TOPICS IN PROGRAMME SPECIFICATION AND DESIGN:

# SPECIFICATION AND DESIGN OF DISTRIBUTED SYSTEMS

I.H. SORENSEN
WOLFSON COLLEGE

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE UNIVERSITY OF OXFORD, SEPTEMBER 1981.

TOPICS IN PROGRAMME SPECIFICATION AND DESIGN :
**SPECIFICATION AND DESIGN OF DISTRIBUTED SYSTEMS**

Ib Holm Sorensen
Wolfson College

A thesis submitted for the degree of Doctor of Philosophy in the University of Oxford, September 1981.

**ABSTRACT.**

This thesis presents a method for *specifying*, *analysing* and *refining* the designs of *distributed systems*. Distributed systems are systems which consist of several autonomous process components.

The characteristics of the specification method employed in this thesis can be summarised as follows – 1) The structure of a system's specification indicates the structure of the system's realisation, 2) A design is specified entirely in terms of the permissible activity across the interfaces between process components (i.e. the communications); such a specification gives the rules for the behaviour of each process component and postpones decisions about its internal structure, 3) permissible activity is described in terms of predicates on the history of past communications.

This specification method will be shown to allow important questions about the behaviour of a distributed system to be posed early in the design process; in particular designs will be analysed with respect to termination and absence of deadlocks.

The specification method can be employed to describe systems in different degrees of detail, and it is demonstrated that a specification can evolve to a stage close to realisation using a stepwise refinement method which ensures that the important properties are maintained.

**Table of Contents.**

**Acknowledgement.**

I am indebted to my colleagues at the University of Aarhus, Denmark, in particular Peter Kornerup and Michael Spier for supporting my application for the grant which enabled me to join the software engineering research team at Oxford University. I am also grateful to Professor Tony Hoare for accepting me as a D.Phil. student in his department.

I would like to thank the staff at the Programming Research Group, Oxford University for their valuable guidance in research and general assistance during the preparation of this thesis. Thanks are especially given to Jean Raymond Abrial, Tim Clement, Bernard Sufrin and Cliff Jones.

It should be mentioned that the work presented in this thesis is a consequence of Jean Raymond Abrial's pioneering research in the area of formal specifiations. His insight and suggestions have inspired many of the ideas presented in this thesis.

Special thanks are due to Hilary Darragh for enriching my English language and to Tim Clement and Steve Schuman for their constructive criticisms while editing my thesis.

Ib Holm Sorensen



# 1. INTRODUCTION.

The tools and methods used in the creation of computerised systems are being intensively studied by many researchers with the aim of improving our capability for constructing 'good' computer systems.

The quality of such systems depends to a large extent on the methods used for documention during the development process.

The *documentation* of a system has three functions: firstly, it identifies the *needs* which the system is intended to fulfil; secondly, it specifies the decisions taken by the designers of the system as to *how* those needs are to be fulfilled; and thirdly, it gives a detailed description of how the design decisions are realised. The documentation of the needs must be *precise* in order for it to serve as a basis for a contract between the customer and the supplier. The documentation of the design and the realisation must be precise in order to *test* and *verify* whether a suggested implementation fulfills that contract. Furthermore, since the *needs* will change over time it is important that all parts of the documentation (including the programs which document the realisation) are easy to modify.

These concerns have inspired the development of numerous methodologies for the creation of software. Current techniques use *informal languages* or *diagrams* as description tools for documenting the requirements, the design decisions and the structure of the realisation. The disadvantage of such techniques lies in the ambiguity inherent in the notational framework. Among other things, this ambiguity makes it impossible to *formally test* a design against a requirement specification, or a realisation against the design decisions (*program verification*). *Theoretical* or *formal* tools which do not have these drawbacks have been proposed, but system developers are reluctant to use such formal methods because they are claimed to be inadequate for the documentation of *real systems* - a claim which is understandable since their usefulness has yet to be demonstrated on large commercial products.

But, it is incontestable that formal methods are necessary, since only the use of unambiguous formal descriptions enables us to *verify* whether decisions taken in the development process are consistent with the original specification of the needs. Formal methods for developing systems which consist of modules running in isolation and having well-defined starting and stopping points are fairly well understood [Jones,14]. These methods, in which modules can be described as mathematical functions, cannot, however, be employed in the development of systems which consist of several interacting autonomous process-components (i.e *distributed systems*). The 'function' computed by a subcomponent in such a system can at any point in time be *interrupted* and influenced by non-deterministically occurring stimuli from the environment or from other concurrently executing process-components. The component itself may interfere with computations within other components depending on some non local information.

The complexity of such systems necessitates the development of alternative formal methods. The aim of this thesis is to contribute to this development by providing tools for specifying, analysing and refining the designs of distributed systems.

## 1.1. THE OUTLINE OF THIS THESIS.

*CHAPTER 2* gives an informal presentation of the *language* used in this thesis. The language is an extension of a conventional set theoretical notation. The extensions are simple syntactical conventions and shorthands, which are introduced in order to improve the writebility and readability of specifications. The language has been developed independently of the special area – specification of distributed systems – to which it is being applied in this thesis. The main aim in the development of this new language has been to provide a uniform notation and a formal framework for reasoning and proving properties about computer systems. The language (in its current form) has been shown to meet this aim (e.g. [Sufrin,21]). It is not within the scope of this thesis to give a formal definition for the syntax and semantics of the notation in question. Hence, chapter 2 gives only an informal overview of the language, and is included mainly to ease the understanding of the specifications in the remaining parts of the thesis.

Specifications of a system include a description of an abstract state-space and a description of the transformations of that state. The state is constrained by axioms. These constraints proscribe those 'abstract' tranformations which would violate the axioms, and allow (by definition) those which would not. In other words, an operational model in which the transformations may or may not occur whenever sufficient pre-conditions are met can be derived from the description of the abstract state and the constraints. The approach, for specifying distributed systems, presented in this thesis allows for systems behaviour to be investigated using such an operational model for describing *non-deterministic systems*. A formal framework for discussing *non-deterministic behaviour* is developed in *CHAPTER 3*. Although the work presented in that chapter is original, it is not entirely novel as it embodies a recasting of familiar concepts (e.g. well-founded relations [Abrial,1] and correctness of non-deterministic constructs [Dijkstra,7]). The theory of non-deterministic systems developed in chapter 3 is generally applicable to the analysis of systems whose behaviour can be characterised by a relation.

*CHAPTER 4* introduces a new approach for describing *distributed systems*. Such systems are described in terms of constraints on the communications which may occur across a set of interfaces (i.e. connections). The constraints are imposed by axioms involving the *past history of communications* along one or more of the connections. A communication history along a single connection is modelled as a *sequence*. The specification of a distributed system provides enough information for a non-deterministic model of the its behavior to be defined and analysed (using the framework developed in chapter 3). Furthermore, the inside of each process-component (an object between a set of interfaces) can be constructed from the the axioms which constrain the activities along its own ***external interfaces***.



In *CHAPTER 5* the specification method is applied to a number of small systems. Questions which are normally of concern when designing distributed systems (mutual exclusion, starvation, deadlock etc.) are discussed in connection with these examples.

The specification method can be used to describe distributed systems at different levels of abstraction. In a very abstract form only a few interfaces are visible – and in the least abstract form all interfaces from the realisation are visible. Hence, it can support a *stepwise development* method. *Chapter 6* gives a method for comparing (the behaviour of) different levels of description, thus giving us the basis for *refining* specifications.

*CHAPTER 7* presents some conclusions and proposes topics for further study.

## 2. THE SPECIFICATION LANGUAGE.

One of the purposes of the documentation of a system (i.e. the specification) is to give a description of one's understanding of a given problem and a description of the set of rules which must be obeyed by any system which is claimed to be a solution to that problem. For a specification to play a useful role in the *development* of systems, statements in the specification must be expressed in a form which is independent of any subsequent realisation, i.e. the statements should only involve *abstract* concepts. Furthermore, to be of any use at all, the statements must be *unambiguous*, so that all readers of the specification (including the author) reach exactly the same understanding of the problem.

These points give rise to the requirement for a language or notation in which abstract concepts can be unambiguously expressed. Since the main concern of mathematics is the precise formulation of abstract concepts, it seems appropriate to look to this discipline for a formal notational framework.

The formal language described here is based on *Set Theory*. This theory has the advantage of being the branch of mathematics which is the most intuitively accessible, while being sufficiently powerful; this is important if specifications are to be used (and read) by other than highly skilled mathematicians. Section 2.1 gives an informal overveiw of our Set theoretical notation.

Experience shows, however, that this notation is most unwieldy as a language for specifying practical systems, because of the complexity that comes from the large number of concepts involved. In order to manage this kind of complexity we use abstract structures (called SCHEMAS), which permit the encapsulation (or grouping) of several concepts into a single *named* structure. These structures can be viewed as pieces of text. Using the names of structures, instead of the structures themselves in statements about a system simplifies the formal text and increases readability without any loss of detail whatsoever, as the abstracted detail can be recovered by replacing the names with the structures they denote. In order to ennance the applicability of such schemas their definitions may be parameterised so that the same schema may be instantiated in different contexts. The *Schema notation* is introduced in section 2.2.

An important characteristic of the approach to system specification presented in this paper is the notion of *Extensibility.* The theoretical framework provided by elementary Set Theory can be extended with user-defined theories (i.e. new axioms which may introduce new constants or new operations). Success in formalising a system is often dependent on finding a sufficiently powerful formal framework (theory) for expressing the statements and rules about that system. Using an inappropriate theory will often lead to a confused and unreadable description; the 'right' theory will provide the insight necessary and a sufficiently powerful notational framework for a description to be clear and readable. Hence a specification exercise will normally involve two distinct tasks. The first is to find the 'right' theory, the second is to apply that theory to the given problem. In section 2.3, examples of extensions to the elementary Set Theory will be given.

## 2.1. THE SET THEORETICAL BASIS.

This section is an overview of the basic notation of the proposed language. No attempt is made to define the semantics of the constructs formally; this and succeeding subsections are intended only to provide enough information about the language to permit discussion about specifications written therein.

### 2.1.1. Sets and Operations on Sets.

The following expression denotes a particular subset of a set X, namely the set of elements for which PRED holds,

$$\{ x:X \mid PRED(x) \} \qquad \text{DEF (2.1)}$$

where PRED(x) stands for a predicate in which x appears free. This is a rewriting of the classical form,

$$\{ x \mid x \in X \wedge PRED(x) \}$$

The set above is *non-empty* if and and only if

$$(\exists\ x : X \mid PRED(x))$$

The explicit construction of a finite set is denoted as usual by enumerating its elements, as in

$$\{ a, b, c, d \}$$

The following expressions denote, respectively, the set of all subsets of the set X and the set of all finite subsets of the set X

$$\mathbb{P}(X), \qquad \mathbb{F}(X)$$

The empty set is denoted by

$$\{\}$$

The number of elements in a finite set X is

$$card(X)$$

We shall use the standard set operations of union, intersection and difference as well as the inclusion and membership operators. They are denoted as usual by

$$\cup \quad \cap \quad - \quad \subset \quad \not\subset \quad \subseteq \quad \in \quad \notin$$



Furthermore

$$\cup \quad \cap$$

will be used as prefix operators to denote the distributed union and intersection respectively. The use of these operators take a special form, e.g. let EXP(i), for all i in an index set I, denote a subset of X, then the following

$$\cup\{ EXP(i) \mid i : I \}$$

is a rewriting of

$$\cup\{ s : \mathbb{P}(X) \mid (\exists\ i : I)( s = EXP(i)) \}$$

The *introduction* (or the *type declaration)* of a variable x, which can only be bound to values from the set X, is written,

$$x : X$$

(see the examples above)

A particular element with the value EXP from

$$\{ x:X \mid PRED(x) \}$$

can be written

$$(\mu\ x:X \mid PRED(x))\ (x = EXP) \qquad \text{DEF (2.2)}$$

provided, of course, that PRED(EXP) holds.

The designation of an *arbitrary* element from the same set is written

$$(\mu\ x : X \mid PRED(x))$$

provided that

$$\{ x : X \mid PRED(x) \} \neq \{\}$$

Construction of a subset of the Cartesian Product

$$X \times Y$$

is written

$$\{ x:X;\ y:Y \mid PRED(x,y) \} \qquad \text{DEF (2.3)}$$

instead of

$$\{(x,y) \mid (x,y) \in X \times Y \wedge PRED(x,y) \}$$

### 2.1.2. Relations and Operations on Relations.

The set of all binary relations from a set X to a set Y is denoted by

$$X \leftrightarrow Y$$

which simply is a shorthand for

$$\mathbb{P}( X \times Y )$$

A finite relation can be constructed explicitly as follows.

$$\{ x_1 \leftrightarrow y_1 , x_2 \leftrightarrow y_2 , x_1 \leftrightarrow y_2 \}$$

which maps $x_1$ to $y_1$ and $y_2$ etc.

Given a binary relation R from X to Y, then the *image* of a subset SX of X through R i.e.

$$\{ y : Y \mid (\exists x : SX) \ (x,y) \in R \}$$

is denoted by

$$R( SX )$$

The *inverse* of a binary relation R from X to Y is denoted by

$$R^{-1}$$

Given a relation $R : X \leftrightarrow Y$ we have

$$R^{-1} \in Y \leftrightarrow X$$

and

$$dom(R) = R^{-1}(Y)$$
$$ran(R) = R(X)$$

which denote the *domain* and the *range* of the relation R.

When expressing the relationship between two elements we use the well-known form

$$x \ R \ y$$



which is syntactically equivalent to

$$y \in R( \{x\} )$$

The *composition* of two binary relations

$$R_1 : X \leftrightarrow Y$$
$$R_2 : Y \leftrightarrow Z$$

is denoted by

$$R_2 \circ R_1$$

which is a relation from X to Z for which

$$(\forall x:X; z:Z) \quad x \ (R_2 \circ R_1) \ z \iff (\exists y:Y)(x \ R_1 \ y \wedge y \ R_2 \ z)$$

### 2.1.3. Functions.

The set of *partial* functions from X to Y is a subset of the set of all relations between X and Y, and is denoted by

$$X \nrightarrow Y$$

$$X \nrightarrow Y =$$
$$\{ r:X \leftrightarrow Y \mid (\forall y:Y \mid y \in ran(r)) \ r(r^{-1}(\{y\}))=\{y\} \}$$

The construction of a partial function 'f' from X to Y, which maps an element in X to an element of Y given by the expression EXP(x), might be written.

$$f = (\lambda x : X \mid PRED(x)) \ (EXP(x)) \qquad \text{DEF (2.4)}$$

Provided that

$$(\forall x:X \mid PRED(x)) \ ( EXP(x) \in Y )$$

we have

$$dom(f) = \{ x:X \mid PRED(x) \}$$

Given two predicates PRE and POST then we may construct a function h of type

$$h : \{x:X \mid PRE(x)\} \nrightarrow Y$$

as follows

$$h = (\lambda\ x{:}X \mid PRE(x))\ ((\mu\ y{:}Y \mid POST(x,y))\ (y = EXP(x)))\qquad \text{DEF (2.5)}$$

We have

$$dom(h) \subseteq \{x{:}X \mid PRE(x)\}$$
$$ran(h) \subseteq \{y{:}Y \mid (\exists\ x{:}X \mid POST(x,y))\}$$

The domain of h can determined to be

$$dom(h) = \{x{:}X \mid PRE(x)\} \cap \{x{:}X \mid (\exists\ y{:}Y \mid POST(x,y))\ y{=}EXP(x)\}$$

The set of *total functions* from X to Y is a subset of all partial functions over these sets, and is denoted by

$$X \rightarrow Y$$

$$X \rightarrow Y = \{\ f{:}X \leftrightarrow Y \mid dom(f) = X\ \}$$

Consider the definition of h above: if

$$(\forall\ x{:}X \mid PRE(x))\ POST(x,EXP(x))$$

then h is a total function i.e.

$$h \in \{x{:}X \mid PRE(x)\} \rightarrow Y$$

hence the 'transformations' denoted by

$$y = EXP(x)$$

can be considered as a realisation of the specification

$$h_{spec} = (\lambda\ x{:}X \mid PRE(x))\ ((\mu\ y{:}Y)(POST(x,y)))$$

(cf. [Jones,14])

Given $f : X \rightarrow Y$, then the single element in the *image* of a single element through a function is written as usual,

$$f(x)$$

where

$$f(x) = (\mu\ y : Y)(y \in f(\{x\}))$$

We can define a function which maps a *non-empty* set into an *abitrary* element of that set as follows

$$\tau : \mathbb{P}(X) \nrightarrow X;$$
$$\tau = (\lambda\ s{:}\mathbb{P}(X) \mid s \neq \{\})\ ((\mu\ x{:}X)(x \in s))$$



## 2.2. THE SCHEMA NOTATION.

The previous section provided a uniform notation for the implicit construction of sets, elements, relations and functions (see DEF(2.1-5))

The *syntactic* schema

$$a{:}\ A;\ b{:}\ B\ .\ .\ .\ z{:}\ Z \mid predicate(a,b,..,z)$$

appears in the characterisation of sets and of the domain and range of functions.

This schema is also used in the quantified expressions of the Predicate Calculus, e.g. the expressions (appearing in section 2.1.1. and 2.1.3)

$$(\exists\ x{:}X \mid PRED(x))$$
$$(\forall\ x{:}X \mid PRED(x))\ (EXP(x) \in Y)$$

are rewritings of

$$(\exists\ x)\ (x \in X \wedge PRED(x)),$$

and

$$(\forall\ x)\ ((x \in X \wedge PRED(x)) \Rightarrow (EXP(x) \in Y))$$

The notation introduced in this thesis allows the *definition* and *naming* of such syntactic schemas.

### 2.2.1. Example of the Usage of Schemas.

Let the Natural Numbers be denoted by

$$\mathbb{N}$$

We can give the name TWONUM to the schema

$$n{:}\mathbb{N}\ ;\ m{:}\mathbb{N} \mid n \geqslant m$$

either by writing

$$TWONUM \triangleq [\ n{:}\mathbb{N};\ m{:}\mathbb{N} \mid n \geqslant m\ ]$$

or by using a *vertical* presentation,

TWONUM

| |
|---|
| $n : \mathbb{N};$ |
| $m : \mathbb{N}$ |
| --- |
| $n \geqslant m$ |

The part above the horizontal line within the box is referred to as the *signature*, the part below as the *axiom* or the *invariant*.

The schema TWONUM can be used as a *textual macro* in

1) The implicit construction of a set.

```
twonum = {TWONUM}
```

i.e.

```
twonum = {n:N; m:N | n ≥ m}
```

2) The implicit construction of a function.

```
subtract : {TWONUM} → N
subtract = (λ TWONUM) (n-m)
```

3) The construction of elements.

```
pair = (μ TWONUM)(n=3; m=2)
```

NB. the definition is 'valid', since 3 ≥ 2

4) In quantified expressions.

```
(∃ TWONUM)(n = m+10)
(∀ TWONUM)(n + 1 > m)
```

5) The definition of *new* schemas, by *extending* existing schemas

INTERVAL

```
TWONUM;
set : ℙ(N)
```

```
(∀ i : s) (m ≤ i ∧ i ≤ n)
```

which is a shorthand for

INTERVAL

```
n,m  : N;
set  : ℙ(N)
```

```
n ≥ m;
(∀ i : s) (m ≤ i ∧ i ≤ n)
```



6) Definition of *new* schemas by *combining* existing schemas.
Let

LIMSET

```
set : ℙ(N);
```

```
card(set) < 256
```

then we might define

SET2

```
TWONUM; LIMSET
```

```
n = card(set);
m = card({ j : set | j ≠ 0 })
```

Note that the combination of two schemas may introduce 'duplicated' declarations in the signature of the resulting schema. Such a schema is identical to a schema in which one of the declarions is removed, e.g. the schema

```
[ INTERVAL; LIMSET ]
```

in which the name set 'occurs' twice is

```
[ INTERVAL; card(set) < 256 ]
```

7) In *Theorems*, as illustrated below.

The signature part of a schema may be empty, hence we can define,

SUM_and_AVG

```
n = x + y ;
m = (x + y)/2
```

NB '/' is integer division, i.e.

```
/ : N × N → N
(∀ i,j,r : N) (r=i/j ⇔ 0 ≤ i - m*j < j)
```

Given the schema,

PARAM

```
x : N; y : N1
```

NB. N1 denotes the *non-zero* natural numbers

we can define a function which gives the *sum* and the *average* of two numbers in the following way.

```
sumavg = (λ PARAM)(μ TWONUM)(SUM_and_AVG)
```

which defines a total function if we can prove the following theorem.

```
x∈N; y∈N1; n = x+y; m = (x+y)/2
⊢
n∈N; m∈N; n≥m
```

which leads to the last important use of the schema notation. The theorem above can be written as follows.

```
PARAM; SUM_and_AVG  ⊢ TWONUM
```

### 2.2.2. Schema Renaming.

The bound variables of a schema (i.e. the variables which appear in the signature part of a schema) may be systematically *renamed*, e.g.

```
TWONUM'
```

denotes the text

```
n' : N; m' : N | n' ≥ m'
```

and

```
TWONUM_ex
```

denotes

```
n_ex : N ; m_ex : N | n_ex ≥ m_ex
```

The schema notation allows individual renaming of one or several variables of the signature, e.g.

```
TWONUM[k/n]
```

denotes



```
k : N; m : N | k ≥ m
```

It is important to understand, however, that

```
{TWONUM} = {TWONUM'} = {TWONUM_ex} = {TWONUM[k/n]}
```

i.e. systematic renaming of bound variables of a schema does not affect the set which corresponds to that schema, cf.

```
(λx)(x+2)  =  (λy)(y+2)
```

Additionally, the schema notation allows for renaming of the free variables in a schema, i.e. given

```
A_NUM ≙ [ n:N | n < Limit ]
```

then

```
A_NUM1 ≙ A_NUM[L1/Limit]
```

denotes a new schema. Note that, if L1 ≠ Limit then

```
{A_NUM} ≠ {A_NUM1}
```

The main use of schema renaming is in the definition of state transformations, where, informally speaking, the same state variables appear twice.

Example:
let us define a simple state as follows.

STATE

```
counter : N
```

A decrement function,

```
decr : {STATE} ⇸ {STATE}
```

can be defined using the renaming facility

```
decr = (λ STATE) (μ STATE')(counter'=counter-1)
```

NB

```
dom(decr) = {STATE | counter > 0 }
```

### 2.2.3. Generic Schemas.

Informal statements or definitions, such as those in section 1 of this paper, often take a *generic* form. Consider the following definition which is generic with respect to X and Y

"let X and Y be sets. The *partial injections* from X to Y are exactly the partial functions from X to Y whose *inverses* are also partial functions."

In the schema notation this concept is formalised, e.g. the above statement will have the form

par_inj [X, Y]

$$f : X \pfun Y$$

---

$$f^{-1} \in Y \pfun X$$

A schema may have several *instances*, e.g. the set of partial injections over the natural numbers is

$$\{\ par\_inj[\mathbf{N},\mathbf{N}]\ \}$$

and the set of partial injections from **N** to the Cartesian Product over **N** is,

$$\{\ par\_inj[\mathbf{N},\mathbf{N}\times\mathbf{N}]\ \}$$

## 2.3. Extensibility.

The importance of the possibility to extend the formal framework in which one wants to present a specification has already been emphasised in the introduction.

The notation provided for extending a theory allows for the introduction of new objects, which are either constructively or axiomatically defined, and for the introduction of new operations on these objects. Such extensions can be grouped and the groups are given names, which enable the writers of a specification to reference theories developed elsewhere.

### 2.3.1. Operations on Relations and Functions.

The elementary Set Theory which was presented in section 2.1.1 can be extended to included a set of useful operations on relations. Let us first introduce the *identity* function (relation) over a subset of any set, i.e.



REL-OP [X]

$$id : \mathbb{P}(X) \to (X \leftrightarrow X)$$

---

$$(\forall\ s : \mathbb{P}(X))$$
$$id(s) = \{\ (x,x) \mid x \in s\ \}$$

This definition is *generic* in X, hence, we have several instances of the identity function. The use of different instances of a definition is illustrated in the next extention which is *generic* with respect to S and T. This extension defines two *domain* restriction operators and a *codomain* restriction operator.

REL-OP1 [S, T]

$$op(\lhd) : (S \leftrightarrow T) \times \mathbb{P}(S) \to S \leftrightarrow T\ ;$$
$$op(\rhd) : (S \leftrightarrow T) \times \mathbb{P}(T) \to S \leftrightarrow T\ ;$$
$$op(\setminus) : (S \leftrightarrow T) \times \mathbb{P}(S) \to S \leftrightarrow T$$

---

$$(\forall\ r : S \leftrightarrow T;\ s : \mathbb{P}(S);\ t : \mathbb{P}(T))$$
$$(\ r \lhd s = r \circ id[S](s) \ \wedge$$
$$\ \ r \rhd t = id[T](t) \circ r \ \wedge$$
$$\ \ r \setminus s = r \lhd (S - s))\ )$$

As for the identity function we have several instances of the domain-restriction operators, e.g.

$$\lhd[\mathbf{N},\mathbf{N}] \in (\mathbf{N} \leftrightarrow \mathbf{N}) \times \mathbb{P}(\mathbf{N}) \to \mathbf{N} \leftrightarrow \mathbf{N}$$
$$\lhd[\mathbf{N},\mathbb{P}(\mathbf{N})] \in (\mathbf{N} \leftrightarrow \mathbb{P}(\mathbf{N})) \times \mathbb{P}(\mathbf{N}) \to \mathbf{N} \leftrightarrow \mathbb{P}(\mathbf{N})$$

The 'parameter-list' of the instantiation of an operator is often omitted, since its 'type' can be determined from the context in which the operator appears, e.g.

let f, g, S be declared as follows

$$f, g : \mathbf{N} \pfun \mathbf{N}\ ;$$
$$S : \mathbb{P}(\mathbf{N})$$

then

$$g = f \lhd S$$

means

$$g = f \lhd[\mathbf{N},\mathbf{N}]\ S$$

The nth iterate of a relation

$$R : S \leftrightarrow S$$

can informally can be written as

$$R^n = \underbrace{R \circ R \circ \ldots \circ R}_{\text{n-times}}$$

The following extensions define the *iteration* operators.

REL-OP2 [X]

op(^) : (X↔X) × **N** → (X↔X) ;
op(*) : (X↔X) → (X↔X)

---

(∀ r:X↔X; n:**N**1)
( r^0 = id(X) ∧
r^n = r^(n-1) ∘ r )
(∀ r:X↔X) r* = ∪{r^n | n : **N**}

NB. the *infix* operator ^ is implied (as usual) by writing the second operand as a superscript to the first operand.

In the following we add the function *overriding* operator to the theory of functions.

FUNC-OP [S T]

op(⊕) : ((S↔T) × (S↔T)) → (S↔T)

---

(∀ g, f : S↔T)( g ⊕ f = (g\dom(f)) ∪ f )

NB. functions are sets to which the Set operations can be applied.

### 2.3.2. Operations on Sequences.

A theory which is often used in the specification of systems is the theory of sequences. In the following we will formalise *sequences* and some operations on them.

Let us first formalise the well known 2-dots, as in

3..m

op(..) : **N** × **N** → ℙ(**N**)

---

(∀ n,m,p : **N**)( p ∈ n..m ⇔ n≤p ∧ p≤m )

*Sequences* are then formalised as a subset of the partial functions, e.g. a *sequence* of length 4 over a set



{a, b, c, d, f, g}

might be defined as,

{(1,a), (2,c), (3,b), (4,a)}

This sequence will be denoted by

⟨ a c b a ⟩

The empty sequence is denoted by

⟨⟩

Sequences are 'formally' added to our theory by the following generic extension.

SEQ [X]

seq,seq1 : ℙ(**N** ⇸ X)

---

(∀ s:seq)
( dom(s) ∈ ℱ(**N**) ∧
dom(s) = 1..card(dom(s)) )

seq1 = seq − {⟨⟩}

In the following we add some operators to our 'current theory'

SEQ-OP1 [X]

# : seq[X] → **N**;
next : seq[X] → (X↔X);
first,last : seq1[X] → X;
ending,
beginning : seq1[X] → seq[X]

---

(∀ s:seq)
(#(s) = card(s) ∧
next(s) = s ∘ succ ∘ s⁻¹ )

(∀ s:seq1)
(first(s) = s(1) ∧
last(s) = s(#(s)) ∧
ending(s) = s ∘ (succ\{0}) ∧
beginning(s)= s⩥(1..#(s)-1) )

The function succ is the successor function over the Natural Numbers.

The *concatenation* operator for sequences is defined by.

SEQ-OP2 [X]

$$op(*) : seq[X] \times seq[X] \rightarrow seq[X]$$

$$(\forall s,s':seq)\ s * s' = s \cup (s' \circ pred^{\#(s)})$$

The function pred is the predecessor function over the Natural Numbers.

## 2.4. NOTATIONAL CONVENTIONS.

The *schema* notation permits the grouping of concepts into named structures. The names are used in the construction of sets, functions etc.. We also allow for the *naming* of *theory-extensions* (as in SEQ and SEQ_OP1 from section 2.3.2) simply in order to be able to reference specific theories elsewhere in the document. The names are also suggestive , i.e. SEQ, SEQ_OP1 and SEQ_OP2 make up the current theory of SEQuences. Even with such a naming facility the cross-referencing in a large document (like a thesis) becomes unmanageable, therefore this thesis makes use of an indexing system, in which all important sections of formal text have been indexed. *Schemas* and *Extensions* are categorised as definitions, hence we use DEF as a qualifier for them (see DEF(3.2)). *Theorems*, *Lemmas* and *Figures* are also isolated from the informal text using 'boxes'; their index-numbers are prefixed with the qualifiers THEOREM, LEMMA and FIGURE respectively. Theorems are when convenient given the name of the *theory* to which they belong, see LOOP:DEF(3.2) and the theorems LOOP:THEOREM(3.3-6).



# 3. NON-DETERMINISTIC SYSTEMS.

The purpose of this chapter is to investigate the properties of non-deterministic behaviour. The results will be used to justify the introduction of a particular approach to the specification of distributed systems.

The descriptions of most systems do not include complete descriptions of control mechanisms. This means that we cannot fully predict the behaviour of the system by looking at its description - the behaviour of our system is *non-deterministic*. The *top-down* development process which takes a system through design stages to a realisation, involves, among other things, adding further decisions about actual behaviour to the description; hence the non-determinism present at the top level might well be resolved at some lower level where a more detailed description of the behaviour is given.

We can therefore expect, even when developing sequential deterministic programs, to be faced with the problem of proving properties of designs which involve non-deterministic behaviour. Systems which are eventually implemented as strictly sequential programs may at a certain stage in the design process best be described using non-deterministic constructs [Dijkstra,8]. The reasons for this are twofold - 1) the designer should not impose in advance any decisions about the system which may have an adverse effect on future refinements and realisations - 2) the properties of a design are often independent of decisions about actual dynamic behaviour, hence a description of these decisions is pure 'noise' and should be left out.

Furthermore, there is a very important class of systems which cannot be realised using sequential control constructs because their actual behaviour is controlled by or synchronized with *external events* (i.e. events activated by the environment) which occur non-deterministically. For example, we expect the control-flow of the program for an air-traffic control system to be influenced by the unpredictable event of an aeroplane entering into the airspace monitored by the system; we expect a terminal multiplexer to acknowledge a key-stroke; we expect the activity of a batch operating system to be interrupted upon entry of a new deck of cards.

System behaviour may also include *internal events* that occur non-deterministically, i.e. may or may not occur whenever the preconditions which determine their possible occurrence are satisfied, as for a communication between components in an multi-process system. Common to the realisations of these systems is the fact that they are composed of several highly independent processing components, whose progress between points of synchronization is unknown (distributed concurrent systems). Such systems cannot be realised using purely sequential control mechanisms, since any decision that attemps to prescribe the order of occurrence for 'inter-process' events in advance (in either the design or implementation) could easily cause unnecessary delays at run-time, and even infinite delays ( i.e. deadlocks).

In the first section of this chapter a simple model for non-deterministic behaviour will be introduced. Definitions of a set of functions, which will be used to characterise non-deterministic behaviour, will be given. This model and these definitions are applied to a very simple example in the section which follows. The last section will summarise the properties of non-deterministic behaviour to be considered, and give a method for proving those properties.

## 3.1. FORMALISATION OF NON-DETERMINISTIC BEHAVIOUR.

Non-deterministic behaviour may be pictured as a *state diagram (a directed graph)* in which the *nodes* represent the *states* of the system and the outgoing edges represent the *events*. A node with two or more outgoing edges represents a state of the system where sufficient conditions for the occurrence of more than one event are satisfied. Nodes with no outgoing edges represent states where no further progress can be made - these states are referred to as `Terminal`-states.

Even for small systems this pictorial method for describing non-deterministic behaviour becomes unmanageable, and can therefore only be used to illustrate special dynamic properties - never complete systems. A more suitable method of description is needed.

The events ( the *state-transitions* ) will be described as *partial functions*. Whenever the system which is being described is in a state belonging to the domain of one of these partial functions, the corresponding event may occur, thus transforming the system. Consequently in such a model the total behaviour is represented by a *relation* (usually referred to as the `Step` - relation) and the `Terminal` - states are states which do not belong to the domain of this relation.

We can define a *schema* for describing such a model :

ND [S] DEF (3.1)

$$\begin{array}{l} \texttt{Events} : \mathbb{F}(\ S \nrightarrow S\ ); \\ \texttt{Step} : S \leftrightarrow S; \\ \texttt{Terminal} : \mathbb{P}(\ S\ ) \\ \hline \texttt{Step} = \cup(\ \texttt{Events}\ ); \\ \texttt{Terminal} = S - \texttt{dom}(\ \texttt{Step}\ ) \end{array}$$

Note that only systems with a finite number of events are considered; most of the results in this chapter do however apply to the more general but unrealistic case where

$$\texttt{Events} : \mathbb{P}(\ S \nrightarrow S)$$

For any given state `s` , `Step({s})` is the set of states which can be reached from `s` in one step, i.e. if `s` is in the domain of two events



`(ev1, ev2)`

then

`Step({s}) = {ev1(s), ev2(s)}`

The relation

$\texttt{Step}^n$

will, applied to state `s` , describe the states which can be reached after `n` steps.

The domain of the relation,

$\texttt{Step}^n \lhd \texttt{Terminal}$

is the set of states in `S` in which the system may be started and possibly terminate after `n` steps. This intuition justifies the following definition:

LOOP [S] DEF (3.2)

$$\begin{array}{l} \texttt{loop} : (\ S \leftrightarrow S\ ) \rightarrow (\ S \leftrightarrow S\ ) \\ \hline (\forall\ r : S \leftrightarrow S) \\ (\texttt{loop}(r) = \cup\{\ r^n \lhd (S - \texttt{dom}(r)) \mid n : \mathbb{N}\ \}) \end{array}$$

The operator `loop` can be used to establish *partial correctness*. ( A system is *partially correct* if it produces the desired result or fails to terminate).

If the system is started (from `s`) within the domain of `loop` for a relation `Step` then the system may terminate, and furthermore if it terminates the resulting state is within `loop(Step)({s})`. This may be expressed as follows:

LOOP [S] THEOREM (3.3)

$$\begin{array}{l} R : S \leftrightarrow S; \\ s : S; \\ s \in \texttt{dom}(\texttt{loop}(R)) \quad \vdash \\ (\ \exists\ n : \mathbb{N}\ )(\ \exists\ s' : S\ )\ ((s\ R^n\ s') \wedge s' \notin \texttt{dom}(R))\ ; \\ \\ (\ \forall\ n : \mathbb{N};\ s' : S\ ) \\ \quad (\ (s\ R^n\ s' \wedge s' \notin \texttt{dom}(R)) \Rightarrow s' \in \texttt{loop}(R)(\{s\})) \end{array}$$

The theorem below is a reformulation of the definition of `loop` , a reformulation which follows directly from the definition of the *transitive closure* ( $R^*$ ) for relations and the distributivity of $\lhd$ over $\cup$.

LOOP — S — THEOREM (3.4)

$R : S \leftrightarrow S \vdash$
$loop(R) = R^{*} \downarrow (S - dom(R))$

A system started in a Terminal-state will remain in that state; in other words states belonging to the complement of the domain of a relation are in the domain of the loop of the same relation. This is a consequence of -

LOOP — S — THEOREM (3.5)

$R : S \leftrightarrow S \vdash$
$loop(R) = id(S - dom(R)) \cup (loop(R) \bullet R)$

A *rule of iteration* for non-deterministic systems may be formulated as follows:

LOOP — S — THEOREM (3.6)

$R : S \leftrightarrow S;$
$INV : \mathbb{P}(S)$
$R(INV) \subseteq INV \vdash loop(R)(INV) = INV - dom(R)$

**proof :**

| | | |
|---|---|---|
| (1) | $R(INV) \subseteq INV$ | (hypothesis) |
| (2) | $R^{*}(INV) \subseteq INV$ | (by induction, as follows) |
| | (a) $R^{0}(INV) \subseteq INV$ | (the basis of the induction) |
| | (b) $R^{n}(INV) \subseteq INV$ | (induction hypothesis) |
| | (c) $R(R^{n}(INV)) \subseteq R(INV)$ | (b) |
| | (d) $R^{n+1}(INV) \subseteq INV$ | (c),(1) |
| (3) | $(R^{*} \downarrow (S-dom(R))(INV)) \subseteq INV$ | ((2) and def $\downarrow$) |
| (4) | $(R^{*} \downarrow (S-dom(R))(INV)) \subseteq (S - dom(R))$ | (def $\downarrow$) |
| (5) | $(loop(R)(INV)) \subseteq (INV \cap (S-dom(R)))$ | (3,4, TH3.4) |
| (6) | $id(S-dom(R)) \subseteq loop(R)$ | (TH3.5) |
| (7) | $id(S-dom(R))(INV) \subseteq loop(R)(INV)$ | (6) |
| (8) | $(INV - dom(R)) \subseteq (loop(R)(INV))$ | (7) |
| (9) | $(loop(R)(INV)) = (INV - dom(R))$ | (5),(8) |

Intuition -

Given a state S. Suppose INV_CON is an invariant condition for a system which behaves according to a relation Step and suppose we can describe S - dom(Step) as { S | TERM_CON }. The invariant condition holds after any number of iterations (occurrences of events) provided it was satisfied upon initialization, hence this condition also holds when ( IF! ) the system terminates; furthermore we know that no more progress is possible i.e. the condition

$INV\_CON \wedge TERM\_CON$

holds, i.e. the system terminates in a state belonging to



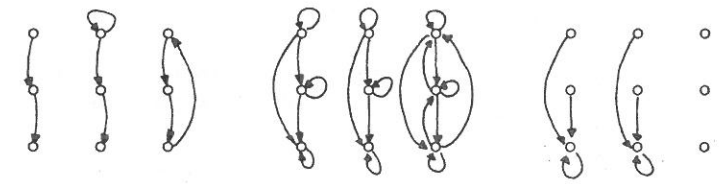
$\{S \mid INV\_CON\} - dom(Step)$

The following graphs (FIGURE 3.7) illustrate the behaviour of a simple system.

We observe the following :

1) A system having the transition-relation Step will never terminate if it is started outside the domain of loop(Step ).
2) A system started within that domain may terminate, but termination is not guaranteed.

Loop and Termination — FIGURE (3.7)




To be able to distinguish between properties such as *possible* termination and *guaranteed* termination we introduce -

CYCLE — S — DEF (3.8)

$Span : (S \leftrightarrow S) \rightarrow \mathbb{P}(S \twoheadrightarrow S);$
$Cycle : (S \leftrightarrow S) \rightarrow (S \leftrightarrow S)$

$(\forall r : S \leftrightarrow S)$
$(Span(r) = \{f : S \twoheadrightarrow S \mid dom(f) = dom(r) \wedge f \subseteq r\} \wedge$
$Cycle(r) = loop(r) \upharpoonright \cap\{dom(loop(sp)) \mid sp : Span(r)\})$

NB. the set of spanning functions for a relation r is *non-empty* ,as

$g = (\lambda x : S \mid x \in dom(r))(\tau(r(\{x\})))$
$g \in Span(r)$

A spanning-function, sp , for a relation R

$sp \in Span(R)$

characterises a particular deterministic behaviour obtained by making decisions in advance about the 'future' non-deterministic behaviour. In other words a particular spanning-function together with a starting state describes a run-time trace.

The set of spanning-functions for a system contains functions which describe 'worst-case' behaviour, e.g. infinite loops. Hence if we start a system in a state from which an infinite loop can be reached then there exists a spanning-function,

say sp, such that the particular starting state is outside the domain of loop(sp). On the other hand if a system is started within the domains of the loops of all spanning-functions, then termination is guaranteed. (as an example, see FIGURE(3.27) and FIGURE(3.28) which illustrate a relation and two of the spanning-functions for that relation). Using these definitions we can give a more precise characterization of non-deterministic behaviour.

A rewriting gives -

CYCLE — S — THEOREM (3.9)

$$\vdash (\forall r : S \leftrightarrow S)$$
$$Cycle(r) = \cup\{ loop(sp) \mid sp : Span(r) \} \upharpoonright \cap\{ dom(loop(sp)) \mid sp : Span(r) \}$$

The next theorem characterises the deterministic behaviour of a system if it is started in a terminal-state ( outside the domain of the step- relation).

CYCLE — S — THEOREM (3.10)

$$\vdash (\forall r : S \leftrightarrow S)$$
$$Cycle(r) = id(S - dom(r)) \cup ((loop(r) \bullet r) \upharpoonright \cap\{dom(loop(sp)) \mid sp : Span(r)\})$$

**proof :**
use DEF(3.8) and THEOREM(3.5) to rewrite loop(r).

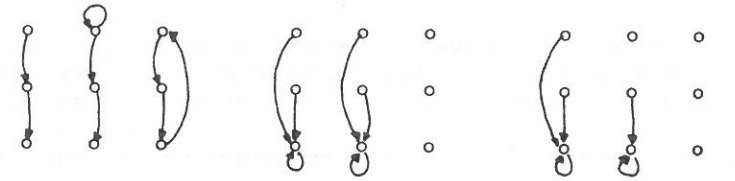
The *rule of iteration* for non-deterministic systems can now be strengthened-

CYCLE — S — THEOREM (3.11)

$$r : S \leftrightarrow S$$
$$INV : \mathbb{P}(S)$$
$$r(INV) \subseteq INV \vdash Cycle(r)(INV) = (INV - dom(r))$$

Furthermore, as mentioned earlier, the domain of Cycle gives the set of states for a system from which it is guaranteed to terminate; this is illustrated by the graphs in FIGURE(3.12) which is a continuation of the description of the small system from FIGURE(3.7) -

Cycle and Termination — FIGURE (3.12)






The following Termination theorem expresses that there is, for each state within the domain of the Cycle of a finite branching relation ( R ), i.e. a relation for which-

$$(\exists n : \mathbb{N})\ (\forall x : dom(R))\ (card(R(x)) < n)$$

an upper limit to the number of transitions which can be performed before a terminal-state is reached. A proof of the theorem can be given using the Rank function, which will be defined in DEF(3.14).

TERMINATION — S — THEOREM (3.13)

$$Events : \mathbb{F}(S \nrightarrow S);$$
$$s : S;$$
$$s \in dom(Cycle(\cup Events)) \vdash$$
$$(\exists Max : \mathbb{N})$$
$$(\forall n{:}\mathbb{N};\ s'{:}S)\ ((s\ (\cup Events)^n\ s') \Rightarrow (n < Max))$$

The Rank function defines the maximum distance to a terminal state - i.e. the maximum distance to the *leaves* of the relation in question. The definition of Rank is justified by definition DEF(3.15), which defines the distance-set , and THEOREM(3.16) in which it is proved that all distance-sets of finite branching relations are finite sets. Hence, a distance-set has a maximum.

RANK — S — DEF (3.14)

$$Rank : \mathbb{F}(S \nrightarrow S) \rightarrow (S \nrightarrow \mathbb{N})$$

$$(\forall events : \mathbb{F}(S \nrightarrow S))$$
$$(\forall s : S \mid s \in dom(Cycle(\cup events)))$$
$$Rank(events)(s) = max(distance\text{-}set(\cup events, s))$$

For a relation R , the distance-set is defined only for elements which are in the domain of Cycle(R) -

DISTANCES — S — DEF (3.15)

$$distance\text{-}set : (S \leftrightarrow S) \times S \nrightarrow \mathbb{P}(\mathbb{N})$$

$$(\forall R : S \leftrightarrow S;\ s : S \mid s \in dom(Cycle(R)))$$
$$distance\text{-}set(R,s) =$$
$$\{ d : \mathbb{N} \mid (\exists sp : Span(R))\ (d = distance(sp,s)) \}$$
**where**
$$distance(sp,s) = min\{ n{:}\mathbb{N} \mid sp^n(s) \notin dom(sp) \}$$
which exists, since sp is functional and
$$s \in dom(loop(sp))$$

The set of spanning functions for a relation may be infinite; however - as the next theorem states - for a finite branching relation R the distance-sets for an element within dom(Cycle(R)) are finite. Hence the definition given for Rank is valid.

S

THEOREM (3.16)

Events: $\mathbb{F}( S \nrightarrow S )$;
s : S
s $\in$ dom( Cycle(∪Events)) $\vdash$
distance-set(∪Events,s) $\in \mathbb{F}(N)$

let us define -
INF = { s : S | distance-set(rel,s) $\notin \mathbb{F}(N)$ }
rel = ∪Events
**proof** : by contradiction, the assumption is
(1) s $\in$ INF
(2) $(\forall x{:}S)( x \notin dom(rel) \Rightarrow$ distance-set(rel,x) = {0})
from which we get via contrapositive
(3) $(\forall x{:}S)\ (( x \in INF ) \Rightarrow x \in dom(rel))$
at least one of the finite number of states reachable from a state in INF must have an infinite distance-set, i.e -
(4) $(\forall x{:}S)( x \in INF \Rightarrow (\exists y{:}S)( x\ rel\ y \wedge y \in INF))$
from (3) we get
(5) $(\forall x{:}S)$
$(x \in INF \Rightarrow (\exists y{:}S)(x\ rel\ y \wedge y \in INF \wedge y \in dom(rel)))$
This allows us to define a function, as follows -
$t = (\lambda x{:}\ INF)(\mu y : S)( y \in INF \wedge x\ rel\ y )$
which gives -
(6) $t \subseteq rel \wedge t \in S \nrightarrow S \wedge t \in INF \rightarrow INF$
The function
$sp = \tau(Span(rel) \oplus t$
is a spanning function for rel (6), and as
s $\in$ dom(t)
we get (as $t \in INF \rightarrow INF$ )
s $\notin$ dom(loop(sp))
therefore
(7) $(\exists f : Span(rel) )\ ( s \notin dom( loop (f) )$
which gives ( THEOREM(3.9) )
(8) s $\notin$ dom(Cycle( rel )
which contradicts the hypothesis.



## 3.2. A BOUNDED BUFFER.

To illustrate the use of the definitions given in the previous section we examine an *abstract design* for a small system with the following properties :

1) The system consumes *input* from the *environment*.

2) All *input* consumed will eventually appear as *output*. Furthermore, the order in which the elements appear in the *output* stream is identical to the order in which they were consumed.

3) Only a limited number of elements can be 'within' the system at any given time.

4) Necessary conditions for an *input-event* to occur are -
   a) the event will not cause the limit to be exceeded.
   b) the *environment* is willing to participate in such an event, i.e. there are elements to consume.

5) A necessary condition for an *output-event* to occur is that there are still elements, previously consumed, left 'within' the system.

This informal specification will be the basis for the *abstract design* presented in this section. In succeeding chapters formal specification techniques will be developed, but to justify these techniques we need first to examine the general properties of non-deterministic systems. It should be emphasized that the following description is not considered to be a *formal specification*. It is a *design* we propose, and we will use the tools from the previous section to investigate how this particular design relates to the informal specification given.

Given the environment -

in_env : seq[X] ; Bound : N1

then an abstract state of the system may be described as follows :

BB

X

DEF (3.17)

unseen, buf, out : seq[X]

#buf $\leqslant$ Bound;
in_env = out * buf * unseen

where unseen describes the un-consumed input, out the produced output and buf the elements 'within' the system.

As operations on this state we propose two non-deterministically occurring events. The necessary condition for an input to occur is described in PRE_INPUT, the state tranformations are described in INPUT .

PRE_INPUT — X — DEF (3.18a)

```
BB ; unseen ≠ ⟨⟩; #buf < Bound
```

INPUT — X — DEF (3.18b)

```
buf'    = buf * ⟨ first(unseen) ⟩
unseen' = ending(unseen);
out'    = out
```

Similarly for output we have

PRE_OUTPUT — X — DEF (3.19a)

```
BB ; buf ≠ ⟨⟩
```

OUTPUT — X — DEF (3.19b)

```
out'    = out * ⟨ first(buf) ⟩
buf'    = ending(buf);
unseen' = unseen
```

If we restrict ourselves to considering a particular environment, e.g. where

```
#( in_env ) = 3 ∧ Bound = 2
```

then a reasonably complete picture of the dynamic behaviour of the system can be given (FIGURE (3.20)). We define the states and the events as follows-

```
bb     =  { BB };
events =  { (λ PRE_INPUT) (μ BB')(INPUT),
            (λ PRE_OUTPUT)(μ BB')(OUTPUT) }
```

or using the definition (3.1)

```
(μ ND[{BB}])
  ( Event = { (λ PRE_INPUT) (μ BB')(INPUT),
              (λ PRE_OUTPUT)(μ BB')(OUTPUT) }
```

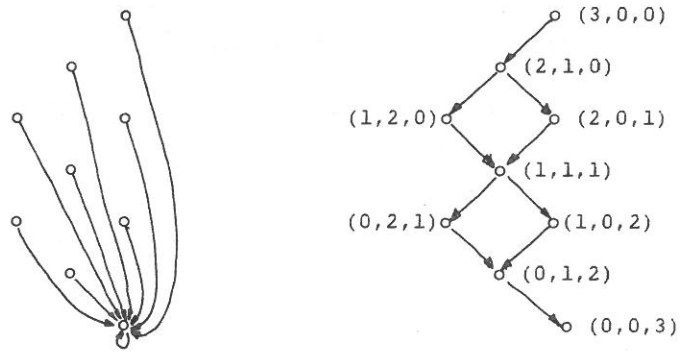
In the diagram the states are represented by triples -

```
( #(unseen), #(buf), #(out) )
```

adjacent to the nodes of the behaviour-graph.



Bounded buffer Behaviour — FIGURE (3.20)




a) loop( ∪ Events)  b) ∪ Events

We now observe the following :

1) Independent of the starting state, it is possible to reach a terminal state in a finite number of steps, i.e.

```
( ∀ b : {BB} ) ( ∃ n : ℕ ) ( ∃ b' : {BB} )
                ( b Step^n b' ∧ b' ∈ ( {BB} - dom(Step)) )
```

where
Step = ∪ Events
in other words the behaviour can be described as a total relation -
dom( loop(Step) ) = {BB}

2) Independent of starting state it is guaranteed that a terminal state will be reached (the graph is loop-free), i.e.
( ∀ sp : Span(Step) ) ( dom( loop(sp) ) = {BB} )
or
dom( Cycle(Step) ) = {BB}

3) If the system is started in a state where the buffer ( buf ) is empty and no output has been produced, then the system will terminate in a state where the output is the original input ( in_env ) - this is formulated as follows.
Given -
Effect = Cycle( ∪Events )
then -
Effect( {BB | out,buf = ⟨⟩; unseen = in_env }) ⊆
{ BB | buf,unseen = ⟨⟩; out = in_env }

The informal discussion above illustrates how the tools provided can be used to express properties about the design of non-deterministic systems.

The *correctness* of our design is expressed in the two following theorems-

1) The *invariant* theorem.
(the *axioms* of BB are invariant under the events)

BUF_INV ——— THEOREM (3.21)

```
⊢ (input ∪ output) ( { BB }) ⊆ { BB } ;
   where :
   input  = (λ PRE_INPUT) (μ BB')(INPUT);
   output = (λ PRE_OUTPUT)(μ BB')(OUTPUT)
```

which is a direct consequence of LEMMA(3.22-23)

Buf_Sys_Invariant ——— LEMMA (3.22a)

```
BB;
unseen ≠ ⟨⟩ ∧ #(buf) < Bound;
buf'= buf * ⟨ first(unseen) ⟩;
unseen'= end(unseen); out'=out
⊢      BB'
```

We may use the descriptions introduced to restate LEMMA(3.22a) more compactly.

Buf_Sys_Invariant ——— LEMMA (3.22b)

```
PRE_INPUT; INPUT   ⊢ BB'
```

Buf_Sys_Invariant ——— LEMMA (3.23)

```
PRE_OUTPUT; OUTPUT ⊢ BB'
```



2) The *total correctness* theorem

Total correctness ——— THEOREM (3.24)

```
Step   : { BB }  ↔  { BB }
Step   = (λ PRE_INPUT) (μ BB')(INPUT) ∪
                 (λ PRE_OUTPUT)(μ BB')(OUTPUT)
⊢
(Cycle(Step))({BB | out,buf=⟨⟩; unseen=in_env }) ⊆
              {BB | unseen,buf = ⟨⟩; out = in_env }
```

**proof :**

Note, the following proof is conditional as the proof of (1) is assumed.
The omitted part of the proof will be given in section 3.3.2.
let us define

```
Effect  = Cycle(Step);
Initial = { BB | out,buf = ⟨⟩, unseen = in_env }
Final   = { BB | unseen,buf = ⟨⟩; out = in_env }
(1) dom(Effect) = {BB}
(2) Initial ⊆ dom(Cycle(Step))                              (1)
(3) Step({BB}) ⊆ {BB}                                  (TH 3.21)
(4) Effect(Initial) ⊆ {BB} - dom(Step)                 (TH 3.11)
(5) Effect(Initial) ⊆
      {BB | (unseen=⟨⟩ ∨ #buf=Bound) ∧ buf=⟨⟩}
(6) Effect(Initial) ⊆
      {BB | unseen=⟨⟩ ∧ buf=⟨⟩ } ∪
      {BB | #buf=Bound ∧ buf=⟨⟩ }
(7) {BB | #buf=Bound ∧ buf=⟨⟩ } = {}                   (Bound∈ℕ1)
(8) Effect(Initial) ⊆
      {BB | (unseen= ⟨⟩ ∧ buf=⟨⟩) }                  (6),(7),(BB)
    using the invariant we get
(9) Effect(Initial) ⊆ Final
```

Consider a slightly modified version of the bounded buffer with a *PUTBACK* facility.
The *putback-event* is described as follows :

PRE_PUTBACK ——— X ——— DEF (3.25a)

```
BB ; #(buf) = Bound
```

PUTBACK ——— X ——— DEF (3.25b)

```
unseen' = ⟨last(buf)⟩ * unseen ;
buf' = beginning(buf);
out' = out
```

The dynamic behaviour of the system described as follows -

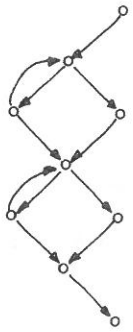
BUF_with_PUTBACK — X — DEF (3.26)

```
in_env : seq[X]
Bound  : ℕ1
Step   : {BB} ↔ {BB}
```

```
Bound = 2    ; #( in_env ) = 3;
Step  =  (λ PRE_INPUT)   (μ BB')(INPUT)   ∪
         (λ PRE_OUTPUT)  (μ BB')(OUTPUT)  ∪
         (λ PRE_PUTPACK)(μ BB')(PUTBACK)
```

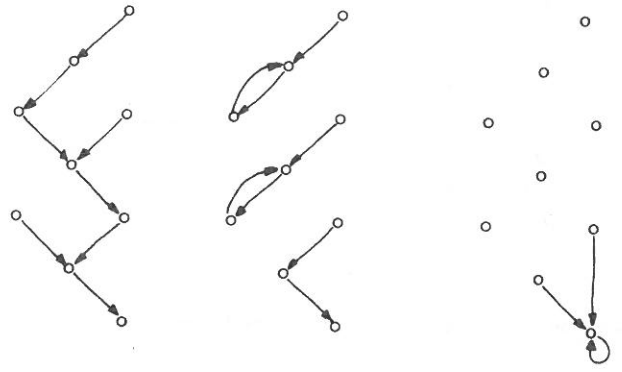
is illustrated in -

Bounded Buffer with putback — FIGURE (3.27)



This system is not guaranteed to terminate as an infinite sequence of 'input - putback' events may occur. FIGURE(3.28) illustrates two of the spanning-functions and the *partial function* Cycle( Step ).

Putback-Buffer behaviour — FIGURE (3.28)



a) Two Spanning Functions.



b)Cycle(Step)



The dynamic behaviour of the system may be characterized formally:

X — THEOREM (3.29)

```
BUF_with_PUTBACK;
f1 , f2 : {BB} ⇸ {BB};
f1 = Cycle( Step );
f2 = Loop( Step )                ⊢

dom( f2 ) = {BB};
dom( f1 ) = {BB | #(buf * unseen) < Bound}
```

which reads - independent of the starting state the system may possibly terminate; termination cannot be guaranteed,unless the number of elements yet to be produced is less than the bound.

### 3.3 FORMALISATION OF NON-DETERMINISTIC SYSTEMS.

The definitions and theorems in section 3.1 gave a formal characterization of non-deterministic behaviour.

The important theorems comprise- 1) [THEOREM(3.13): Termination] which describes the proper starting states for a system, i.e. the states from which terminal-states will be reached - 2) [THEOREM(3.11): CYCLE] which is used to further characterize the states in which a particular non-deterministic system (NDS) terminates.

It should be mentioned that termination properties are considered to be an important concept when designing systems consisting of several independent processing components. *Distributed programs* which are intended to co-operate in solving a specific problem, such as finding a root for a particular function, are expected to terminate.

Systems which ideally should run continuously - such as Operating Systems, Distributed Data Base Systems or Network Communication Systems - should be designed to terminate *only* if the system has been stopped intentionally (operator stop) or if a failure occurs within a vital component of the system, and furthermore, only when all current activity unaffected by the possible failure is properly terminated.

This approach requires that systems for which we prove termination are assumed to operate in a finite environment, i.e. the number of incoming requests (cards, OS-commands, failure-messages) is bounded.

The termination properties which are built into the design of these kinds of systems will, in the implementation, be provided by *close-down* procedures.

We use the following classical concepts to formally discuss termination properties for non-deterministic systems :

Final - states: states in which a system is intended to terminate.
Deadlock - states: terminal states, which are not foreseen Final states.
Initial - states: the states from which the system may be started.

The important categories of non-deterministic systems can be described as follows :

1) The unrestricted system, which may deadlock or encounter an infinite loop.

```
NDS ____________________ X ____________________ DEF (3.30)
|   Events   : 𝔽( X ⇸ X );
|   Terminal : ℙ(X);
|   Initial,
|   Final,
|   Deadlock : ℙ(X)
|-------------------------------------------------
|   Terminal = X - dom( ∪ Events );
|   Deadlock = Terminal - Final
|_________________________________________________
```

2) The halting NDS (include freedom from loops).

```
HALTING_NDS ____________ X ____________ DEF (3.31)
|   NDS[X]
|-------------------------------------------------
|   Initial ⊆ dom( Cycle( ∪ Events ) )
|_________________________________________________
```

intuition - THEOREM(3.13): Termination

3) The *totally correct* system. A system is considered to be totally correct if it can be proved to terminate (loop-free) and that all possible Terminal states, reachable from the Initial states, are acceptable Final states (deadlock-free).

```
WELL_HALTING_NDS ____ X ____ DEF (3.32)
|   HALTING_NDS[X]
|-------------------------------------------------
|   (Cycle( ∪ Events ))(Initial) ⊆ Final
|_________________________________________________
```

4) The very 'robust' system which behaves 'correctly' independent of the starting state, the system may compute the wrong result if it is started outside the Initial state, but it will never encounter an infinite loop or deadlock.



```
NON_BREAKABLE_NDS ____ X ____ DEF (3.33)
|   NDS
|-------------------------------------------------
|   dom(Cycle( ∪ Events )) = X;
|   Terminal ⊆ Final
|_________________________________________________
```

These properties are indeed sufficient to guarantee *total correctness* as we have :

```
____________________ X ____________________ THEOREM(3.34)
|   NON_BREAKABLE_NDS  ⊢  WELL_HALTING_NDS
|-------------------------------------------------
|   let Step = ∪ Events
|
|   proof :
|   (1) dom(Cycle(Step)) = X                    (NON_BREAKABLE)
|   (2) Initial ⊆ dom(Cycle(Step))                       (1)
|   (3) X - dom(Step) ⊆ Final                   (NON_BREAKABLE)
|   (4) ran(Cycle(Step)) ⊆ X - dom(Step)          (TH (3.11))
|   (5) (Cycle(Step))(Initial) ⊆ Final                (3),(4)
|   (6) WELL_HALTING_NDS                              (2),(5)
|_________________________________________________
```

### 3.3.1 Deadlock and the Invariant Condition.

Using the new definitions the statement of correctness [THEOREM(3.24): Bufsys] for the design of the small Buffer System in section 3.2 may now be reformulated (NB. THEOREM(3.35) is stronger than THEOREM(3.24)).

```
BUFSYS ____________________ X ____________________ THEOREM(3.35)
|   in_env  : seq[X] ;
|   Bound   : ℕ1
|   NDS[ {BB[X]} ];
|   Events  = { (λ PRE_INPUT)(μ BB')(INPUT),
|               (λ PRE_OUTPUT)(μ BB')(OUTPUT) }
|   Initial = { BB | out=⟨⟩; buf=⟨⟩; unseen=in_env }
|   Final   = { BB | buf=⟨⟩; unseen=⟨⟩; out=in_env }
|   ⊢
|   NON_BREAKABLE_NDS[ {BB} ]
|_________________________________________________
```

The proof of THEOREM(3.24) illustrates the importance of the *Invariant theorem* in proving WELL_HALTING properties.

If the small buffer system is started outside Initial-states ( e.g. with a non-empty buffer ) then the Final-states are still the only Terminal-states.

however the function computed by the system is no longer the intended identity for sequences. This property, which is the NON_BREAKABLE property, is directly related to the *invariants*, which constrain the states.

If the abstract state ( DEF(3.17) - BB ) had been -

BB2 — X — DEF (3.36)

```
unseen,buf,out : seq[X]
-----------------------
#(buf) ⩽ Bound
```

i.e. without the important invariant

$$in\_env = out * buf * unseen$$

which says that activities performed by the system preserve the ordering of the data within the data-stream and do not introduce extraneous elements, then the corresponding NDS[{BB}] ( constructed as in THEOREM(3.35) Bufsys ) could have been proved to be a WELL_HALTING_NDS by the invariant theorem and observing that the invariant holds in the initial state.

However, such a system does not have the NON_BREAKBALE_NDS property, as a Final state cannot be reached if the system is started in a state where

$$in\_env \neq out * buf * unseen$$

In general we should always construct NON_BREAKABLE_NDS designs. If a design can be proved to have the WELL_HALTING property only, then there may exist states - satisfying the invariant - from which the system may deadlock or loop indefinitely. This is indeed undesirable. Later modifications to the system, (e.g. addition of operations which can be proved to keep the invariant, but may take the system to new states not previously reachable from Initial) may affect the behaviour of remaining 'unmodified' parts of the system - and can cause deadlocks.

A 'breakable-nds' design shows that there are properties of the system which the designer has not observed or considered.

There are several techniques for removing deadlock-possibilities.

Given -

$$Final : \mathbb{P}(\{S\})$$
$$Step : \{S\} \leftrightarrow \{S\}$$
**where**
$$\{S\} - dom(Step) \not\subseteq Final$$

then there is a possibility for deadlock. This possibility must be detected and removed by deciding whether to -



1) accept the deadlock, i.e. extend the Final - states.
2) provide a deadlock detection solution, i.e. extend the relation Step by adding one or more new events.
3) remove the deadlock possibility by limiting the abstract space, i.e. add invariants.

as both -

$$\{S\} - dom(Step \cup new\text{-}function) \subseteq \{S\} - dom(Step)$$

and

$$\{S \mid new\text{-}invariant\} - dom(Step) \subseteq \{S\} - dom(Step)$$

the changed design of the system may well be provable to be NON_BREAKABLE.

### 3.3.2. Termination and the Variant Function.

The properties of a HALTING_NDS only guarantee termination provided the starting state was within the predefined Initial-states. As mentioned earlier we should strive to design NON_BREAKABLE_NDS , i.e systems which always terminate.

There is an important group of system components where a 'repeat' mechanism is required; e.g. a communication system should include facilities for *re-transmission* of messages which, due to unreliable communication-lines, have been lost; and a disc-handler should provide a *re-try* mechanism which will be used after the occurrence of a disc-read check-sum error.

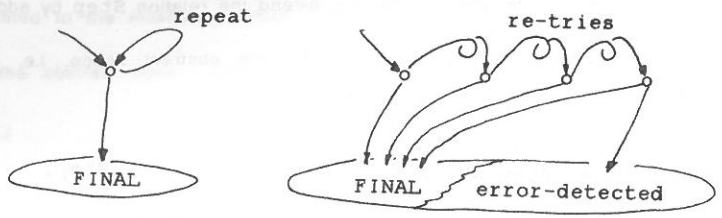
However this 'repeat' facility should not prevent the designer from constructing *totally correct* and non-breakable systems.

Consider the 'behaviours' illustrated in FIGURE(3.37).

a) A partially correct system, which 'only' fails to terminate WHEN the unexpected happens (loops infinitely).
b) The 'repeat' mechanism from system a) is replaced by a well-known mechanism, in which only a limited number of re-tries is performed before the system is stopped within the new, extended Final - state. This mechanism will behave identically for transient errors, but persistent errors will be detected. Note, this modification is done using a similar techniques as those applied in order to prevent deadlock (see previous page). This time in order to satisfy a termination-axiom-

$$dom(Cycle(\ a\text{-}relation\ )) = all\text{-}states$$

Repeat Mechanisms — FIGURE(3.37)




The loop-free property of a NON_BREAKABLE_NDS, requires a proof of-

$$\text{dom}(\text{Cycle}(\text{Step})) = X$$
$$\textbf{where } \text{Step} : X \leftrightarrow X$$

i.e. that the relation Cycle(Step) is a total relation.

The following theorem gives a proof method for finite branching relations-

VARIANT X — THEOREM(3.38)

NDS ⊢
$\text{dom}(\text{Cycle}(\text{Step})) = X \iff$
$(\exists\ v : X \to \mathbf{N})(\forall\ x,x' : X)\ (\ x\ \text{Step}\ x' \Rightarrow v(x') < v(x)\ )$
**where**
$\text{Step} = \cup\ \text{Events}$

**proof :**

a) If Cycle(Step) is total then a variant function exists :

(1) dom( Rank ) = X (DEF(3.14))

(2) $(\forall\ x,x' : X \mid x\ \text{Step}\ x')(\text{Rank}(x') < \text{Rank}(x))$ (DEF(3.15))

i.e. RANK is a variant function.

b) If a variant function for the relation exists then this variant function is indeed a variant function for all spanning-functions for Step , i.e.

(1) $(\forall\ sp : \text{Span}(\text{Step}))(\forall\ x : \text{dom}(sp))\ (v(sp(x)) < v(x))$

(2) $(\forall\ x : X)(\forall\ sp : \text{Span}(\text{Step}))(x \in \text{dom}(\text{loop}(sp))$ (next LEMMA)

(3) dom(Cycle(Step)) = X (TH (3.9))

The well-known theorem on the importance of the variant function for iterations of partial functions -



VARIANT X — LEMMA (3.39)

$f : X \nrightarrow X$
$v : X \to \mathbf{N}$
$(\forall\ x : \text{dom}(f))\ (\ v(f(x)) < v(x)\ )$
⊢
$(\forall\ x : X)(\exists\ n : \mathbf{N})(\exists\ y : X)\ (\ y = f^n(x) \wedge y \notin \text{dom}(f)\ )$

**proof :**

(1) for an arbitrary x : X we define the following partial function:

$g_x = (\lambda\ n : \mathbf{N} \mid f^n(x) \in \mathbf{N})(\ v(f^n(x))\ )$

because $(0, v(x)) \in g_x$ we can define :

$m_x = \min(\text{ran}\ g_x)$

$n_x = \min\ g_x^{-1}(\{m_x\})$

hence, we can define

(2) $y_x = f^{n_x}(x)$

under the assumption that

$y_x \in \text{dom}(f)$

we get (as v is a decreasing function)

$g_x(\ n_x+1\ ) < \min(\text{ran}\ g_x)$ ,

which is a contradiction, therefore

(3) $y_x \notin \text{dom}(f)$

hence according to (1), (2) and (3) we get

$(\forall\ x : X)(\exists\ n : \mathbf{N})(\exists\ y : X)\ (\ y = f^n(x) \wedge y \notin \text{dom}(f)\ )$

Taking the proof-method for proving freedom from loops into account, we can now reformulate our definition of NON_BREAKABLE_NDS -

NON_BREAKABLE_NDS X — DEF (3.40)

Events : $\mathbb{F}(\ X \nrightarrow X\ )$
Initial,Final : $\mathbb{P}(X)$
Var : $X \to \mathbf{N}$

$(\ X - \text{dom}(\ \cup\ \text{Events}\ ))\ \subseteq\ \text{Final}$
$(\forall\ f : \text{Events})(\forall\ x : \text{dom}(f))\ (\ \text{Var}(f(x)) < \text{Var}(x)\ )$

Example:

The statement of correctness for the small Buffer System (section 3.2) may now be formulated-

Buf_Sys — X — THEOREM (3.41)

```
in_env  : seq[X]
Bound   : N1
Events  : F( {BB} ⇸ {BB} )
Initial,
Final   : P( {BB} )
Var     : {BB} → N
Events  = { (λ PRE_INPUT)(μ BB')(INPUT),
            (λ PRE_OUTPUT)(μ BB')(OUTPUT) }
Initial = { BB | out,buf = ⟨⟩ }
Final   = { BB | unseen,buf = ⟨⟩ }
Var     = (λ BB) ( 2 * #(unseen) + #(buf) )
⊢
NON_BREAKABLE_NDS[ {BB} ]
```

Note! the proof of the total correctness of the Buffer System ( which was conditional in THEOREM(3.24) ) may now be completed by proving that the given variant function is indeed a variant function for all participating events.

We must prove-

Buf_Sys_Variant — X — LEMMA (3.42)

```
PRE_INPUT; INPUT    ⊢
(2*#(unseen')) + #(buf') < (2*#(unseen)) + #(buf);
```

Buf_Sys_Variant — X — LEMMA (3.43)

```
PRE_OUTPUT; OUTPUT  ⊢
(2*#(unseen')) + #(buf') < (2*#(unseen)) + #(buf);
```

LEMMA(3.22) and LEMMA(3.23) guarantee that

```
Events  ∈  F( {BB} ⇸ {BB} )
```

We can prove the deadlock-free property -

```
( X - dom( ∪ Events )) ⊆ Final
```

as in THEOREM(3.24) or by proving -

Buf_sys_termination — X — LEMMA (3.44)

```
in_env   : seq[X];
Bound    : N1;
BB;
(unseen=⟨⟩  ∨  #(buf)=Bound) ∧ buf=⟨⟩;
⊢
( unseen=⟨⟩ ∧ buf=⟨⟩ ∧ out=in_env )
```



which expresses that the Buffer System terminates within the acceptable final states only.

The method illustrated here for proving non-deterministic designs correct can be summarised as follows:

1) State the NON_BREAKABLE properties as in THEOREM(3.41).
2) Prove the Invariant Lemmas, (see LEMMA(3.22-23)).
3) Prove the Variant Lemmas, ( LEMMA(3.42-43)).
4) Prove the Deadlock-Lemma, ( LEMMA(3.44))

Even for small systems it may be difficult to find a Variant Function. The reason may be that the *Invariants* which constrain the state-space are insufficient, whence we are faced with another difficult task - to find proper Invariants.

However, it is easy to prove or disprove that a particular function is a Variant Function for a given set of partial functions.

# 4. SPECIFICATIONS OF DISTRIBUTED SYSTEMS.

The remaining parts of this thesis present a new approach for specifying *DISTRIBUTED SYSTEMS.*

Distributed systems are systems which are composed of several concurrently operating processing components. As mentioned in chapter 3 these systems fall into the category of *non-deterministic systems*. The formal framework which we developed in the previous chapter will be applied to the analysis of distributed systems, and we will see (in section 4.4) how the difficulties in proving termination (i.e. finding a variant function) are reduced owing to the method used for specification of the behaviour of such systems.

## 4.1.MOTIVATION FOR DISTRIBUTION.

A specification of a distributed system must document the decisions taken with respect to the distribution of data- and control-structures onto independent processing units. The reasons for designing distributed systems may vary:

a) The decisions may be directly dictated by demands with respect to *security* (e.g. certain data must only be available at a specific geographical location) or *reliability* (e.g. the introduction of redundancy).
b) The decisions may also be made by the designer in order to satisfy *efficiency requirements* (e.g. 1) in a computer network the response time for accessing data is dependent on the 'distance' to the location of the accessed data, and 2) the throughput of a system can be increased by utilising a *piped* processor arrangement.)
c) *Maintenance* and *modification* of running systems are other factors which may affect decisions with regard to the physical distribution of control- and data-structures.
d) Finally, a decision to provide a distributed system for solving a specific task may simply be made because of the availability of 'off the shelf' hardware- or software-units (e.g. special purpose microprocessors).

The approach presented here for documenting the distribution is based on the classical idea [Parnas,20] of specifying a system in terms of descriptions of its subcomponents and the relationship between those subcomponents. In our approach each component of the *description* corresponds to a single component or a set of components in the *realisation.*

## 4.2. THE INDEPENDENT MODULE.

When describing a subcomponent of a system it is essential to draw a distinction betwen its externally observable behaviour and its internal composition. If we have a complete description of a component's *external connections* and a description of the behaviour along those connections there is nothing more of interest we can learn by looking at the inside of the component. The main problem, however, is to describe the external behaviour entirely in terms of those connections, i.e. without any reference to the internal structure. For example, the specification of external behaviour of the *Buffer* from section 3.2 does not qualify as it involves an abstract structure (buf) which represents the inside of the described component.

### 4.2.1. The Classical Independent Module.

The classical independent module can be described through an *input output* relationship, i.e. in terms of its external behaviour only. A specification of a module through an input-output relationship gives a complete description of the behaviour of a realisation of such a module because we assume that any realisation follows two principles:

1) there are only two kind of external connections - *input* and *output* - and these connections are *activated* alternately.

2) any output value produced is exclusively derivable from the module's functional definition and from the value of the most recent input presented, i.e. the module is *stateless*.

### 4.2.2. The Process Module.

The modules under consideration here do not follow either of these simple principles. For the modules in question the ability to participate in a communication along an external connection is not governed by a simple data-independent control mechanism, and a value communicated along a connection is not determined solely by the most recent values communicated along another set of connections. At any point in time both the ability to communicate and the actual value communicated along a particular connection can be dependent on information in the *past history* of the behaviour along any set of external connections. The distinction between the classical independent module and the modules we are interested in - hereafter called *Process Modules* - can be compared to the distinction between combinatorial logic circuits (e.g. AND-gates) whose output depends only on the most recent input, and sequential logic circuits whose output depends on past input (the history of the input activity), e.g. the output of a flip-flop is determined by the parity of the number of input-signals previously transmitted to it.



NB, a reasonable realisation of a module whose behaviour is dependent on its past behaviour memorises only a minimal set of facts about the past only - e.g. the flip-flop has a 1-bit memory which represents the *parity of the number of input signals*.

The independent module's behaviour can be specified by giving:
1) The type of input and output parameters.
2) A functional definition of the output produced, i.e.

```
output = F(input)
```

The process module's behaviour can be specified by giving:
1) The type of the objects which can be communicated along its external connections.
2) A set of axioms involving one or more of the past histories of the behaviour along those external connections.

Example:
We will give a specification of a process (P1) with two external connections (in and out) which for each element (x) communicated along the input connection (in) will produce a single result (F(x)) along the output connection (out) before any new input can be consumed.
Note the similarities with the assumptions made for a realisation of independent memoryless modules.

Given F : X → Y, we will define the behaviour of our process by, 1) modelling the history of the communications along the input connection as a *sequence* over elements of type X, 2) modelling the history of the communications along the output connection as a *sequence* over elements of type Y and 3) giving two axioms involving these abstractions. We have-

```
P1 ____________________ X  Y ____________________ DEF(4.1)
|     in  : seq[X];
|     out : seq[Y]
|-------
|     #in - #out ≤ 1;
|     out ⊆ F • in
|_____________________________________________________
```

where the first axiom restricts the number of computations P1 can carry out 'simultaneously'. In this example only one computation can be carried out at a time, hence the input connection will be *blocked* after a value has been passed along the input connection and the corresponding result not yet produced along the output connection. The second axiom states that all values produced by P1 can be obtained by applying F to a previously received input element.

Assume that we have observed

a, b, c, and d

being communicated along the in connection in that order, then

$$in = \langle a\ b\ c\ d \rangle$$

The history of the communication along the out connection must then be

$$\langle F(a)\ F(b)\ F(c) \rangle$$

or

$$\langle F(a)\ F(b)\ F(c)\ F(d) \rangle$$

if

$$out = \langle F(a)\ F(b)\ F(c) \rangle$$

further input must be rejected as the occurrence of an input will invalidate the first of the axioms. However P1 may produce F(d), as this event will not invalidate any of the given axioms.

We shall in the following use a graphical notation to give an informal overview of the interaction between the process-modules of a system.

The process P1 will be represented by

FIGURE 4.2


which may be interpreted either as a *box* representing process component which realises module P1, or simply as an illustration of the fact that P1 is a description which involves axioms on in and out only.

### 4.2.3. The Readiness Principles.

A realisation of a process module must - just like a sequential realisation of a classical independent module - follow certain principles. As a process module contains information concerning the scheduling of the activities along its external connections the principles take a different form from those for the independent module.

The principles are:

1) At any point in time the realisation of a process-module must not participate in a communication (i.e. extend a history) if the occurrence of this communication would invalidate the axioms given (for the histories of the activities along the external connections). We might say - 'only communications which are permitted to happen can happen'.



2) If an occurrence of a communication along a specific connection can take place without invalidating the external invariants then the realisation of the module must eventually, and without further activity along other connections, be willing to participate in this particular communication. We might say - 'an event which is expected to occur can eventually take place.'

No attempts will be made in this thesis to formalise these principles, for example by giving proof-rules for a specific realisation technique. We will concentrate on discussing the correctness, or rather the consistency, of specifications. However in order to illustrate how these principles can be used in statements about the correctness of an implementation we will give a simple realisation of module P1.

A realisation is outlined using the tasking facilities of the programming language *Ada*.

P1 — X Y — EXAMPLE (4.3)

```
task body P1 is
 x' : X ;  y'  : Y;
 function F(x : in X) return Y is ...;
begin
  loop
    accept input(x : in X) do x':=x ; end ;
      y':= F(x');
    accept output(y : out Y) do y:=y' ; end;
  end loop;
end;
```

This is a realisation of the process-module P1 because

1) before the occurrence of an *input* we can assert that
$$\#in - \#out = 0$$
hence an occurrence of an input will not invalidate the axioms of the specification P1.

2) before the occurrence of an *output* we can assert that
$$\#in - \#out = 1 \quad ,$$
$$y' = F(last(in)) \qquad \text{and}$$
$$out = beginning(in)$$
hence the occurrence of the output will not invalidate the axioms of P1 as the occurrence of output will update the history of communications along the out connection as follows :
$$out := out * \langle F(last(in)) \rangle$$

3) after the occurrence of an input we have
$$\#in - \#out = 1$$
hence we expect only an output to happen, which in fact the realisation will eventually be willing to perform.

4) after initiation of the task, as well as after the occurrence of an output, an input is expected to happen - which the realisation will eventually be willing to participate in.

1) and 2) ensure that the first of the principles is followed, while 3) and 4) ensure that the second principle is followed.

The example presented here as an introduction to the approach of specifying distributed systems could have been described simply by giving its functional specification, and by assuming that the module would be realised as a conventional *memoryless* software module. However, as we shall see in the next section, a small modification will give a specification of a module with a different dynamic behaviour - a behaviour which cannot be described through a simple input output relationship.

### 4.3. DISTRIBUTED SPECIFICATIONS AND DISTRIBUTED REALISATIONS.

Consider the specification -

P2 — X Y — DEF (4.4)

```
in : seq[X];  out : seq[Y]
-----------------------------
δ ≤ 2 ;  out ⊆ F • in
where
δ = #in - #out
```

which is a specification of a process which computes the function F. However in contrast to the process P1 two computations can be carried out simultaneously. A realisation of this module does not follow the principles given for the classical independent module, e.g. whenever

```
#in - #out = 0
```

the process may participate in two consecutive inputs.

A realisation could be -



P2 — X Y — EXAMPLE (4.5)

```
task body P2 is
 i,o,δ : INTEGER := 0;
 queue : array (0..1) of X;
 function F(x : in X) return Y is ...;
begin
 loop
  select
    when δ < 2 ⇒ accept input(x : in X) do
                   queue(i) := x;
                 end;
                 δ := δ+1; i := (i+1) mod 2;
    or
    when δ ≠ 0 ⇒ accept output(y : out Y) do
                   y := F(queue(o));
                 end;
                 δ := δ-1; o := (o+1) mod 2;
  end select;
 end loop;
end P2;
```

or if we can decompose the computation of F - i.e. we have

$$F = H \bullet G$$

we might decide to provide the following *distributed* realisation -

P2 — X Z Y — EXAMPLE (4.6)

```
task body P2_1 is
  t : X; z : Z;
  function G(x : in X) returns Z is ...;
begin
   loop accept input(x : in X) do t:=x ; end;
        z := G(t);
        P2_2.internal_com(z);
   end loop;
end P2_1;
```

```
task body P2_2 is
  t : Z; y' : Y;
  function H(z : in Z) return Y is ...;
begin
   loop accept internal_com(z : in Z) do t:=z ; end;
        y' := H(t);
        accept output(y : out Y) do y:=y' ; end;
   end loop;
end P2_2;
```

EXAMPLE(4.5) gives a *sequential non-distributed* realisation of DEF(4.4). The external operations (input and output) will always happen in sequence although the actual order in which they happen is unpredictable (the run-time environment

determines the order). EXAMPLE(4.6) gives a *non-sequential distributed* realisation of the same specification. The actual occurrence of input and output cannot be related (in time) unless there are some (unknown) constraints which synchronise the part of the environment which produces input with the part which consumes output.

Consider the *distributed* realisation of P2 - in the development (construction) of the task P2_1 it is of no importance how the 'inside' of P2_2 is constructed. The only part of P2_2 which is of interest in developing P2_1 is the common interface between these two processes. P2_1 can be viewed as a realisation of the specification-

P1_1 [X Z] DEF (4.7)

```
in : seq[X];  com : seq[Z]
-----------------------------
#in - #out ≤ 1;  com ⊆ G • in
```

and P2_2 can be viewed as a realisation of the specification-

P2_2 [Z Y] DEF (4.8)

```
com : seq[Z]; out : [Y]
-----------------------------
#com - #out ≤ 1; out ⊆ H • com
```

The following *compound* specification

P2-Dis [X Z Y] DEF (4.9)

```
P2_1[X,Z] ; P2_2[Z,Y]
```

which expands into

P2-Dis [X Z Y] DEF (4.10)

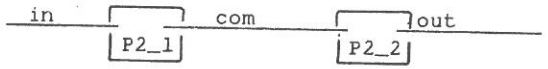
```
in : seq[X]; com : seq[Z]; out : seq[Y]
-----------------------------
#in - #com  ≤ 1;  com ⊆ G • in;
#com - #out ≤ 1;  out ⊆ H • com;
```

is a *distributed specification* as the set of axioms of the description can be divided into two groups, a group which involves the in and com connections only and a group which involves the com and out connections only.

We can use our graphical notation to illustrate the distribution-



FIGURE (4.11)




This illustration has an obvious alternative interpretation: two processes communicating along the connection com.

We may attempt to prove that the *distributed realisation* (EXAMPLE(4.6)) is a valid implementation of its *non-distributed specification*, - which means that we must demonstrate that the realisation follows the *readiness principles* with respect to its specification DEF(4.4). Such a proof will not be straightforward since the correctness of, for example, the task P2_1 (i.e. a proof that it behaves correctly along its connection in) depends on the correctness of the task P2_2. Furthermore even if we assume the correctness of P2_2 a proof of P2_1 cannot be carried out without referring to the internal behaviour of the realisation of P2_2. This is indeed undesirable as 1) independent development of subcomponents is impossible and 2) the future modification of the local mechanisms of any module may invalidate global proofs or proofs for other modules.

Proofs should be localised, i.e. the tasks P2_1 and P2_2 (from EXAMPLE(4.6)) should be verified against the specifications DEF(4.7) and DEF(4.8) respectively. However proving that the two tasks are 'locally' correct does not guarantee that the tasks - when running in parallel - realise the non-distributed specification DEF(4.4).

Such a guarantee can be gained by proving that the distributed specification DEF(4.9) is a *valid decomposition* of the non-distributed specification DEF(4.4). A decomposition is valid if, from the sole assumption that each process-component follows the *readiness principles* with respect to its local specification, it can be proved that the processes when co-operating will follow the readiness principles with respect to the global specification. It is the designer's responsibility to provide such a proof.

Decomposition (or distribution) of specifications will be treated in detail in section 6 of this thesis.

Obviously the correctness of the behaviour of a system which consists of interacting processes relies on more than just the proofs made with respect to decomposition at the abstract level. We must have a guarantee that the processes of a realisation are properly interfaced, i.e. co-operating processes must agree on a common communication protocol. Furthermore we must be certain that the interprocess control mechanisms provide an appropriate service. The communication protocol selected by the *implementor* and the mechanisms for communication provided by the underlying machinery must guarantee that, a) a communication can only take place between processes if all involved processes are willing to participate, and b) at least one of the 'currently' possible communications eventually takes

place. These aspects of the correctness of a system reflect the obvious assumption a designer makes about future realisations; they will be referred to as the *connectivity assumptions* and will not be treated further in this thesis.

The techniques presented in this section provide the designer with a method for describing dynamic behaviour at an abstract level, i.e. it is possible for him to express WHAT (i.e. which operations) will take place WHEN without determining the HOW. Furthermore by *distributing* his specification (*partitioning* his design) he can document his decisions about WHERE the operations will be performed. In section 4.5 we will demonstrate how such *distributed specifications* can be used as a basis for the analysis of dynamic behaviour.

## 4.4. ENVIRONMENTS FOR SYSTEMS.

A precise description of the environment in which a system is expected to operate is as important as a description of the subcomponents of which the system consists. The environment as well as any subcomponent of a system must obey certain well-documented rules. The behaviour of the environment for a system can be described as a separate independent system, i.e. as a set of assertions involving the past histories of the communications along the connections to the system for which it serves as an environment. The environment may consist of several independent parts and they should, if this is the case, be described using a distributed formulation.

A description of a system and an environment can be 'joined' together - thereby giving a complete description of the activity along the connections between the system and its environment.

As mentioned in section 3.3. we will investigate the behaviour of systems in *finite environments* only, i.e. we assume that the part of the environment which initiates new activities within the system only performs a finite number of requests. Such an 'inputting' environment must cease to communicate after having participated in a finite number of communications.

In the following we will give examples of environments, some of which will be used as environments for the systems presented in chapter 5.

### 4.4.1. A Requesting Environment.

Consider an environment which issues commands along a single connection to a system. The environment never commits itself to one particular command, but will be capable of 'participating' in any command which satisfies certain criteria. The environment promises to issue a limited number of requests.

Given a set of commands R

and a limit L : ℕ1



then we can formalise such an environment as follows

REQUESTER — R — DEF (4.11)

```
requests : seq[R]
-----------------
#requests ≤ L
```

Example:
REQUESTER may describe a human user of an interactive operating system who requests certain services - save-a-file, edit-a-file, send-a-message, etc. The user communicates with the *front-end* of the command interpreter, which ensures that only intelligible commands are passed between the user and the operating system.

Given intelligible_cmds : ℙ(R), then the description of the front-end module might be

FRONT END — R — DEF(4.13)

```
requests : seq[R]
-----------------
ran(requests) ⊆ intelligible_cmds
```

While the FRONT-END ensures that unintelligible commands are rejected, the user (REQUESTER) promises that his session is finite, i.e. when

#requests = L

he will not issue any more requests.

### 4.4.2. A Selfcommitting Environment.

Consider a component of an environment which communicates a sequence of data-elements or requests along a single connection. At any point in time this component commits itself to participate in one particular communication only.

Given

in_env : seq[E]

then the environment which transmits elements from ran(in_env) along a connection in in the order they appear in the sequence in_env, can be described as-

PRODUCER — E — DEF (4.14)

```
in : seq[E]
-----------
in ⊆ in_env
```

A PRODUCER environment may be used to describe:
a) A card-reader connected to a batch operating system.
b) A programme which 'sends' a predefined sequence of (machine-) instructions to a processor.

If the system receiving requests from a PRODUCER environment cannot recognise the element currently presented by this environment, then no further interaction between the system and this particular environment can take place.

example:
a) If the sequence of cards in a card-reader does not conform to some agreed rules then the task of reading the cards can only be terminated *abnormally*.
b) A program containing unrecognisable op-codes will be *aborted* by the processor.

(NB, a REQUESTER environment is expected to 'try again' if a particular request is rejected)

### 4.4.3. An Interactive Environment.

An environment may interact with a system along several connections, and the environment's activity along one connection may be influenced by previous activity along other connections.

Example:
Consider an environment from which *jobs* (J) may *enter* into a system. The jobs may eventually *leave* the system. The obvious rule imposed by the environment is that only jobs outside the system can *enter* or *re-enter*.

The following specification

INTERACT[J] DEF (4.15)

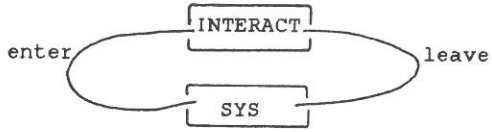
```
enter, leave : seq[J]
-----------------------------------------------
(∀ j:JOBS)
    ( card(enter↓{j}) - card(leave↓{j}) ≤ 1) )
```

describes an environment which guarantees that only jobs outside the system will enter.

Assume we have a description SYS of a system with the external connections *leave* and *enter*. The compound system consisting of this SYS and INTERACT can be illustrated as:



FIGURE (4.16)




## 4.5. AN ANALYSIS OF THE BEHAVIOUR OF DISTRIBUTED SYSTEMS.

The purpose of a description of a module is twofold:
1) The description should provide the *implementer* or the *designer* with sufficient information to *construct* the *inside* of the module. The way in which the implementor selects a *state* and constructs an *algorithm* in order to realise a specification was informally illustrated in section 4.3 (see the realisation EXAMPLE(4.5) of the specification DEF(4.4) ). The module, however, does not have to be implemented as a *sequential process* (see EXAMPLE(4.6)). The designer may *propose* a distributed realisation by decomposing his specification.

2) The description should allow the designer to *analyse* the module's behaviour in co-operation with other modules before taking any decisions with respect to the internal structure of that module.

The second of these objectives will be treated in this section.

In the following we will see how important characteristics of a modules behaviour in co-operation with other modules can be derived solely from its specification, i.e. knowledge about its *externally observable behaviour*. The dynamic properties of a module which are proved as a consequence of the decisions taken with respect to its external behaviour are *inherited* by any implementation of that module, i.e. these properties do not have to be re-proved when the designer or implementor settles on a particular realisation.

The dynamic properties of a system which will be discussed are:
1) Termination Properties.
   Can a system whose constituents are realisations of well-defined modules be proved to terminate (solely by referring to the specification of the modules)?
2) Deadlock Properties.
   Can we prove that such a system only terminates in a predefined acceptable state?

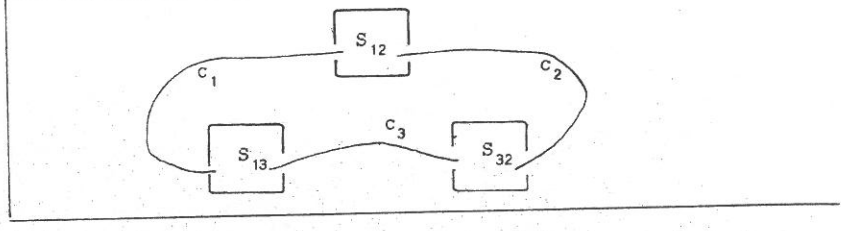
Chapter 3 comprehensively discussed these properties for *sequential non-deterministic systems*. Distributed systems which consist of several independent processing components cannot however be characterised as sequential systems.

Operations within such a system may happen in *parallel*, simply because operations are unrelated.

In order to discuss the dynamic properties of distributed systems we base our dynamic model on the *OBSERVATIONS* of the behaviour. Without any lost of generality observations can be sequentialised; hence we can apply the formal framework developed in chapter 3.

Example:
Consider a description of a system with 3 visible connections-

FIGURE (4.17)




The information which can be externally observed to change is-

CHAN — M — DEF (4.18)

$c_1, c_2, c_3$ : seq[M]

where M denotes the messages communicated between the modules, and $c_1$, $c_2$ and $c_3$ denote the history of observations along the visible connections.

$S_{ij}$ is the specification of a module which communicates along $c_i$ and $c_j$. We have a guarantee (if the realisation is correct) that the communications along $c_i$ and $c_j$ will not invalidate $S_{ij}$. Hence if the system is started in a state satisfying $S_{12}$, $S_{32}$ and $S_{31}$ then these will be satisfied in any state the system can ever reach. The system's observable states can therefore be described as-

SYSTEM — M — DEF (4.19)

CHAN[M]

---

$S_{12}$; $S_{23}$; $S_{31}$

When a communication takes place, a *single message will be appended to an existing history*, hence an occurrence of a communication along the visible connection $c_1$ can be described as-



M — DEF (4.20)

$hist_1 : M \rightarrow ( \{CHAN\} \rightarrow \{CHAN\} )$

---

$(\forall\ m{:}M)$
$hist_1(m) =$
$(\lambda\ CHAN)(\mu\ CHAN')(c_1' = c_1{}^\frown\langle m\rangle;\ c_2' = c_2\ ;\ c_3' = c_3\ )$

Furthermore, if the system is started in a state satisfying the axioms of SYSTEM then only communications keeping the axioms invariant will take place (according to the *readiness principles* in section 4.3.2). Hence we can refine our description of the communication along connection $c_1$ by *restricting* its *domain* and *co-domain* as follows-

M — DEF (4.21)

$obs_1 : M \rightarrow ( \{SYSTEM\} \nrightarrow \{SYSTEM\})$

---

$(\forall\ m{:}M)( obs_1(m) = (hist_1(m) \upharpoonright \{SYSTEM\}) \downarrow \{SYSTEM\} )$

Assuming that the number of messages is finite i.e. -

$MES : \mathbb{F}(M)$

then the total behaviour of SYSTEM can now be described as-

OBSERVATIONS — M — DEF(4.22)

$Events : \mathbb{F}(\{SYSTEM\} \nrightarrow \{SYSTEM\})$

---

$Events = \cup\{obs_1(m) \mid m{:}MES\} \cup$
$\cup\{obs_2(m) \mid m{:}MES\} \cup$
$\cup\{obs_3(m) \mid m{:}MES\}$

So far we have a description of a state (DEF(4.19)) and a description of the transformations of that state (DEF(4.22)). By giving a description of the acceptable final states we can discuss the NON-BREAKABLE properties as stated in DEF(3.33) or DEF(3.40).

Let

$Step : \{SYSTEM\} \leftrightarrow \{SYSTEM\}$
$Step = \cup\ Events$

and

$Final : \mathbb{P}(\{SYSTEM\});$
$Initial : \mathbb{P}(\{SYSTEM\})$

then *termination* is guaranteed if we can find a variant function, i.e. if

$$Var : \{SYSTEM\} \rightarrow \mathbb{N}$$
$$(\forall s,s' : \{SYSTEM\})(\ s\ Step\ s' \Rightarrow Var(s') < Var(s))$$

and the system will be *deadlock-free* if

$$\{SYSTEM\} - dom(Step) \subseteq Final$$

NB.

1) Both the initial and the final state of the system are expressed in terms of the history of the visible behaviour.

2) The initial state for most of these systems will be a state where no communication has taken place, i.e. all history components of the state are *empty sequences*. For SYSTEM we have-

$$Initial = \{SYSTEM \mid c_1, c_2, c_3, = \langle\rangle\}$$

3) The proof of termination relies on the existence of a variant function; for dynamic behaviour based entirely on the history of observations it is normally straightforward to find a variant function.
E.g. for the system presented here we have-

$$inc\_func = (\lambda\ SYSTEM)(\#c_1 + \#c_2 + \#c_3\ )$$

which obviously satisfies

$$(\forall\ s,s' : \{SYSTEM\})$$
$$(s\ Step\ s' \Rightarrow inc\_func(s') > inc\_func(s)$$

hence the existence of a decreasing variant function relies on the existence of an *upper limit* for the total number of communications which can take place within the system. The existence of the upper limit will normally rely on an *assumption* about the environment.

4) A discussion of the deadlock properties of a system relies on the *consistency* of the decisions taken with respect to the dynamic behaviour (i.e. the *predicates* of the *histories*) and the definition of the final state. The next chapter will demonstrate how the *consistency check* is carried out.



# 5. EXAMPLES OF SPECIFICATIONS.

The approach presented in the previous chapter is intended to be applicable to real-life engineering problems in the area of *distributed* computing. It is not, however, intended to be suitable for examining all aspects of big distributed systems. The properties which will be documented and analysed are only those which concern co-operation between independent process components, that is, the approach will be used to give a precise formal description of the decisions made with respect to the distribution of data- and control-structures between the process components on the one hand, and a precise formal description of the interaction between these components on the other.

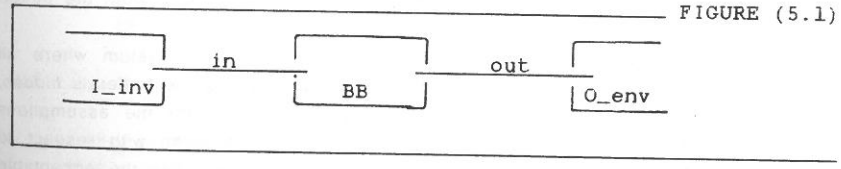
In this chapter the approach is illustrated through a set of simple examples. In order to compare the new approach, where behaviour is described through predicates on the histories of observed communications, with the more traditional approach, where behaviour is described through a relation (or a set of partial functions) over an abstract state, section 1 gives a new specification of the Bounded Buffer (from section 3.2.). The succeeding three sections will discuss miscellaneous systems (buffers and queues) and concepts (mutual exclusion, deadlock etc.) which are of concern when designing distributed systems. The final section will give a specification of a communication network.

## 5.1. A BOUNDED BUFFER.

The informal description of the bounded buffer given in the beginning of section 3.2. was used to justify the design of a small system which was described in terms of 1) an *abstract state* (BB:DEF(3.17)) representing the inside of a Bounded Buffer through a buf component and 2) two *state transformation* functions (DEF(3.18-19)). Using the new approach we can give a description which replaces the informal specification in section 3.2.. We can formally describe a bounded buffer without suggesting an internal state for the buffer and without (misleadingly) assuming that only two *indivisible* operations make up the *dynamics* of the system.

The Bounded Buffer system can be illustrated as follows-

FIGURE (5.1)




i.e. the system consists of three independent operating components, two of which belong to the *environment*.

We have-

1) The input environment is described as—

$I\_env[X] \triangleq [\ in{:}seq[X] \mid \#in \leq Max\ ]$ DEF(5.2)

for some

$Max \in \mathbf{N}1$

We assume that the inputting environment produces only a finite number of elements.( we might have used REQUESTERDEF(4.11)).

2) The buffer-system is described as-

$BB[X] \triangleq [in,out{:}seq[X] \mid out \sqsubseteq in;\ \#in-\#out \leq B]$ DEF(5.3)

for some

$B \in \mathbf{N}1$

which ensures that-

a) The order in which the elements are being produced along out is identical to the order in which they were consumed.
b) only a limited number of elements can be within the buffer at any time.

3) The part of the environment which consumes elements from the buffer can be described as-

$O\_env[X] \triangleq [\ out{:}seq[X]\ ]$ DEF(5.4)

i.e. an environment which accepts any element from X at any point in time.

4) The total system is described through the compound-

$BB\_SYS[X] \triangleq [\ I\_env[X];\ BB[X];\ O\_env[X]\ ]$ DEF(5.5)

An *acceptable final state* for the system can be described as-

$BB\_Final[X] \triangleq [BB\_SYS[X] \mid \#in=Max;\ in=out\ ]$ DEF(5.6)

i.e. a state where

a) all input submitted to the buffer from the environment is consumed by the buffer.
b) all input consumed by the buffer has appeared as output from the buffer.

BB_SYS and BB_Final constitute a description of a buffer system where all information except the *interface* between the environment and the buffer is *hidden*. I_env and O_env are components of BB_SYS and document the assumptions made about the environment. BB describes the decisions taken with respect to the component we are about to implement. BB_Final describes the acceptable termination-states of that system.



The *consequences* of the decisions taken so far are:

1) The occurrence of a (visible) communication along the in connection can be described as-

$input[X] =$
$\quad (\lambda\ x : X)$
$\quad\quad (\lambda\ BB\_SYS)(\mu\ BB\_SYS')(in'=in^{\frown}\langle x\rangle;\ out'=out)$

DEF(5.7)

NB. compare this definition with DEF(4.21).

A communication along the out connection can be described as-

$output[X] =$
$\quad (\lambda\ x : X)$
$\quad\quad (\lambda\ BB\_SYS)(\mu\ BB\_SYS')(in'=in;\ out'=out^{\frown}\langle x\rangle)$

DEF(5.8)

The domains of these operations can be derived from the axioms of the BB description.

Example:
The element x can be observed to be consumed by a system b_sys only if

$b\_sys : \{BB\_SYS\}$

and

$b\_sys \in dom(input(x))$

We have -

a) An input can occur whenever we are in a state belonging to

$\cup\{\ dom(input(x)) \mid x : X\ \} =$
$\{BB\_SYS \mid \#in+1 \leq Max\ ;\ (\#in+1) - \#out \leq B\ \} =$
$\{BB\_SYS \mid \#in < Max;\ \#in-\#out < B\ \}$

and

b) An output can occur whenever we are in a state belonging to

$\cup\{\ dom(output(x)) \mid x : X\ \} =$
$\{BB\_SYS \mid (\exists\ x : X\ )(\ out\ ^{\frown}\langle x\rangle \sqsubseteq in)\ \} =$
$\{BB\_SYS \mid out \neq in\ \}$

2) The system *terminates* because we can present a decreasing variant function

$Var_{BB} : \{BB\_SYS\} \rightarrow \mathbf{N}$
$Var_{BB} = (\lambda\ BB\_SYS)(2*Max - \#in - \#out)$ DEF(5.9)

3) The system is *deadlock free.*
The system will terminate in a state which can be described as-

```
Terminal =
{BB_SYS} - ∪{ dom(input(x))  | x:X }
         - ∪{ dom(output(x)) | x:X }
=
{BB_SYS} - {BB_SYS | in<Max; #in-#out<B }
         - {BB_SYS | out≠in }
```

and the system is deadlock free if

```
Terminal ⊆ Final
```

which is a direct consequence of the following theorem-

THEOREM (5.10)

```
        X
BB_SYS[X] ⊢
(in≥Max ∧ out=in ) ∨ (#in-#out ≥ B ∧ out=in)
⇒
(#in=Max ∧ in=out )
```

( the proof relies on B being *non-zero* )

This completes the *consistency-check* for the decisions taken at the top level of design for a bounded buffer.

NB.
The systems we are concerned with will normally have an additional important property - they are *restartable.* If input is temporarily *blocked* (no requests are submitted to the system for some time) then the system will *deadlock* in a state in which new input must be accepted. Furthermore in such a state all received input must have been output (i.e. all requests have been fulfilled). This property can be formally stated, for BB_SYS, as

```
{BB_SYS} - dom(∪{output(x)|x:X})  ⊆
           dom(∪{input(x)|x:X}) ∩ {BB_SYS|in=out}
```

For BB_SYS the *restartable* theorem is-

Restartable THEOREM (5.11)

```
        X
BB_SYS[X] ⊢
in = out ⇒ (#in-#out < B) ∧ (in=out)
```

The approach for *specifying* distributed systems, and for checking whether the decisions taken are *consistent* can be summarised as follows:



1) First the designer should decide at which level of detail he wants to present the system. The system is then *partitioned* by the choice of *interfaces* across which we may have *connections.*

2) Each part of the system is then described in terms of invariant conditions on the history of the communication along a chosen set of connections.

3) The *acceptable final states* for the system are described using predicates on the histories of communications.

4) *Termination* is proved by presenting a *variant function.*

5) The *consistency-check* is done by either
   a) proving a deadlock-free theorem, or
   b) proving a restartable theorem.

## 5.2. MISCELLANEOUS SYSTEMS.

In order to illustrate the use of the assertional technique for describing behaviour a collection of specifications for some simple, well-known process-components is given in the following sections.

### 5.2.1. Buffers.

We weaken in turn each of the assertions given for the bounded buffer described in the previous chapter, and get-

a) An unbounded fifo.

```
FIFO[X] ≙ [in,out:seq[X] | out⊆in ]                DEF (5.12)
```

By placing this component in the environment proposed for the bounded buffer of section 5.1. we get-

```
FIFO_SYS ≙ [ I_env; FIFO; O_env ]                  DEF (5.13)
```

With the following definition of *acceptable final* states-

```
FIFO_FINAL ≙ [ FIFO_SYS | #in=Max; in=out ]        DEF (5.14)
```

we can prove that the compound system FIFO_SYS will terminate and is deadlock-free.
In terms of final results the two systems BB_SYS and FIFO_SYS are identical, i.e.

```
{BB_FINAL} = {FIFO_FINAL}
```

However the two systems are not identical as they have been specified to have different dynamic behaviour. E.g. in a state where

```
#in - #out = B
( B is the bound for BB_SYS )
```

then a BB_SYS will *reject* input until an output has occurred, while the FIFO_SYS can *accept* an input.

b) An Unordered Buffer.

```
Unordered_Buffer[X] ≙
[in,out:seq[X] | (∀x:X) card(out↓{x})≤card(in↓{x})]
                                                    DEF (5.15)
```

An Unordered_Buffer behaves like an unbounded buffer along the in connection. However while the output produced by the FIFO_SYS at any point in time can be determined to be-

```
in(#out + 1)
```

the Unordered_Buffer component can produce any element x for which-

```
card(out↓{x}) < card(in↓{x})
```

c) A Bounded Unordered Buffer-

```
Bounded_Unordered_Buffer[X] =
 [Unordered_Buffer | #in - #out ≤ B ]          DEF (5.16)
```

for some B : **N1**

This buffer behaves like a bounded buffer along the in connection, and like an unordered buffer along the out connection.

### 5.2.2. Queues.

Jobs within an operating system are given unique identification codes in order for the different parts of the system (schedulers for controling the sharing of resources) to distinguish between those jobs. The histories of the *arrival* and the *departure* to and from the queues of jobs (wait-queues, ready-queues, spooling-queues) within an operating system can therefore be described as-

```
Id_seq[X] = { s:seq[X] | s⁻¹ ∈ X ⇸ N }
```

Hence for any connection or channel in such a system,

```
chan : id_seq[X]
```

the function defined as

```
rel_time : X ⇸ N
rel_time = chan⁻¹
```



gives the relative time of the occurrence of communications along chan.

The id_seq will in the following be used to describe the behaviour of various queues. Note that the following descriptions do not include a description of the environment, thus a complete discussion of the behaviour of the component presented cannot be given.

a) Unordered Queue.

```
QUEUE[J] ≙
[arrive,depart:id_seq[J] | ran(depart) ⊆ ran(arrive)]
                                                    DEF (5.17)
```

which states the obvious property that only jobs which have arrived in a queue can depart from it. Note that the time of departure is not related to the time of arrival. A job within a QUEUE may have its departure delayed unreasonably often, because as long as more than one job is within the queue it is non-deterministic which job actually leaves the queue, hence if the rate of input is high a job can be passed over time after time. By adding a description of priority mechanisms to the QUEUE specification we can impose a fairer scheduling for departures.

b) Ordered Queues.

```
FIFO_QUEUE[J] ≙
     [ QUEUE[J] | depart ⊆ arrive ]                DEF (5.18)
```

In a correct implementation of a FIFO_QUEUE the jobs will depart in the order of arrival. This scheduling mechanism may however have undesirable run-time characteristics. Unnecessary delays may occur if the process component which receives jobs along the departure connection is temporarily unwilling to accept the next job, but could have serviced another queued job.

An alternative design would be-

c) Limited-overtaking (or 'fair') queue

```
FAIR_QUEUE ____________J____________ DEF (5.19)
|   QUEUE[J]
|-------------
|   1+#depart - min(arrival_times) ≤ L
|   where
|   arrival_times = dom(arrive ↓ queued);
|   queued        = ran(arrive) - ran(depart)
|_____________________________________
```

( for some limit L which is non-zero)

The FAIR_QUEUE describes a queue which ensures that no jobs can be overtaken by more than L other jobs.

Example:
let

$$L = 3;$$
$$arrive = \langle j1\ j2\ j3\ j4\ j5\ j6 \rangle$$
$$depart = \langle j2\ j4\ j3 \rangle$$

then

$$1+\#depart - min(arrival\_times) = 1+3 - 1 = 3$$

and the departure of a job other than the job $j1$ would invalidate the axioms given. If $j1$ departs we get-

$$depart = \langle j2\ j4\ j3\ j1 \rangle$$

and

$$1+\#depart - min(arrivaltimes) = 1+4 - 5 = 0$$

## 5.3. CRITICAL SECTIONS AND MUTUAL EXCLUSION.

Consider the mutual exclusion problem. This is a problem of devising a scheduling mechanism for concurrently operating jobs or processes which ensures that a job does not *enter* a *critical section* where it makes use of a shared resource if other jobs, currently within their critical section, have used up the resource.

Let us first formalise *Mutual Exclusion*.

Let P be a set of processes. The mutual exclusion can be described in terms of a relation over P, i.e.

Mutex [P] DEF (5.20)

$$exclude : P \leftrightarrow P$$

---

$$id(P) \cap exclude = \{\};$$
$$exclude^{-1} = exclude$$

which states that-

a) nobody can exclude themselves (the Mutex relation is *irreflexive*).
b) the exclusion is mutual (the Mutex relation is *symmetric*).



Examples:

1) Let us describe a small readers/writers system with three readers and one writer.

$$RW\_system = \{ r_1, r_2, r_3, w_1 \}$$

The readers exclude the writer and the writer excludes all readers, i.e.

$$RW : RW\_system \leftrightarrow RW\_system$$
$$RW = \{ r_1 \leftrightarrow w , r_2 \leftrightarrow w , r_3 \leftrightarrow w , w \leftrightarrow r_1 , w \leftrightarrow r_2 , w \leftrightarrow r_3 \}$$ DEF (5.21)

and

$$RW \in \{Mutex[RW\text{-}System]\}$$

2) Let

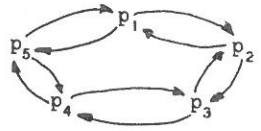
$$Phil = \{ p_1, p_2, p_3, p_4, p_5 \}$$

and

$$PH : Phil \leftrightarrow Phil$$
$$PH = \{ p_1 \leftrightarrow p_2, p_2 \leftrightarrow p_3, p_3 \leftrightarrow p_4, p_4 \leftrightarrow p_5, p_5 \leftrightarrow p_4, p_4 \leftrightarrow p_3, p_3 \leftrightarrow p_2, p_2 \leftrightarrow p_1 \}$$ DEF (5.22)

which can be illustrated as-

FIGURE (5.23)




then the relation PH is a Mutex relation, where each individual excludes *neighbours*.

A simple system which monitors a set of jobs which alternately execute outside and inside a critical section, can be characterised in terms of two components:

1) The non-critical section can be described as-

ENV [J] DEF (5.24)

$$enter, leave : seq[J]$$

---

$$(\forall\ j:J)(\ card(enter\downarrow\{j\}) - card(leave\downarrow\{j\}) \leq 1 \wedge card(enter\downarrow\{j\}) \leq MAX(j)\ )$$

for some $MAX \in J \rightarrow \mathbb{N}$, which ensures that all jobs only *enter* the critical section a finite number of times.

2) Given a Mutex relation M. then a critical section can be described as-

CS[J] DEF (5.25)

```
enter,leave : seq[J]
---
M(inside) ∩ inside = {};
(∀ j:J) (card(enter↓{j}) ≥ card(leave↓{j}))
where
inside =
   {j:J| card(enter↓{j}) > card(leave↓{j})}
```

which constrains the behaviour along the *enter* connection so that no job can get inside if another job to which it is related through the mutual exclusion relation, is currently inside. The obvious property that no jobs can leave before they enter is also included in the description.

The final states of the system-

SYS ≙ [ ENV; CS ] DEF (5.26)

are the states where all jobs have terminated and are outside the critical section, i.e.

FINAL[J] DEF (5.27)

```
SYS
---
(∀j:J)( card(enter↓{j}) = MAX(j)  ∧
        card(leave↓{j}) = MAX(j)  )
```

The system SYS terminates and is deadlock free.

NB. taking the relation PH in DEF(5.22) as the mutex relation we have a top level specification of the *dining philosophers* problem, and taking RW in DEF(5.21) we have a description of a small readers/writers system.

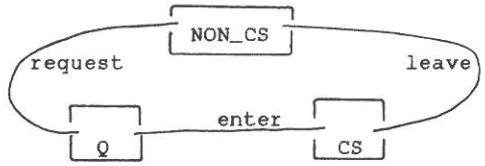
A property which is normally discussed in connection with mutual exclusion (in addition to deadlock properties) is the absence of *starvation*. Starvation may occur when a process (or job) can be excluded from entering the critical section by two or more other independent processes. The two processes can either intentionally co-operate in excluding the process or accidently be scheduled in such a way that one of the two processes will always be within the critical section.

How can we change our specification in such a way that we impose a scheduling strategy that prevents starvation?

Consider the following system-



FIGURE (5.28)




where NON_CS makes up the environment for the system being designed and is defined as-

NON_CS[P] ≙ ENV[P][request/enter] DEF (5.29)

and represents a collection of independent processes (in their non-critical section) which alternately *request-to-enter* and *leave* the critical section.

Q represents a queue of processes which have requested permission to enter the critical section but are temporarily delayed in doing so. The Q can be described as-

a) An unordered queue (see DEF(5.17) )

Q[P] ≙
QUEUE[P][request/arrive, enter/depart] DEF (5.30)

which does not prevent starvation.

b) A fifo queue (see DEF(5.18) )

Q[P] ≙
FIFO_QUEUE[P][request/arrive,enter/depart] DEF (5.31)

which prevents starvation but may have undesirable run-time characteristics.

c) A limited overtaking queue (see DEF(5.19) )

Q[P] ≙
FAIR_QUEUE[P][request/arrive,enter/depart] DEF (5.32)

which prevents starvation and only causes unnecessary delays in situations where a process has been overtaken too often.

The system which is illustrated in FIGURE(5.28) (with the Q being defined as a FAIR_QUEUE) constitutes a top level specification of a system which schedules several independently excecuting processes in such a way that -

1) no two processes which are related through a given Mutex relation are within the critical section at the same time,
2) The system is deadlock free, with respect to the following definition of the final state-

FINAL_CS(P) DEF (5.33)

NON_CS[P]; Q[P]; CS[P]

---

(∀ p:P)
( card(request↓{p}) = MAX(p) ∧
card(request↓{p}) = card(enter↓{p}) ∧
card(enter↓{p}) = card(leave↓{p}) )

3) No processes are being starved. By introducing the queue into the specification we introduced new operations which enabled us 1) to make a distinction between a request to enter and the actual entering of the critical section, 2) to define starvation and 3) to suggest a 'dynamics' which prevents it.

## 5.4. PREVENTION OF DEADLOCK.

An inconsistency in the decisions made with respect to - 1) the invariants of the system and 2) the final state - will result in *Deadlock*, i.e. if a system is realised following the rules given for the *behaviour* of the system (the invariant conditions) then this system cannot be guaranteed to reach a *final state*. The importance of the relationship between invariant conditions and the definition of the final state has been discussed in section 3.3.1.

The approach for specifying distributed systems introduced in the preceding chapters enables the designer to investigate deadlock properties solely from the information provided in the specification. If a specification cannot be proved to be deadlock free the designer should either change the *behaviour* of the system by modifying the invariant of the system or change the definition of the *final state*.

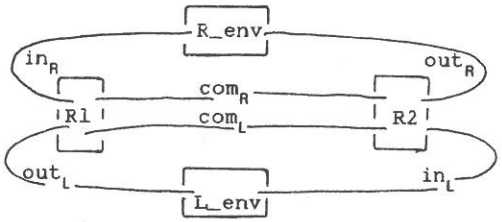
We will in this section give an example of a small system which cannot be prooved to be deadlock free from its first specification. A modification to the specification which *strengthens* the invariant conditions will ensure that behaviour which may lead to deadlock is avoided.

Consider a distributed system consisting of four independent process components. Two of the components are non-sharable resources (R1, R2), and two are components in the environment. The first environment component (R_env) is in control of starting a sequence of jobs (R_jobs). Each R_job will, when initiated, first occupy resource R1. Next it acquires resource R2. On completion it signals the R_env. The other environment (L_env) controls a sequence of L_jobs. The L_jobs behave like R_jobs except for the order in which they acquire the resources R1 and R2. An L_job requires access to R2 before it requires access to R1.

The system may be illustrated as follows:



FIGURE (5.34)




Given two initial sequences:

$init_R$ : seq[R];
$init_L$ : seq[L]

we define

$R\_env[R] \triangleq [in_R, out_R : seq[R] \mid in_R \subseteq init_R ]$ DEF (5.35)

$L\_env[L] \triangleq [in_L, out_L : seq[L] \mid in_L \subseteq init_L ]$ DEF (5.36)

i.e. the R_env will transmit all elements from $init_R$ along $in_R$ in the order they appear in $init_R$ and L_env will transmit elements from $init_L$ along $in_L$.

R1 is described as-

R1(R L) DEF (5.37)

$in_R, com_R$ : seq[R];
$com_L, out_L$ : seq[L]

---

$com_R \subseteq in_R; out_L \subseteq com_L;$
$\delta l_R + \delta l_L \leq 1$
**where**
$\delta l_R = \#in_R - \#com_R$
$\delta l_L = \#com_L - \#out_L$

The first two axioms state that the resource-controller R1 will transmit the jobs received along $in_R$ to the channel $com_R$ and the jobs received along $com_L$ to the $out_L$ channel. The third axiom ensures that only a single job can use the resource at any point in time.

Similarly for R2 we have-

R2 — R L — DEF (5.38)

$in_L, com_L$ : seq[L];
$com_R, out_R$ : seq[R]

---

$com_L \subseteq in_L$; $out_R \subseteq com_R$;
$\delta 2_R + \delta 2_L \leq 1$
**where**
$\delta 2_L = \#in_L - \#com_L$
$\delta 2_R = \#com_R - \#out_R$

In a final state for the system-

RL_SYS — R L — DEF (5.39)

R_env[R,L];
L_env[R,L];
R1[R,L];
R2[R,L]

we expect all jobs originally in the input queues ($init_L$ and $init_R$) to have completed their execution, i.e.

RL_FINAL — R L — DEF (5.40)

RL_SYS[R,L]

---

$in_R = init_R$ ; $in_L = init_L$;
$in_R = out_R$ ; $in_L = out_L$

NB. if we are in a final state then all resources are free, i.e

$$RL\_FINAL \vdash \delta 1_R = \delta 1_L = \delta 2_R = \delta 2_L = 0$$

We now observe-

1) The specified system will terminate. A variant function can be defined as -

$$Var = (\lambda\ RL\_SYS)\ (\#init_R*3 - \#in_R - \#com_R - \#out_R + \#init_L*3 - \#in_L - \#com_L - \#out_L) \qquad \text{DEF (5.41)}$$



2) The terminal states for the system can be described as -

RL_TERM — R L — DEF (5.42)

RL_SYS[R,L]

---

$(in_R=init_R \vee \delta 1_R=1 \vee \delta 1_L=1)$; (1)
$(\delta 1_R=0 \vee \delta 2_R=1 \vee \delta 2_L=1)$; (2)
$(\delta 1_R=0)$; (3)
$(in_L=init_L \vee \delta 2_R=1 \vee \delta 2_L=1)$; (4)
$(\delta 2_L=0 \vee \delta 1_R=1 \vee \delta 1_L=1)$; (5)
$(\delta 2_L=0)$ (6)

NB.
(1) denotes the states where activity along $in_R$ are blocked.
(2) denotes the states where activity along $com_R$ are blocked.
(3) denotes the states where activity along $out_R$ are blocked.
(4,5,6) denote similar conditions for L_jobs.

3) The system cannot be guaranteed to reach a final state as we cannot prove-

RL_TERM[R,L] ⊢ RL_FINAL[R,L]

However we can prove-

R L — THEOREM (5.43)

RL_TERM[R,L] ⊢
$(in_R=init_R \wedge in_L=init_L \wedge in_R=out_R \wedge in_L=out_L)$
∨
$(\delta 1_R=1 \wedge \delta 2_L=1)$

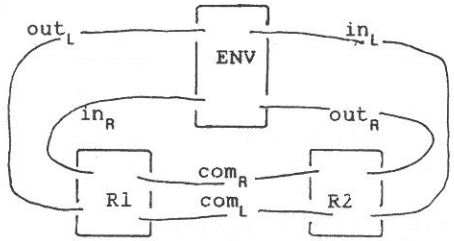
which states that a RL_SYS will terminate in either an acceptable final state or in a state where an R_job occupies R1 (waiting for R2) and an L_job occupies R2 (waiting for R1).

There are several ways of modifying the specification in order to eliminate undesirable deadlocks (see section 3.3.1.). For this particular system it seems reasonable to maintain the given specification of the final state and solve the 'deadlock' by strengthening the invariant conditions thereby modifying the dynamics of the system.

Two solutions will be given in the following:

1) The deadlock can be avoided by simply making further assumptions about the environment. If we can rely on the environment never to start an R_job while an L_job is in progress and vice versa, then the system will be deadlock-free. Such a system may be illustrated as-

RL_SYS1 DEF (5.44)




where R1 and R2 are unchanged and-

ENV R L DEF (5.45)

R_ENV[R]; L_ENV[L]

$\delta_R = 0 \vee \delta_L = 0$
**where**
$\delta_R = \#in_R - \#out_R$
$\delta_L = \#in_L - \#out_L$

The new system-

RL_SYS1[R,L] ≙
[ ENV[R,L] ; R1[R,L] ; R2[R,L] ] DEF (5.46)

with the final state-

RL_FINAL1 R L DEF (5.47)

RL_SYS1[R,L]

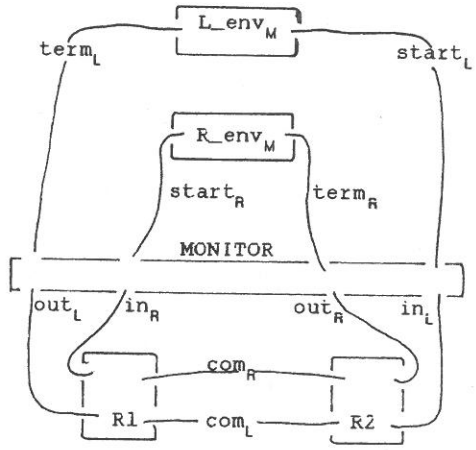
$in_R = init_R$ ; $in_L = init_L$;
$in_R = out_R$ ; $in_L = out_L$

terminates and can be proved to be deadlock free.



2) The behaviour of the system can also be changed by adding a *monitor* which coordinates the L-traffic and the R-traffic. Such a system can be illustrated as-

RL_SYS2 DEF (5.48)




where R1 and R2 are unchanged, and the environments are changed to communicate with the introduced monitor, i.e.

$L\_env_M[L]$≙
$L\_env[L][start_L/in_L,\ term_L/out_L\ ]$ DEF (5.49)

$R\_env_M[R]$≙
$R\_env[R][start_R/in_R,\ term_R/out_R\ ]$ DEF (5.50)

The MONITOR is a buffer between the resource controllers and the environment and delays R_jobs if an L_job is in progress and vice versa.

MONITOR R L DEF (5.51)

$start_R, in_R, out_R, term_R$ : seq[R];
$start_L, in_L, out_L, term_L$ : seq[L]

$term_R \subseteq out_R$ ; $term_L \subseteq out_L$ ;
$in_R \subseteq start_R$ ; $in_L \subseteq start_L$
$\delta_R = 0 \vee \delta_L = 0$
**where**
$\delta_R = \#in_R - \#out_R$
$\delta_L = \#in_L - \#out_L$

RL_SYS2 can with an obvious definition for the final state be proved to be deadlock-free.

Note that we used the result from the analysis of the first system (RL_SYS) to *guide* design modifications. However the process of removing deadlock from a design cannot in general be automated. The presence of deadlock reveals that the designer has not fully understood the consequences of his design. His design is *inconsistent* and must be modified, but exactly what must be changed is solely his decision.

## 5.5. A COMMUNICATION NETWORK.

In this section a specification of a communication network will be developed. The design decisions taken will be formally stated and motivated. A formal description of the behaviour of a single station (node) will be given, as well as a description of the static properties of the network as a whole. These descriptions will be used to verify that the *global behaviour* of a network, which consists of stations whose *local behaviour* complies with the given specification, agrees with some independently stated *rules* for networks.

### 5.5.1 A Discussion of a Communication Network.

A *NETWORK* consists of a collection of *STATIONS*. In this formal description the set of stations will be denoted by-

$$\mathbf{St}$$

The value which is communicated over the NETWORK is called a *PACKET*. Packets are denoted by-

$$\mathbf{Pk}$$

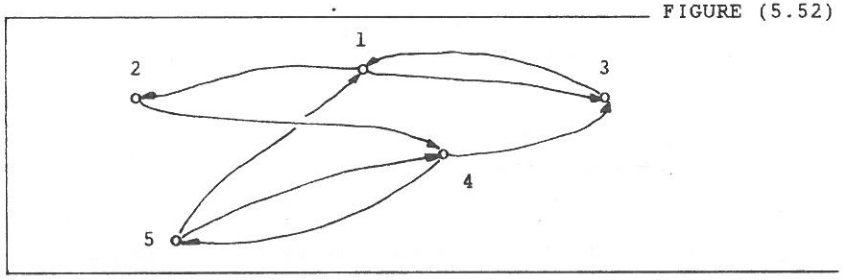
All packets have a *SOURCE* station and a *DESTINATION* station-

$$source, dest \ : \mathbf{Pk} \rightarrow \mathbf{St}$$

Example:
Consider the network illustrated in Figure (5.52) consisting of five stations,

$$\{ 1, 2, 3, 4, 5 \}$$

FIGURE (5.52)






Given a set of packets,

$$\{ a, b, c \}$$

We might have,

$$source = \{ (a,1), (b,5), (c,2) \}$$
$$dest = \{ (a,5), (b,3), (c,5) \}$$

The set of packets which originates in station i is

$$source^{-1}(\{i\})$$

The set of packets which can be consumed by i is

$$dest^{-1}(\{i\})$$

For the example above we get,

$$source^{-1}(\{1\}) = \{a\}$$
$$dest^{-1}(\{5\}) = \{a,c\}$$

Let us first formalise some obvious properties for stations in a network. Stations either *consume* or *transmit* packets. In order to formalise this statement we introduce the *history* of the input performed by each station-

$$In : \mathbf{St} \rightarrow seq[\mathbf{Pk}]$$

and a *history* of the output performed by each station-

$$Out: \mathbf{St} \rightarrow seq[\mathbf{Pk}]$$

The output from station i, Out(i), will be written $Out_i$ similarly for In.
We can now state some requirements for the behaviour of a station.

a) Only packets originating in (or previously received by) a station can be transmitted along the outgoing connections for that station. We have-

$$(\forall\ i{:}\mathbf{St})(ran(Out_i) \subseteq source^{-1}(\{i\}) \cup ran(In_i))$$

b) A packet arriving at its destination must be consumed- i.e. it must not be retransmitted, therefore we strengthen a)

$$(\forall\ i{:}\mathbf{St})$$
$$(ran(Out_i) \subseteq (source^{-1}(\{i\}) \cup ran(In_i)) - dest^{-1}(\{i\}))$$
DEF (5.53)

c) If we assume that packets are distinguishable- the fact that no packet can be transmitted from a particular station more than once can be formalised-

$$(\forall\ i{:}\mathbf{St})(\ Out_i^{-1} \in \mathbf{Pk} \twoheadrightarrow \mathbf{N}\ )$$ DEF (5.54)

b) and c) record the decisions taken with respect to the behaviour of a station, ignoring the fact it may have several *input channels* and several *output channels*. We may say that a station's behaviour is like the behaviour of an *unlimited unordered buffer*, i.e. a station will always accept incoming packets, but these packets may be stored for later transmission. Such a system is called a *store and forward* system.

Alternative designs will be discussed below:

1) Limited buffering capacity.
In order to describe stations with limited buffering capacity we might have added the in variant, (for some limit L),

$$(\forall\ i{:}St)\ (card(ran(In_i) - dest^{-1}(\{i\})) - card(Out_i) \leqslant L)$$

However, a limit on the buffer-store for a station in a network may cause the network to deadlock.

2) Scheduling.
No policy for scheduling the outgoing packets has been suggested. Hence we allow for the possibility of *unfair scheduling* where a packet within a station may be *starved* (overtaken by other packets an unreasonable number of times).

A *FIFO* scheduling might be described as follows:

For a *relay-station* i where

$$dest^{-1}(\{i\}) = source^{-1}(\{i\}) = \{\}$$

a *FIFO* ordering could be imposed as usual by requiring that the history sequence describing the output is a *prefix* of the sequence describing the input, i.e.

$$Out_i \subseteq In_i$$

For a *terminal-station* i , where

$$dest^{-1}(\{i\}) \neq \{\} \quad \vee\ source^{-1}(\{i\}) \neq \{\}$$

the *ordering* axiom could be expressed as follows:

$$ignore(source^{-1}(\{i\}))(Out_i) \subseteq ignore(dest^{-1}(\{i\}))(In_i)$$

where



X

DEF (5.56)

```
ignore[X] : ℙ(X) → seq[X] → seq[X]

(∀ S:ℙ(X); s:seq[X])
     ran(ignore(S)(s)) = ran(s) - S  ∧
     (∃ f:Monotone) (ignore(S)(s) = s ∘ f)
where
Monotone = {f:ℕ→ℕ | (∀x,y:ℕ) x≤y ⇒ f(x)≤f(y)}
```

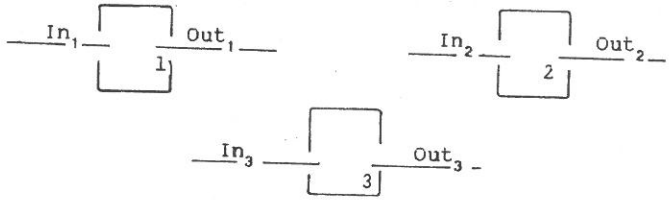
NB ignore removes elements from a sequence. The result is a new sequence where the order of the remaining elements is the same as the ordering between these elements in the original sequence, e.g.

$$ignore(\{a,c\})(\langle a\ b\ c\ d\ e\rangle) = \langle b\ d\ e\rangle$$

A *FIFO* scheduling policy could cause unnecessary runtime delays, because the station to which a scheduled packet is to be transmitted could be temporarily engaged in other communications. For these reasons we will not impose a *FIFO* scheduling policy.

DEF(5.53) and DEF(5.54) gave some dynamic properties for a station, without any reference to the topology of the network, i.e. so far we have only described a collection of dis-connected stations-

FIGURE 5.57




The connections between the individual stations are described by:

$$Network : St \leftrightarrow St$$

where we assume that the number of connections is finite, i.e.

$$Network \in \mathbb{F}(St \times St)$$

hence we might get:

NW

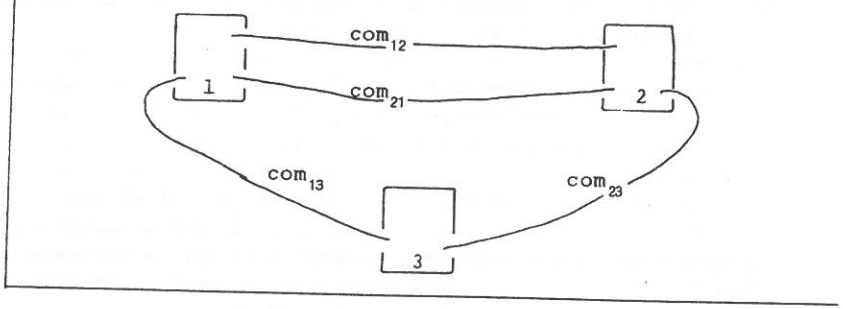



FIGURE 5.57a

When packets are sent along the connections of the network, the information which changes is:

```
Communication_between : St × St  ⇸ seq[Pk]     where
dom(Communication_between) = Network
```

that is, when a packet is sent from station i to station j, the *history sequence*

```
Communication_between(i,j)
```

will be extended.

Let $com_{ij}$ denote (as in FIGURE(5.57) ) the

```
Communication_between(i,j)
```

The correct behaviour of a station depends on its capability to transmit each packet along a proper connection. We will in what follows discuss two possibilities for proper routing of packets.

a) A packet should not be sent to a station from which the packet's destination cannot be reached, i.e.

```
(∀ i,j:St  | (i,j) ∈ Network)
    (∀ p:Pk  | p ∈ ran(com_ij)) (j,dest(p)) ∈ Network*
```

NB $Network^*$ is a description of the stations which are connected through the network.

The axiom above does not prevent a packet from 'entering a loop' or 'traveling' erratically round the network without ever reaching its destination. We therefore strengthen the axiom:

```
(∀ i,j:St | (i,j) ∈ Network)
    (∀ p:Pk  | p ∈ ran(com_ij))
        ( (j, dest(p))  ∈ Network* ∧
          distance(Network)(j,dest(p)) <
              distance(Network)(i,dest(p))  )
```

where



X — DEF (5.58)

```
distance  : (X ↔X) → (X × X ⇸ N)

(∀ r : X↔X)
   (∀i,j:X| (i,j) ∈ r*)
   ( distance(r)(i,j) = min{n:N | (i,j) ∈ r^n} )
```

This condition insures that all packets visit a finite number of stations, and travel along a route of minimal length.

b) Condition a) does not allow a station to send a packet along a connection which is not on the shortest route for that packet. The following specification allows a station to choose a longer but possibly faster route. It is still guaranteed that all packets will reach their destinations in a finite number of steps.

We first formalise the concept of a proper route for a packet, which is a route within the network where no station is visited more than once, hence

X — DEF (5.59)

```
proper_routes : (X ↔ X)  → ℙ(seq[X])

(∀ r : X ↔ X )
  proper_routes(r) =
  {rt:seq[X]  | next(rt)  ⊆ r; rt^-1  ∈ X ⇸ N }
```

NB next is defined in section 2.3.2.

for the network illustrated in FIGURE(5.57a) we have:

```
⟨1 2 3⟩    ∈ proper_routes(NW)
⟨1 3⟩      ∈ proper_routes(NW)
⟨3 1⟩      ∉ proper_routes(NW)
⟨1 2 1 3⟩  ∉ proper_routes(NW)
```

The requirements for a packet router for the network,

```
Packet_router : Pk ⇸ seq[St]
```

can now be formalised:

All routes are proper routes:

```
ran(Packet_router) ⊆ proper_routes[St](Network)
```

Exactly the packets which can reach their destination from their source can be given a route by the router:

$$dom(Packet\_router) = \{p:Pk \mid (source(p),dest(p)) \in Network^{*}\}$$

A route for a packet 'starts' at its source and 'ends' at its destination:

$$dest = last \circ Packet\_router$$
$$source = first \circ Packet\_router$$

NB. separate packets sent between the same source and destination may take (or be given) different routes.

The use of output channels is determined by the packet router in the following way:

$$(\forall\, i : St)\ (\forall\, j:St \mid (i,j) \in Network)\ (\forall\, p:Pk \mid p \in ran(com_{ij}))\ (i,j) \in next(Packet\_router(p))$$

### 5.5.2. A Specification of a Network.

The decisions taken in the previous section will be summarised in this section.

The *static* properties of the network presented are described in DEF(5.60). A description of `source` and `destination` (introduced above) is no longer necessary as the source and destination of packets are the `first` and the `last` stations along their route. `Packets` are all the packets ever processed by the Network, hence we must require that at least these packets have a route within the network. `Start_in` gives for each station the set of packets originating in that station. Previously we used the *inverse* of the `source` function to describe these packets.

St Pk — DEF (5.60)

$$Network : \mathbb{F}(St \times St)$$
$$Packet\_router : Pk \rightarrow seq[St]$$
$$Packets : \mathbb{P}(Pk)$$
$$Start\_in : St \rightarrow \mathbb{P}(Pk)$$

---

$$ran(Packet\_router) \subseteq proper\_routes(Network);$$
$$Packets \subseteq dom(Packet\_router);$$
$$Start\_in = (first \circ Packet\_router)^{-1} \downarrow Packets$$

The *dynamic* properties are described in DEF(5.61). The properties are described in terms of *invariant conditions* on the histories of the communications between



the stations. Axioms (2) and (3) are substituted for the axioms DEF(5.53) and DEF(5.54) respectively; they have been simplified as the `Packet_router` axiom (4), insures that-

a) a packet cannot 'leave' its destination,

b) a packet can 'leave' a station along only one outgoing connection.

NETWORK — St Pk — DEF (5.61)

$$com : (St \times St) \rightarrow seq[Pk]$$

---

$$dom(com) = Network; \quad (1)$$
$$(\forall\, i : St)\ (\forall\, j : St \mid (i,j) \in Network)$$
$$(ran(com(i,j)) \subseteq Start\_in(i) \cup IN(i)) \wedge \quad (2)$$
$$com(i,j)^{-1} \in Pk \rightarrow \mathbb{N} \wedge \quad (3)$$
$$((\forall\, p:Pk \mid p \in ran(com(i,j)))\ (i,j) \in next(Packet\_router(p))) \quad (4)$$

where $IN(i) = \cup\{ran(com(k,i) \mid k : Network^{-1}(\{i\})\}$

The initial state of a network system is a system where no communication has taken place-

INITIAL_NW — St Pk — DEF (5.62)

NETWORK[St, Pk]

---

$$(\forall\, (i,j): St \times St \mid (i,j) \in Network)\ (com(i,j) = \langle\rangle)$$

The final state of a network system is a system where all packets have reached their destination-

FINAL_NW — St Pk — DEF (5.63)

NETWORK[St, Pk]

---

$$(\forall\, i:St \mid i \in dom(Network))\ (\forall\, p:Pk \mid p \in (Start\_in(i) \cup IN(i)) - OUT(i))\ (i = last(Packet\_router(p)))$$

where $IN(i) = \cup\{ran(com(k,i) \mid k : Network^{-1}(\{i\})\}$

and $OUT(i) = \cup\{ran(com(i,j) \mid j : Network(\{i\})\}$

### 5.5.3. An Analysis of the Network.

The given network will be analysed with respect to *termination* and *deadlock*. Under the assumption that a finite number of packets are being 'submitted' to the network system, i.e.

$$Packets \in \mathbb{F}(Pk)\ ,$$

we will verify that

1) The operations of the system cease after the occurence of a finite number of communications (*termination*).
2) The network system only terminates in an acceptable final state – as described in FINAL-NW (*deadlock-free*).

**Termination.**

Termination is guaranteed if we can present a *decreasing variant function*.
A communication between station i and station j can be described as-

Observations — Pk — DEF (5.64)

$OBS_{ij}$ : **Pk** → {NETWORK} ↛ {NETWORK}

$OBS_{ij}$ = (λ p:**Pk**) (λNETWORK)(μ NETWORK')
(com' = com ⊕ {(i,j) → com(i,j) * ⟨p⟩})

NB, ( ⊕ ) is the *function overriding* operator.

According to the definition above a communication will *increase* the length of the *history* of the communications along a single connection. Therefore the following function must be an *increasing variant function* –

```
Vl  : {NETWORK} → N
Vl  = (λ NETWORK) (Sigma(com, card ∘ ran) )
```

where Sigma is defined as

X — DEF (5.65)

Sigma : (𝔽(X) × (X↛N)) ↛ N

(∀ s:𝔽(X); f:X↛N | s ⊆ dom(f); s ≠ {})
Sigma(s,f) = f(τ(s)) + Sigma(s-{τ(s)},f)
Sigma({},f) = 0

According to axiom (3) of NETWORK a packet can appear only once along any connection, hence

```
(∀ nw:NETWORK)  Vl(nw) ≤ card(Packets)*card(Network)
```

Therefore the following function is a *decreasing variant function*–

```
V  : {NETWORK} → N
V  = (λ nw:NETWORK)
          ((card(Packets)*card(Network)) - Vl(nw))
```

**NB, in a termination state 's' we do not guarantee that V(s) = 0.**



**Deadlock.**

We can describe the activity along channel (i,j) as follows:

Pk — DEF (5.66)

$activity\_along_{ij}$ : {NETWORK} ↔ {NETWORK}

$activity\_along_{ij}$ = ∪{ $OBS_{ij}$(p) | p : **Pk**}

A termination state of a network is a state where no activity can take place, and can be described as:

DEF (5.67)

Terminal : 𝔽({NETWORK})

Terminal =
{NETWORK} - ∪{dom($activity\_along_{ij}$)|(i,j) : Network}

A system is *deadlock-free* if it only terminates in acceptable final states.

For this network system we have:

THEOREM (5.68)

⊢ Terminal ⊆ {FINAL_NW}

The proof of THEOREM(5.68) is directly derivable from LEMMA(5.70) and LEMMA(5.71)

We first describe the states of the network in which a communication is expected to take place.

Pk — LEMMA (5.69)

⊢ dom($activity\_along_{ij}$) =
{NETWORK | (∃ p:**Pk**)
(p ∈ (Start_in(i)∪IN(i))-OUT(i)) ∧
(i,j) ∈ next(Packet_router(p)) }

⊢ ∪{dom($activity\_along_{ij}$) | (i,j) : Network} =
{NETWORK | (∃ p:**Pk**) (∃ (i,j):Network)
(p ∈ (Start_in(i)∪IN(i))-OUT(i)) ∧
(i,j) ∈ next(Packet_router(p)) }

NB the functions IN and OUT are defined as in FINAL_NW DEF(5.63).

LEMMA(5.70) describes the termination states:

**Pk** — LEMMA (5.70)

```
⊢ Terminal =
    {NETWORK | (∀ (i,j):Network) (∀ p:Pk)
                 (p ∉ (Start_in(i)∪IN(i))-OUT(i)) ∨
                 (i,j) ∉ next(Packet_router(p))  }
```

LEMMA(5.71) states that a *termination* state is an *acceptable final* state.

**St Pk** — LEMMA (5.71)

```
NETWORK;
(∀ (i,j):Network)
 (∀ p:Pk)
  (p    ∉ (Start_in(i)∪IN(i))-OUT(i) )  ∨
  (i,j) ∉ next(Packet_router(p))
⊢ FINAL_NW
```

**proof:**

```
(∀ i:St  | i ∈ dom(Network))
    (∀ p:Pk  | p ∈ (Start_in(i)∪IN(i)) - OUT(i) )
        (∀ j:St | (i,j) ∈ Network)
```

we deduce from the hypothesis

(1) `(i,j) ∉ next(Packet_router(p))`

as p is 'within' the buffer, we conclude

(2) `i ∈ ran(Packet_router(p))`

from (1) , (2) we get

(3) `i = last(Packet_router(p))`

which according to DEF(5.63) completes the proof



# 6. DECOMPOSITION.

A complex system is easier to comprehend if it is described through several specifications presenting the system in increasing degrees of detail. For that reason, and in order to make the development process more manageable as well as to make the end-product easier to modify, it has become a convention to describe (if not develop) systems in a *top-down* fashion. A system is first described in terms of a limited set of modules and their inter-relationship. When first presented the modules are specified solely in terms of their *external interfaces*, i.e. the services they provide for the surrounding modules. In a more refined description the decisions made with respect to the *internal structure* of each module will be documented. During the development the designer must verify that the internal structure provided for each module is *consistent* with the definition of its external interface.

The independent module (as described in section 4.2.1) which is specified through an *input-output* relation can be *decomposed* by giving a set of new independent modules. From the description of the input-output relation for each individual sub-module we can infer the input-output relation computed by the compound system, giving us a method for analysing whether the behaviour of the *refined* system agrees with the behaviour of the system it is intended to *implement*. The techniques for calculating the input-output relation computed by a compound system are based on the assumption that each individual module-realisation runs in *isolation*.

As mentioned in section 4.3, it is important to provide a comparable *refinement* and *development* method for systems whose constituents are process-modules. However, these modules, which run in *parallel* and which, during their execution, can *inspect* and *influence* the execution of one another, cannot be described through a simple input-output relation. Hence, the refinements methods and the techniques for validating a decomposition will take a different form from the methods used for modules running in isolation.

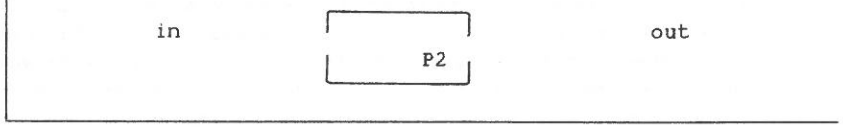
A process-module is refined by replacing one specification of the module with another specification, in which the assertions about the histories of the communication are decomposed and partitioned. A refinement reflects the decision to realise a process-module using two or more concurrently operating process components (a decision taken, for example, in order to increase performance). A refinement will normally increase the number of visible connections as some local communications between the modules of the refinement will be added to the description.

## 6.1. INTRODUCTORY EXAMPLE.

This section will introduce the decomposition method by means of an example.

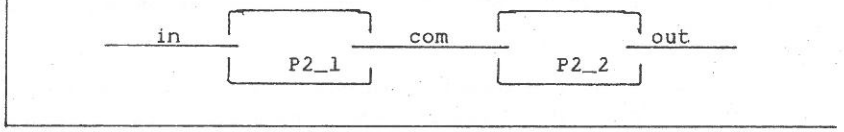
Consider the system presented in section 4.2. (DEF(4.4)) which can be illustrated-

P2_MODULE FIGURE (6.1)




Its decomposed (distributed) counterpart from section 4.3. (DEF (4.9)) can be illustrated

P2_SYSTEM FIGURE (6.2)

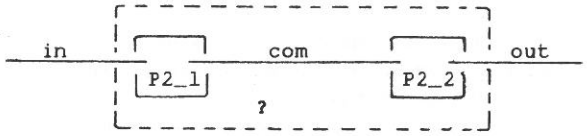



Informally, we will say that a distributed system (like P2_SYSTEM) is an *implementation* (or valid decomposition) of a module (like P2_MODULE) if its externally observable behaviour (i.e. for this example the behaviour along in and out) is indistinguishable from the behaviour of the module it implements. Hence, after decomposing a module, we must compare the behaviour which can be derived from the new compound system (after having *hidden* internal communications) with the expected behaviour of the module. For the example above, we must answer the question, is-

$$? = P2$$

where

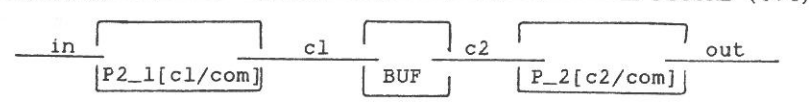
FIGURE (6.3)






Before formalising the notion of indistinguishability, consider, for comparison, the following system-

P2_SYSTEM2 FIGURE (6.4)




where

BUF[X] ≙ [ c1,c2:seq[X] | c2⊑c1; #c1-#c2≤1] DEF (6.5)

In an *Environment* where

$$\#in \leq Limit$$

for some Limit, and with a final state requirement that

$$out = F \circ in$$

We can prove that all three systems- P2_MODULE, P2_SYSTEM and P2_SYSTEM2 - are *totally correct* systems, i.e. they terminate and are deadlock free. However, the dynamic behaviour of P2_SYSTEM2 is different from the dynamic behaviour of P2_SYSTEM and P2_MODULE, e.g. P2_SYSTEM2 may be observed to consume three consecutive inputs before the activity along the in connection is blocked, while P2_MODULE and P2_SYSTEM can consume at most two before blocking. Hence a system, consisting of P2_SYSTEM (or P2_MODULE) in an environment which insists on performing three inputs before participating in an output, will deadlock.
NB, such an environment could be defined by

ENV[X] ≙ [in,out:seq[X] | #in≤3 ⇒ out=0]

If P2_SYSTEM2 is placed in the same environment no deadlock will occur.

As the dynamic behaviour of P2_SYSTEM2 is distinct from that of P2_MODULE, we will *not* accept

P2_1[c1/com]; BUF; P2_2[c2/com]

as a valid decomposition (or distribution) of

P2

A method for proving that [P2_1; P2_2] is a valid decomposition of P2 is given in the next section.

## 6.2. VALID DECOMPOSITION.

Let us consider a system which is specified in terms of predicates on its behaviour along external connections only, i.e.

$$M\_SPEC \triangleq [\ ext\_con \mid PRED_{spec}\ ] \qquad \text{DEF (6.6)}$$

A refinement (decomposition) of this module will expose some internal interfaces and the connections across them. Furthermore a new set of predicates will be given, now involving both the communications along the external connection and the internal connections. We have

$$M\_IMP \triangleq [\ ext\_con;\ int\_con \mid PRED_{imp}\ ] \qquad \text{DEF (6.7)}$$

Example:
For the system described in the previous section we have a specification of a module (introduced in section 4.3.),

P2_SPEC — DEF (6.8)

```
in,out : seq[X]
---------------
#in-#out ≤ 2; out ⊆ F•in
```

and a description of an implementation of that module

P2_IMP — DEF (6.9)

```
in,out,com : seq[X]
-------------------
#in-#com ≤ 1; com ⊆ G•in;
#com-#out ≤ 1; out ⊆ H•com
```

which is a distributed description because the axioms can be divided in to two groups - one group involving in and com only, and one involving com and out only. Furthermore we claim that this is an *implementation* (valid decomposition) of the specification P2_SPEC.

An analysis of a system consisting of interconnected modules will be carried out without any reference to the internal structure of the modules, i.e. a top-level specification for each module (like M_SPEC) will be used as a basis for the analysis. When refining a system we provide an implementation for each individual module (like M_IMP for M_SPEC) and, in order to ensure that the global dynamic properties (termination and deadlock) are maintained, we must give a guarantee, for each module, that the behaviour of the implementation *simulates* the specified behaviour along the external connections. In other words, the behaviour along the external connections (ext_con) which can be derived from M_IMP must simulate the behaviour along these connections as derived from M_SPEC.



In order to compare the two systems (M_IMP and M_SPEC) we introduce a *projection* function-

$$project : \{M\_IMP\} \rightarrow \{M\_SPEC\}$$

NB. this is analogous to a *retrieve-function* [Jones,14] or an *abstraction-function* [Hoare,9]

For the system introduced above we can define the projection function as follows-

$$project = (\lambda\ P2\_IMP)(\mu\ P2\_SPEC')(in'=in;out'=out) \qquad \text{DEF (6.10)}$$

Let

$$x1,\ x2 : seq[Y]$$

be the only external connections for M_SPEC (and M_IMP) then a communication of a value $y$ along the x1 connection can be described as-

1) According to the M_SPEC description-

$$comx1_{spec}(y) = (\lambda\ M\_SPEC)(\mu\ M\_SPEC')(x1'=x1*\langle y\rangle;\ x2'=x2) \qquad \text{DEF (6.10)}$$

2) According to the M_IMP description-

$$comx1_{imp}(y) = (\lambda\ M\_IMP)(\mu\ M\_IMP')(x1'=x1*\langle y\rangle;x2'=x2) \qquad \text{DEF (6.11)}$$

In the following we will use the examples introduced above to illustrate the method for proving that an implementation is a valid decomposition of its specification.

We must prove-

1) A *Consistency* property.
External activity (as performed by the implementation) must only occur when it is expected to occur, i.e if a communication (along an external connection) can take place according to the *rules* imposed by the implementation (e.g. M_IMP) then the same communication must also be permitted by the rules imposed by the specification (M_SPEC).

Example 1:
For the communication along x1 in the system described by M_IMP, we must prove-

$$(\forall\ y:Y)\ (\ project(dom(comx1_{imp}(y)) \subseteq dom(comx1_{spec}(y))\ )$$

which can be verified simply by proving-

M_IMP ⊢ M_SPEC

Intuition - If the axioms of the specification are kept invariant by the operations performed by the implementation then these operations (extension of a history sequence) are also 'allowed' to take place by the specification (NB. an operation can take place whenever its occurrence does not invalidate the axioms).

Example 2:
For the system P2_IMP we must prove-

P2_consistency ——— THEOREM (6.13)

P2_IMP ⊢ #in-#out ≤ 2; out ⊆ F•in

NB:
Because we require that an implementation is *consistent* with its specification we can always describe an implementation as an extension of its specification, e.g. P2_IMP could have been defined as

P2_IMP [X Z Y] ——— DEF (6.14)

P2_SPEC[X,Y];
com : seq[Z]

#in-#com ≤ 1; com ⊆ G•in;
#com-#out ≤ 1; out ⊆ H•com

2) An *adequacy* property.
When an external communication is expected to take place (according to the rules of the specification), then either this communication or an internal communication can take place (according to the rules of the implementation).

Example 1:
For the communication along x1 in M_IMP we have-

$$(\forall\ y{:}Y)\ \big(\ project^{-1}(dom\ comxl_{spec}(y)) \subseteq dom(comxl_{imp}(y) \cup dom(internal\_activity)\ \big)$$

where internal_activity is a relation describing internal communications.



Example 2:
For the system P2_IMP we must prove

P2_adequacy ——— THEOREM (6.15)

P2_IMP ⊢
δ < 2 ⇒ δ1 < 1 ∨ (δ1 = 1 ∧ δ2 < 1) ;
δ ≠ 0 ⇒ δ2 ≠ 0 ∨ (δ2 = 0 ∧ δ1 ≠ 0)
**where**
δ = #in - #out;
δ1 = #com - #in;
δ2 = #out - #com

which states -
a) If an input is expected to occur, i.e
$\delta < 2$
then either the input can be performed, i.e.
$\delta 1 < 1$
or an internal communication can take place, i.e.
$\delta 2 < 1$

b) If an output is expected to occur, i.e.
$\delta \neq 0$
then an output can take place , i.e.
$\delta 2 \neq 0$
or an internal comunication can take place, i.e.
$\delta 1 \neq 0$

3) A *Halting* property.
The adequacy property does not guarantee that a communication which is expected to take place will ever happen, because non-terminating internal activity (*infinite chatter*) could prevent the system from ever reaching a state which allows the occurrence of the external communication.

We must verify that internal activity will terminate if the system is *unstimulated* (no external activities will occur). This property can be verified by presenting a decreasing variant function for the *internal* activity.

Example:
The following function is a variant function for system P2_IMP

P2_var = (λ P2_IMP)( #in - #com ) DEF (6.16)

Summary-
The *consistency* property ensures that communications along the external connection only occur when they are expected to occur. The *adequacy* and *halting* properties ensure that a communication which is expected to occur can occur before the

occurrence of any other external communication. Furthermore, the occurrence can only be delayed by a finite number of internal events. (NB, compare these properties with the *readyness* principles from section 4.2.3).

An implementation which satisfies all three properties with respect to a specification is said to be a *valid decomposition* of that specification. Hence, P2_IMP is a *valid decomposition* of P2_SPEC (see THEOREM(6.13), THEOREM(6.15) and DEF(6.16)).

## 6.3. LIMITATIONS OF THE METHOD.

In this section we will define some systems which consist of two or more process-modules. The examples are given mainly to reveal the limitations of the method for decomposition (or distribution) presented in the two previous sections. Suggestions for surmounting these limitations will be given.

### 6.3.1. Piped Systems.

As the example (P2) presented in the previous sections illustrated, an *implementation* (or refinement) where a single process-module is replaced by a set of process-modules in a *pipe-line* arrangement is easily expressible in the provided framework. Furthermore the tools proved to be adequate for checking the behaviour of such pipe-line arranged systems against an abstract description in which no reference is made to the (local) communications between the individual stages of the pipe-line.

The following example is a continuation of the bounded buffer example from section 5.1.

We will apply the method developed in the previous section to the implementation of a bounded buffer as specified by BB, (DEF(5.3)). The example is similar to the example (P2) presented previously, and does not add to the explanation of the method. However the example is included to give yet another overview of the approach for decomposing systems and to illustrate the convienience of the schema-notation used as a development tool.

let

```
B1,B2 : N1
```

be given. In order to verify that

PIPE DEF (6.17)

```
BB[com/out;B1/B]; BB[com/in;B2/B]
```



is an implementation of

```
BB[ B1+B2/B ]
```

we must prove

1) a *consistency* theorem, which is

THEOREM (6.18)

```
PIPE ⊢ BB[B1+B2/B]
```

2) an *adequacy* theorem, which for this system can take the form

THEOREM (6.19)

```
PIPE;
#in - #com = 0 ∨ #com - #out = B2
⊢
δ < B1+B2 ⇒ δ1 < B1;
δ ≠ 0 ⇒ δ2 ≠ 0
where
δ  = #in - #out;
δ1 = #in - #com;
δ2 = #com - #out
```

3) a *halting* property, which is ensured by the presentation of a variant function,

$$Var_{Pipe} = (\lambda\ PIPE)(\#in - \#com)$$

### 6.3.2. Unordered Pipe.

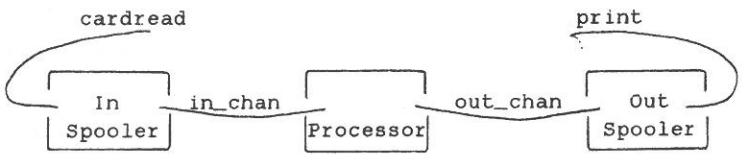
The unordered queue introduced in section 5.2.2. (DEF(5.17)) can, as mentioned in that section, conveiniently represent queues of jobs in an operating system.

let

```
In_Spooler[J]  ≙ QUEUE[J][cardread/arrive;in_chan/depart];
Processor[J]   ≙ QUEUE[J][in_chan/arrive;out_chan/depart];
Out_Spooler[J] ≙ QUEUE[J][out_chan/arrive;print/depart]
```

be components of a simplified batch operating system, which can be represented as

BATCH FIGURE (5.20)




BATCH is indeed an implementation of SYSTEM, where

SYSTEM ≙ QUEUE[cardread/arrive;print/depart] DEF (5.21)

let us now consider a slightly modified version of BATCH. By restricting the buffering capacity of the Out_Spooler, i.e. for some L : **N**1 we may have

Out_Spooler2 ≙ [ Out_Spooler | #out_chan-#print≤L ]

and by defining a new batch system as

BATCH2 ≙ [In_Spooler; Processor; Out_Spooler2] DEF (5.22)

we get a system whose behaviour *cannot* be explained by an abstraction which does not refer to the internal communications between Processor and Out_Spooler. BATCH2 is indeed distinct from BATCH because BATCH allows for any job currently within the system to get priority and to run to completion (i.e. being served by the printer), while this may not be possible for system BATCH2. in a state where

$$\#out\_chan - \#print = L$$

BATCH2 cannot accept further requests along out_chan before another job has left the print-spooler queue. Hence a job currently in the input-spooler queue or in the processor queue cannot be completed before at least one other job has been completed.

BATCH2 therefore (according to the rules stated in the previous section) is not an implementation of SYSTEM. However if we can *rely* on the environment never to insist on the completion of a particular job, then, intuitively, BATCH2 will be a perfect implementation of SYSTEM. This can be formalised.

Let O_ENV(p) describe the environment's restriction on the printer connection (p). If we can prove



THEOREM (5.23)

$$(\forall\, i{:}\mathbf{N};\ j,j'{:}J)\ (\forall\, p\_s : Print\_Stream)\ (\{i\to j\}\in p\_s) \Rightarrow ((p\_s \oplus \{i\to j'\}) \in Print\_Stream)$$

**where**

Print_Stream = {p : seq[J] | O_ENV(p)}

which states that the given environment does not insist on the completion of any particular job, then we will accept BATCH2 as an implementation of SYSTEM.

NB, if we do not make this (or a similar) assumption about the environment then BATCH2 cannot be considered to be a refinement of SYSTEM because BATCH2 might reject requests which could be accepted by SYSTEM, i.e. introducing BATCH2 as a refinement for SYSTEM may also introduce deadlock possibilities.

### 6.3.3. A Resource Monitor.

Consider the system in DEF(5.44) whose constituents are R1 from DEF(5.37), R2 from DEF(5.38) and ENV from DEF(5.45).

The subsystem

RL_PIPE ≙ [ R1; R2 ] DEF (6.24)

does not implement

RL_SYS R L DEF (6.25)

$$in_R,\ out_R : seq[R];\quad in_L,\ out_L : seq[L]$$

$$out_R \subseteq in_R;\ out_L \subseteq in_L;\quad \delta_R + \delta_L \leq 2$$

**where**

$$\delta_R = \#in_R - \#out_R;\quad \delta_L = \#in_L - \#out_L$$

as the RL_PIPE will deadlock in the situation where

$$\delta 1_R = 1 \wedge \delta 2_L = 1$$

However, the system

RL_PIPE2 — DEF (6.26)

RL_PIPE

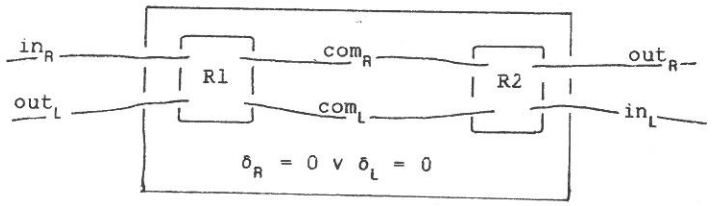
$\delta_R = 0 \vee \delta_L = 0$
**where**
$\delta_R = \#in_R - \#out_R$;
$\delta_L = \#in_L - \#out_L$

does implement

$RL\_SYS2 \cong [\ RL\_SYS \mid \delta_R=0 \vee \delta_L=0\ ]$ DEF (6.27)

RL_PIPE2 is a system consisting of three processes, one of which (a Monitor) does not refer to the local communications between the resource controllers R1 and R2. This system can be illustrated as

FIGURE (6.28)






## 7. CONCLUSION.

The goal of this thesis has been to develop a method for *specifying*, *analysing* and *refining* the designs of distributed systems.

A new approach for *specifying* distributed systems has been proposed in chapter 4 to accomplish this goal. The approach can be used to specify distributed systems in different degrees of detail; i.e. a description with few *visible internal interfaces* is considered to be more abstract than a description in which all interfaces from the future realisation take part. The systems are described through *predicates* on the *histories* of communications across the visible interfaces, and the modularisation (or distribution) of the system is determined by the possible grouping of these predicates (*localisation of knowledge*).

A method for *analysing* the behaviour of distributed designs has been proposed in sections 4.5 and 5.2. The analysis is based on a technique presented in chapter 3. It should be pointed out that the formal framework developed (in chapter 3) for analysing *non-deterministic* systems consists mainly of reformulations of familiar concepts (transition-systems: [Keller,16], [Lamswerde,17]; well-founded relations: [Abrial,1]; and correctness of non-deterministic constructs: [Dijkstra,7]). The techniques presented in chapter 3 are, as illustrated in section 3.2. (A Bounded Buffer) applicable to descriptions based on abstractions of internal states as well as to descriptions based on abstractions of external behaviour (section 5.1. A Bounded Buffer).

An important aspect of the method presented is its use in *refinement* or *step-wise development* of distributed systems. Chapter 6 introduced a simple method for comparing different levels of descriptions using projections as *retrieve-functions* [Jones,14]. It was demonstrated that the refinement method is especially suitable for refinements where a single process-module is replaced by several modules arranged in a *pipe-line*. Note that when abstracting details of the protocols (*hand-shaking*, *synchronisations* etc.) used for communications between process-components, the number of systems falling into this category is quite large. Some limitations of the method have been pointed out, together with informal suggestions, for overcoming them. From example 6.3.2.(Unordered Pipe) we learned that some *internal interfaces* (i.e. internal communications) of compound systems can be so integral to the system's behaviour that they cannot be *hidden* without losing information about the true behaviour along the external connections of the system. Instead of hiding such internal events (and introducing an unmanageable kind of non-determinism) a 'solution', in which assumptions are made about the behaviour of future environments, is proposed. (*rely-conditions*).

## 7.1. RELATED WORK.

Communication between cooperating processes has become an area of concern especially since the introduction of multiprogramming systems. In such systems the possibility for breaking down large tasks into smaller communicating (concurrently operating) processes provides a useful abstraction tool. The advance in technology - i.e. the introduction of inexpensive microprocessors - has further stimulated research in this area.

The basis for a formal treatment of cooperating processes was laid with the introduction of *semaphores* [Dijkstra,6]. Since then, several programming languages which embody the notion of processes have been suggested and high level language constructs, which provide the programmer with an abstract (and 'secure') tool for managing process communication and synchronisation, have been proposed. Examples of earlier work in this area are to be found in the work of [Hoare,10] and [Brinch Hansen,2] on *monitors* and [Campell,3] on *path expressions*. More recently and certainly more notably, there are the high-level communication facilities provided through CSP [Hoare,11] and Ada [13].

Verification of concurrent programs has also been investigated (e.g. [Owicki,19]. Verification of programs requires a *specification* with which the program can be compared, which suggests that it is important to have a framework (or a style) in which specifications can be expressed. The methods for specifying *sequential* programs are well understood; however, methods for specifying *distributed* (concurrent) programs are still in their infancy. For distributed systems which consist of several cooperating components the properties to be specified consist not only in WHAT is to be computed but also WHEN and WHERE (see section 4.3). Hence an approach for specifying - like the approach presented in this thesis - *must* be concerned with *time* in one form or another.

In this thesis sequences of past communications (i.e. histories of communications) across interfaces are used to give the needed notion of *time*.

In the area of *program verification* we find the use of *communication sequences* as *ghost* (or mythical) variables introduced in order to verify the correctness of *co-routines* [Clint,5]. Assertions involving the past history of communications along external channels have more recently been used in [Chen,4] and [Hoare,12] (NB, the assertions used in [Hoare,12] also involve the immediate future).

An alternative approach for documenting *time-related* properties for concurrent systems is to be found in the work on *temporal logic* (e.g. [Manna,18]. In this approach the time-related properties are classified into : *invariant* properties, *eventuality* properties and *precedence* properties; these properties can all be expressed using predicates on communication sequences.

The main goal of the work presented in this thesis is *not* to provide a method for *program proving*, but rather to provide a *development* tool for distributed systems, i.e. to provide a uniform formal notation in which design decisions can be documented and analysed, and a method for refining and modifying such systems. Little work has been done in this area. However, notable suggestions have recently been made in [Jones,15] where a 'module' executing in an interfering environment is described using *guarantee-* and *rely* assertions on the information it *shares* with other modules.



## 7.2. FUTURE RESEARCH.

Because the approach presented in this thesis has only been under development for a short time, research into feasibility of the proposed methods cannot be declared complete, nor can the methods themselves.

The tools must be studied further and I am convinced that inspiration for further research can be obtained from 1) [Hoare,12] in which assertions on *past* communications are used to describe intended behaviour for processes, 2) [Jones,15] where the goal is to find development-methods for concurrent systems, and 3) [Manna,18] who uses *temporal logic* in order to abstract direct references to execution sequences.

One aspect of the development process which has been intentionally omitted from this thesis is the problem of *realisation*. The difficulties in realising a communication between two distinct process components have not been considered. In any development process we must at some level of refinment consider the physical properties of the target machinery,. Hence, for distributed systems which consist of communicating process-components we must consider the means of communication provided by the inter-process control mechanisms built into the realisation tool.

The abstract communications of the operational model in terms of which we analyse our specifications may not be easily realisable. Intuitively the difficulties will depend on the realisation tool (compare the facilities provided by Ada or CSP with communications performed entirely by means of shared memory and semaphore-operations). We may reformulate a specification before realisation in order to simplify the task of translating the specification into a program.

Example:
Consider the communications between Q and CS in FIGURE(5.28).
At any point in time Q will be willing to communicate any 'job' which does not invalidate the axioms of Q, e.g.

```
{ j1, j2, j3, j4 }
```

and at any point in time CS is willing to 'receive' the set of jobs which will not invalidate the *mutual* exclusion property, e.g.

```
{ j2, j3 }
```

If the intersection of the two sets is non-empty then a communication will take place. Such a communication is not directly realisable. However, we can *reformulate* the specification and obtain a description which is 'closer' to a realisation.

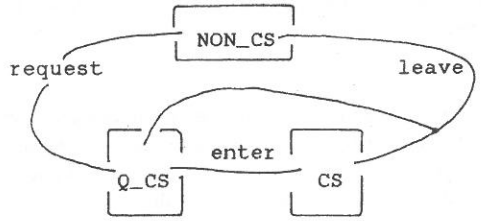
Let us define (use DEF(5.32) for Q)

Q_CS $\overset{J}{=}$ DEF(7.1.)

```
Q;
leave : seq[J]
|-----------------------------------------
M(inside) ∩ inside = {}
where
inside = {j:J | card(enter↓{j}) > card(leave↓{j})}
```

as replacement for Q.which gives

FIGURE (7.2)




NB. the modification does *not* change the properties of the system as no axioms have been added. Furthermore the mutual exclusion axiom from CS can now be removed as this axiom will be 'obeyed' by Q_CS. Hence, the communications between Q_CS and CS are much simpler as all knowledge about the communications along enter has been localised within Q_CS.

Further investigations must be made in the area of realisation of specifications.

It is most pressing, however, to apply the methods presented to a wider range of distributed systems, not only in order to improve the methods but also to explore their limitations and to identify the category of systems for which they are suitable.



**List of References.**


[1] J.R. Abrial:
'The Specification Language Z: Basic Library'.
Report from Oxford University Computing Laboratory, 1980.

[2] P. Brinch Hansen:
'Operating System Principles'
Prentice-Hall, 1973.

[3] R.H. Campbell and A.N. Habermann:
'The Specification of Process Synchronization by Path Expressions'.
Lecture notes in Comp.Sci. Vol 16.
Springer Verlag, 1974.

[4] Zhou Chao Chen and C.A.R. Hoare:
'Partial Correctness of Communicating Processes and Protocols'.
Proc. International Conference on Distributed Computing, 1981.

[5] M. Clint:
'Program proving: Coroutines'.
Acta Informatica 2, 1973.

[6] E.W. Dijkstra:
'Cooperating Sequential Processes'.
Programming Languages, Academic Press, New York, 1968.

[7] E.W. Dijkstra:
'Guarded Commands, Non-determinacy and Formal Derivation of Programs'.
CACM Vol 18 No 8, 1975.

[8] E.W. Dijkstra:
'A Discipline of Programming'.
Prentice-Hall Int. Series in Automatic Computation, 1976.

[9] C.A.R. Hoare:
'Proof of Correctness of Data Representations'.
Acta Informatica 1, 1972.

[10] C.A.R. Hoare:
'Monitors: An Operating System Structuring Concept'.
CACM Vol 17 No 10, 1974.

[11] C.A.R. Hoare:
'Communicating Sequential Processes'.
CACM Vol 21 No 8, 1978.

[12] C.A.R. Hoare:
'A Calculus of Total Correctness for Communicating processes'.
Oxford University Computing Laboratory, PRG Monograph no 23, 1981.

[13] Reference Manual,
'Rationale for the Design of the GREEN Programming Language'
Honeywell, inc. Minneapolis, 1979.

[14] C.B. Jones:
'Software Developmant - A Rigorous Approach'.
Prentice-Hall Int. Series in Computer Science, 1980.

[15] C.B. Jones:
'Development method of Computer Programs Including a Notion of Interference'
D.Phil. thesis, University of Oxford, 1981.

[16] R.M. Keller:
'Formal Verification of Parallel Programs'.
CACM Vol 19 No 7, 1976.

[17] A. von Lamswerde and M. Sintzoff:
'Formal Derivation of Strongly Correct Concurrent Programs'.
Acta Informatica 12, 1979.

[18] Z.Manna and A. Pnuelli:
'Verification of Concurrent Programs: The Temporal Framework'.
Lecture notes for The International Summer School, Munich, August 1981.

[19] S. Owicki and D Gries:
'Verifying properties of Parallel Programs: An Axiomatic Approach'.
CACM Vol 19 No 5, 1976.

[20] P.L. Parnas:
'On the Criteria to be used in Decomposing Systems into Modules'.
CACM Vol 15 No 12, 1972.

[21] B. Sufrin:
'Formal Specification of a Display Editor'.
Oxford University Computing Laboratory, PRG Monograph no 21, 1981.