th>
<th>am</th>
</tr>
</thead>
<tbody>
<tr>
<td>at: P(S[6])</td>
<td>ap: P(S[5])</td>
<td>am: P(S[3])</td>
</tr>
<tr>
<td>t := 0</td>
<td>t := 0</td>
<td>t := 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d</th>
<th>dp</th>
<th>dm</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt: P(tobacco)</td>
<td>dp: P(paper)</td>
<td>dm: P(match)</td>
</tr>
<tr>
<td>P(mutex)</td>
<td>goto dt</td>
<td>goto dp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>d1</th>
<th>d2</th>
<th>d3</th>
</tr>
</thead>
<tbody>
<tr>
<td>dl: P(S[1])</td>
<td>d2: P(S[2])</td>
<td>d3: P(S[4])</td>
</tr>
<tr>
<td>goto dl</td>
<td>goto d2</td>
<td>goto d3</td>
</tr>
</tbody>
</table>

--- initially: s = mutex = 1
- tobacco = paper = match = smoker = smokerp = smokerm = 0
- semaphore array S[1..6] = 0
** stands for some operation performed by the process

**endspecification**

**definition** Parnas_Habermann definition

Here one can insert the five page proof by Nico Habermann [Ha73].

**enddefinition**

**verifcation** abstraction_correctness2

The problem statement from the outer verification of correctness is inherited at this point, but in addition we are required not to use conditional statements. Obviously Parnas_program contains no such conditional statements.

Note:
The problem could be seen as implementing the abstract specification into the concrete one using no conditional statements.

Here we again need to show that the given specifications are in some more abstract sense equivalent to the specification Parnas_program. Applying the translation of abstraction_correctness1 we obtain the intermediary implementation specification. That leaves the translation
of the effects of the assignment statements on the variable t. t is always set to zero before new ingredients are supplied. So the effect of the increments of the three pusher processes can only be:

t=1, if Q10 is first;
after which
t=3, if Q11 is next;

or
t=5, if Q12 is next;

t=2, if Q11 is first; etc.

replacing these constants for t in the intermediary and adding the path P15 which expresses the feedback the various increments have on t, results in the Janicki_Parnas_solution.

Similar indications of abstraction correctness can be given for the other two specifications.

```plaintext
intermediary implementation_specification

P1:path tobaccoV; tobaccoP end
P2:path paperV; paperP end
P3:path matchV; matchP end
P4:path smokertV; smokertP end
P5:path smokerP; smokerP end
P6:path smokerV; smokerV end
P7:path sP; sV end
P8:path mutexP; mutexV end
Q1:process sP; paperV; matchV end
Q2:process sP; tobaccoV; paperV end
Q3:process sP; matchV; tobaccoV end
Q4:process smokertP; sV end
Q5:process smokerP; sV end
Q6:process smokerV; sV end
Q7:process S[6]P; t:=0; smokertV end
Q8:process S[3]P; t:=0; smokerV end
Q9:process S[5]P; t:=0; smokerP end
Q10:process tobaccoP; mutexP; t:=t+1; S[t]V; mutexV end
Q11:process paperP; mutexP; t:=t+2; S[t]V; mutexV end
Q12:process matchP; mutexP; t:=t+4; S[t]V; mutexV end
Q13:process S[1]P end
Q14:process S[2]P end

end_intermediary

end_verification
```
verification correctness_of_solution

Examination of the definition indicates that there are six distinct that is nonequivalent behaviours of the problematic smokers. Three of them lead to deadlock and three of them do not.

Problem: Devise a strategy which will preserve the extendable behaviours of the problematic smokers but eliminate their deadlock behaviours.

Consideration of the vector firing sequences of the Campbell_Lauer_solution shows that if we ignore:

1. the fact that smokers no longer receive the signals about the arrivals of ingredients on the table directly;

2. the firing sequences of the feedback path P8, and the pusher processes Q10-Q12.

we get exactly the periodic vector firing sequences of the problematic smokers.

Abstraction correctness for both patil_problem and parnas_solution implies applied COSY verification of the two "programs".

endverification

enddossier
ACKNOWLEDGEMENTS

I would like to thank Brian Randell and Elizabeth Barraclough for their helpful discussions with me concerning the dossier concept. Brian Hamshere and John Cotronis [C82] must additionally be given credit for having done considerable preparatory work involving the Textform system used to produce this paper and particularly the treatment of the formal notations by means of Textform. Furthermore, Brian Hamshere is responsible for the production of the BCS environment without which much of the contents of the paper could not have been produced as readily. Richard Janicki should be thanked for his inspiring and clarifying influence during the formulation of the ideas presented here. In particular, the formal definition of BCS and the theorem about the full characterisation of a COSY specification by maximally concurrent simulation is due to him. Finally, John Cotronis was involved in development of the formulation and proofs of the Lemma and Fact about concurrent simulation in BCS.

The work reported was supported from grants from the Science and Engineering Council of Great Britain.


[LH81] P E Lauer, B Hamshere :A computer based environment for the design and analysis of highly parallel and distributed
computing systems, ASM/88, Computing Laboratory, University of Newcastle upon Tyne, 1981.


[Par75] D L Parnas: On a solution of the cigarette smoker's problem (without conditional statements); CACM 18, 3, 1975.


[S81] M W Shields: On the non-sequential behaviour of a class of


52