# A FORMAL METHODOLOGY FOR THE VERIFICATION OF CONCURRENT SYSTEMS

Philip John Clarke

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

SEPTEMBER 1993

The University of Aston in Birmingham

A Formal Methodology for the Verification of Concurrent Systems

Philip John Clarke

Submitted for the degree of Doctor of Philosophy, 1993

## Summary of Thesis

There is an increasing emphasis on the use of software to control safety critical plants for a wide area of applications. The importance of ensuring the correct operation of such potentially hazardous systems points to an emphasis on the verification of the system relative to a suitably secure specification. However, the process of verification is often made more complex by the concurrency and real-time considerations which are inherent in many applications.

A response to this is the use of formal methods for the specification and verification of safety critical control systems. These provide a mathematical representation of a system which permits reasoning about its properties. This thesis investigates the use of the formal method Communicating Sequential Processes (CSP) for the verification of a safety critical control application. CSP is a discrete event based process algebra which has a compositional axiomatic semantics that supports verification by formal proof. The application is an industrial case study which concerns the concurrent control of a real-time high speed mechanism.

It is seen from the case study that the axiomatic verification method employed is complex. It requires the user to have a relatively comprehensive understanding of the nature of the proof system and the application. By making a series of observations the thesis notes that CSP possesses the scope to support a more procedural approach to verification in the form of testing.

This thesis investigates the technique of testing and proposes the method of Ideal Test Sets. By exploiting the underlying structure of the CSP semantic model it is shown that for certain processes and specifications the obligation of verification can be reduced to that of testing the specification over a finite subset of the behaviours of the process.

The scope of the method of Ideal Test Sets is clearly defined by a number of syntactic rules for processes and specifications. In particular care is taken to ensure that the method is compatible with existing proof techniques. The thesis then uses Ideal Test Sets to verify the case study, and is thus able to draw conclusions on the suitability and limitations of the method of Ideal Test Sets.

**Key Words:** Formal Methods, Test Sets, Communicating Sequential Processes, Safety Critical Systems, Verification.

*To Rebecca, for always*

Acknowledgements

I would like to thank my supervisor David Holding for his support, encouragement and guidance during my time at Aston University. In addition I would like to express gratitude to my colleagues and the staff in the Department of Electronic Engineering for the stimulating environment which is fostered there. Also to Rebecca for her love.

This research was carried out under a UK Science and Engineering Research Council Studentship.

# LIST OF CONTENTS

Page

CHAPTER ONE: INTRODUCTION

CHAPTER TWO: FORMAL SPECIFICATION AND VERIFICATION

CHAPTER FIVE: CATENARY FUNCTIONS

CHAPTER SIX: IDEAL TEST SETS

CHAPTER SEVEN: A SYNTAX OF RSPEC

**CHAPTER EIGHT: CONCLUSIONS**

## LIST OF FIGURES

## Glossary of Notation

Throughout this thesis there is a consistent notation for certain variable data types. For example a trace is always represented by one of the letters r, s, t, u, or v. Although this usage is made explicit during the course of the text the following glossary is intended as a guide to the reader.

### Variable Labels Used.

| Notation | Usage | Comments |
|---|---|---|
| Q, R, S, T, $\Re$ | Predicates | |
| M, N, E | Predicates | Used in the conditional statements for categories $\bar{\alpha}$ and $\bar{\beta}$. |
| r, s, t, u, v | Trace variables | |
| **a**, **b**, ... | Events | Expressions in bold courier text represent events. |
| A, B, C, D | Sets of events | Also used to represent process alphabets, which are themselves sets of events. |
| **A, B, C, D** | Sets of traces | Note the different font from the above sets of events. |
| i, j, k, l, m, n | Integers | k is also used as a real number in the definition of metric spaces. |
| P, X | Process variables | |

| | |
|---|---|
| F, G, H | Process functions |
| I | Ideal Test Set |
| Obs | Behaviours of a process |

All of the above notations may be indexed and still retain their respective representations. E.g. $R_2$ represents a predicate.

New Notation Introduced.

| | | |
|---|---|---|
| ^ | Set catenation | Defined in Section 5.5 |
| * | Dedicated event | Defined in Section 5.6 |
| DED | Dedicated process | Defined in Section 5.6 |
| $\mathcal{D}_{\mathcal{F}}$ | Special subset of F(STOP) | Defined in Section 5.6 |
| $\alpha$, $\beta$, $\gamma$, $\delta^n$, $\sigma^n$ | Semantic categories | Defined in Chapter 6 |
| $\overline{\alpha}$, $\overline{\beta}$ | Semantic categories | Defined in Section 7.11 |
| *Head*(s) | Head of a trace s | Defined in Section 7.9.3.2 |
| *Tail*(s) | Tail of a trace s | Defined in Section 7.9.3.3 |
| $\underset{P}{\geq}$ | Prefix ordering | Defined in Section 7.9.5 |
| $\underset{s}{\geq}$ | Suffix ordering | Defined in Section 7.9.5 |

Standard Notation

| | |
|---|---|
| $\mathbb{P}$(A) | Powerset of set A |
| $\mathbb{N}$ | Natural numbers |

| | |
|---|---|
| $\bigvee_{j=1}^{n} (Q_{ij})$ | Disjunction of indexed terms $Q_{ij}$ |
| $\bigwedge_{j=1}^{n} (Q_{ij})$ | Conjunction of indexed terms $Q_{ij}$ |

## CSP expressions

| | |
|---|---|
| $\parallel$ | Parallel |
| ${}_{A}\parallel_{B}$ | Alphabetized parallel |
| ; | Sequential composition |
| $\Box$ | Deterministic choice |
| $\sqcap$ | Nondeterministic choice |
| $\interleave$ | Interleaving |
| $\rightarrow$ | Prefix |
| $\backslash$ | Hiding |
| $f(P)$ | Change of symbol |
| $\mu P.F(P)$ | Fixed point of function $F$ |
| STOP | Deadlocking process |
| SKIP | Skip process |
| $\sqsubseteq$ | Partial ordering on traces |
| $P \upharpoonright n$ | Process $P$ restricted to the first $n$ events |

## Trace expressions

| | |
|---|---|
| #s | Length of s |
| s^t | Concatenation of s and t |
| s↾A | Restriction of s to set A |
| *Last*(s) | Last event in s |
| *First*(s) | First event in s |
| s(i) | The $i^{th}$ event in trace s |
| ✓ | Termination event |
| <> | Empty trace |
| **a** **<u>in</u>** s | The event **a** is in the trace s |
| Σ | The set of all events |

CHAPTER ONE

# INTRODUCTION

## 1.1 Introduction.

The general objective of this thesis is to investigate the use of formal methods in the verification of concurrent systems, with a particular bias towards an application to controllers for high speed machinery.

In this thesis, to provide a common basis for the work, it is useful to outline some of the general terms which are used. The word concurrent is used to describe systems which have the potential for parallel execution. This loosely describes those systems which are able to perform more than one action at the same time. This is in contrast to sequential systems which are required to perform their actions one after another.

A specification is a requirement which a particular system has to meet. It is usually a statement about the execution of the system. Specifications can be expressed in many ways. They are often expressed in terms of a verbal or written contract which dictates what the system must and must not do. Such specifications which express the system's requirements in plain English (or other language) are called natural language specifications. Alternatively a system's requirements may be expressed in a more exact manner, for example in terms of particular states the system can reach or particular sequences of labelled actions it may perform. Requirements written in such a mathematically precise manner are called formal specifications. From the human point of view, natural language requirements are usually simpler to comprehend than formal specifications, but they are often less precise and may obscure ambiguity.

Verification is the task of establishing that a system satisfies a particular requirement. In this thesis verification will be discussed in terms of the truth of a particular specification relative to a system. As with specification there are different approaches to verification depending on how the system and specification are structured.

## 1.2 Motivations.

A safety critical system is one in which incorrect operation has the potential to cause serious injury or loss of life [ Anderson 81 ]. There are an increasing number of concurrent applications in the medical, nuclear, aerospace and other industries which can be labelled as safety critical [ Leveson 93 ]. With respect to typical applications of high speed mechanisms, such as packaging machines, hazard free operation has conventionally been achieved by using centralized power trains which link all the relevant actuators. However, it is recognized that there are benefits for the design and flexibility of such systems if these conventional mechanisms are replaced by sets of independent software-synchronized actuators. The advantages of using such independent drives can only be realized if the drives are properly coordinated and synchronized to ensure safety critical operation.

The need to achieve a high level of confidence in the underlying control logic of such machinery points to an emphasis on verification and in particular on formal proof methods of verification [ Froome 88 ]. In the case of independent drive machinery any proposed formal method must include the concept of concurrency and the notion of real-time.

There are a number of possible formal proof methods currently available. Chapter 2 provides a discussion of the underlying principles and some specific examples of contemporary techniques. However for the purposes of the work presented in this thesis the particular formal method adopted is Communicating Sequential Processes, or CSP [ Hoare 85 ]. The reasons for this choice are given in more detail in Chapter 2 but essentially they can be summed up as follows

- CSP allows a formal description of concurrency by utilising an approach based on the communication between a number of interacting sequential processes.

- It possesses a range of process interpretations to cope with different system properties such as nondeterminism and time.

- It has a mature approach to verification based on an axiomatic proof system. Moreover, this system has the quality that the obligation to verify a large process relative to a specification can be translated into an obligation to verify a set of smaller processes relative to respective specifications. That is the system is compositional.

An alternative approach to formal proof is that of testing. This involves comparing the actual response of a system against its desired response as given by the specification [ Lanski 89 ]. As a rule testing is more procedural and easier to automate, whereas formal proof can be more concise and have a greater scope.

Although the approaches of formal proof and the concept of testing can be seen to be different, they do not by necessity exclude one another. In fact they can be complementary if formal rules are used to reduce the range of testing necessary to establish the truth of a specification, or alternatively if testing is used to establish some of the axioms necessary for factorizing a formal proof [ Bernot 91 ].

With respect to CSP much of the theoretical research to date has centred on improvements to the semantic theory such as higher semantic models and treatment of singular operators [ Davies 92, Roscoe 88a, Reed 90 ], or applying the techniques to the solution of specific problems [ Jackson 89 ]. Little has been published on using the formalism supported by CSP as a means to assist in the process of testing.

The theoretical contribution of this thesis was motivated by an acknowledgement that the methods proposed by CSP for the specification and verification of CSP possess a wide and useful scope and have the potential to assist in the development of safety critical systems. In particular there was an early realization that the formalism of CSP could be used to support a theory of testing, which in turn could serve to improve the approach to verification. This culminated in a desire to investigate methods of structuring such a theory, and applying them to the solution of problems.

## 1.3 Approach.

The approach adopted for the research contained in this thesis was to first establish a familiarity with the formal method CSP and the existing approaches to verification it supported, particularly with respect to its use in a control environment. This was achieved by considering a case study taken from the manufacturing industry. The advantage of this was that, as well as illustrating the use of CSP in the specification, design and verification of a controller, it would provide a tangible example with which to make observations about the methods of CSP and to draw comparisons with any developed techniques.

As a result of the case study a number of observations were made which proved to be useful in guiding the research. The first observation was that the axiomatic proof system of CSP constituted a powerful verification tool. The range of formal specifications to which it could be applied, that of continuous behavioural specifications [ Olderog 86 ], was sufficient to capture all the necessary controller requirements. However, it was also noted that the axiomatic proof system was complex. To effectively apply it required both an in depth understanding of the theory of CSP and a thorough knowledge of the system at hand in order to both propose suitable initial axioms and factorize the subsequent proof. It was also noted that one of the most complex factors of this axiomatic proof system was its treatment of recursion.

The second observation was less conventional. It transpired that the structure of the recursive sequential processes involved in the application controller was similar to that of a loop. That is they acted as a process which repeated one of the same limited set of behaviours in sequence. When two process behaviours are joined in sequence they are said to be catenated. Thus in terms of the behaviours of a process it was noted that certain recursive process structures could be linked to the concept of catenation.

The third observation was that in the axiomatic proof system developed for CSP the inference rule associated with recursion was based on the principle of mathematical induction. The underlying principle of mathematical induction is that the truth of a statement over a large, possibly infinite, domain can be inferred from its truth over a particular value or set of values.

The final observation was that a number of the specifications used in the case study had particular inductive properties over the catenation of process behaviours. For example if a certain specification S were true for a behaviour b then it could be shown that S was true for all process behaviours which at some point acted as b.

When taken together, these observations suggested that there was a class of processes and specifications for which the particular links between recursion, induction and catenation could be exploited to benefit. Coupled with an understanding of the structure of CSP they led the author to consider the following points

- It is observed that for certain processes recursion is linked to catenation and for certain specifications catenation is linked to induction. What is the specific nature of these links? How can they be explicitly defined and which processes and specifications exhibit them?

- If such a connection can be clearly established and understood, how may the interrelationships between the concepts of recursion, induction and catenation be exploited? Specifically, is it possible to use the principles which underlie mathematical induction to infer that a recursive process satisfies a specification by demonstrating that the specification satisfies a particular finite set of process behaviours?

It is the desire to investigate and answer these questions which provided the motivation for the method of Ideal Test Sets presented in this thesis.

## 1.4 Aims.

With respect to the observations made in the previous section, it is now possible to state some of the aims of this thesis

- The main aim of the thesis is to investigate methods by which the proof obligations necessary for verifying systems may be reduced. Specifically, the thesis aims to investigate how the axiomatic proof obligation on a particular process/specification pair can be reduced to testing a subset of the process behaviours over which the truth of a specification can be determined.

- Because the approach will be limited to a particular class of processes and specifications, it is imperative to define the boundaries of any methods proposed. It is envisaged that this would take the form of a range of syntactic definitions.

- It is recognized that the limitations of any methods put forward imply that a verification procedure based upon them may not be complete. It is therefore important that any restrictions in the scope of the methods can be compensated for by resorting to existing techniques. That is the approach should be compatible with current proof systems.

- In order to be assured of the methods proposed, they should be supported by a mathematical theory. Wherever appropriate, definitions should be given in a precise mathematical manner, and theorems should be justified by proofs. The rigour of such an approach will instil confidence.

## 1.5 Summary of the Thesis.

The main body of the thesis commences with Chapter 2 which opens by looking at current approaches to the formal specification and verification of systems. The underlying concepts and terminology employed are described and the different approaches to specification and semantic interpretation are discussed. Of particular note are the concepts of safety and liveness specifications, correctness and compositionality. The thesis then moves on to address different approaches to verification. Axiomatic proof techniques are discussed and the importance of the soundness and completeness of a logical system is stressed.

All axiomatic systems possess an underlying structure based on mathematical logic. The thesis outlines three main types of logic, namely propositional, predicate and modal. The basic points of each are outlined and theorems which relate to later work are presented.

The second chapter concludes by providing examples of specific formal methods which are in current use. Six methods are discussed and the rationale of choosing Communicating Sequential Processes is given.

Chapter 3 provides an introduction to the notation and concepts of CSP. It opens by describing the initial proposal for CSP as a programming language for distributed systems. It outlines some of the observations made about CSP, particularly with respect to its use of a synchronous communications primitive, and looks at some of the executable implementations of CSP such as occam. The chapter then moves on to discuss proof systems for CSP. It compares the earlier approaches to axiomatic proof based on specifications on the state of a process with later methods which specify a system's behaviour by sequences of interprocess communications. Then the formal notation which is employed by the rest of the thesis is given. Concepts such as events, traces, alphabets and catenation are formally defined. The syntactic operators are related and their effect on the traces of a process is described. It is shown how the formal treatment of the traces of a process leads to a mathematical semantic representation of traces in the form of the traces domain $M_T$. Finally it is demonstrated how this mathematical representation can be expressed as a metric space in order to support a theory of recursive processes.

Chapter 4 relates some of the higher semantic models which have been developed for CSP and details their respective properties. In particular the untimed failures ($M_F$) and timed failures ($TM_F$) domains are discussed in detail because of their use in a following case study. The case study itself involves the coordination of a high speed slider and drum mechanism. The system is initially specified by a natural language specification which is translated into a formal set of requirements and CSP models which satisfy them. To meet the specification two CSP models are proposed, an untimed model and a timed model. The timed model is necessary because of the timing constraints which have to be placed upon the system to ensure hazard free operation. Each of these models and their respective specifications are verified with the existing axiomatic proof system for CSP.

Chapter 5 looks at automated methods for assisting in the verification of complex systems. The two main approaches of model checking and theorem proving are outlined. Particular attention is paid to the concept of an ideal test, where a subset of behaviours can be used to determine the correctness of a specification relative to a system by exhaustive testing. These ideal tests suggest a means of verification based on exploiting the link between recursion and catenation exhibited by the case study of the previous chapter. In order to clarify the bounds of such a link, the concept of Catenary functions is introduced and a precise semantic definition of these functions is provided. An investigation is made to determine which of the CSP operators given in Chapter 3 are Catenary. The chapter ends by using this analysis to provide a syntactic definition for a Catenary function and discussing some of the limitations.

Chapter 6 extends the concept of ideal tests discussed in Chapter 5 to a formal definition of an Ideal Test Set for a CSP process and specification. An Ideal Test Set is defined as a subset of the behaviours of a process over which the truth of a specification can be determined. In order to structure a method of generating Ideal Test Sets the chapter looks at how specifications distribute over catenation. It introduces four categories, $\alpha$, $\beta$, $\gamma$, $\delta$, into which particular specifications can be placed by virtue of their properties over catenation. The chapter then investigates how these categories may be used as a basis for a method to structure Ideal Test Sets for particular process/specification pairs. In order to provide some syntactic rules relating to the limits of the specifications for which the method of Ideal Test Sets is suitable the syntax RSPEC is developed. The chapter concludes with the important result that for any process/specification pair where the process is defined by a Catenary function and the specification is defined by RSPEC then there is always a finite Ideal Test Set. Furthermore a procedure for generating such Ideal Test Sets for RSPEC is given.

Chapter 7 completes the definition of RSPEC presented in the previous chapter. By a series of examples the expressibility of RSPEC is examined and it is discovered that there is scope for improvement. The approach to improvement is given in the form of monotonic trace endomorphisms. These are functions which take a trace as an argument, return a trace as a value and which are monotonic on a given partial order. By using these functions in the place of the trace variables of RSPEC a new syntax for specifications is developed, that of ERSPEC. To support the extensions provided by ERSPEC two further categories relating the properties of specifications over catenation are introduced. It is shown how these new categories, $\overline{\alpha}$ and $\overline{\beta}$, are resolved with the existing theory of Ideal Test Sets to develop an extended theory. The chapter concludes with the result that it is possible to generate an Ideal Test Set for a process/specification pair where the process is defined by a Catenary function and the specification belongs to ERSPEC. A procedure for generating this Ideal Test Set is given.

Chapter 8 presents a complete syntax of ERSPEC and illustrates by means of a number of examples how it can be used to specify and verify system properties. The thesis then returns to the case study of Chapter 4 and demonstrates how the method of Ideal Test Sets can be used to verify the specifications which were previously established by axiomatic proof. The chapter then draws some conclusions about the suitability of the methods proposed and suggests some proposals for further work.

## 1.6 Novelty of the Thesis.

The novelty of the work in this thesis lies in the method of Ideal Test Sets and the theory which has been developed to support it. The extension of the existing notion of an ideal test to that of the formally defined Ideal Test Set is work presented in this thesis for the first time, as is the syntactic and semantic definitions of a Catenary function, the special set $\mathcal{D}_{\mathcal{F}}$ and the definition of $\mathcal{HTR}$ functions. The semantic categories $\alpha$, $\beta$, $\gamma$, $\delta$, $\bar{\alpha}$ ,$\bar{\beta}$, and the theorems and definitions relating to their closures and their use in the structure of Ideal Test Sets is also novel work, and was partially presented in [ Clarke 92b ]. In addition the the syntaxes RSPEC and ERSPEC which are used to indicate the limits of suitable specifications are introduced here.

The application of CSP to the Arbor drum problem is work originated by the author and was initially presented in [ Clarke 92a ]. The notations $\underset{s}{\geq}$, $\underset{p}{\geq}$ for partial ordering over traces are unique to this thesis but are derived from the basic principles of an ordering over traces presented in [ Hoare 85 ]. The concept of a dedicated event is based loosely on the special qualities exhibited by the termination event.

CHAPTER TWO

# FORMAL SPECIFICATION AND VERIFICATION

## 2.1 Introduction to Formal Methods.

As systems become more complex by their size or structure, it becomes increasingly difficult to manage them without resorting to some form of assistance. Formal methods aim to provide just such assistance by providing the field of software engineering with a firm scientific basis similar to that which exists for other engineering disciplines.

A formal method is a mathematically based technique for describing and reasoning with a system's properties [ Pagan 81 ]. It provides the software engineer with a framework with which to make assertions about a systems performance, and to prove rigorously properties about that system [ Stoy 77 ].

Formal methods facilitate the detection and correction of errors throughout the software development cycle. A possible result of an error is a failure [ Anderson 81 ] and the consequences of failures are undesirable, not only in terms of financial cost but also in terms of injury to personnel [ Leveson 93, New Sci 89 ]. Formal methods detect errors by revealing ambiguity, incompleteness and inconsistency throughout a systems development from specification through to design and verification.

There are at present a number of formal methods available, some established and others under development. Each has it own intended scope of application and accordingly adopts a particular approach. Between them they cover all phases of system development from specification and design through to implementation and verification. This thesis concentrates on two particular stages of development, those of specification and verification, with an emphasis on the latter.

This chapter addresses the area of formal specification and verification. It opens by discussing some of the terms and concepts commonly used in formal methods. Then it outlines the three main approaches to formal description: operational, denotational and axiomatic. The underlying principles of formal verification are related and a brief description of relevant mathematical logics is provided. Finally the chapter cites some specific examples of formal methods.

## 2.2 Formal Specification Language.

In general terms, the mathematical basis for a formal method is provided by a formal specification language [ Wing 90 ].

**Definition 2.1**: A formal specification language is a triple

$$\{ \mathtt{Syn}, \mathtt{Sem}, \mathtt{Sat} \} \qquad \text{Eq. 2.1}$$

where `Syn` is the languages syntactic domain,
`Sem` is the languages semantic domain,
`Sat` is a satisfies relation between them ($\mathtt{Sat} \subseteq \mathtt{Syn} \times \mathtt{Sem}$).

■

The syntax of a formal specification language is essentially its notation. It consists of a set of symbols along with a set of rules which determine how these symbols may be composed. A syntactic expression is called a sentence of a formal language. The semantics of a formal specification language is a universe of objects which attach an interpretation or meaning to a syntax. The satisfies relation comprises of a set of precise rules which define the relation between a specification and the semantic objects.

There are two broad, but not mutually exclusive, classes of formal methods: Property-Oriented and Model-Oriented [ Cohen 86 ].

### 2.2.1 Model-Oriented Specification Languages.

In the Model-Oriented or State-Based approach, a system is explicitly specified by a model giving its mathematical structure, most often in terms of sets, functions and relations. Initially a highly abstract formal model is proposed for the system in hand. The original model then undergoes a series of refinements, most often by the use of mathematical transformations, which result in a more detailed and less abstract model. With each refinement step the data structures involved are reified ( the process of adding more implementation detail ) and built upon. For the purposes of factorizing correctness proofs it is important that refinements follow a logical structure.

### 2.2.2 Property-Oriented Specification Languages.

In a property-oriented method the user specifies a system by stating a set of properties, or rules, which that system must satisfy. This is a more indirect method than the model-based approach because there is no explicit representation of the system. The rules which are used form a property-oriented specification fall into two categories, axiomatic and algebraic.

Axiomatic specification stems from work on the proof of correctness for abstract data types [ Hoare 69 ] where predicate logic assertions were made on the preconditions and post-conditions of an operation (Section 2.4.3). With an axiomatic specification, the system behaviour is expressed indirectly as a set of axioms written as predicates on a set of abstract data types, which the system must satisfy.

An algebraic specification defines a system property in terms of functions of an abstract algebra. Thus it still expresses the properties as axioms, but in the form of algebraic equations rather than predicate assertions.

### 2.2.3 Classification of Specifications.

The taxonomy of system specifications divides it into two broad categories. They are Safety Properties and Liveness Properties [ Lamport 77 ]. They are described as follows:

**Safety Properties**: Dictate those actions which a system must not perform.

**Liveness Properties**: Dictate those actions which a system must perform.

Thus, informally, safety properties stipulate that bad things will not happen, while liveness properties stipulate that good things will happen. A more formal treatment is presented by Alpern & Schneider [ Alpern 85 ].

## 2.3 Semantics of Formal Specification Languages.

The semantics of a formal specification language is the interpretation or meaning that is attached to a syntactic sentence of that language [ Nielson 92 ]. There are three established approaches to semantics: Operational Semantics, Denotational Semantics and Axiomatic Semantics.

### 2.3.1 Operational Semantics.

An operational approach to defining the semantics of a sentence of a formal specification language is concerned with how that sentence is executed, rather than merely what the effect of executing it is. This is achieved by defining an abstract machine which interprets a sentence by passing through a sequence of discrete states of that abstract machine. Thus the sentence is represented by an explicit sequence of computational operations, which represents how the states are modified during execution.

### 2.3.2 Denotational Semantics.

The denotational approach is concerned with the effect of executing a sentence of a formal specification language. With this approach a semantic valuation function [ Woodcock 88 ] is defined for each syntactic category of the language. The meaning of a sentence is represented by mappings to mathematical objects, such as sets. Thus there is no explicit concept of a computation sequence; the semantics are taken as an abstract mathematical form.

### 2.3.3 Axiomatic Semantics.

The axiomatic approach views a specification language's semantics as a mathematical theory for that language. That is a system in which properties about a sentence of the specification language can be expressed and proved or disproved. These properties are formal expressions called formulae. A formula may be either true or false; a true formula is called a theorem. The purpose of an axiomatic semantics is to determine which formulae are theorems. The semantics is made up of three kinds of component

**Syntactic Rules**: Determine the well-formed formulae which express the properties about the statements of the specification language.

**Axioms**: Basic theorems, which are accepted without proof.

**Inference Rules**: Mechanisms for deducing new theorems from established ones.

An axiomatic semantics provides a firm basis upon which to factorize program proofs.

## 2.4 Concepts of Verification.

Verification is concerned with proving that a specification is correct relative to some particular system, hence the term correctness [ Meyer 90 ]. The system could be an implementation, such as a program or piece of hardware, or it could be another formal specification. As a convention for this chapter such a system will be represented as a sentence of an arbitrary formal specification language.

Correctness itself is characterized by two approaches, namely partial and total correctness. These are defined as follows

**Partial Correctness**: When it can be shown that if a system terminates then it satisfies a particular requirement.

**Total Correctness**: When it can be shown that a system will terminate and it satisfies a particular requirement.

If a process is proved partially correct and also proved to terminate, then it has been shown to be totally correct. Termination is essentially a liveness property in that it states that a process will eventually terminate. It is this liveness which makes the above taxonomy of correctness relevant. The proof of liveness properties, and hence total correctness, is usually different in nature than the proof of safety properties [ Meyer 90 ].

### 2.4.1 Motivations.

Formal verification lies at the heart of the need for a formal method. Just as it is important to be able to define precisely the requirements of a system in terms of a formal specification, so it is necessary to be able to deduce with confidence that that system will be able to fulfil those requirements. Dijkstra [ Dijkstra 81 ] stresses the importance of a mathematical approach to verification by highlighting three important requirements of verification. They are

**Generality of Scope**: A good verification technique should have the ability to prove a wide range of properties about a system.

**Precision of Definition**: The technique should be able to represent system properties precisely without vagueness or ambiguity.

**Confidence in Proof**: The technique must provide a rigour which inspires confidence in the results achieved.

These three desirable properties are embodied by a mathematical approach to verification.

### 2.4.2 Compositionality.

Compositionality asserts that a process should be verifiable in terms of the axioms of its syntactic sub-processes [ de Roever 85 ]. Consider a process PROC with syntactic sub-processes $P_1,..,P_n$ about which a specification S must be verified. A compositional, or modular, proof system advocates that proving the correctness of PROC relative to S should be decomposed to verifying the correctness of each $P_i$ relative to a specification $S_i$ ( for $i = 1$ to n). Here $S_i$ is a specification of $P_i$ which is independent of PROC. This decomposition allows the prover to deal with processes as smaller, more manageable units. Once the specifications have been established for the syntactic sub-processes, there then exist a set of inference rules which allow the processes and specifications to be reconstituted in order to establish the correctness proof for the full system.

Apart from splitting the proof process into smaller modules, compositionality has an added attraction given by the Principle of Compositionality [ de Roever 85 ].

**Principle Of Compositionality**: A Correctness Proof for a process should be similarly structured as that process, showing that a proof of a process is composed of the proofs of the constituent parts of that process.

If the inference rules are based on the syntactic operators, and the initial axioms are provided by specifications which hold for the syntactic sub-processes, then the principle of compositionality implies syntax directed proofs [ Lamport 84 ]. That is the proof has a structure which can be determined by the syntactic definition of the process.

### 2.4.3 Foundations.

The early work in the field of formal verification was done by Floyd [ Floyd 67 ] and Hoare [ Hoare 69 ]. Floyd, in a paper to the American Mathematical Society, presented his method of inductive assertion. This attaches assertions about the state of a system represented by a flow chart at points on the chart. The assertion would be true whenever execution reached that point. For a looped flow chart it was proved that if an assertion is initially true then it will be true when execution next reaches it, thus permitting an inductive proof. It was also suggested that a programming language could be defined in terms of proof rules.

This suggestion was taken up by Hoare who defined a small programming language in terms of a logical system of axioms for each syntactic construct. The method formally characterizes a specification in the form

$$\{Q\} \quad P \quad \{R\} \qquad \text{Eq. 2.2}$$

Here Q and R are predicates, called the precondition and post-condition respectively, and P is a program. Equation 2.2 has the notation that if execution of P is begun in a state satisfying Q then, upon termination, P will satisfy R.

Hoare's method placed great emphasis on axiomatic techniques and the soundness of the proof system ( Section 2.5.2.2 ). By aiming to provide an axiomatic inference rule for each of the various program constructs it laid the foundation for deductive program proofs as well as compositional proof systems. The compositionality of the axiomatic approach is seen as an advantage in that it allows a modular approach to design and it permits the structuring of a proof around the syntactic definition of the process.

### 2.4.4 Parallel Verification.

It was recognized by Owicki and Gries [ Owicki 76 ] that although both the inductive assertion method and the axiomatic approach addressed partial correctness for sequential programming structures, they do not account for stronger concepts such as parallelism and total correctness.

Owicki's work on the correctness of parallel program structures has since become the seminal basis for concurrent verification techniques. This extends the axiomatic approach for sequential programs in order to represent parallelism as described by two languages, GPL and RPL. GPL starts and terminates several sequential programs together within a cobegin .. coend statement. The control and expression of parallelism is achieved via the primitive await construct.

Partial correctness in GPL is established by considering each sequential process in isolation and establishing pre-conditions and post-conditions in the standard axiomatic fashion. Then the proofs of the programs must be shown to be interference-free, i.e. that no await statement from one program interferes with the proof of another. If the processes are not interference-free then examples illustrate [ Barringer 85a ] that they can be transformed by auxiliary variables to achieve interference-free proofs. Once this property has been established the pre-conditions and post-conditions of the parallel program are the logical conjunction of the pre-conditions and post-conditions of the constituent sequential programs. The second language, RPL, employs a shared read/write variable method.

A method for proving total correctness in parallel processes has been suggested by Flon and Suzuki [ Flon 81 ]. This proves the total correctness of a parallel program by recognizing the equivalence between a parallel program communicating through shared memory and a non-deterministic sequential form. Total correctness specifications for the parallel program are then translated into total correctness properties for non-deterministic sequential programs and solved with a given proof system due to Gries [ Gries 81 ]

Current approaches to verification of parallel systems tend to be specific to particular formal methods. Examples are given in Section 2.7.

## 2.5 Methods of Verification.

Section 2.4 discussed in general terms the background for verification and its application to parallel systems. It is now appropriate to consider in more detail what different approaches there are to verification.

### 2.5.1 Direct Proofs of Partial Correctness.

As well as the syntax (`Syn`) and semantics (`Sem`) of a formal specification language Definition 2.1 illustrates the existence of satisfies relations (`Sat`) between the two. These relations, or consequence closures [ Cohen 86 ], form the basis for the rules of logical deduction which permit reasoning over the semantics of a system and thus allow verification.

Examples have shown [ Nielson 92 ] how such consequence closures can be used to achieve proofs of partial correctness in both operational and denotational semantic models.

#### 2.5.1.1 Operational Verification.

The operational approach to verification is by symbolic execution [ Darringer 78 ]. Here the actual object values representing initial states and input are expressed as metavariables called symbolic object values. Symbolic execution generates a trace of the system. The semantics are then derived as a function, $\Phi$ say, of the symbolic object values explicitly from the symbolic execution traces. To show the correctness of a specification it is first translated into a function, $\Psi$ say, of symbolic values. The proof of correctness is then a matter of algebraically establishing equivalences between $\Psi$ and

Φ. Because the semantics are independent of any particular initial and final states symbolic execution obviates the need for testing with specific subsets of data.

One important limitation of the operational approach is its reliance on generating a trace of the execution. For conditional constructs all alternative traces must be generated which increases the computation involved. Furthermore, recursive constructs may lead to infinite traces. This latter case, however, may be overcome with inductive arguments [ Berg 82 ] .

### 2.5.1.2 Denotational Verification.

With the denotational approach to verification the semantics of each syntactic operator of the formal language are defined by semantic valuation functions. The semantics of a formal sentence are then generated by evaluating all the relevant semantic valuation functions. This is achieved algebraically by solving a set of fixed point equations [ Stoy 77 ] to result in a function, say $\xi$. A specification is expressed as a function of symbolic values, say $\varphi$. The correctness of that specification is established by algebraically establishing equivalences between $\xi$ and $\varphi$.

Although it has obvious similarities with the operational approach in establishing algebraic equivalences, the denotational technique differs in the way in which it constructs the semantic function. The hallmark of the denotational semantics is the compositional nature of its semantic valuation functions.

### 2.5.2 Axiomatic Verification.

The direct verification methods described above are often too detailed to be of practical use for complex systems. What is required is a verification method which allows abstraction from such semantic detail as well as supplying a convenient mechanism to support proofs of correctness. Axiomatic semantics provides a basis for this.

The idea of the axiomatic approach is to associate semantics of formal languages with logical assertions. These are analogous to the axioms and inference rules of a logical calculus and such they provide the ideal medium for factorizing verification proofs.

Initial axiomatic semantics [ Hoare 69 ] were able to prove partial correctness properties. If restricted to non-looping sequential syntactic sentences it is reasonable to assume total correctness because such structures always terminate. However, with the introduction of recursive constructs this assumption is not generally true and it was found that the axiomatic semantics had to be extended to permit proofs of total correctness [ Nielson 92 ].

### 2.5.2.1 Inference Rules.

An inference rule is the means by which a set of initial axioms, (premises) are composed to form a new theorem (conclusion). Typically, this is written:

$$\begin{array}{l} R_1 \\ \underline{R_2} \\ \Rightarrow \quad R_3 \end{array} \qquad \text{Eq. 2.3}$$

Equation 2.3 is interpreted as " If $R_1$ is true and $R_2$ is true then $R_3$ is true ". A syntactic inference rule is one which is associated with a particular syntactic construct of a formal language, and which defines the conclusions given the initial premises.

The purpose of inference rules is to provide the mechanism for proofs. A proof is defined as a sequence of valid statements, each accompanied by a justification for its validity, which lead from initial premises to a final conclusion [ Backhouse 86 ]. The justification for each step of a proof may come from an inference rule or from an axiom.

### 2.5.2.2 Soundness and Completeness of an Inference System.

An axiomatic semantics provides the user with a means of proving properties about a process by applying inference rules to a set of established axioms. However, to have faith in any such semantics and to assess how useful they may be, there are two criteria on the semantics, namely soundness and completeness [ Borowski 89 ]

**Soundness**: A system is sound if the application of the inference rules does not allow a contradiction to be proved from the axioms.

**Complete**: A system is complete if every property can be proved or disproved by suitable application of the inference rules to the axioms.

These properties are not trivial to prove and they are important in terms of the applicability of the axiomatic semantics. Soundness is the more important of the two. It is intrinsic to any deductive system. If an axiomatic semantics is not sound then the theorems which it is used to prove have no value. Completeness is a secondary consideration. An incomplete axiomatic semantics is restricted by those theorems which it is unable to prove. However, the validity of those which it may proved is not dependent on completeness. For example, Gödel's famous incompleteness theorem [ Hamilton 78 ] asserts that mathematical arithmetic as a whole is incomplete, yet there are still valid arithmetic theorems.

## 2.6 Mathematical Logics.

Up to now the discussion of axioms and inference rules has been generic. It is important to note, however, that most formal methods are based on one of three established fields of mathematical logic. They are Propositional Logic, Predicate Logic and Modal Logic, and in themselves they form a hierarchy. Predicate logic is an extension of Propositional Logic, and Modal Logic is an extension of Predicate Logic.

### 2.6.1 Propositional Logic.

A proposition is a statement of a formal language which is either true or false and which remains so regardless of any environmental considerations. Like any formal language, propositional logic has a precisely defined syntax, given by Definition 2.2, and a semantic interpretation. A statement of propositional logic is interpreted by its truth table which maps the statement to a member of the set {TRUE,FALSE}.

**Definition 2.2**: Propositional calculus has the following syntactic definition in BNF

```
PROPOSITION ::=  PROPOSITION
                 | "¬", PROPOSITION
                 | "(", PROPOSITION, "∧", PROPOSITION, ")"
                 | "(", PROPOSITION, "∨", PROPOSITION, ")"
                 | "(", PROPOSITION, "⇒", PROPOSITION, ")"
                 | "(", PROPOSITION, "⇔", PROPOSITION, ")";
```

Furthermore the connectives are labelled as follows

| | |
|---|---|
| ∨ | Disjunction |
| ∧ | Conjunction |
| ⇒ | Conditional |
| ⇔ | Biconditional |
| ¬ | Negation ■ |

In broad terms a proposition can be thought of as a set of basic statements, or simple propositions, compounded together under a set of operators called connectives. This set of connectives is { $\wedge$ , $\vee$ , $\Rightarrow$ , $\Leftrightarrow$ , $\neg$ }. For example consider two simple propositions Q and R. The statement ' $Q \wedge R$ ' is interpreted as TRUE if both Q and R are TRUE, but FALSE otherwise. This is illustrated by the truth table, given in Fig 2.1 for all the connectives.

| Q | R | $Q \wedge R$ | $Q \vee R$ | $Q \Rightarrow R$ | $Q \Leftrightarrow R$ | $\neg Q$ |
|---|---|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE | FALSE | FALSE | FALSE |
| FALSE | TRUE | FALSE | TRUE | TRUE | FALSE | TRUE |
| FALSE | FALSE | FALSE | FALSE | TRUE | TRUE | TRUE |

*Fig. 2.1 Truth Table For Propositional Logic.*

2.6.1.1 Expressing Propositions in Normal Form.

Two propositions are equivalent if they both have the same truth table. For example the propositions ' $Q \Rightarrow R$ ' and '$\neg Q \vee R$ ' have the same truth table and are thus equivalent. It can be seen from this example that every occurrence of the conditional connective ($\Rightarrow$) may be replaced with the negation ($\neg$) and disjunction ($\vee$) connectives. A similar result for the biconditional connective ($\Leftrightarrow$) leads to Theorem 2.1 [ Hamilton, 78 ].

**Theorem 2.1**: Every proposition is equivalent to a proposition in which the only connectives occurring are from the subset { $\wedge$ , $\vee$ , $\neg$ }. This subset is known as an adequate set of connectives. ■

*Proof*

See [ Hamilton 78 ] □

This theorem leads to two important corollaries on the standardization of propositional statements.

**Corollary 2.1**: Every statement which is not a contradiction is equivalent to a restricted statement of the form

$$(\bigvee_{i=1}^{m} (\bigwedge_{j=1}^{n} Q_{ij})) \qquad \text{Eq. 2.4}$$

where m, n are non-negative integers and each $Q_{ij}$ or $\neg Q_{ij}$ is an elementary proposition (See Section 6.3 for an explanation of the notation). This is termed the disjunctive normal form. ■

**Corollary 2.2**: Every statement which is not a tautology is equivalent to a restricted statement of the form

$$(\bigwedge_{i=1}^{m} (\bigvee_{j=1}^{n} Q_{ij})) \qquad \text{Eq. 2.5}$$

where m,n are non-negative integers and each $Q_{ij}$ or $\neg Q_{ij}$ is an elementary proposition. This is termed the conjunctive normal form. ■

### 2.6.2 Predicate Logic.

Whereas the propositional logic was concerned with the absolute truth of a statement, predicate logic is concerned with the relative truth of a statement. A predicate is an expression which ascribes a property to some thing, called the subject. More formally it is a mapping from a set of objects (subject) to the space {TRUE, FALSE}. Predicates are combined with connectives in the same manner as propositions, but there are two additional operators associated with predicates, the universal quantifier and the existential quantifier.

#### 2.6.2.1 Universal Quantifier.

For a predicate R and set G of the same type as the subject of R, the universal quantifier typically has the notation

$$\forall g \in G \bullet R(g) \qquad \text{Eq. 2.6}$$

This is interpreted as 'for all members of the set G predicate R holds'.

#### 2.6.2.2 Existential Quantifier.

For a predicate R and set G of the same type as the subject of R, the existential quantifier typically has the notation

$$\exists g \in G \bullet R(g) \qquad \text{Eq. 2.7}$$

This is interpreted as ' There exists at least one member of G for which R holds '.

### 2.6.3 Modal Logics and Temporal Logic.

Modal Logic adds another dimension to description by predicates, and as such it is an extension of predicate logic [ Manna 88 ]. Predicate logic is appropriate for describing static situations in the sense that it makes statements about basic objects. Modal Logic extends this notion to that of dynamic change from one situation, or world, to another. It portrays a set of static situations (a universe of worlds) and relates the rules of change between them.

Modal Logics attempt to define different notions of truth, such as truth by necessity or probabilistic truth. There exist a number of modal logics, one of the most common and useful of the modal logics in terms of real-time systems is temporal logic [ Pnueli 86 ]. Temporal logic is an expression of how properties alter through the passage of time. As well as the standard notation of predicate calculus it possesses temporal operators. There are variants of temporal logic with different temporal operators. Most commonly the operators ○,◇, and □ appear. When interpreted with respect to a sequence of states, with R as a predicate on the state of the system interpretations are

| | |
|---|---|
| ○R - [next] | states that R will hold in the next state. |
| ◇R - [eventually] | states that R will hold in some future state. |
| □ R - [henceforth] | states that R will hold for all future states. |

For example, for predicates Q, R the temporal logic sentence $R \Rightarrow \Diamond Q$ says that if R holds in the current state, then Q will eventually hold.

## 2.7 Examples of Formal Methods.

This section looks at a selection of current formal methods for specification and verification. The choice of methods has been made on two criteria. First, the popularity of the particular method and secondly the approach to verification. The exception to these is PAISLey, which lacks effective verification but provides an example of an executable specification language.

### 2.7.1 VDM.

The Vienna Development Method, or VDM, evolved from research carried out at IBM Vienna Laboratories in the 1970's. The first book on the subject appeared in 1978 [ Bjørner 78 ] and since then VDM has achieved a level of maturity and acceptance [ Jones 86 ]. It has been taught widely and used in a number of applications. Examples include studies in the field of telecommunications and the specification of electronic mail systems [ VDM 87 ].

VDM possesses more scope than a formal specification language in that it also provides rules and procedures which outline the progression through every stage of system development. The system development involves a series of refinements to the initial model, and from this it can be seen that VDM adopts a model-oriented approach to specification. Each refinement step consists of adding more implementation detail until a desired level is achieved. Within the framework of its development technique VDM has a precisely defined syntax and semantics. The syntactic notation for VDM was originally the metalanguage META-IV, this has since been extended to a British standard notation VDM-SL (Specification Language) [ ISO 91 ]. The underlying semantics of VDM is denotational, mapping the syntactic constructs to set-theoretic data types. System specifications are expressed as predicates in a first order predicate calculus which forms the basis of a logical inference system.

Each VDM model consists of a set of modules which each possess an internal state represented by variables. VDM operations may be defined which are able to use and amend the values of these variables. The use of modules illustrates object orientated techniques for VDM. Object-Oriented Design (OOD), in which a system is characterized as a set of objects that pass messages to one another, can then be exploited [VDM 87].

### 2.7.2 Z and the Refinement Calculus.

Z is a formal specification language developed at the Programming Research Group at Oxford. It is a popular, well used formal method with published case studies [ Hayes 87 ]. Z takes a model-oriented approach to specification with an underlying denotational semantics based on set theory. It uses first order predicate logic and axiomatic description to capture and prove the correctness of specifications. Thus in many ways Z is similar to VDM. It differs in two main respects. Primarily, VDM is a development method, concerned with aspects of design such as proof obligations etc. Z is simply a specification and verification method. Secondarily, Z possesses a major notational difference in the schema [ Lightfoot 91 ]. This is essentially a means of expressing part of a specification with an emphasis on its readability and subsequent inclusion in natural language documents. An example of a schema is:

```
┌── SCH ──────────────────
│ a, b : ℕ
├─────────────────────────
│ a < b
└─────────────────────────
```

*Fig 2.2 A Z Schema.*

This schema is called SCH and it declares two variables a and b, with the property that a must always be less than b. This has an equivalent linear form

$$\mathrm{SCH} \triangleq [\, a, b : \mathbb{N} \mid a < b \,]$$

where ≙ indicates textual equivalence. In addition to providing clarity such schemas can be regarded as units and manipulated by a set of Z operators which are analogous to those of predicate calculus.

It has been noted that although Z has been widely used to describe several large systems its use has been significantly less in the later stages of the development cycle, particularly implementation. A response to this is the refinement calculus [ King 90 ]. This is based on Dijkstra's language of guarded commands [ Dijkstra 76 ] and is similar in notation to Z. The refinement calculus is intended to be used in conjunction with Z. Z is used in the specification and design stages, where its schema calculus can be used to structure and simplify work. Then the laws of refinement calculus are employed to develop the abstract Z model into an implementation. The Refinement Calculus is currently at an early stage of development.

### 2.7.3 Concurrent Theories.

The two above techniques are directed solely towards the analysis and design of sequential systems. There exist techniques to extend VDM to cover concurrent systems. VDM has been incorporated with time and CCS to create the MOSCA specification Language [ Toetenel 92 ]. However, these approaches are still in an early stage of development. As extensions of sequential methods they were not originally intended to model concurrency. The next sections provide examples of techniques designed with concurrency in mind.

### 2.7.4 Petri Nets.

Petri nets were developed by Petri in the early 1960's at GMD Bonn, Germany [ Petri 66 ]. They are a popular technique and are the subject of extensive research and introductory texts [ Murata 89, Peterson 81 ].

Petri nets are influenced by automata theory and were the first general theory of concurrency. They are a powerful technique which models concurrency by representing the causal relationships between events and conditions. A Petri net takes the form of directed bi-partite graphs [ Wilson 84 ] with two types of nodes: places which represent the state of the system, and transitions which represent the system's actions. The distribution of tokens, indicated by black dots, indicates the current state. Tokens move around the net, reflecting system dynamics, by the firing of transitions. The firing rules are

i) A transition is enabled when each of its input places has at least one token.

ii) A transition may only fire if it is enabled.

iii) Upon firing, one token is removed from each input place and a token is placed at each output place.

The destination of arcs from a transition to a place are output places, the origins of arcs from places to a transitions are input places.

The mathematical representation of a Petri net is achieved via a graph-theoretic incidence matrix and state equation. The incidence matrix **A** captures the causal properties of a net by relating the input and output places of each numbered transition. The firing of transition `i` is then represented by the elementary vector $u_k = [\ 0\ 0\ ..\ 1\ ..\ 0\ ]$, where the 1 is the `i`$^{th}$ element. It is then possible to generate a sequence of markings $M_0$, $M_1$, .. with the state equation

$$M_k = M_{k-1} + \mathbf{A}^T u_k \qquad \text{Eq. 2.8}$$

The reachability of a Petri net is the set of markings which it is possible to achieve from an initial marking $M_0$ by firing a legitimate sequence of transitions. The state equation enables the construction of a graph with markings at the nodes and transitions at the edges. By starting at the initial marking it is then possible to trace the reachability of the net by following the edges of this graph. This graph is called the reachability tree. To establish invariant properties of a Petri net the reachability tree is used to compare those properties against the reachable states. However, it is important to note that while this method of verification has rigour, it cannot be described as formal. This is because Petri nets do not possess a suitable mathematical structure for generating correctness proofs. It should also be noted that the method lacks the ability to establish liveness properties.

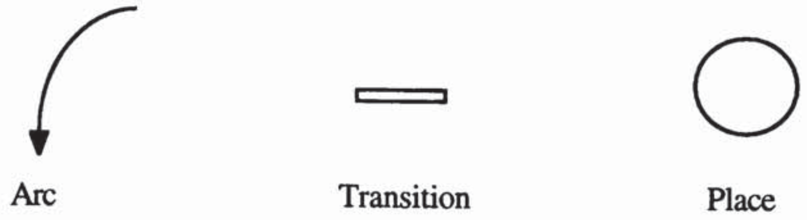


*Fig 2.3 Graphical Syntax of Petri nets.*

There exist two main extensions to standard Petri net theory to represent real-time properties. Time Petri Nets [ Merlin 76 ] attach a minimum and maximum delay to each transition. Timed Petri nets [ Berthomieu 83 ] associate a discrete time period to each transition such that after being enabled that transition must fire within this period.

### 2.7.5 PAISLey.

PAISLEY ( Process-oriented, Applicative and Interpretable Specification Language ) was developed at the AT&T Bell Laboratories in New Jersey, USA, in the late 1970's and early 1980's [ Zave 82 ]. It is formed from merging the concepts of two models of digital computation, asynchronous processes and functional programming [ Zave 86 ]. PAISLey aims to be a simple and coherent language, with relevance to education as well as real applications. It illustrates many central concepts such as concurrency, data flow and real-time constraints.

PAISLey is an executable specification language. That is, it forms the basis of a software development process which translates a specification into an implementation. A PAISLey specification is written as a set of functional definitions, the behaviour of which is represented by a notional dataflow paradigm. This serves to guarantee the integrity of data by controlling its flow and additionally maximizes concurrency.

Inter-process communication and thus concurrency is effected by special functions called exchange functions. The nature of these functions provides PAISLey with a synchronous communications primitive, but unlike the CSP input/output operator ( Section 2.7.6 ) exchange functions permit two way flow of data.

An important stage in the software development cycle is verification. Limited verification is attainable by representing the control flow of a PAISLey specification as a Petri net. It is then possible to verify properties via a reachability tree in the usual fashion. The underlying semantics of PAISLey is operational but they have not yet been captured in the form of proof rules [ Zave 91 ]. Thus there exists no axiomatic inference system. Additionally, the nature of the exchange functions also serve to prohibit the composition of two PAISLey processes into one equivalent process.

PAISLey is able to represent real-time systems by imposing property-oriented timing constraints. These form part of the original PAISLey syntax and semantics and they relate to the evaluation time of functions.

### 2.7.6 Communicating Sequential Processes.

Communicating Sequential Processes, or CSP, was developed by C.A.R. Hoare in the late 1970's at Queen's University, Belfast and later at the Programming Research Group at Oxford University. The seminal paper was published in 1978 [ Hoare 78 ] and was followed by a book [ Hoare 85 ] in which the original theory has been much refined and clarified.

CSP is a discrete event based process algebra which adopts a denotational approach to semantics, after the fashion of the lambda-calculus [ Stoy 77 ]. It is influenced by Dijkstra's language of guarded commands [ Dijkstra 76 ]. The underlying theory is set based with processes being represented in an observational manner in terms of traces, or sequences of events which they perform. In his book, Hoare discusses two styles of denotational semantics, the traces model and the failures model. The traces model is the simpler of the two and describes a process in terms of the traces it performs. The failures model extends this concept by describing a process in terms of its failures. A failure is an ordered pair which relates a trace to the set of actions which the process may be unable to engage in after having performed that trace. These two semantic models form the basis of a larger hierarchy [ Reed 90 ].

As well as providing for parallelism, interleaving and communication, the syntax of CSP also distinguishes between two types of choice, non-deterministic and deterministic. Specifications in CSP are represented as predicates over a particular semantic domain. Correctness is established with an axiomatic proof system which infers complex properties from basic initial axioms. This proof system is compositional, and there exist well defined proof rules associated with each syntactic operator which provide syntax directedness to the proofs.

The motivation and history of CSP are discussed in detail in the following chapter. However, it is important to mention one influential extension to CSP, and that is Timed CSP [ Davies 89a ]. TCSP extends the syntax of conventional CSP by only one operator, a delay primitive. But it has associated with it a range of semantic domains which give it a powerful capability for real-time description. TCSP also represents specifications as predicates under an axiomatic proof technique.

### 2.7.7 Calculus of Communicating Systems.

The calculus of Communicating Systems, or CCS, was developed in the late 1970's by R. Milner at the University of Edinburgh. The first book on the subject was published in 1980 [ Milner 80 ] and since then CCS and its extension, Synchronous Calculus of Communicating Systems, or SCCS, have become recognized algebraic techniques for the formal description of parallel processes.

The underlying semantics of CCS is operational, based on labelled transition sequences. Later versions of the semantics are also compositional [ Milner 89 ]. The semantic theory of CCS is intentionally biased in favour of interaction or communication rather than the state of a machine. CCS represents a system as a collection of agents which communicate with each other. Agents perform actions which are either external actions - which are observable to the external environment, or hidden actions - which are not observable. Associated with each agent is a label, which comprises all the external actions which that agent may perform. CCS also possesses operators for the parallel composition of agents (conjunction), choice between agents (summation) and hiding actions from the environment (restriction).

CCS and CSP have many aspects in common. Comparisons can be made with the above constructs and the alphabets, processes, events and operators of CSP. However, there do exist subtle but important distinctions between CSP and CCS [ Brookes 83 ]. They differ in their approach to verification.

CCS relies heavily on the concept of bisimulation relations. A bisimulation relation establishes a class of semantic equivalence between agents. There are broadly two categories. Weak bisimulation indicates that two agents have a class of equivalence in their external behaviour, whereas Strong bisimulation indicates equivalence of external and internal behaviour. Specifications are conventionally represented in an algebraic form. The approach to verification in CCS is to establish algebraically a bisimulation relationship between a process and a specification. This contrasts with the axiomatic approach of CSP (above). (As an alternative to algebraic specification a simple specification language called process logic, or PL, has been proposed. This can represent specifications in a predicate logic form, but is only able to represent safety properties [ Milner 89 ] ). Another difference between the process algebras is the inability of CCS to explicitly distinguish between deterministic and non-deterministic choice.

CCS is considered to have a more elegant algebraic approach than CSP. This is typified by the fact that CCS eschews the conventional method of inter-process communication by data transmission. Instead it employs a combination of synchronization with a non data passing primitive and summation to give the power to express the communication of variables of any kind. This has the advantage that it restricts CCS to a purer calculus, where conditional statements are expressed in algebraic forms.

The extended form of CCS, SCCS, aims to clarify some of the arbitrary features of CCS which are caused by its asynchrony [ Milner 83 ]. That is the fact that concurrent agents perform actions at unknown relative rates. SCCS insists on synchrony of agents, by which actions proceed in lockstep at defined time intervals given by a conceptual global clock. It has been shown that CCS is derivable from SCCS, and thus SCCS forms a more comprehensive but more complex process algebra [ Scholefield 90 ]. The 'clock ticks' of SCCS provide a basis for temporal reasoning.

Another approach to incorporating real-time properties into CCS is suggested by Tofts [ Tofts 90 ] in his weak Timed CCS, or wTCCS. This extends CCS by introducing primitives for asynchronous and synchronous delay and defines them as operational semantics, separate from the untimed actions.

## 2.8 The Rationale of Choosing CSP.

This thesis requires a means of analysing the control logic of high speed machinery. To achieve this three important considerations were identified. First a method was needed that possessed the ability to express a wide range of system properties and the means to reason about such properties with confidence. Secondly the method should be able to cope with the fact that many contemporary control systems are distributed in nature. The complexities inherent in distributed systems also imply that a method which permitted the user to abstract away from unnecessary detail would be advantageous. Thirdly, the critical nature of timing constraints in high speed control points to an emphasis on a technique with well developed temporal properties.

Dijkstra [ Dijkstra 81 ] has noted that precision, scope and confidence are enhanced by a mathematical or formal approach to system description. This points to the use of a formal method for carrying out the necessary analysis. Thus a formal method which can deal with concurrency, allows abstraction from detail and has developed temporal properties is sought.

CSP has a representation of concurrency and communication. This is coupled with a formal approach to specification and verification. Since its development in the 1970's

CSP has been the subject of research and application [ Jackson 89 ]. This has served to refine the techniques, placing them on a firm theoretical basis and increasingly giving CSP the status of a mature formal method.

The clear distinctions between syntax and semantics in CSP permit a hierarchy of semantic domains for interpretation [ Reed 90 ]. Within this hierarchy there exist different semantic domains with the power to express such properties as nondeterminism, divergence, deadlock and stability. The advantage of such a hierarchy is that it permits a user to select an appropriate domain depending on the properties to be proved. Additionally the extensional approach of this hierarchy, in which complex models are direct extensions of simpler models, inherently permits contrasting properties, such as safety and liveness, to be expressed in one semantic domain [ Hoare 91 ].

Another important factor in choosing CSP is its axiomatic proof system. Inference rules are associated with each syntactic operator and this permits both a compositional approach to proof and the ability to use the structure of the syntax to factorize a proof.

The timed extension to CSP permits real-time properties to be specified and verified. TCSP adopts most of the conventions of CSP. It has the same style of proof system, a hierarchy of semantic domains which can be related to those of CSP, a similar syntax and similar representation of specifications. There also exist well defined mappings between the timed and untimed semantics which permit specification by refinement [ Schneider 90 ]. CSP also has a well developed theory of recursion with a fixed point representation of recursive processes. This is based on establishing metric spaces over each of the semantic domains [ Reed 90 ].

Thus CSP and TCSP are established formal methods with axiomatic proof systems, a hierarchy of domains which permit abstraction, an established real-time representation and a mathematical treatment of recursion. These properties make CSP an attractive formal method for the analysis of real-time control systems.

## 2.9 Summary.

This chapter has discussed formal methods of specifying and verifying control systems, and described some of the different approaches available. It has outlined the mathematical concepts and notations often used in formal methods. The first two specific examples cited, Z and VDM, serve to illustrate two sequential formal methods which are well used and understood. These methods are similar but differ in their intended applications. VDM is primarily a development method whereas Z is a specification and verification tool.

The chapter then addressed different methods of modelling and studying parallel systems. Petri nets proved to have a simple graphical structure, which provides an approachable user interface and adds to the designer's understanding of the system. They also possess a well defined matrix based semantics which makes an operational approach to verification convenient to implement in the form of reachability trees. However, pure Petri net theory is only effectively able to communicate the flow of control information between concurrent processes, as opposed to data. PAISLey is not so restricted, its dataflow paradigm gives it a good basis upon which to describe the flow of both data and control communications. In terms of verification, both techniques possess equal flexibility. In their unextended forms neither support an axiomatic semantics.

Two other methods which are based on the concept of describing parallel systems in terms of their interprocess communications were then considered, CCS and CSP. Both methods are process algebras and support a highly mathematical approach to the description of systems. To different extents they eschew a state based approach in favour of a transitional approach. However, they differ in their approach to to verification and semantic style. CCS adopts an operational semantics and favours an algebraic approach to verification based on the concept of bisimulation. CSP has a denotational semantics and has an axiomatic approach to verification. While CCS possibly advocates a more mathematical approach than CSP (it has fewer primitive operators) CSP has in its favour a wide range of semantic interpretations and a more mature real-time theory.

In conclusion it was felt that CSP constitutes a useful and productive method for the specification, analysis and verification of real time control logics. Chapter 3 provides a more detailed and constructive review of CSP, its origins and present form.

CHAPTER THREE

# COMMUNICATING SEQUENTIAL PROCESSES

## 3.1 Introduction.

Fundamental concepts which underlie CSP are modularity and compositionality. Systems are perceived as processes which may readily be decomposed into parallel subprocesses that are able to interact with one another as well as their common environment. The properties of such subprocesses are easier to analyse than those of the whole process. The simplicity of construction of a process from the parallel composition of its subprocesses is placed on a par with the sequential composition of statements in a conventional programming language [ Hoare 85 ].

This chapter aims to illustrate the principles and notation which constitute the formal language CSP. It opens by discussing the origins of CSP as a proposed programming style for distributed systems. This introduces some of the fundamental concepts of parallelism, such as communication and guarded commands. The contemporary observations and criticisms made about this proposal are introduced and discussed. These points are illustrated by examples of specific languages which, in varying degrees, implement the concepts behind CSP.

The chapter then considers methods for the verification of processes in CSP. Two main axiomatic approaches are studied and discussed. The first adopts an approach to specification using precondition and post-condition predicates. Methods of this type employ proof systems based on establishing non-interference between the proofs of the constituent sequential processes. The second considers a system where specifications are written as predicates on transitional trace models. This system obviates the need to establish non-interference between the proofs of sequential processes.

The text then moves on to consider the development of CSP as a mathematical theory for the description of concurrency. A semantics for the theoretical description of CSP processes is described. This trace semantics is based on the sequence of transitions and communications which a process exhibits externally. It upholds a more rigorous definition and understanding of a syntax for CSP. This syntax is outlined and shown to extend the original concepts of CSP. Particular attention is paid to the recursion operator. It is demonstrated how correctly structured recursion leads to an avoidance of unchecked process behaviour, or divergence. This is supported by a topological metric space representation for CSP which allows a distinction to be drawn between divergent and non-divergent processes.

Finally an axiomatic proof system for CSP based on the trace semantics is outlined and described in terms of its inference rules.

## 3.2 CSP as a Programming Language.

Hoare's seminal paper on CSP [ Hoare 78 ] advocated a new programming style, designed to cater for the needs of distributed programming. The paper suggests that the parallel composition of sequential processes which communicate with one another is a fundamental basis for a programming technique. The approach is characterized by its treatment of parallelism, communication and alternative commands.

### 3.2.1 Parallel Composition.

The approach of CSP to parallelism is to define a number of sequential processes and then combine them in parallel permitting them to communicate. In Hoare's notation, each sequential process is assigned a label or process name and a command list. This takes the form

*label* :: *commandlist* Eq. 3.1

A construct is required to specify the concurrent execution of the constituent sequential processes. This is achieved with the parallel command, denoted syntactically by the || operator. The parallel command follows Dijkstra's parbegin construct [ Dijkstra 76 ]. Upon execution, the constituent processes in a parallel command all start simultaneously. The command will only terminate after all its sequential processes have terminated.

For example, consider the following process [*west* :: DISASSEMBLE ‖ *X* :: SQUASH ‖ *east* :: ASSEMBLE] Eq. 3.2

This consists of three processes, named *west*, *X* and *east* respectively, all running concurrently. The capitalized words represent the command lists which the respective sequential processes execute.

### 3.2.2 Communication.

Communication between concurrent sequential processes is specified by the input and output commands. These effect communication by transferring a variable value between processes. Both input and output commands have to explicitly name their source and destination processes respectively. They have the following syntax

*source.label* ? *variable* - Input Command Eq. 3.3

*destination.label* ! *variable* - Output Command Eq. 3.4

An inter-process communication may occur when the following three criteria are met

i) An input command in one process specifies as its source the name of the other process.

ii) An output command in the other process specifies as its destination the name of the first process.

iii) The target variable of the input command matches the type of the output command variable.

These three criteria ensure that communication is synchronous because one process must wait until the other is ready before data is transferred. There is explicitly no provision made for automatic buffering.

Input and output commands fail if their sources or destinations are terminated. A possible consequence of this is one process waiting forever for a synchronized communication which may never occur. This would lead to deadlock.

### 3.2.3 Alternative and Guarded Commands.

A guarded command has the notation

G → CL Eq. 3.5

where $G$ is a guard and $CL$ a command list. A guarded command is executed if and only if the execution of the guard does not fail. A guard may take the form of either a Boolean expression or an input command, or a combination of the two. If the Boolean expression is false the guard fails, if true it succeeds. In the case of an input command the guard succeeds upon successful execution of the input communication.

Guarded commands are used to provide a mechanism for making decisions in sequential processes. CSP provides a similar such mechanism with the alternative construct, denoted by the term $\Box$. For guarded commands $G_i \rightarrow CL_i$ , where $i = 1$ to $n$, this has the notation:

$$G_2 \rightarrow CL_2 \;\Box\; G_2 \rightarrow CL_2 \;\Box \ldots \Box\; G_n \rightarrow CL_n \qquad \text{Eq. 3.6}$$

This construct specifies the execution of exactly one of the constituent guarded commands. The alternative command fails if all the guards fail. If more than one guard succeeds simultaneously the choice between them is arbitrary. By allowing input commands to act as guards in the alternative construct, one sequential process is able to affect the flow of another sequential process in parallel.

### 3.2.4 Observations.

In their appraisal of CSP, Kieburtz and Silberschatz [ Kieburtz 79 ] raise two main objections to the proposed notation. These refer to the synchronous communication primitives and the absence of output guards in guarded commands.

It is recognized that, with asynchronous communications primitives, a source process is not required to wait on a destination process after it sends a communication. This implies that the parallel execution of an asynchronous system will be greater than that of a synchronous system such as CSP. Asynchronous communications are restricted only by their causality. Thus Kieburtz and Silberschatz infer that a synchronous communications primitive will degrade a systems performance relative to its potential. They put forward an alternative in the form of input/output ports with memory. This essentially constitutes an asynchronous communications primitive with a buffer able to hold a single message.

There are three responses to the use of an asynchronous communications primitive. First, in his original paper, Hoare asserts that buffered communication is not primitive. It can be represented in terms of conventional CSP. Secondly, the proposed buffers of Kieburtz and Silberschatz have a finite capacity. As a system increases in size and complexity the number of communications is liable to increase rapidly. This would eventually lead to overflowing buffers and thus present separate problems. Finally, the primitive synchronizes the behaviour of concurrent processes about their communications. This limits the non-deterministic behaviour a concurrent system may exhibit because of different process execution speeds, thus facilitating design and analysis. It also gives a process an indication of the current state of its partner processes.

The second criticism is levelled at the lack of a syntactically defined output guard. This involves allowing an output command to act as a guard in a guarded command. For example, consider the statement

```
PROCESSA ! message → PROCESSBLIST                    Eq. 3.7
```

This constitutes a process which will wait until it has successfully sent a message to PROCESSA, and then continue with the command list PROCESSBLIST.

Hoare and others [ Bernstein 80, Silberschatz 81 ] agree that the addition of an output guard would increase the symmetry of CSP. Most commentators also appreciate, however, that the inclusion of output guards is not as straightforward as it initially appears. The problem lies in an effective implementation rather than the theory [ Silberschatz 79 ]. Bernstein [ Bernstein 80 ] proposes just such an implementation. He postulates an algorithm for output guards based on prioritizing the guarded command in the alternative construct. Silberschatz [ Silberschatz 81 ] proposes a broader solution. He disputes Hoare's insistence that input/output commands must name their source and destination explicitly. Instead he suggests that communication and synchronization should be handled through ports. Each process is assigned a set of ports which it is then said to own. Two processes communicate when one process names a port owned by the other and sends a message by that port. These concepts lay the groundwork for a more efficient implementation of the communication and synchronization constructs. More relevantly they permit the inclusion of both input and output commands in guards.

## 3.3 Implementations of the CSP programming language.

The proposals for concurrent programming languages which CSP espouse provide a basis for an effective notation. However, it has been noted [ Hull 86 ] that an emphasis on notation should be followed by efforts to achieve a workable implementation of the language.

This section considers three such implementations. The first two, COSPOL and CSP/80 are presented here because they are examples of implementations specifically derived from the principles of CSP. Both were intended solely as academic projects with a limited scope of application. The third example, occam, is a commercially available language which is strongly influenced by the principles of CSP. It is well established and has a wide range of applications

### 3.3.1 COSPOL.

COSPOL - "COmmunicating Sequential PrOcess Language and implementation" - was developed by T.J. Roper and C.J. Barter from the University of Adelaide in 1979 - 1981 [ Roper 81 ]. The language is strongly motivated by the proposals for CSP and is written in standard Pascal. It has a parallel command and a message based communications protocol. Nondeterminism is controlled using Dijkstra's guarded commands. As in CSP there is no provision for output guards.

One important respect in which COSPOL differs from CSP is that it does not employ synchronous communications. Instead it uses an automatic buffering method that allows a process which has sent a message to proceed without awaiting a response.

Another difference between CSP and COSPOL is the latter's early use of ports. Ports were suggested by Silberschatz [ Silberschatz 81] as an alternative approach to the explicit process naming convention of CSP communications. Each process declares local port names and is designated as the owner of those ports. If two processes wish to communicate then they must be connected by a common port which is owned by one of the processes. The advantage of ports is the added identification which they supply to communications. For example if two processes exchange multiple messages it may be useful to know which messages come from which parts of the processes. This can be achieved by directing those messages through appropriate ports.

COSPOL uses a combination of process naming and ports. Messages are accepted for input and placed in a suitable 'slot' depending on their construction. These slots correspond to ports. However, process naming is supported by the fact that a process outputs information on the construction of a message to the input process label.

### 3.3.2 CSP/80.

CSP/80 was developed by M. Jazayeri and colleagues at the University of North Carolina in the late 1970's [ Jazayeri 80 ]. They wished to develop a language which would exhibit the properties of CSP.

Like CSP, CSP/80 has a synchronous communications primitive and a nondeterministic alternative construct for selection between processes. Unlike CSP, however, CSP/80 effects communication and synchronization by using ports and channels. A channel connects two ports, one in each process, in one direction only. Each sequential process declares its own ports and types them. Channels are also

PAGE
NUMBERING
AS ORIGINAL

A new version of occam, sometimes called occam3, is expected to be released by Inmos soon. The literature that exists [ Barret 90, Edwards 91 ] indicates that occam3 will have stronger data typing and floating point arithmetic to keep up with hardware advances such as the IMS T9000 transputer. There is no indication that occam3 will include recursive calls or output guards.

## 3.4 Proof Systems For Communicating Sequential Processes.

In addition to suggesting a style for concurrent programming, Hoare's original paper acknowledged the need for formal proof techniques to assist in the design and verification of correct parallel programs. The groundwork for an axiomatic approach to the partial correctness of parallel programs had already been laid [ Owicki 76 ]. The next natural step lay in applying these techniques to Communicating Sequential Processes.

### 3.4.1 Precondition/Post-condition Proof Systems.

Two approaches to the verification of CSP processes were formulated at approximately the same time. Because they were both influenced by the same work their methods have much in common. The first is by Apt, Francez and de Roever [ Apt 80 ]. Their system was able to prove partial correctness and deadlock freedom. System properties are represented as preconditions and post-conditions on the system state. A CSP process is then expressed as n sequential processes $P_1, .., P_n$ operating in parallel with the notation

$$[\ P_1 \parallel P_2 \parallel \ldots \parallel P_n\ ] \qquad \text{Eq. 3.8}$$

Proofs are then presented for the sequential processes in separation. In these separate proofs each precondition and post-condition refers only to the variables of the process in which the statement occurs. Axioms and proof rules relating to the sequential constructs of CSP are given and allow the proof of properties for these sequential components. However, these rules are only meaningful in the context of parallel composition, they do not capture the complete meaning of the constructs. When viewed in isolation it is not yet possible to ascertain whether or not these sequential processes will cooperate.

Sequential proofs cooperate if, when placed together, they fulfil the requirements of the Input and Output commands expected by the sequential proofs. Proofs cooperate provided that

i) The inter-process communications match one another in sequence.

ii) The variables of the preconditions and the post-conditions do not overlap between processes.

Once cooperation has been established the inference rule for the parallel composition may be employed. Here the preconditions and the post-conditions of the parallel process are simply the logical conjunction of the preconditions and post-conditions of the constituent sequential process. A formal justification for the proof method is provided in a later paper which also includes soundness and completeness of the axioms [ Apt 83 ].

Levin and Gries paper [ Levin 81 ] proposes a similar proof technique also based on precondition/post-condition specifications. Again proofs are factorized for the sequential processes and then combined in parallel. Just as for Apt *et al* there is an onus on the proof system to show that parallel processes cooperate with their communications, a property Levin and Gries term satisfaction. The paper makes use of inference rules attached to each syntactic operator and improves the technique by allowing the proof of total correctness in the sense that a process can be shown to terminate in the absence of deadlock. Additionally, output guards are considered as a permissible primitive, reflecting the trend.

In this proof system, however, variables used in the post-conditions and preconditions are allowed to overlap between the concurrent sequential processes. These variables are called shared auxiliary variables. This places an extra proof obligation on the parallel composition of processes, that of proof of non-interference. The proof of non-interference is necessary because of the possibility of the execution of one process to affect the assertions, and thus the sequential proof, of another process. The property of non-interference involves showing that each process is invariant over any parallel execution.

Although the proof system proposed by Apt *et al* does not explicitly require non-interference proofs, this is because it is essentially a restricted form of the Levin and Gries techniques. The obligation to prove such properties between parallel processes before performing correctness proofs increases the effort required by the verifier. This is a major drawback of these state based proof systems. There have been attempts to reduce the complexity of the proofs, such as Prasad [ Prasad 84 ], but they were restricted by the need to prove cooperation.

### 3.4.2 Proof Methods Based on Sequences of Communications.

The need for non-interference is a result of the structure of concurrent languages. Specifically the parallel execution of commands can no longer be expressed as a function from the initial state to the final state at it was in a sequential process. During the execution time of a command a parallel process may alter its state, and the difference between initial and final states ceases to be the result of a single command.

Some methods circumvent this problem by orienteering their proof systems around the sequences of messages exchanged by the processes. This method is suggested by Misra and Chandy [ Misra 81 ] and is indirectly achieved by Soundararajan [ Soundararajan 84 ] in a proof system with similarities to that of Levin and Gries. He reasons about hidden variables which correspond to the sequences of messages which pass between processes. Consequently, he proposes an axiomatic method which simplifies the proofs by removing the obligations of cooperation and non-interference. Murtagh [ Murtagh 87 ] concedes this point but maintains that concentrating on the sequence of communications alone limits the approaches to constructing proofs.

Hoare and others [ Hoare 81, Hoare 85, Brookes 84] have adopted just such an approach to a proof method for CSP. System specifications are expressed as assertions on a trace based semantic model. A set of axiomatic inference rules associated with the syntactic operators are derived, and there is no requirement for either cooperation or non-interference to be established.

## 3.5 An Introduction to the Notation of CSP.

In order to structure a proof system for CSP based on assertions made over the communications passed within a process, it is necessary to possess an appropriate semantic model for CSP processes.

An early approach to a semantics based on the transitional behaviour of a process is adopted in [ Hoare 81 ]. Here the state of a process is considered to be defined by the current sequence of interprocess communications. In later works [ Brookes 84, Hoare 85 ] the scope of the semantics is extended to all transitions or events which the process exhibits externally.

It is noted from these works that CSP has undergone a change in tone from its original description. By developing a formal trace based semantics and axiomatizing all of the constructs, CSP is placed on a more rigid theoretical basis. This affords a more precise and comprehensive understanding of the nature of existing constructs, and in addition it has led to the introduction of new notation. For example, in [ Hoare 85 ] CSP has an interleaving operator, a recursive operator, two different choice operators, a hiding operator and allows output guards; all of which were not present in the original definition. From its proposal as a programming style CSP has shifted emphasis from implementation towards mathematical description.

This next section describes this extended CSP formally in terms of the traces which processes perform. Each syntactic operator is described first informally and then in terms of a rule which precisely defines what effect that operator has on the traces of its operand processes. Section 3.6 then illustrates how the description of CSP processes in terms of their traces can be formalized into a complete semantic model.

( This change in emphasis is reflected by a comparison with the programming language occam which was described in Section 3.3.3. It is sometimes informally stated that occam is an implementation of CSP. This is true to the extent that occam embodies virtually all of the proposals for a programming language made in [ Hoare 78 ] and has a broadly similar notation. However, it is also noted that occam has as yet no provision for recursion or output guards. The reason given for the absence of recursion was the secure operation of the transputer under compiler memory constraints. Section 3.2.4 made the point that output guards are also difficult to implement. Yet the formal language described in this and subsequent sections possesses both recursion and output guards. In addition occam does not, as yet, allow multiple process synchronisation on communications. Thus it would not be true to say that occam constituted an implementation of the formal language CSP.)

### 3.5.1 Processes, Events and Traces.

Following Hoare, the unit of CSP is the process. A process is an entity which is able to perform actions or tasks, and which is able to interact with its environment and other processes. A process may be made up of smaller constituent subprocesses, or it may form part of a larger process.

In the description of a process it is convenient to isolate and label the actions in which it may engage. Such actions are called events. An event is a discrete action or package of actions which a process may perform. Events are the external representation of the progress of the process, they denote the internal behaviour. They are considered to be instantaneous.

A trace is an ordered sequence of events. The trace of a process is a sequence of events which a process has performed in the order in which they occurred. The set of traces of a process P, say, is the set of all traces in which P may engage. This takes the notation

**Definition 3.1:** The set of all possible traces exhibited by a process P is represented by the notation

$$\textit{Traces}(\texttt{P}) \qquad \text{Eq. 3.9}$$

■

The following conventions of notation apply in this text: A trace variable will be expressed in lower case text, such as s or t. A process variable will be expressed in uppercase text, such as P or X. An event will be expressed in lower case bold text, such as **up** or **down**. A trace is explicitly expressed as a sequence of events by separating those events by commas and placing the expression in chevron brackets, such as

$$\texttt{s} = < \textbf{up}, \textbf{down} > \qquad \text{Eq. 3.10}$$

A set of events, as distinct from an ordered sequence, is expressed within curved brackets, such as

$$\{ \textbf{up}, \textbf{down}, \textbf{left}, \textbf{right} \} \qquad \text{Eq. 3.11}$$

( Some of these notations may also be used to represent other quantities, such as sets or real variables. Wherever this is the case the distinctions will be made clear ).

### 3.5.2 Alphabets and Empty Traces.

The alphabet of a process is the set of those events which that process has the potential to perform. The alphabet is a pre-defined property of a process and may not be changed dynamically.

The special set $\Sigma$ is defined as the set of all possible events. That is the union of the alphabets of every process. For a set of events A, say, the star of A represents the set of all traces which contain only events from A. This is denoted

$$\texttt{A}^* \qquad \text{Eq. 3.12}$$

Thus the set $\Sigma^*$ represents the set of all traces which some process can perform.

The empty trace is a trace with no events in it, representing the action of a process which has done nothing externally. It is expressed as

$$< > \qquad \text{Eq. 3.13}$$

Since there is a point in the execution of every process at which it has not done anything, it is logical to conclude that the empty trace belongs to the traces of every process.

### 3.5.3 Trace Operations.

Certain operations are defined over traces. The operations defined here are relevant to later work.

### 3.5.3.1 Catenation.

Catenation is the joining of two or more traces to form a single trace. This is a simple yet important concept both in CSP and in the work which follows. It is formally defined

**Definition 3.2:** Let s, t be traces. Then the catenation of s and t is represented

$$s\hat{}t \qquad \text{Eq. 3.14}$$

■

Catenation does not affect the ordering of the events in the operand traces. In Definition 3.2 trace t simply follows on from trace s. For example

$$\langle a, b, c, d \rangle \hat{} \langle w, x, y \rangle = \langle a, b, c, d, w, x, y \rangle \qquad \text{Eq. 3.15}$$

For traces s, t, u catenation obeys the following laws

i) $s\hat{}\langle\rangle = \langle\rangle\hat{}s = s$ Eq. 3.16

ii) $s\hat{}t = \langle\rangle \quad \Leftrightarrow \quad s = \langle\rangle \wedge t = \langle\rangle$ Eq. 3.17

iii) $s\hat{}(t\hat{}u) = (s\hat{}t)\hat{}u$ Eq. 3.18

iv) $s\hat{}t = u\hat{}t \quad \Leftrightarrow \quad s = u$ Eq. 3.19

v) $s\hat{}t = s\hat{}u \quad \Leftrightarrow \quad t = u$ Eq. 3.20

### 3.5.3.2 Event Restriction.

Sometimes it is useful to be able to filter out certain events from a trace and confine it to a particular set of events. This can be achieved by the restriction operator.

**Definition 3.3:** Let s be a trace and A be a set of events. The restriction of s to A is denoted

$$s \upharpoonright A \qquad \text{Eq. 3.21}$$

■

As an example, consider the trace < **a, b, d, c, b, d** > restricted to the set { **a, b** }

$$< \mathbf{a, b, d, c, b, d} > \upharpoonright \{ \mathbf{a, b} \} = < \mathbf{a, b, b} >$$

Restriction is distributive over catenation, and restriction to two sets results in restriction to the union of those sets. For traces s, t and sets of events A, B

i) $(s\hat{\ }t)\upharpoonright A \quad = \quad (s\upharpoonright A)\hat{\ }(t\upharpoonright A)$ Eq. 3.22

ii) $(s\upharpoonright A)\upharpoonright B \quad = \quad s\upharpoonright(A \cup B)$ Eq. 3.22

### 3.5.3.3 First Event of a Trace.

The first event of a trace s is denoted by the expression

$$\mathcal{F}irst(s) \qquad \text{Eq. 3.24}$$

As an example consider the trace < **right, left** >. The first event in this is given by

$$\mathcal{F}irst(< \mathbf{right, left} >) \quad = \quad \mathbf{right} \qquad \text{Eq. 3.25}$$

If s is the empty trace then the convention adopted in this text will be to represent the first event of s as a null symbol. That is

$$\mathcal{F}irst(<>) \quad = \quad \emptyset \qquad \text{Eq. 3.26}$$

### 3.5.3.4 Last Event of a Trace.

The last event of a trace s is denoted by the expression

$$\mathcal{L}ast(\mathtt{s}) \qquad \text{Eq. 3.27}$$

As an example consider the trace < **right**, **left** >. The last event in this is given by

$$\mathcal{L}ast(<\mathtt{right}, \mathtt{left}>) \quad = \quad \mathtt{left} \qquad \text{Eq. 3.28}$$

If s is the empty trace then the convention adopted in this text will be to represent the last event of s as a null symbol. That is

$$\mathcal{L}ast(<>) \quad = \quad \emptyset \qquad \text{Eq. 3.29}$$

### 3.5.3.5 Event Inclusion.

Event inclusion indicates that a particular event is contained within a trace. For a trace s and an arbitrary event **a** the statement

$$\mathtt{a}\ \underline{\mathbf{in}}\ \mathtt{s} \qquad \text{Eq. 3.30}$$

is TRUE if **a** is one of the events in the trace s and FALSE otherwise. For example

$$\mathtt{road}\ \underline{\mathbf{in}} < \mathtt{road}, \mathtt{tree}, \mathtt{car} > \ \equiv \quad \text{TRUE} \qquad \text{Eq. 3.31}$$

## 3.6 Syntactic Operators.

The syntactic operators are used to define processes in a compact and precise manner. They follow a set of syntactic rules which are summarized at the end of the chapter. Before describing the syntax it is useful to outline the fundamental CSP processes STOP.

### 3.6.1 STOP.

STOP is the deadlocking process. Deadlock occurs when a process is unable to make any further external progress and thus is unable to perform any more events. STOP is the process which does nothing. This is reflected by its set of traces.

In the process P ; X a trace t of X will only occur after a trace s which effects the successful completion of P. That is s must end with the termination event, so there is some trace s' such that $s' = s\hat{}\langle\checkmark\rangle$. Thus, since t follows s a trace of P ; X is given by $s'\hat{}t$. (Note that the termination event has been removed from $s'\hat{}t$ because the successful completion of P results in the commencement of X).

This reasoning gives rise to the following rule which relates the effect of the sequential operator on the traces of its operand processes.

$$\text{i)}\quad \mathit{Traces}(\,P\,;\,X\,) = \{\, s \mid s \in \mathit{Traces}(\,P\,) \wedge \neg \langle\checkmark\rangle\ \underline{\textbf{in}}\ s\,\}$$
$$\cup \{\, s\hat{}t \mid s\hat{}\langle\checkmark\rangle \in \mathit{Traces}(\,P\,) \wedge t \in \mathit{Traces}(\,X\,)\,\} \qquad \text{Eq. 3.38}$$

The termination event leads on to the definition of another fundamental CSP Process. SKIP is the process which starts, does nothing and then terminates successfully. The traces of SKIP are

$$\mathit{Traces}(\,\mathrm{SKIP}\,) = \{\ \langle\rangle, \langle\checkmark\rangle\ \} \qquad \text{Eq. 3.39}$$

## 3.6.4 Alphabetized Parallel Operator.

Consider two processes, P and X say, which are combined in parallel so they may interact. Interactions between them may be regarded as events which require the simultaneous participation of both processes. Such events are called synchronized events. Like a synchronous communication an event **a**, synchronized between P and X, will not occur until both processes are ready to engage in it.

Synchronous events are those which are of concern to all the processes which engage in them. They are those events which are common to the alphabets of those processes. Let A and B be the alphabets of processes P and X respectively. Then the alphabetized parallel combination of P and X is denoted

$$P\ {}_{A}\|_{B}\ X \qquad \text{Eq. 3.40}$$

Here both P and X proceed, synchronizing on the events common to both their alphabets, i.e. those in the set $A \cap B$.

Let s be a trace of ( $P\,{}_{A}\|_{B}X$ ). Every event in s which belongs to A has been performed by P and thus ( $s \upharpoonright A$ ) is a trace of P. Similarly ( $s \upharpoonright B$ ) is a trace of X. Furthermore every event in s must be in either A or B. This reasoning implies the following rules

i) $$Traces(\,\mathrm{P}_A\|_B\mathrm{X}\,) = \{\,\mathrm{s} \mid (\,\mathrm{s}\restriction \mathrm{A}\,) \in Traces(\,\mathrm{P}\,) \wedge (\,\mathrm{s}\restriction \mathrm{B}\,) \in Traces(\,\mathrm{X}\,) \wedge \mathrm{s} \in (\,\mathrm{A} \cup \mathrm{B}\,)^*\} \qquad \text{Eq. 3.41}$$

ii) $$\text{Alphabet of } \mathrm{P}_A\|_B\mathrm{X} = \text{Alphabet of } \mathrm{P} \cup \text{Alphabet of } \mathrm{X} \qquad \text{Eq. 3.42}$$

Parallelism in CSP is sometimes denoted by the || symbol without the alphabet subscripts [ Hoare 85, Brookes 84 ]. In this text this symbol will be taken to mean that the processes on either side each have the same alphabet and so must synchronize on every event each performs. That is it is a special case of the alphabetized parallel operator

$$\mathrm{P} \,\|\, \mathrm{X} \quad \equiv \quad \mathrm{P}_A\|_A\mathrm{X}\,, \qquad \mathrm{A} = \text{Alphabet of } \mathrm{P} \cup \text{Alphabet of } \mathrm{X} \qquad \text{Eq. 3.43}$$

3.6.5 Deterministic choice.

Deterministic choice defines a mechanism by which a process can make a choice between two courses of action dependent on the environment of the process. The environment of a process is the circumstances and surroundings in which it is placed. For example the environment of a networked computer may consist of the current user, the status of the network server or the number of people using the network. The environment may relate only to the other processes composed in parallel.

The deterministic choice between two processes P and X is denoted

$$\mathrm{P} \;\square\; \mathrm{X} \qquad \text{Eq. 3.44}$$

Suppose the environment in which ( P □ X ) is placed allows the first event of P to occur but not the first event of X. Then process P will proceed instead of process X. If the opposite is true, that the first event of X is possible but not that of P then process X will proceed and not process P. By using synchronous events as guards for processes the choice between them can be controlled. For example, for the set A = { **a**, **b** } consider the process

$$(\,\mathbf{a} \rightarrow \mathrm{SKIP}\,)\,{}_A\|_A\,(\,\mathbf{a} \rightarrow \mathrm{P} \;\square\; \mathbf{b} \rightarrow \mathrm{X}\,) \qquad \text{Eq. 3.45}$$

Both **a** and **b** are synchronized events. The right hand process is required to choose between ( **a** → P ) and ( **b** → X ). Its environment, in this case the left hand process, will only permit the first event of ( **a** → P ) to proceed. Thus

$$( \mathbf{a} \rightarrow \mathtt{SKIP} )\, {}_{A}\|_{A}\, ( \mathbf{a} \rightarrow \mathtt{P} \,\square\, \mathbf{b} \rightarrow \mathtt{X} ) \equiv ( \mathbf{a} \rightarrow \mathtt{P} ) \qquad \text{Eq. 3.46}$$

If s is a trace of P, then s is also a possible trace of ( P □ X ) since P may be selected. Similarly if s is a trace of X, it is a trace of ( P □ X ).' This gives rise to the rule

i) $\mathit{Traces}( \mathtt{P} \,\square\, \mathtt{X} ) = \mathit{Traces}( \mathtt{P} ) \cup \mathit{Traces}( \mathtt{X} )$ Eq. 3.47

### 3.6.6 Nondeterministic Choice

Sometimes a choice between processes is unaffected by the current environment. There is no way of telling from the external actions of any of the processes what the outcome will be and thus the choice becomes arbitrary. This is termed nondeterministic choice. For two processes P and X nondeterministic choice is represented by

$$\mathtt{P} \sqcap \mathtt{X} \qquad \text{Eq. 3.48}$$

Either process P or X will proceed in ( P⊓X ), the choice is not controlled.

The nondeterministic operator is infrequently used in the description of processes; nondeterminism arises more naturally from the use of the deterministic choice operator. This can be seen by considering a situation which occurs if, in the construct ( P □ X ), both the first events of processes P and X are allowed by the environment. In this case the choice between P and X becomes nondeterministic. To illustrate this consider the equality

$$( \mathbf{a} \rightarrow \mathtt{P} \,\square\, \mathbf{a} \rightarrow \mathtt{X} ) = ( \mathbf{a} \rightarrow \mathtt{P} \sqcap \mathbf{a} \rightarrow \mathtt{X} ) \qquad \text{Eq. 3.49}$$

Nondeterministic choice has the following rule

i) $\mathit{Traces}( \mathtt{P} \sqcap \mathtt{X} ) = \mathit{Traces}( \mathtt{P} ) \cup \mathit{Traces}( \mathtt{X} )$ Eq. 3.50

### 3.6.7 Interleaving.

Sometimes it is convenient to join two processes together concurrently without them directly interacting or synchronizing with one another. In this case, at any point an action of the composite process is an action of only one of the subprocesses. This form of composition is called interleaving and for processes P and X is denoted

$$P \mid\mid\mid X \qquad \text{Eq. 3.51}$$

A trace of ( P ||| X ) is an arbitrary interleaving of a trace from P with a trace from X. To define the meaning of this more precisely consider a sequence ( trace ) r say. A subsequence of r is a sequence derived from r by selecting certain of its terms ( events ) and retaining their order [ Borowski 89 ]. Define the complement of a subsequence as being the sequence of terms in the original sequence which were not selected for subseq(r), with their original order. This are denoted

subseq(r) - A Subsequence of r Eq. 3.52

comsubseq(r) - The Complement of subseq(r) Eq. 3.53

These lead to the derivation of the interleaved set of two traces, s and t say. This is the set of all traces which may be interleaved from s and t and is defined

$$\text{Interleave}(s, t) = \{u \mid s = \text{subseq}(u) \wedge t = \text{comsubseq}(u)\} \qquad \text{Eq. 3.54}$$

In turn these definitions allow the following rule for interleaving to be stated

$$\text{i)} \quad \mathit{Traces}(P \mid\mid\mid X) = \{ s \mid s \in \text{Interleave}(t, u) \wedge t \in \mathit{Traces}(P) \wedge u \in \mathit{Traces}(X) \} \qquad \text{Eq. 3.55}$$

### 3.6.8 Change of Symbol and Hiding.

Change of symbol denotes an injective function which maps the process alphabet to some other set. Its purpose is to relabel certain events. Because it is injective it has an inverse function which restores events to their original labelling. For a process P and a change of symbol function $f$ the operator has the notation

$$f(P), \qquad (f^{-1}(P) \text{ is the inverse function}) \qquad \text{Eq. 3.56}$$

Hiding is an operator which conceals events from anything other than the process which performs them. If P is a process and A is a set of events to be concealed then the hiding operator is denoted

$$P \setminus A \qquad \text{Eq. 3.57}$$

Equation 3.57 represents a process which behaves like P except that the occurrence of any event in A is hidden.

### 3.6.9 Syntactic Rules in Backus-Naur Form.

The above operators follow a precise set of syntactic laws so that they may represent a formal language in the true sense [ Wing 90 ]. These laws are expressed below in standard Backus-Naur form [ Woodcock 88 ]. The function F represents a CSP function and the function $f$ represents a change of symbol function.

**Definition 3.4:** The syntax of CSP as used in this thesis is given by the following BNF form. (This broadly follows the definition in [ Brookes 84 ].)

*event* ::= "**a**", "**b**", ...* list of names *... ;

*eventset* ::= "{", *event*, *event*, ... "}";

*process* ::= STOP | SKIP |
*event*, "→", *process* |
$process_{eventset}$, "||", $_{eventset}process$ |
*process*, "□", *process* |
*process*, "⊓", *process* |
*process*, ";", *process* |
*process*, "||", *process* |
*process*, "|||", *process* |
*process*, "\", *eventset* |
"$f$(", *process*, ")" | "$f^{-1}$(", *process*, ")" |
"μP.F( P )" ;

The notation μP.F( P ) is discussed in Section 3.8 ■

## 3.7 The Traces Semantic Model.

The theoretical treatment given to the operators of the previous section forms the basis for a semantic description of every syntactic process in CSP. This semantics is defined as follows

**Definition 3.5:** A process is defined as a pair ( A, T ), where

A is a set of symbols - the Alphabet of the Process

T is a subset of $A^*$ - the Traces of the Process

Furthermore, T is subject to the rules

i) $\langle\rangle \in T$ Eq. 3.58

ii) $\forall s, t \bullet (s\hat{}t \in T \Rightarrow s \in T)$ ( Prefix closure ) Eq. 3.59

■

The first rule ensures that the empty trace is included. All traces engage in the empty trace before they have done anything, and thus all processes contain it. Prefix closure means that if a process can perform a trace then it can perform all prefixes of that trace. That is if s^t is a trace of process P then P can also perform the trace s.

The semantic domain in which processes are so defined is called the Traces Model, denoted by $M_T$. $M_T$ is the set of all possible pairs ( A, T ). The traces model is the simplest semantic representation in CSP, and is used as the basis for the others which are introduced later in the text.

$M_T$ is a denotational semantics. This is seen from the fact that in each of the Subsections 3.6.1 to 3.6.7 dealing with syntactic operators a mapping from the semantics of the operands to those of the composite process was supplied.

## 3.8 Recursion and Fixed Point Treatment.

There is one more CSP operator to be discussed. This is the recursion operator and it is used extensively in later chapters. It has a complex theory underlying it and so merits discussion.

A process is recursive if it can call itself as a subprocess. For example, consider the simple process expressed by

$P = \mathbf{a} \rightarrow P$ Eq. 3.60

### 3.8.2 Metric Space Representation.

The concepts of recursion and fixed points are generic to other fields. To represent them in a mathematical sense it is convenient to use the branch of Topology known as Metric Spaces , an introduction to which can be found in [ Bryant 85 ].

#### 3.8.2.1 A Metric Space.

A metric space is defined as a pair ( U, d ), where U is a set and d is a function which relates the 'distance' between two points in U. The function d is called a metric. The formal definition of a metric space is as follows

**Definition 3.6:** Let U be a set and let d be a function such that $d : U \times U \rightarrow \mathbb{R}$, ( $\mathbb{R}$ is the set of real numbers ). Let x, y, z be members of U. Then the pair ( U, d ) is said to be a metric space if and only if the following hold

i) $\forall x, y \in U \bullet \quad d(x, y) \geq 0$ Eq. 3.64

ii) $\forall x, y \in U \bullet \quad d(x, y) = 0 \Leftrightarrow x = y$ Eq. 3.65

iii) $\forall x, y \in U \bullet \quad d(x, y) = d(y, x)$ Eq. 3.66

iv) $\forall x, y, z \in U \bullet \quad d(x, y) + d(y, z) \geq d(x, z)$ Eq. 3.67

■

The term 'distance' is often used for the metric because it is sometimes convenient to envisage a metric space as two dimensional real space, with the metric defined as the spatial distance between points in that space.

#### 3.8.2.2 Contraction Mappings and Nonexpansions.

A function which reduces the metric between any two points in a metric space is called a contraction mapping. More formally

**Definition 3.7**: For a metric space ( U, d ) a contraction is a mapping $f: U \rightarrow U$ with the property that, for some real number $k < 1$,

$\forall x, y \in U \bullet \ d( f(x), f(y) ) \leq k.d( x, y )$ Eq. 3.68

■

sequence of points generated by repeatedly applying $f$ to $x$ converges to a point $x'$ then the sequence of $y$ points will converge to $f(x')$. Thus $x' = f(x')$. This means that the limit of the sequence produced by repeatedly applying $f$ to any member of $U$ will be the fixed point of $f$.

Additionally, the contraction mapping $f$ can only have one fixed point solution in the set $U$. Suppose there were two distinct fixed points in $U$, say $x'$ and $y'$, and suppose $f$ were applied to each of them. Then because $f$ is a contraction they would have to move nearer. To do this at least one point would have to alter position under $f$ and thus would not be a fixed point of $f$.

These ideas are presented more formally by the following theorem

**Theorem 3.2**: ( Banach's Fixed Point Theorem ). Let $f : U \rightarrow U$ be a contraction mapping over the ( complete† ) metric space ( $U, d$ ). Then $f$ has a unique fixed point solution. Furthermore, if $x_1$ is any point of $U$ then the sequence

$$x_1, x_2 = f(x_1), x_3 = f(x_2), \ldots \qquad \text{Eq. 3.70}$$

converges to that unique fixed point. ■

*Proof*

See Appendix A: Theorem A.2 □

### 3.8.3 CSP as a Metric Space.

Roscoe [ Roscoe 82 ] has shown that processes of CSP can be structured as a complete metric space. For the traces semantic domain given above this is the metric space ( $M_T$, $d_T$ ).

$M_T$ is the traces model defined in Section 3.7 above. Prior to defining the metric $d_T$ additional notation is required. Let $P$ be a process and $n$ a natural number. The notation

$$P\restriction n \qquad \text{Eq. 3.71}$$

represents the process which behaves like $P$ for its first $n$ events, and then stops. The metric $d_T$ over the space $M_T$ is defined for two processes $P$ and $X$ in $M_T$ as

$$d_T( P, X ) = \frac{1}{n}, \qquad n = \max\{ m \mid P\restriction m = X\restriction m \} \qquad \text{Eq. 3.72}$$

† For a definition of completeness see Appendix A.

Roscoe has shown that the above metric complies with the necessary conditions 3.8.2.1 (i) - (iv) and that the metric space is complete.

### 3.8.4 Syntactic Operators as Contractions and Nonexpansions.

The CSP operators outlined in Sections 3.6.1 to 3.6.7 had certain related functions which demonstrated their effect on the traces of a process. These relations over the set of traces constitute mappings from the space $M_T$ onto itself [ Roscoe 82 ].

A preliminary result for this interpretation of the syntactic operators being associated with semantic mappings is that the prefix operator is a contraction on the metric space ( $M_T$, $d_T$ ). To illustrate this consider any two processes P and X from $M_T$. Suppose

$$d_T( P, X ) = \frac{1}{n_1} \qquad n_1 \geq 0 \qquad \text{Eq. 3.73}$$

From the definition of the metric $d_T$, it can be seen that the maximum value of m such that $\{m \mid P\lceil m = X\lceil m\}$ is $n_1$. Prefixing both processes with the same event **a** will have the effect of adding 1 to $n_1$. Therefore

$$d_T( \mathbf{a} \rightarrow P, \mathbf{a} \rightarrow X ) = \frac{1}{n_1+1} \leq \frac{1}{n_1} \qquad \text{Eq. 3.74}$$

The prefix operator is the only contraction in CSP over the metric space ( $M_T$, $d_T$ ). However, the following functions [ Roscoe 82 ] are nonexpansive: Parallel Composition, Deterministic Choice, Nondeterministic Choice, Interleaved Composition, Sequential Composition and Change of Symbol.

Hiding, however, is expansive ( not nonexpansive ). This is illustrated by considering the CSP function F( P ) = P\{ **a** } applied to the processes ( **a** → STOP ) and ( **a** → SKIP ). First the metric between ( **a** → STOP ) and ( **a** → SKIP ) is given by

$$d_T( \mathbf{a} \rightarrow \text{STOP}, \mathbf{a} \rightarrow \text{SKIP} ) = 1 \qquad \text{Eq. 3.75}$$

After applying F the metric becomes

$$d_T( F( \mathbf{a} \rightarrow \text{STOP} ), F( \mathbf{a} \rightarrow \text{SKIP} ) )$$
$$= \quad d_T( \text{STOP}, \text{SKIP} ) \quad = \quad \infty \qquad \text{Eq. 3.76}$$

$$\therefore \quad d_T(\mathbf{a} \rightarrow \mathtt{STOP}, \mathbf{a} \rightarrow \mathtt{SKIP})$$
$$\not\geq \quad d_T(\mathtt{F}(\mathbf{a} \rightarrow \mathtt{STOP}), \mathtt{F}(\mathbf{a} \rightarrow \mathtt{SKIP})) \qquad \text{Eq. 3.77}$$

### 3.8.5 The Uniqueness of Fixed Pointed CSP Equations.

Thus all the syntactic operators for CSP so far described, with the exception of the hiding operator, are nonexpansive and one, the prefix operator, is contractive. From Theorem 3.1 (ii) it may be concluded that any CSP function composed from operators excluding hiding is nonexpansive. Furthermore it is possible to make use of Theorem 3.1 (iii) to establish a result for contractions.

That is, if F is a CSP function which maps the space of all processes onto itself and all the occurrences of the operand of F are, directly or indirectly, guarded by a prefix operator, and furthermore if F does not contain the hiding operator, then F is a contraction over the metric space ( $M_T$, $d_T$ ) [ Brookes 84 ]. This provides a method of determining whether or not F is a contraction by examining the syntax of the expression.

By applying the Fixed Point Theorem 3.2 to this it can be deduced that an equation of the general recursive form

$$\mathtt{P} = \mathtt{F}(\mathtt{P}) \qquad \text{Eq. 3.78}$$

will have a unique fixed point solution if F complies with the above syntactic rules for contraction. A unique solution means that the process does not diverge.

## 3.9 Specifications.

An observation of a process is a finitely describable experiment to which that process can be subjected [ Olderog 86 ]. Specifications in CSP may be expressed as predicates over the observations of a process. A process P satisfies a specification S if S holds for every observation made of P. This is denoted

$$\mathtt{P}\ \mathbf{sat}\ \mathtt{S} \qquad \text{Eq. 3.79}$$

For the case of the traces model $M_T$, an observation is a trace of the process and a specification is written as a predicate on the set of traces of the process. That is

$$\mathtt{P}\ \mathbf{sat}\ \mathtt{S} \quad \Leftrightarrow \quad \forall\, \mathtt{s} \in \mathit{Traces}(\mathtt{P}) \bullet \mathtt{S}(\mathtt{s}) \qquad \text{Eq. 3.80}$$

Specifications expressed as predicates on a semantic domain are called behavioural specifications.

### 3.9.1 Satisfiable and Continuous Specifications.

Consider a specification R. R is said to be satisfiable if there exists some process P such that R(P) is true. Furthermore R is said to be continuous if its truth or falsehood can be determined by examining the finite restrictions of its argument [ Brookes 84 ]. That is R is continuous if it satisfies the condition

$$\forall P \in \mathbf{CSP}, n \in \mathbb{N} \bullet \exists X \bullet (P\lceil n = X\lceil n) \Rightarrow (R(X) \Rightarrow R(P)) \qquad \text{Eq. 3.81}$$

An example of a non-continuous specification is the requirement for an infinitely recursive process that "The process will eventually perform the event **a**". No bound has been set on when **a** may occur, and thus its non-occurrence can only be determined by considering every single trace the process may perform, which may be an infinite number.

All behavioural specifications in $M_T$ are continuous [ Roscoe 82 ]. That is if R is a behavioural specification on the traces of a process then R is continuous.

## 3.10 The Proof System For $M_T$.

The nature of behavioural specifications gives rise to a set of inference rules [ Hoare 85 ]. These allow the properties of the behaviours of a composite process to be deduced from the behaviours of the syntactic subprocesses. The inference rules have been derived directly from the semantic definitions of syntactic operators provided earlier in the chapter. Thus there is a rule associated with each operator. The set of rules provided for the traces semantic model, $M_T$, allows the proof obligation on any composite CSP process to be transferred to an obligation on its component subprocesses. Thus the proof system is both axiomatic and compositional.

### 3.10.1 General Rules.

Three rules embody the general properties of the proof system. For a processes P and predicates $R_1$, $R_2$ they are

i) P **sat** TRUE Eq. 3.82

ii) $\underline{\mathtt{P}\ \mathbf{sat}\ R_1 \wedge \mathtt{P}\ \mathbf{sat}\ R_2}$

$\Rightarrow \mathtt{P}\ \mathbf{sat}\ (R_1 \wedge R_2)$ Eq. 3.83

iii) $\underline{\mathtt{P}\ \mathbf{sat}\ R_1 \wedge (R_1 \Rightarrow R_2)}$

$\Rightarrow \mathtt{P}\ \mathbf{sat}\ R_2$ Eq. 3.84

### 3.10.2 Syntactic Rules.

The following are the inference rules associated with the syntactic operators in CSP. Here P, X are processes with alphabets A, B respectively. $R_1, R_2$ are predicates over $M_T$ and $s, s_1, s_2$ are trace variables. For a definition of the Tail operator see Section 7.9.2.3.

**Rule for Prefix Operator**

$\underline{\mathtt{P}\ \mathbf{sat}\ R_1\qquad\qquad\qquad\qquad\qquad\qquad}$

$(\mathbf{a} \rightarrow \mathtt{P})\ \mathbf{sat}\ (s = \langle\rangle \vee (\mathit{First}(s) = \mathbf{a} \wedge R_1(\mathit{Tail}(s))))$ Eq. 3.85

**Rule for Alphabetized Parallel Operator**

$\underline{\mathtt{P}\ \mathbf{sat}\ R_1 \wedge \mathtt{X}\ \mathbf{sat}\ R_2\qquad\qquad}$

$(\mathtt{P}\ {}_A\|_B\ \mathtt{X})\ \mathbf{sat}\ (R_1(s \upharpoonright A) \wedge R_2(s \upharpoonright B))$ Eq. 3.86

**Rule for Deterministic Choice**

$\underline{\mathtt{P}\ \mathbf{sat}\ R_1 \wedge \mathtt{X}\ \mathbf{sat}\ R_2\ }$

$(\mathtt{P} \Box \mathtt{X})\ \mathbf{sat}\ R_1 \vee R_2$ Eq. 3.87

**Rule for Nondeterministic Choice**

$\underline{\mathtt{P}\ \mathbf{sat}\ R_1 \wedge \mathtt{X}\ \mathbf{sat}\ R_2\ }$

$(\mathtt{P} \sqcap \mathtt{X})\ \mathbf{sat}\ R_1 \vee R_2$ Eq. 3.88

**Rule for Interleaving Operator**

$$\begin{array}{l} \underline{P \ \mathbf{sat}\ R_1 \ \wedge\ X \ \mathbf{sat}\ R_2 \qquad\qquad\qquad\qquad\qquad\qquad} \\ (P \mathbin{|||} X)\ \mathbf{sat}\ (\exists\, s_1, s_2 \bullet s_1 \in \mathrm{subseq}(s) \ \wedge\ s_2 \in \mathrm{comsubseq}(s) \\ \wedge\ R_1(s_1) \ \wedge\ R_2(s_2)) \end{array} \qquad \text{Eq. 3.89}$$

**Rule for Sequential Operator**

$$\begin{array}{l} \underline{P \ \mathbf{sat}\ R_1 \ \wedge\ X \ \mathbf{sat}\ R_2 \qquad\qquad\qquad\qquad\qquad} \\ (P\,;X)\ \mathbf{sat}\ R_1(s) \ \vee\ (s_1 \in \mathit{Traces}(P) \ \wedge\ \checkmark\ \mathbf{in}\ s_1 \ \wedge \\ s = (s_1 \upharpoonright \{\Sigma - \checkmark\})^\frown s_2 \ \wedge\ R_2(s_2)) \end{array} \qquad \text{Eq. 3.90}$$

3.10.3 The Recursive Inference Rule.

The structuring of $M_T$ as a complete metric space allows the construction of an inference rule relating to the fixed point processes of recursive CSP equations. This rule is given here as a theorem. The theorem can be seen to follow immediately from the definitions of a contaction on $M_T$ and a continuous specification. A treatment and a proof of this theorem can be found in [ Brookes 84 ].

**Theorem 3.3:** Let $F$ be a CSP function which is a contraction on the metric space $(M_T, d_T)$. Furthermore let $R$ be a satisfiable continuous specification over $M_T$ such that $STOP$ **sat** $R$ ($R$ is satisfiable). If $R$ is such that for all processes $P$

$$\forall P \bullet (R(P) \quad \Rightarrow \quad R(F(P))) \qquad \text{Eq. 3.91}$$

then the fixed point of $F$ satisfies $R$. That is

$$\mu P.F(P)\ \mathbf{sat}\ R \qquad \text{Eq. 3.92}$$

## 3.11 Summary

This chapter has outlined some of the principles behind Communicating Sequential Processes. It opened by discussing the early work on CSP as a proposal for a programming style for distributed systems. The concepts of parallelism and communication were discussed and observations about the suitability of the approach were made.

CSP has subsequently influenced the development of parallel programming. In particular three languages which implement its fundamental principles were considered, CSP/80, COSPOL and occam. Most notable of these was occam because of its commercial availability and wide area of application.

The emphasis then turned to formal verification techniques for CSP. It was shown how different approaches to the proof of correctness for CSP processes were proposed. Early approaches were based on a state model oriented method of verification, using preconditions and post-conditions on the state of the process. These approaches were broadly compositional in that they associated an inference rule with each of the syntactic operators and they permitted proofs to be factorized in isolation for each of the sequential subprocesses involved. However, to varying degrees they all necessitated a proof of cooperation between the sequential subprocesses before an overall process proof could be established. In contrast later verification methods concentrated on specifying the system requirements in terms of sequences of interprocess communications. This removed the obligation of establishing cooperation.

The refinement of this latter approach has led to the development of a formal denotational trace based semantics for CSP. It is noted that this semantics, which is set theoretic, allows the effect of each CSP operator to be described in a rigorous mathematical fashion. By concentrating on this mathematical description and laying less emphasis upon considerations of implementation later models of CSP have allowed new operators such as interleaving and recursion to be defined and modelled.

The chapter describes the traces semantic model in terms of the effect each of the semantic operators has on the traces of the operand process(es). Particular attention is paid to the mathematical description of recursion. Specifications are shown to be represented as predicates over the traces model and then finally a formal proof system is given.

CHAPTER FOUR

# HIGHER SEMANTIC MODELS AND APPLYING THEM.

## 4.1 Introduction.

The purpose of this chapter is to investigate how CSP may be used in the specification and verification of a controller for a real-time industrial application.

The traces model of the previous chapter proved to be a concise and useful means of explaining the underlying concepts of CSP. However, the traces model is limited with respect to its ability to express certain system properties which are manifest in the real world. In particular some of these properties, such as time and liveness requirements, are necessary for a suitable treatment of the intended controller application. For this reason the chapter opens by outlining some of the higher semantic models which have been developed to express complex system properties.

The chapter considers two such semantic models, the failures model and the timed failures model. The failures model is able to distinguish between deterministic and nondeterministic choice and is able to represent liveness properties. The timed failures model is, in addition, able to incorporate time into a process description. The fundamental concepts of events and traces are redefined with respect to time to accommodate this.

It is shown how these models along with other semantic interpretations can be structured into a well defined hierarchy which delineates the relationship between domains.

The chapter then moves on to evaluate how these different semantic models can be used in the specification and verification of a controller. The application involves the coordination of high speed machinery, typically in a safety critical environment. It requires a controller which must make the system function correctly to a high level of confidence. To achieve this confidence the system's operational requirements are considered and expressed formally as a behavioural specification in CSP. This specification is then used to develop processes representing the controller's underlying logic. Two processes are proposed, an untimed process and a timed process. The correctness of these processes relative to their formal specification is established by using the axiomatic proof system of CSP.

## 4.2 Failures Model.

The need for higher semantic models for CSP processes is highlighted by two restrictions which limit the ability of the traces model to specify systems.

One restriction is the inability of $M_T$ to distinguish between deterministic and nondeterministic choice. Inspection of the behavioural definitions and inference rules for each choice operator in $M_T$ show that there is no semantic difference between them. This can lead to difficulties, especially if two processes are informally interpreted as distinct but shown to be semantically equivalent in $M_T$. For example, consider the two processes PRO1 and PRO2 below, where **a**, **b** are events and A = { **a**, **b** }

$$\mathtt{PRO1} \;=\; \mathbf{a} \rightarrow \mathtt{SKIP}\ {}_{A}\|_{A}\ \mathbf{a} \rightarrow \mathtt{SKIP} \;\Box\; \mathbf{b} \rightarrow \mathtt{SKIP} \qquad \text{Eq. 4.1}$$

$$\mathtt{PRO2} \;=\; \mathbf{a} \rightarrow \mathtt{SKIP}\ {}_{A}\|_{A}\ \mathbf{a} \rightarrow \mathtt{SKIP} \;\sqcap\; \mathbf{b} \rightarrow \mathtt{SKIP} \qquad \text{Eq. 4.2}$$

PRO1 is equivalent to a process which performs an event **a** and then terminates successfully. PRO2 is equivalent to a process which will either perform event **a** and terminate or will simply deadlock on its first step. That is it is equivalent to $((\mathbf{a} \rightarrow \mathtt{SKIP}) \sqcap \mathtt{STOP})$. PRO1 and PRO2 are thus interpreted as distinct. However examination of their traces shows

$$\mathit{Traces}(\mathtt{PRO1}) = \mathit{Traces}(\mathtt{PRO2}) = \{\ \langle\,\rangle, \langle\,\mathbf{a}\,\rangle, \langle\,\mathbf{a}, \checkmark\,\rangle\ \} \qquad \text{Eq. 4.3}$$

Another limitation in $M_T$ is that, using the **sat** relation, it is only possible to specify safety properties. This can be demonstrated by considering any process P with a specification R expressed over *Traces*(P). If P **sat** R then R must be true for all traces in P. By Definition 3.2(i) the empty trace belongs to every process P . Because the empty trace is equivalent to the traces of the process STOP [ Hoare 85 ] it is seen that for an arbitrary process P

$$\mathit{Traces}(\mathtt{STOP}) \subseteq \quad \mathit{Traces}(\mathtt{P}) \qquad \text{Eq. 4.4}$$

Thus it can be concluded that the specification R must also satisfy the process STOP. Since R satisfies the process which does nothing it can only be a safety specification.

These problems can be overcome by the use of the failures model [ Brookes 84, Hoare 85 ] (Sometimes called the Refusals model [ Olderog 86 ]). This is based on the concepts of Refusals and Failures.

Let B be a set of events which are offered initially by the environment of a process P say. Then if it is possible for P to deadlock on its first step when placed in this environment B is said to be a refusal of P. In the above example { **b** } is a refusal of PRO1 - the process cannot perform event **b** on its first step and thus if this were offered by the environment then the process would deadlock. The set of all refusals of a process P is denoted

$$\mathit{Refusals}(\mathrm{P}) \qquad \text{Eq. 4.5}$$

The concept of refusals leads directly to the idea of failures. A failure is a relation between a trace of a process and the set of events which that process refuses to perform immediately after having engaged in the trace. For a more precise definition, define (P/s) as being that process which behaves exactly as process P would after having engaged in trace s. Then the set of all failures of process P can be expressed

$$\mathit{Failures}(\mathrm{P}) = \{\ (\mathrm{s},\ \mathrm{B}) \mid \mathrm{s} \in \mathit{Traces}(\mathrm{P}) \ \wedge\ \mathrm{B} \in \mathit{Refusals}(\mathrm{P/s})\ \} \qquad \text{Eq. 4.6}$$

Note that the set of all failures of a process is an extension of the set of all traces of a process, because the failures contains the traces. If Domain represents the domain of a relation, then

$$\mathit{Traces}(\mathrm{P}) = \mathrm{Domain}(\mathit{Failures}(\mathrm{P})) \qquad \text{Eq. 4.7}$$

These definitions lead to a mathematical representation for a semantic domain to represent the failures of a process [ Hoare 85 ].

**Definition 4.1**: A process is defined as a pair $(\mathrm{A}, \mathcal{F})$ where

A is any set of symbols - the Alphabet of the Process

$\mathcal{F}$ is a relation between $\mathrm{A}^*$ and $\mathbb{P}(\mathrm{A})$ - the Failures of the Process

Furthermore, if s, t are traces, B, C sets of events and **a** an arbitrary event, then $\mathcal{F}$ is subject to the rules

i) $(\langle\,\rangle, \{\}) \in \mathcal{F}$ Eq. 4.8

ii) $(s\hat{\ }t, B) \in \mathcal{F} \;\Rightarrow\; (s, \{\}) \in \mathcal{F}$ Eq. 4.9

iii) $(s, C) \in \mathcal{F} \wedge B \subseteq C \;\Rightarrow\; (s, B) \in \mathcal{F}$ Eq. 4.10

iv) $(s, B) \in \mathcal{F} \wedge \mathbf{a} \in A$

$\Rightarrow\; (s\hat{\ }\langle\mathbf{a}\rangle, \{\}) \in \mathcal{F} \vee (s, B \cup \{\, \mathbf{a}\, \}) \in \mathcal{F}$ Eq. 4.11

■

Rule (i) indicates that every process, before it starts, is in a position where it has performed no external events. Rule (ii) ensures the prefix closure of the traces. Rule (iii) implies that all the failures related to one trace form a powerset. Rule (iv) states that for some trace s of a process either event **a** is a possible next event, in which case s^<**a**> is a trace of the process, or event **a** is not a possible next event, in which case { **a** } will be refused.

The space of all processes represented by their failures in this way is termed the Failures Model, and has the notation $M_F$. The usual representation of a failure in this text will be (s, ℵ) where s is a trace and ℵ the refusal associated with s.

It can now be seen how refusals allow a formal semantic distinction to be drawn between the two types of choice. Consider the refusals of the processes (P□X) and (P⊓X).

$$Refusals(P \,\Box\, X) \quad = \quad Refusals(P) \cap Refusals(X) \qquad \text{Eq. 4.12}$$

$$Refusals(P \sqcap X) \quad = \quad Refusals(P) \cup Refusals(X) \qquad \text{Eq. 4.13}$$

In addition, the reasoning which showed that $M_T$ could only support safety properties does not apply to $M_F$. The Failures of a process STOP with an alphabet A is

$$Failures(STOP) = (\langle\rangle, \mathbb{P}(A)) \qquad \text{Eq. 4.14}$$

For an arbitrary process P it does not follow that

$$Failures(STOP) \quad \subseteq \quad Failures(P) \qquad \text{Eq. 4.15}$$

Thus a specification which satisfies P in $M_F$ does not by necessity satisfy STOP, and so is not by necessity a safety specification.

**Rule for Alphabetized Parallel Operator in $M_F$**

$$\begin{array}{l} P \textbf{ sat } S_1(s, \aleph_1) \\ \underline{X \textbf{ sat } S_2(s, \aleph_2) \qquad\qquad\qquad\qquad} \\ P \;_A\|_B\; X \textbf{ sat } S_1(s \upharpoonright A, \aleph_1) \wedge S_2(s \upharpoonright B, \aleph_2) \end{array}$$

Eq. 4.19

where $\aleph_1$, $\aleph_2$ are the respective refusals.

**Rule for the Recursive Operator**

Let F be a CSP function which is a contraction on the metric space ($M_F$, $d_F$). Furthermore let R be a continuous predicate over $M_F$. If R is such that for all processes P

$$\forall P \bullet (R(P) \quad \Rightarrow \quad R(F(P)))$$

Eq. 4.20

then the fixed point of F satisfies R. That is

$$\mu P.F(P) \textbf{ sat } R$$

Eq. 4.21

## 4.3 Timed CSP.

There is a need to be able to express parallel systems, such as Communicating Sequential Processes, with time incorporated. For example real-time applications inevitably involve circumstances where correctness may depend on even the most subtle of timing considerations. The recognition for such a need has led to the development of Timed CSP, or TCSP.

A number of models have been proposed which allow CSP to be applied to real-time systems. Zwarico and Lee [ Zwarico 85 ] suggest adding simple timing constraints to the Traces Model. Boucher and Gerth [ Boucher 87 ] extend the untimed failures semantics of Brookes *et al* [ Brookes 84 ] in a manner which specifically aims to preserve the recursive theory.

It is the timed models of Reed and Roscoe [ Reed 86 ] and subsequent refinements [ Reed 87, Davies 89a, Schneider 90 ] which are considered in this text. This is because they are essentially extensions of the untimed Failures Model so far discussed and thus can be treated with similar semantics and inference rules. They may also be represented as metric spaces under a suitable metric and there exist well documented examples of their use in expressing specifications and factorizing correctness proofs [ Jackson 89, Davies 89b ].

The following assumptions are made about timing in a distributed system

i) There is a non-zero lower time bound between the occurrence of any two events in a sequential process. This is to prevent singularities such as the occurrence of an infinite number of events in a finite time.

ii) The times at which events occur in the system relate to a global clock. Time passes at the same rate in each process.

### 4.3.1 Extending The Syntax.

The syntax of TCSP is the same as that of CSP except in one regard; the introduction of the time delay operator, WAIT. For a time interval $\tau$ the process (WAIT $\tau$) represents a process which starts, waits for time $\tau$, and then terminates successfully with the termination event.

### 4.3.2 Timed Events, Traces and Refusals.

A timed event in TCSP is a pair $(\tau, \mathbf{a})$ where $\mathbf{a}$ is an event which occurs at time $\tau$. The time domain is continuous and is represented by the positive real numbers $\mathbb{R}^+$. A timed trace is a sequence of timed events arranged in their chronological order.

In the Timed Failures Model of TCSP, processes are identified with sets of timed failures. A timed failure is a pair $(s, \aleph)$ where s is a timed trace and $\aleph$ is a timed refusal. A timed refusal is a set of timed events. The presence of a timed event $(\tau, \mathbf{a})$ in the refusal of $(s, \aleph)$ indicates that, after having engaged in trace s, the process in question may refuse to perform event $\mathbf{a}$ at time $\tau$.

The above definitions are sufficient to develop another mathematical interpretation of processes in terms of the timed failures of a process. Informally a TCSP process is a pair $(A, \mathcal{T})$ where A is any set of symbols representing the alphabet and $\mathcal{T}$ is a relationship between the set of all timed traces and the set of all timed refusals. For a more precise definition in the style of Definition 4.1 the reader is referred to [ Reed 90 ]. This model is called the Timed Failures Model or $TM_F$. In this text the mapping from a TCSP syntax to the semantic domain $TM_F$ is represented by the semantic function $f_T$.

### 4.3.3 Specifications on $TM_F$.

As with $M_T$ and $M_F$, specifications in $TM_F$ are expressed as predicates. For a process P with behavioural specification R this is equivalent to

$$\mathtt{P}\ \mathbf{sat}\ \mathtt{R} \quad \Leftrightarrow \quad \forall\, (\mathtt{s}, \aleph) \in f_T(\mathtt{P}) \bullet \mathtt{R}(\mathtt{s}, \aleph) \qquad \text{Eq. 4.22}$$

To facilitate the representation of specifications in $TM_F$ some extra timed operators are required [ Schneider 90 ]. The expression *end*(s) gives the time at which the trace s performed its last event. In addition the expression *end*($\aleph$) gives the time of the last timed refusal in the set $\aleph$. The expression *exev*(s) extracts the set of all event names from a timed trace (timed events with the time removed).† For example

$$end(<(1, \mathtt{up}), (2.54, \mathtt{down})>) \quad = \quad 2.54 \qquad \text{Eq. 4.23}$$

$$end(\{ (2, \mathtt{left}), (3, \mathtt{left}) \}) \quad = \quad 3 \qquad \text{Eq. 4.24}$$

$$exev(<(1, \mathtt{up}), (2.54, \mathtt{down})>) \quad = \quad \{ \mathtt{up}, \mathtt{down} \} \qquad \text{Eq. 4.25}$$

All behavioural specifications in $TM_F$ are continuous [ Schneider 90 ].

### 4.3.4 $TM_F$ as a Complete Metric Space.

$TM_F$ can be represented as a complete metric space under a suitable metric [ Reed 87 ]. If P is a TCSP process and $\tau \in [\,0,\infty)$, then define $\mathtt{P}(\tau)$ as representing the behaviour of P up to time $\tau$. That is

$$\mathtt{P}(\tau) \quad \equiv \quad \{ (\mathtt{s}, \aleph) \mid (\mathtt{s}, \aleph) \in f_T(\mathtt{P}) \wedge end(\mathtt{s}) < \tau \wedge end(\aleph) < \tau \} \qquad \text{Eq. 4.26}$$

The complete metric on $TM_F$ is defined

$$d_{TF}(\mathtt{S}_1, \mathtt{S}_2) = 2^{-\mathtt{tm}}, \quad \text{where } \mathtt{tm} = \max\{ \tau \mid \mathtt{S}_1(\tau) = \mathtt{S}_2(\tau) \} \qquad \text{Eq. 4.27}$$

All of the syntactic operators of CSP, except hiding, have been shown to be nonexpansive under $d_{TF}$, and the prefix operator is a contraction. The same syntactic rules for establishing contraction from a function's syntax apply as for $M_T$. That is if every occurrence of the process variable in a function is guarded then that function has a unique fixed point.

† Note: In [ Schneider 90 ] *exev* is represented by the symbol $\sigma$. However this symbol is used later in a different context and so is changed to avoid confusion.

There is a sound and complete compositional proof system for $TM_F$ which associates an inference rule with each of the syntactic operators [ Schneider 90 ]. This proof system allows the proof for a composite TCSP process to be reduced to a set of subproofs for component subprocesses. For a complete set of the inference rules and a proof of their soundness and completeness the reader is directed to [ Schneider 90 ]. However, one rule in particular will be used later on in this chapter and so is reproduced here.

For processes $P_1$ and $P_2$ with alphabets A, B respectively the rule associated with the alphabetised parallel operator is

$$\begin{array}{l} \left.\begin{array}{l} P_1 \textbf{ sat } S_1(s, \aleph) \\ P_2 \textbf{ sat } S_2(s, \aleph) \\ exev(s_1, \aleph_1) \subseteq A \ \wedge \ exev(s_2, \aleph_2) \subseteq B \\ \wedge \ exev(\aleph_3) \in \Sigma - (A \cup B) \\ S_1(s_1, \aleph_1) \ \wedge \ S_2(s_2, \aleph_2) \\ \wedge \ s \ \in Traces(P_1 {}_A\|_B P_2) \end{array}\right\} \quad S(s_3, \aleph_1 \cup \aleph_2 \cup \aleph_3) \\ \hline P_1 {}_A\|_B P_2) \textbf{ sat } S(s, \aleph) \end{array}$$

Eq. 4.28

## 4.4 Other Semantic Domains and their Hierarchy.

In addition to the three semantic interpretations so far discussed (Traces, Failure, Timed Failures) there are other semantic models with the ability to distinguish different characteristics of a system. Some of these include the concept of the stability of a trace. The stability of a trace is an indication of whether and when a process, after engaging in that trace, will recover and be able to perform further events. This is divided into two categories

i) Timed stability: This associates with each trace a particular time at which the process, after having engaged in that trace, ceases any internal activity and becomes able to perform another event.

ii) Untimed Stability: This associates with each trace a particular value according to whether or not the process will be able to stabilise and perform another event after having engaged in that trace. If the process may continue then the value is set to zero, if not then the value is set to infinity.

Some of the other semantic models available in CSP are

- Failures-Divergence Model - Represents a process as its set of failures and divergences. A divergence of a process is any trace of that process after which the process diverges, or behaves chaotically [ Brookes 85 ].

- The Failures-Stability Model ($M_{FS}$) - Represents a process as a set of observations of the form $(s, \phi, \aleph)$, where $s$ is an untimed trace, $\aleph$ the set of refusals of that trace and $\phi$ the untimed stability of the process.[ Reed 90 ].

- Timed Failures-Stability Model ($TM_{FS}$) - Represents a process as a set of observations of the form $(s, \zeta, \aleph)$, where $s$ is a timed trace, $\aleph$ the refusals of $s$ and $\zeta$ the timed stability of $s$ [ Reed 90 ].

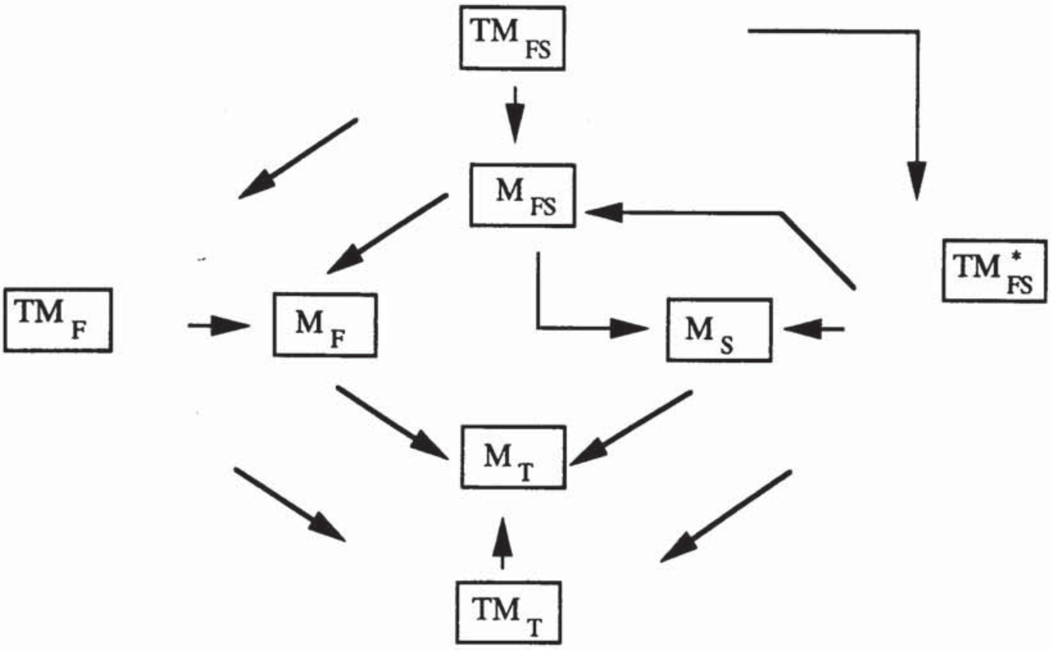
- Timed Traces Model ($TM_T$) - Represents a process as the set of its timed traces [ Boucher 87 ].

- The (Untimed Failures) - (Timed Stability) Model ($TM^{*}_{FS}$) - Represents a process as a set of observations of the form $(s, \zeta, \aleph)$, where $s$ is a timed trace, $\zeta$ is the time at which the process stabilises after $s$, and $\aleph$ is the powerset of untimed events which the process is unable to perform after $s$, that is $\aleph$ is the untimed refusal set.

- Untimed Stability Model ($M_S$) - Represents a process as a set of observations of the form $(s, a)$, where $s$ is an untimed trace of the process and a is the untimed stability of that process after engaging in $s$.

Reed [ Reed 90 ] has shown that the semantic definitions of all of these models can be tailored so that they form a precise hierarchical structure. This hierarchy is based around a set of projection mappings between semantic models which preserve behavioural information. This supports a method whereby the design of complex timed systems can be achieved by first considering simple semantics and then refining them. This structure is presented in *Figure 4.1*.

For the majority of the above domains it is possible to deduce a sound and complete inference rule associated with each operator which leads to a compositional proof system. An exception to this is the Timed Failures-Stability Model $TM_{FS}$. Blamey [ Blamey 89 ] has shown that certain inference rules derived from semantic definitions in $TM_{FS}$ are incomplete. He suggests overcoming this by using a semantics which defines a process in terms of its timed failures and the times at which it is not stable, i.e. instability sets.



| | | | |
|---|---|---|---|
| $M_T$ | - Untimed Traces | $TM_T$ | - Timed Traces |
| $M_S$ | - Untimed Stability | $TM_F$ | - Timed Failures |
| $M_F$ | - Untimed Failures | $TM_{FS}$ | - Timed Failures Stability |
| $M_{FS}$ | - Failures Stability | $TM_{FS}^*$ | - Untimed Failures Timed Stability |

*Figure 4.1: Hierarchy of Semantic Domains*

## 4.5 The Application.

The chapter now illustrates the use of the formal methods associated with CSP and TCSP by considering a specific industrial example. It shows how employing those formal methods leads to the specification, design and verification of an embedded logic controller for a real-time system.

To achieve this it is initially important to establish the objectives and approach. These are summarised by the following points

- Consider and analyse a specific example of a real-time logic controller required for an industrial application.

- Produce a tangible expression of the requirements, or specification, of the system.

- Express this specification in as precise and unambiguous mathematical form as possible.

- Produce and express formally a process for a system which aspires to meet the specification.

- Show that a formal mathematical representation of the process satisfies the specification. That is show the correctness of the process relative to the specification.

- Appraise the process and the specification with respect to the requirements of the customer. Incorporate any improvements in both which have resulted from the study of the formal representations and the correctness proofs.

### 4.5.1 The Application and its Appeal.

The application chosen involves controlling part of a high speed manufacturing plant used for packaging. The plant itself is made by Molins Engineering PLC of Coventry in the UK. Specifically it concerns the control of a slider and drum mechanism. The slider is a moving arm which periodically inserts into an aperture on the periphery of a high speed rotating drum. The desired operation of the system is that the rotating drum should decelerate from spinning and stop, then the slider is inserted while the drum is stationary. The slider then performs whatever function is required in the aperture and is retracted from the drum, which then proceeds to accelerate and spin again [ Clarke 92a ].

Traditionally the individual actions of the system were coordinated by gears and cams powered by a central driving mechanism. A disadvantage of this arrangement is its inflexibility. Changing the product on a manufacturing machine often entails rearranging the precision gear and cam mechanisms. This involves stopping the machine, stripping it down and installing alternative mechanisms. All of this is time during which the machine is not producing. Additionally, introducing a new product may involve tooling a new set of gears or cams. It was proposed to replace the central drive with a set of independent software controlled drives. The coordination previously supplied by the gears and cams was to be replaced by software and inter-process communications.

It is worth mentioning the motivation behind the choice of this application. To a certain extent an industrial example will be less "well behaved" than an idealized example. An idealized example would be better able to illustrate the subtleties and nuances of the formal language and could be formulated so as to avoid any difficult circumstances. However, because the true aim here is to both illustrate the use of CSP and evaluate it as a tool for real-time systems, an idealized example would limit the effectiveness of any appraisal. An industrial application illuminates the theory of CSP in a more relevant context. Additionally it was noted that in the literature there is a lack of publications which apply these techniques to manufacturing systems.

### 4.5.2 Specification.

The plant involves the coordination of machinery which rotates at speeds and accelerations of up to 14 rad/sec and 800 rad/sec$^2$ respectively. Consequently both slider and drum have significant inertia. *Figure 4.2* and *Figure 4.3* supply the intended velocities and motion profiles of the mechanism. Additionally it is typically placed in a safety critical environment where a failure may potentially result in serious injury or loss of life to personnel. Because of this the overriding factor in the specification of the controller must be to ensure hazard free operation.

The main hazard presented by the mechanism is that of a collision between the slider and drum. This is caused by two circumstances. First, the drum accelerating from rest while the slider is still within its periphery. Second, the slider inserting while the drum is still rotating. Consequently the specification must stipulate that these circumstances do not arise. This is expressed in a natural language form as

i) The drum must not start to rotate if the slider is inserted.

ii) The slider must not insert if the drum is still rotating.

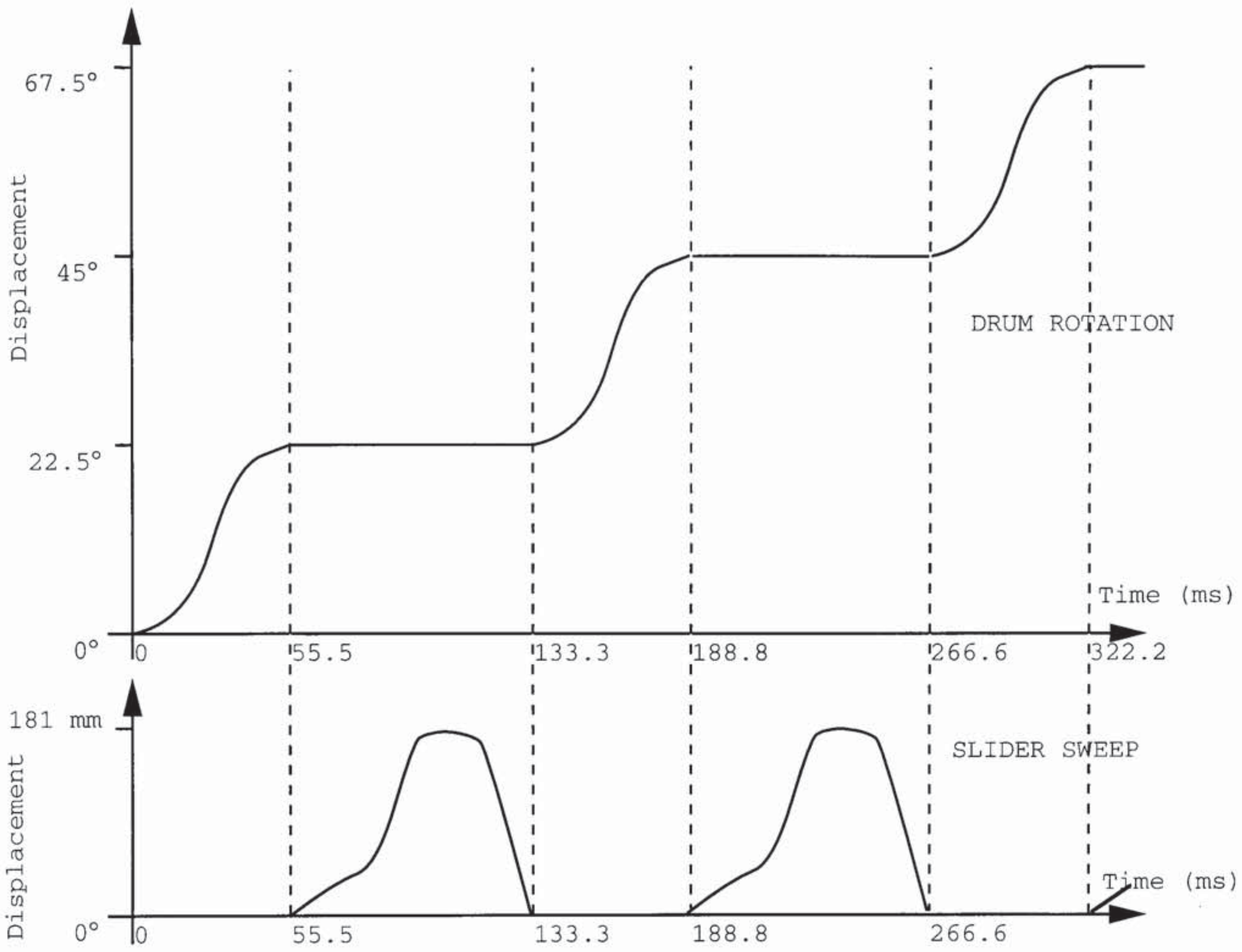


*Figure 4.2: Time/Displacement Profiles for Arbor and Drum*

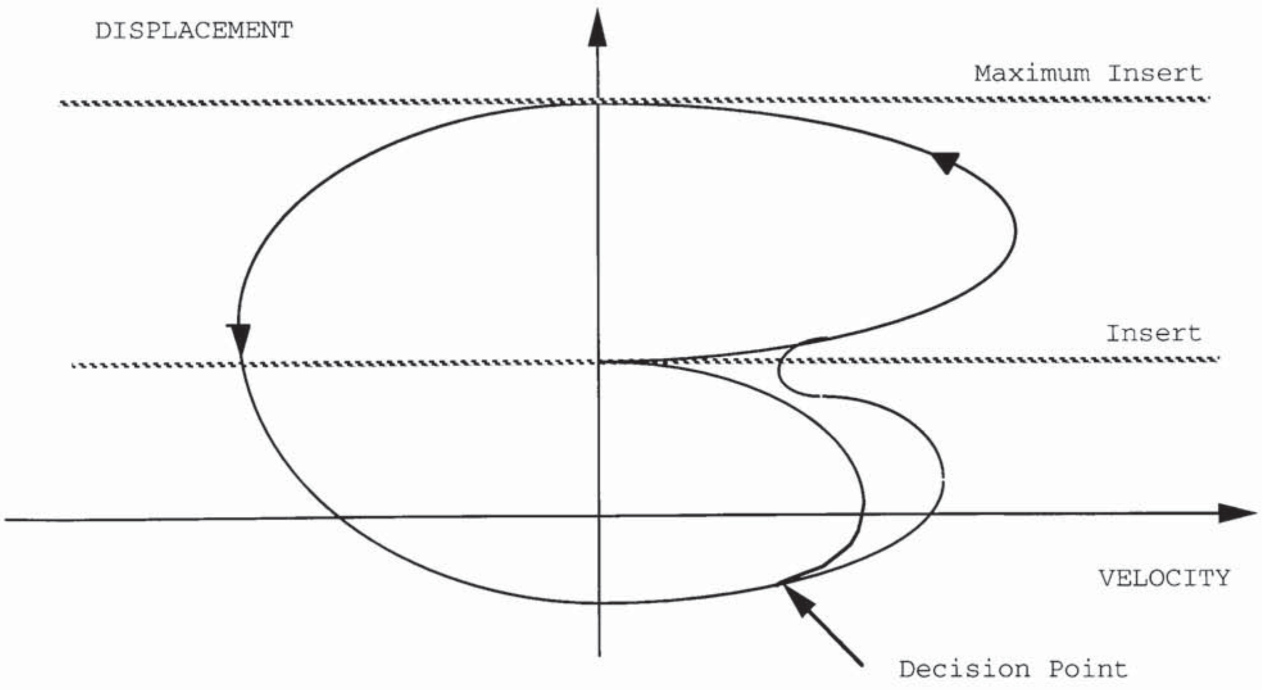


*Figure 4.3: Positioning of Decision Point in Slider Motion Cycle*

One of the simplest means of preventing a collision between the slider and drum would be to prohibit their synchronous motion. A sequential control structure would ensure that the slider did not move until the drum was stationary and in position, and that the drum did not start to move until the slider was at rest and not inserted. However, there are points in the motion profiles of both slider and drum at which concurrent motion is possible without a collision occurring. Not to utilise these points would be an inefficient use of the time resource and thus to increase the operation speed of the mechanism it is advantageous to maximise concurrent motion. Consequently the natural language specification is extended to include

iii) Concurrent motion must be employed whenever possible provided it does not compromise the hazard free operation of the mechanism.

At this point an intuitive feel for the design of the controller starts to appear. If synchronous motion is to be allowed then the mechanism has roughly two phases of operation. One where neither component impinges on the motion of the other and thus each component can move freely, and one where the components do impinge upon each others motion and thus certain conditions must be met to prevent collision. When the mechanism passes from the free phase to the restricted stage it is apparent that the control logic will have to make a decision about the next course of action. This choice will be based on whether it is proper to proceed and allow the slider to insert into the drum, or whether it is improper. Essentially the controller will reach a decision point at which it must either commit the mechanism to insertion because it is secure to do so, or abort the insertion procedure because the environmental circumstances may lead to collision.

The implications of introducing a decision point correctly require consideration. The controller must always be able to make a delay-free decision to either commit or abort. If it is unable to do so then control over the system is diminished and a collision may occur. This is expressed by

iv) At its decision point the slider must always be able to commit or abort.

Also, because the slider is travelling towards the periphery of the drum a decision has to be made in sufficient time so that, if required, the slider's momentum can be overcome before it encounters the periphery of the drum. This requirement can be expressed as the natural language specification

Specification (ii) implies that the process may only perform the event **enter** if the drum is stationary. That is

$$\mathcal{Last}(\mathtt{s}) = \mathtt{enter} \quad \Rightarrow$$
$$\neg\,(\mathcal{Last}(\mathtt{s}\upharpoonright\{\ \mathtt{dstart}, \mathtt{dstop}\ \}) = \mathtt{dstart}) \qquad \text{Eq. 4.32}$$

For simplicity it is noted that specifications (i) and (ii) are equivalent to the specification

$$\mathcal{Last}\,(\mathtt{s}\upharpoonright\{\ \mathtt{enter}, \mathtt{extract}\ \}) = \mathtt{enter}$$
$$\Rightarrow \qquad (\mathcal{Last}\,(\mathtt{s}\upharpoonright\{\ \mathtt{dstart}, \mathtt{dstop}\ \}) \neq \mathtt{dstart}) \qquad \text{Eq. 4.33}$$

### 4.5.4 Formalizing Untimed Liveness Specifications.

Specification (iv) is concerned with establishing what a process will do, and as such it constitutes a liveness specification. However, Section 4.2 demonstrated that the traces model $M_T$ was insufficient for the description of liveness properties. As a result it is necessary to employ a higher semantic model for the representation of specification (iv). The semantic model employed is the failures model $M_F$.

The logic controller will have a decision point at which it must decide between continuing with the present motion or invoking an abort procedure. In order to allow maximum freedom of motion for the controller this decision point is situated at the point just before the slider is irrevocably committed to inserting into the drum by its inertia. That is when the slider can still just be brought to rest before it inserts. The actions of the slider are then considered to be its approach to the periphery of the drum, labelled as the event **approach**, and then either an event indicating committal, **commit**, or one indicating abortion, **abort**. In terms of the traces of the control process the decision point would be after **approach** and before either **commit** or **abort**.

Specification (iv) stipulates that at its decision point the controller must always be able to order the actions **commit** or **abort** at the next step. This would be satisfied if it could be shown that either commit or abort were not failures at the decision point. That is

$$\text{DEC. POINT} \Rightarrow \qquad \text{ABORT IS NEXT} \vee \text{COMMIT IS NEXT} \quad \text{Eq. 4.34}$$

As mentioned, the slider is at decision point between **approach** and **commit/abort**.

The conclusion was that a method of design by directly translating the formal specification into a formal model is not suitable in these circumstances. Instead a more informal method of design was adopted. It was felt that the application was small and well understood, and that a controller could be designed without recourse to structured design methods. Thus, using the formal representations of specifications (i), (ii) and (iv) as a basis and the informal requirement of specification (iii) as a guide, an intuitive iterative approach to design was adopted. That is a process was postulated, and compared with the formal specification to ensure it complied. Then, through understanding of the operation and requirements of the system an improved model was put forward, and compared. The informality of this approach implies that the correctness of the system cannot be ensured by virtue of its design. As a result a greater emphasis on verification is needed, as it is then the only assurance that the process functions correctly relative to its specification.

This method of speculation and refinement eventually led to the design of an untimed control logic for the slider drum mechanism. This is presented in *Figure 4.4*

$$(\mathtt{SLIDER}\ {}_{A}\|_{B}\ \mathrm{CON})\ {}_{A\cup B}\|_{C}\ \mathrm{DRUM}$$

A, B, C are alphabets given as:

$$\begin{aligned}\mathtt{A} = \{\ & \mathtt{abort}, \mathtt{accelerate}, \mathtt{allow}, \mathtt{approach}, \mathtt{commit},\\ & \mathtt{decelerate}, \mathtt{enter}, \mathtt{extract}, \mathtt{perform}, \mathtt{slow}\ \}\end{aligned}$$

$$\mathtt{B} = \{\ \mathtt{abort}, \mathtt{allow}, \mathtt{commit}, \mathtt{dstart}, \mathtt{dstop}, \mathtt{extract}\}$$

$$\mathtt{C} = \{\ \mathtt{dstart}, \mathtt{dstop}\ \}$$

SLIDER, CON and DRUM are recursive processes defined by:

$$\begin{aligned}\mathtt{SLIDER} &= \mu P.F_{um}(P)\\ \mathrm{CON} &= \mu P.G_{um}(P)\\ \mathrm{DRUM} &= \mu P.H_{um}(P)\end{aligned}$$

$F_{um}$, $G_{um}$, $H_{um}$ are CSP functions defined by

$$\begin{aligned}F_{um}(P) = \quad & \mathtt{approach} \rightarrow ((\mathtt{commit} \rightarrow \mathtt{enter} \rightarrow \mathtt{slow} \rightarrow\\ & \qquad \mathtt{perform} \rightarrow \mathtt{extract} \rightarrow P)\\ & \qquad \Box\\ & \qquad (\mathtt{abort} \rightarrow \mathtt{decelerate} \rightarrow \mathtt{allow} \rightarrow\\ & \qquad \mathtt{accelerate} \rightarrow \mathtt{enter} \rightarrow \mathtt{slow} \rightarrow\\ & \qquad \mathtt{perform} \rightarrow \mathtt{extract} \rightarrow P))\end{aligned}$$

$$\begin{aligned}G_{um}(P) = \quad & \mathtt{dstart} \rightarrow (\mathtt{dstop} \rightarrow P \ \Box\ \mathtt{abort} \rightarrow \mathtt{dstop} \rightarrow P)\\ & \Box\ \mathtt{commit} \rightarrow \mathtt{extract} \rightarrow P\\ & \Box\ \mathtt{allow} \rightarrow \mathtt{extract} \rightarrow P\end{aligned}$$

$$H_{um}(P) = \quad \mathtt{dstart} \rightarrow \mathtt{dstop} \rightarrow P$$

*Figure 4.4: Untimed Model*

This priority structure was subsequently introduced into the design of the controller, and lead to the timed controller model given in ***Figure 4.5*** overleaf.

### 4.5.7 Liveness in the Timed Model.

The untimed model was only able to determine the untimed behaviour of the process. As a result of extending the untimed model to incorporate time it is now possible to reason more clearly about its potential to make timely decisions. Specification (v) requires that a decision be reached within a prescribed time. Thus after reaching the decision point, the process must perform either **`commit`** or **`abort`** within time interval $\eta$ say.

The Timed Failures Model can determine what a process may not do at a particular time, and what a process may be able to do at a particular time. But the semantics alone cannot stipulate that a process will perform a particular event at a particular time. This is because the formal treatment of processes always assumes them to be in an environment which can affect the occurrence of events. To say that a process will perform an event requires knowledge about both the process and its environment.

Environmental conditions can be represented as predicates on the observations of a process [ Jackson 89 ]. The environment of the logic controller is considered to be the position/motion controllers upon which it sits. One factor to consider in such an arrangement is that every event recorded by the process is translated into a command to the motion controller and each of these commands takes time to be implemented.

Two assumptions are made about the interaction of the physical system with its environment. First the events **`commit`**, **`abort`** and **`allow`** occur as soon as they become able to. Secondly the time interval $\partial$ is so arranged that the events **`dstop`**, **`dstart`** have time to stabilise. Therefore the events **`dstart`** and **`dstop`** occur immediately after `WAIT` $\partial$ terminates.

Say the mechanism reaches the decision point at time $\tau_a$. Then the specification insists that after reaching this point it must perform either **`commit`** or **`abort`** within a certain prescribed time, say $\eta$. This is equivalent to saying that if the decision point starts at time $\tau_a$ after a trace `s` then there is some time $\tau'$, where $\tau_a < \tau' < \tau_a+\eta$, such that either ($\tau'$, **`abort`**) is not a refusal of `s` or ($\tau'$, **`commit`**) is not a refusal of `s`. This is expressed as the behavioural specification

$$\begin{aligned}&(\mathit{Last}\,(s\restriction\{\ \mathtt{approach},\mathtt{commit},\mathtt{abort}\ \}) = (\mathtt{approach})\\ &\wedge\ \mathit{end}(s\restriction\{\ \mathtt{approach},\mathtt{commit},\mathtt{abort}\ \}) = \tau_a)\ \Rightarrow\ (\exists\,\tau'\bullet\\ &(\tau_a < \tau' < \tau_a+\eta)\ \wedge\ ((\tau_a,\mathtt{commit}) \notin \aleph) \vee (\tau_a,\mathtt{abort}) \notin \aleph)))\qquad \text{Eq. 4.41}\end{aligned}$$

The proof that equation 4.41 satisfies the process $(\mathtt{SLIDER}^{T}\ {}_{A}\|_{B}\ \mathtt{CON}^{T})\ {}_{A\cup B}\|_{C}\ \mathtt{DRUM}^{T}$ as given in ***Figure 4.5*** is presented in outline in Section 4.7.3.

$$(\mathtt{SLIDER}^{T}\ {}_{A}\|_{B}\ \mathrm{CON}^{T})\ {}_{A\cup B}\|_{C}\ \mathrm{DRUM}^{T}$$

where, with alphabets $\mathtt{A}$, $\mathtt{B}$, $\mathtt{C}$ as for the untimed model respectively

$$\begin{array}{lcl}
\mathtt{SLIDER}^{T} & = & \mu\mathtt{P}.\mathtt{F}_{\mathtt{tm}}(\mathtt{P}) \\
\mathtt{CON}^{T} & = & \mu\mathtt{P}.\mathtt{G}_{\mathtt{tm}}(\mathtt{P}) \\
\mathrm{DRUM}^{T} & = & \mu\mathtt{P}.\mathtt{H}_{\mathtt{tm}}(\mathtt{P})
\end{array}$$

$\mathtt{F}_{\mathtt{tm}}$, $\mathtt{G}_{\mathtt{tm}}$ and $\mathtt{H}_{\mathtt{tm}}$ are TCSP functions defined by

$$\begin{array}{ll}
\mathtt{F}_{\mathtt{tm}}(\mathtt{P}) = & \mathbf{approach} \rightarrow ((\mathbf{commit} \rightarrow \mathbf{enter} \rightarrow \mathbf{slow} \rightarrow \\
& \quad \mathbf{perform} \rightarrow \mathbf{extract} \rightarrow \mathtt{P}) \\
& \quad \Box \\
& \quad (\mathbf{abort} \rightarrow \mathbf{decelerate} \rightarrow \mathbf{allow} \rightarrow \\
& \quad \mathbf{accelerate} \rightarrow \mathbf{enter} \rightarrow \mathbf{slow} \rightarrow \\
& \quad \mathbf{perform} \rightarrow \mathbf{extract} \rightarrow \mathtt{P}))
\end{array}$$

$$\begin{array}{lll}
\mathtt{G}_{\mathtt{tm}}(\mathtt{P}) = & \mathtt{WAIT}\ \partial\ ;\ (\mathbf{dstart} \rightarrow & (\mathtt{WAIT}\ \partial\ ;\ (\mathbf{dstop} \rightarrow \mathtt{P}) \\
& & \Box \\
& & \mathbf{abort} \rightarrow \mathbf{dstop} \rightarrow \mathtt{P})) \\
& \Box\ \mathbf{commit} \rightarrow \mathbf{extract} \rightarrow \mathtt{P} & \\
& \Box\ \mathbf{allow} \rightarrow \mathbf{extract} \rightarrow \mathtt{P} &
\end{array}$$

$$\mathtt{H}_{\mathtt{tm}}(\mathtt{P}) = \quad \mathbf{dstart} \rightarrow \mathbf{dstop} \rightarrow \mathtt{P}$$

*Figure 4.5: Timed Model*

## 4.6 Summary.

The control of the Slider-Drum mechanism with independently communicating microprocessors has three main properties which point to using TCSP/CSP for an analysis. First it is a distributed system which requires parallel actions coordinated by communications. CSP provides a framework around which such a system can be specified and studied.

Secondly the system will operate within hard time constraints and therefore requires a semantic model which is able to capture and reason about timing. This is provided by the Timed Failures Model. Finally the communications involved are concerned with the flow of control information rather than the transmission of data. Because of this such communications can be expressed easily in terms of synchronous events. This leads to a clearer mathematical representation of the system as a process in CSP.

An alternative approach to the design of such a controller has been undertaken by Sagoo and Holding [ Sagoo 90 ]. Their approach specifies the system using Temporal Petri nets. By introducing temporal logic into the Petri net description the systems behaviour can be explored in terms of the state reachability tree. They propose a Petri net description of a control logic and show that it does not have reachable hazardous states and that it conforms to a set of timing constraint. This employs a consistent temporal logic proof technique to show that the control logic satisfies its specification and does not cause hazardous operation.

This chapter has described the use of CSP and TCSP in the specification and verification of a controller for a hard real-time distributed system. The application, which is representative of the type of system commonly found in flexible manufacturing machinery, includes both time-critical and safety-critical functions. It has been shown that CSP and TCSP can be used to reason about such properties and to verify hazard free operation. The design permits the drum and slider to move as freely as possible within the constraints of the specification.

The main advantage of formal verification is the high level of confidence it provides in a system. Naturally, an axiomatic proof system depends not only on the soundness of the inference rules but on the truth of initial axioms. The compositionality of the proof system permits axioms to be established more easily by process decomposition. However, the complex nature of the proof system leaves it prone to error and requires a high level of skill and understanding on the part of the designer. The techniques described in this paper are most advantageously applied to well specified and well understood problems.

The hierarchy of semantic domains provided by CSP and TCSP permits levels of abstraction. Existing domains allow for nondeterminism, divergence, time and stability. However, the notation adopted, P **sat** S, is restricted to behavioural (continuous) specifications. Behavioural specifications possess a wide scope for specifying system properties, but it is noted that there are higher logics, such as temporal and other modal logics, which have a greater capacity for specification. There is currently work under way to develop a temporal logic based calculus for CSP [ Barringer 85b, Davies 92 ].

## 4.7 Proofs.

This section presents proofs for the formal specifications expressed in the above text. It consists of three subsections. The first two give the proofs for the safety and liveness properties of the untimed model The third section outlines the proof approach for the timed liveness properties.

### 4.7.1 Proof of the Safety Properties of the Untimed Model.

To prove

$$\begin{aligned}&(\mathtt{SLIDER}\ {}_{A}\|_{B}\ \mathrm{CON})\ {}_{A\cup B}\|_{C}\ \mathrm{DRUM}\ \mathbf{sat}\\&(\mathit{Last}\,(s\upharpoonright\{\ \mathtt{enter},\mathtt{extract}\ \}) = \mathtt{enter})\\&\Rightarrow\qquad(\mathit{Last}\,(s\upharpoonright\{\ \mathtt{dstart},\mathtt{dstop}\ \}) \neq \mathtt{dstart})\end{aligned}\qquad\text{Eq. 4.42}$$

The initial axioms proposed for each process are

$$\begin{aligned}&\mathtt{SLIDER}\ \mathbf{sat}\ \Xi_1,\\&\Xi_1(s)\ \equiv\\&(\mathit{Last}\,(s\upharpoonright\{\ \mathtt{enter},\mathtt{extract}\ \}) = \mathtt{enter})\\&\Rightarrow\\&(\mathit{Last}\,(s\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{commit}\\&\vee\ \mathit{Last}\,(s\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{allow})\end{aligned}\qquad\text{Eq. 4.43}$$

$$\begin{aligned}&\mathrm{CON}\ \mathbf{sat}\ \psi_1,\\&\psi_1(s)\ \equiv\\&(\mathit{Last}(s\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{commit}\\&\vee\ \mathit{Last}\,(s\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{allow})\\&\Rightarrow\\&(\mathit{Last}\,(s\upharpoonright\{\ \mathtt{dstart},\mathtt{dstop}\ \}) \neq \mathtt{dstart})\end{aligned}\qquad\text{Eq. 4.44}$$

$$\mathtt{DRUM}\ \mathbf{sat}\ \Phi_1,$$

$$\Phi_1(\mathrm{s}) \equiv \mathrm{TRUE} \qquad \text{Eq. 4.45}$$

It is necessary to establish the truth of the above axioms. The defining functions $\mathrm{F_{um}}$, $\mathrm{G_{um}}$ and $\mathrm{H_{um}}$ of the processes SLIDER, CON and DRUM are composed of nonexpansive monotonic operators (Prefix and Deterministic Choice). Each occurrence of the process variable is guarded and thus these processes represent the unique fixed points of contractions in $\mathrm{M_T}$.

The predicates $\Xi_1$, $\psi_1$ and $\Phi_1$ are behavioural specifications in $\mathrm{M_T}$ and are thus continuous. All are satisfiable by the process STOP.

Consider some process P which satisfies predicate $\Xi_1$. Application of the semantic definitions for the deterministic choice and prefix operators on $\mathrm{M_T}$ yields the result that

$$\begin{aligned}
&\mathit{Traces}(\mathrm{F_{um}}(\mathrm{P})) = \{\, \mathrm{s} \mid \mathrm{s} \in \mathit{Traces}(\mathrm{F_{um}}(\mathrm{STOP}) \vee (\mathrm{t} \in \mathit{Traces}(\mathrm{P}) \wedge \\
&(\mathrm{s} = <\mathbf{\texttt{approach}}, \mathbf{\texttt{commit}}, \mathbf{\texttt{enter}}, \\
&\qquad\qquad \mathbf{\texttt{slow}}, \mathbf{\texttt{perform}}, \mathbf{\texttt{extract}} >\hat{}\ \mathrm{t} \\
&\vee\ \mathrm{s} = <\mathbf{\texttt{approach}}, \mathbf{\texttt{abort}}, \mathbf{\texttt{decelerate}}, \mathbf{\texttt{allow}}, \\
&\qquad\qquad \mathbf{\texttt{accelerate}}, \mathbf{\texttt{enter}}, \mathbf{\texttt{slow}}, \\
&\qquad\qquad \mathbf{\texttt{perform}}, \mathbf{\texttt{extract}} >\hat{}\mathrm{t}))\} \qquad \text{Eq. 4.46}
\end{aligned}$$

Analysis of the above leads to the conclusion that if predicate $\Xi_1$ holds for *Traces*(P) then it will hold for all traces in $\mathrm{F_{um}}$(P).

In a similar way it can be seen that the function $\mathrm{G_{um}}$ is a monotonic contraction and that $\psi_1$ is a satisfiable behavioural specification. For a process P which satisfies predicate $\psi_1$ an analysis of the semantic definition

$$\begin{aligned}
&\mathit{Traces}(\mathrm{G_{um}}(\mathrm{P})) = \{\, \mathrm{s} \mid \mathrm{s} \in \mathit{Traces}(\mathrm{G_{um}}(\mathrm{STOP}) \vee (\mathrm{t} \in \mathit{Traces}(\mathrm{P}) \wedge \\
&(\mathrm{s} = <\mathbf{\texttt{dstart}}, \mathbf{\texttt{dstop}} >\hat{}\mathrm{t}\ \vee\ \mathrm{s} = <\mathbf{\texttt{dstart}}, \mathbf{\texttt{abort}}, \mathbf{\texttt{dstop}} >\hat{}\mathrm{t} \\
&\vee\ \mathrm{s} = <\mathbf{\texttt{allow}}, \mathbf{\texttt{extract}} >\hat{}\mathrm{t} \\
&\qquad \vee\ \mathrm{s} = <\mathbf{\texttt{commit}}, \mathbf{\texttt{extract}} >\hat{}\mathrm{t})) \qquad \text{Eq. 4.47}
\end{aligned}$$

leads to the deduction that if $\psi_1$ holds for *Traces*(P) then it holds for all traces in $\mathrm{G_{um}}$(P). Thus

$$\Xi_1(\mathrm{P}) \ \Rightarrow\ \ \Xi_1(\mathrm{F_{um}}(\mathrm{P})) \qquad \text{Eq. 4.48}$$

$$\psi_1(\mathrm{P}) \ \Rightarrow\ \ \psi_1(\mathrm{G_{um}}(\mathrm{P})) \qquad \text{Eq. 4.49}$$

The conditions for applying the recursion rule are met, and so SLIDER **sat** $\Xi_1$ and CON **sat** $\psi_1$ are established.

DRUM **sat** $\Phi_1$ holds trivially as $\Phi_1$ is true for all processes. Now that the initial axioms are established it is possible to infer results about the sequential processes composed in parallel. CON and SLIDER are combined with the alphabetized parallel operator to form the process

$$\mathtt{SLIDER}\ {}_{\mathtt{A}}\|_{\mathtt{B}}\ \mathtt{CON} \qquad \text{Eq. 4.50}$$

The alphabetized parallel inference rule on $M_T$ gives

$$\begin{aligned}&\mathtt{SLIDER}\ \mathbf{sat}\ \Xi_1 \ \wedge\ \mathtt{CON}\ \mathbf{sat}\ \psi_1\\ \Rightarrow\quad &(\mathtt{SLIDER}\ {}_{\mathtt{A}}\|_{\mathtt{B}}\ \mathtt{CON})\ \mathbf{sat}\ (\Xi_1(\mathtt{s}\upharpoonright\mathtt{A})\ \wedge\ \psi_1(\mathtt{s}\upharpoonright\mathtt{B}))\end{aligned} \qquad \text{Eq. 4.51}$$

Thus,

$$\begin{aligned}&\Xi_1(\mathtt{s}\upharpoonright\mathtt{A}) \equiv \mathit{Last}\ ((\mathtt{s}\upharpoonright\mathtt{A})\upharpoonright\{\ \mathtt{enter},\mathtt{extract}\ \}) = \mathtt{enter}\\ &\Rightarrow (\mathit{Last}((\mathtt{s}\upharpoonright\mathtt{A})\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{commit}\\ &\vee\ \mathit{Last}((\mathtt{s}\upharpoonright\mathtt{A})\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{allow})\end{aligned} \qquad \text{Eq. 4.52}$$

$$\begin{aligned}&\psi_1(\mathtt{s}\upharpoonright\mathtt{B}) \equiv (\mathit{Last}\ ((\mathtt{s}\upharpoonright\mathtt{B})\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{commit}\\ &\vee\ \mathit{Last}\ ((\mathtt{s}\upharpoonright\mathtt{B})\upharpoonright\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}) = \mathtt{allow})\\ &\Rightarrow \mathit{Last}\ ((\mathtt{s}\upharpoonright\mathtt{B})\upharpoonright\{\ \mathtt{dstart},\mathtt{dstop}\ \}) \neq \mathtt{dstart}\end{aligned} \qquad \text{Eq. 4.53}$$

By noting that

$$\begin{aligned}\mathtt{A}\ \supset\ (&\{\ \mathtt{enter},\mathtt{extract}\ \}\\ &\cup\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \})\end{aligned} \qquad \text{Eq. 4.54}$$

$$\begin{aligned}\mathtt{B}\ \supset\ (&\{\ \mathtt{commit},\mathtt{allow},\mathtt{extract}\ \}\\ &\cup\{\ \mathtt{dstart},\mathtt{dstop}\ \})\end{aligned} \qquad \text{Eq. 4.55}$$

and applying 3.5.3.2 (ii), it is seen that

$$\psi_1(\mathtt{s}\upharpoonright\mathtt{B})\ \equiv\ \psi_1(\mathtt{s}) \qquad \& \qquad \Xi_1(\mathtt{s}\upharpoonright\mathtt{A})\ \equiv\ \Xi_1(\mathtt{s}) \qquad \text{Eq. 4.56}$$

Thus,

$$\mathtt{SLIDER}\ {}_{A}\|_{B}\ \mathrm{CON}\ \textbf{sat}\ (\Xi_1(\mathtt{s})\ \wedge\ \psi_1(\mathtt{s})) \qquad \text{Eq. 4.57}$$

From the tautology $\Phi_1$ it follows that

$$((\mathtt{SLIDER}\ {}_{A}\|_{B}\ \mathrm{CON})\ {}_{A\cup B}\|_{C}\ \mathrm{DRUM})\ \textbf{sat}\ (\Xi_1(\mathtt{s})\ \wedge\ \psi_1(\mathtt{s})) \qquad \text{Eq. 4.58}$$

By combining $\Xi_1(\mathtt{s})$ and $\psi_1(\mathtt{s})$ it can be seen that

$$\begin{aligned}
&((\mathtt{SLIDER}\ {}_{A}\|_{B}\ \mathrm{CON})\ {}_{A\cup B}\|_{C}\ \mathrm{DRUM})\ \textbf{sat}\\
&\mathcal{L}ast\,(\mathtt{s}\upharpoonright\{\ \mathtt{enter},\ \mathtt{extract}\ \}) = \mathtt{enter}\\
&\Rightarrow \quad \mathcal{L}ast\,(\mathtt{s}\upharpoonright\{\ \mathtt{dstart},\ \mathtt{dstop}\ \}) \neq \mathtt{dstart} \qquad \text{Eq. 4.59}
\end{aligned}$$

Thus the safety properties are established for the untimed model.

### 4.7.2 Proof of the Liveness Properties for the Untimed Model.

To prove that

$$\begin{aligned}
&(\mathtt{SLIDER}\ {}_{A}\|_{B}\ \mathrm{CON})\ {}_{A\cup B}\|_{C}\ \mathrm{DRUM}\ \textbf{sat}\\
&\mathcal{L}ast\,(\mathtt{s}\upharpoonright\{\ \mathtt{approach},\ \mathtt{commit},\ \mathtt{abort}\ \}) = \mathtt{approach}\\
&\Rightarrow \quad (\mathtt{abort} \notin \aleph\ \vee\ \mathtt{commit} \notin \aleph) \qquad \text{Eq. 4.60}
\end{aligned}$$

The initial axioms proposed for each process are

$$\begin{aligned}
&\mathtt{SLIDER}\ \textbf{sat}\ \Xi_2,\\
&\Xi_2(\mathtt{s},\ \aleph) \equiv \mathcal{L}ast\,(\mathtt{s}\upharpoonright\{\ \mathtt{approach},\ \mathtt{commit},\ \mathtt{abort}\ \}) = \mathtt{approach}\\
&\Rightarrow (\mathcal{L}ast(\mathtt{s}\upharpoonright\{\ \mathtt{allow},\ \mathtt{commit},\ \mathtt{extract},\ \mathtt{abort}\ \}) = \mathtt{extract}\\
&\vee\ \mathtt{s}\upharpoonright\{\ \mathtt{allow},\ \mathtt{commit},\ \mathtt{extract},\ \mathtt{abort}\ \} = \langle\ \rangle)\\
&\wedge\ (\mathtt{abort} \notin \aleph\ \wedge\ \mathtt{commit} \notin \aleph) \qquad \text{Eq. 4.61}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{CON}\ \textbf{sat}\ \Psi_2,\\
&\Psi_2(\mathtt{s},\ \aleph) \equiv (\mathcal{L}ast\,(\mathtt{s}\upharpoonright\{\ \mathtt{allow},\ \mathtt{commit},\\
&\qquad\qquad\qquad \mathtt{extract},\ \mathtt{abort}\ \}) = \mathtt{extract}\\
&\vee\ \ \mathtt{s}\upharpoonright\{\ \mathtt{allow},\ \mathtt{commit},\ \mathtt{extract},\ \mathtt{abort}\ \} = \langle\ \rangle)\\
&\Rightarrow \quad \mathtt{abort} \notin \aleph \vee \mathtt{commit} \notin \aleph \qquad \text{Eq. 4.62}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{DRUM}\ \textbf{sat}\ \Phi_2,\\
&\Phi_2 \equiv \mathrm{TRUE} \qquad \text{Eq. 4.63}
\end{aligned}$$

The defining functions $F_{um}$ and $G_{um}$ are composed of monotonic operators which are nonexpansive on $M_F$. All occurrences of the process variable are guarded by a prefix operator and thus the $F_{um}$ and $G_{um}$ are contractions. Both $\Xi_2$ and $\Psi_2$ are behavioural specifications and thus continuous.

Analysis of the functions in *Figure 4.4* will show that

$$\Xi_2(\mathtt{P}) \;\Rightarrow\; \Xi_2(\mathtt{F}_{\mathtt{um}}(\mathtt{P})) \qquad \text{Eq. 4.64}$$

$$\psi_2(\mathtt{P}) \;\Rightarrow\; \psi_2(\mathtt{G}_{\mathtt{um}}(\mathtt{P})) \qquad \text{Eq. 4.65}$$

All the conditions hold for the recursive rule to deduce that $\mathtt{SLIDER}$ **sat** $\Xi_2(\mathtt{s}, \aleph)$ and $\mathtt{CON}$ **sat** $\psi_2(\mathtt{s}, \aleph)$.

The alphabetized parallel inference rule on $M_F$ gives

$$\begin{aligned}
&\mathtt{SLIDER}\ \textbf{sat}\ \Xi_2(\mathtt{s},\ \aleph_{1)}\ \wedge\ \mathtt{CON}\ \textbf{sat}\ \Psi_2(\mathtt{s},\ \aleph_2)\\
\Rightarrow\quad &(\mathtt{SLIDER}\ {}_{\mathtt{A}}\|_{\mathtt{B}}\ \mathtt{CON})\ \textbf{sat}\ \mathfrak{R}(\mathtt{s},\ \aleph_1 \cup \aleph_2)
\end{aligned} \qquad \text{Eq. 4.66}$$

$$\mathfrak{R} \equiv (\Xi_2(\mathtt{s}\restriction\mathtt{A},\ \aleph_{1)}\ \wedge\ \Psi_2(\mathtt{s}\restriction\mathtt{B},\ \aleph_{2)}) \qquad \text{Eq. 4.67}$$

Here $(\mathtt{s},\ \aleph_1 \cup \aleph_2)$ is the failures relation. This is applied to give

$$\begin{aligned}
&\mathtt{SLIDER}\ {}_{\mathtt{A}}\|_{\mathtt{B}}\ \mathtt{CON}\ \textbf{sat}\\
&(\mathit{Last}\ ((\mathtt{s}\restriction\mathtt{A})\restriction\{\ \mathtt{approach, commit, abort}\ \}) = \mathtt{approach}\\
\Rightarrow\quad &(\mathit{Last}((\mathtt{s}\restriction\mathtt{A})\restriction\{\ \mathtt{allow, commit, extract,}\\
&\qquad\quad \mathtt{abort}\ \}) = \mathtt{extract}\\
&\quad \vee\ (\mathtt{s}\restriction\mathtt{A})\restriction\{\ \mathtt{allow, commit, extract, abort}\ \} = <\ >)\\
&\quad \wedge\ (\mathtt{abort} \notin \aleph_1\ \wedge\ \mathtt{commit} \notin \aleph_1)\\
\wedge\\
&(\mathit{Last}\ ((\mathtt{s}\restriction\mathtt{B})\restriction\{\ \mathtt{allow, commit, extract, abort}\ \}) = \mathtt{extract}\\
&\vee\ (\mathtt{s}\restriction\mathtt{B})\restriction\{\ \mathtt{allow, commit, extract, abort}\ \} = <\ >)\\
\Rightarrow\quad &\mathtt{abort} \notin \aleph_2\ \vee\ \mathtt{commit} \notin \aleph_2)
\end{aligned} \qquad \text{Eq. 4.68}$$

Applying 3.5.3.2 (ii) and reducing this gives

$$\begin{aligned}
&\mathtt{SLIDER}\ {}_{\mathtt{A}}\|_{\mathtt{B}}\ \mathtt{CON}\ \textbf{sat}\\
&\mathit{Last}\ (\mathtt{s}\restriction\{\ \mathtt{approach, commit, abort}\ \}) = \mathtt{approach}\\
&\Rightarrow (\mathtt{abort} \notin \aleph_1\ \wedge\ \mathtt{commit} \notin \aleph_1)\\
&\wedge\ (\mathtt{abort} \notin \aleph_2\ \vee\ \mathtt{commit} \notin \aleph_2)
\end{aligned} \qquad \text{Eq. 4.69}$$

To outline the proof of equation 4.72 it is necessary to establish the preliminary result

$$
\begin{array}{l}
(\mathtt{SLIDER}^{\mathtt{T}}\ {}_{\mathtt{A}}\|_{\mathtt{B}}\ \mathtt{CON}^{\mathtt{T}}) \\
\mathbf{sat} \\
\mathcal{Last}(\mathtt{s}) = (\tau_a, \mathtt{approach}) \\
\Rightarrow \\
(\mathcal{Last}(\mathtt{s}\restriction \mathtt{A}) = (\tau_a, \mathtt{approach}) \wedge \mathtt{s}\restriction \mathtt{B} = \langle\ \rangle) \\
\\
\vee\ (\mathcal{Last}(\mathtt{s}\restriction \mathtt{A}) = (\tau_a, \mathtt{approach}) \wedge \mathcal{Last}(\mathtt{s}\restriction \mathtt{B}) = (\tau_b, \mathtt{dstop})) \\
\vee\ (\mathcal{Last}(\mathtt{s}\restriction \mathtt{A}) = (\tau_a, \mathtt{approach}) \wedge \mathcal{Last}(\mathtt{s}\restriction \mathtt{B}) = (\tau_b, \mathtt{extract})) \\
\vee\ (\mathcal{Last}(\mathtt{s}\restriction \mathtt{A}) = (\tau_a, \mathtt{approach}) \wedge \mathcal{Last}(\mathtt{s}\restriction \mathtt{B}) = (\tau_b, \mathtt{dstart}))
\end{array}
\qquad \text{Eq. 4.74}
$$

where $\tau_a$, $\tau_b$ are such that

$$\tau_a - \tau_b < \quad \partial \qquad \text{Eq. 4.75}$$

This can be established by inspection of *Figure 4.5* and a realisation that the events **dstart** and **dstop** occur immediately after the successful termination of each preceding process WAIT $\partial$.

Suppose that after the occurrence of the event **approach** at time $\tau_a$ the sequential process SLIDER takes a time $\lambda_a$ to recover before it is able to perform its next event. Then it can be seen that for the time period $[\tau_a, \tau_a+\lambda_a]$ the process SLIDER will refuse all events. After time $\tau_a+\lambda_a$ however the process may continue and either perform event **commit** or event **abort** ( see *Figure 4.5* ). Thus the refusals of any trace of SLIDER (with alphabet A) which ends in the timed event ($\tau_a$, **approach**) is given by the specification

$$\mathtt{Q} \quad \equiv \quad \mathcal{Last}(\mathtt{s}) = (\tau_a, \mathtt{approach}) \Rightarrow \quad \aleph = \aleph_a \qquad \text{Eq. 4.76}$$

$$
\begin{array}{rl}
\aleph_a = & \{[\tau_a, \tau_a+\lambda_a] \times \mathbb{P}(\mathtt{A})\} \\
& \cup \{\ [\tau_a+\lambda_a, \infty] \times \mathbb{P}(\mathtt{A}-\{\ \mathtt{commit},\ \mathtt{abort}\ \})\}
\end{array}
\qquad \text{Eq. 4.77}
$$

If CON has not performed any events then it can be seen from *Figure 4.5* that for an interval of $\partial$ served by the WAIT $\partial$ command CON is able to perform either **commit** or **allow** on its next step. After a time $\partial$ the process WAIT $\partial$ terminates and the only option open to CON is to perform the event **dstart** immediately. The refusals of this are summed up by the specification on CON

$$\mathrm{R}_1 \quad \equiv \quad \mathrm{s} = \langle\,\rangle \quad \Rightarrow \quad \aleph = \aleph_{\langle\,\rangle}$$ Eq. 4.78

$$\aleph_{\langle\,\rangle} = \{\ [0, \partial] \times \mathbb{P}(\mathrm{B} - \{\ \mathbf{commit}, \mathbf{allow}\ \})\ \}$$ Eq. 4.79

A similar piecewise analysis can be used to establish the following specifications for CON.

$$\mathrm{R}_2 \quad \equiv \quad \mathit{Last}(\mathrm{s}) = (\tau_\mathrm{b}, \mathbf{dstop}) \quad \Rightarrow \quad \aleph = \aleph_{\mathtt{dstop}}$$ Eq. 4.80

$$\begin{aligned}\aleph_{\mathtt{dstop}} = \quad & \{\ [\tau_\mathrm{b}, \tau_\mathrm{b}+\lambda_1] \times \mathbb{P}(\mathrm{B})\ \} \cup \\ & \{\ [\tau_\mathrm{b}+\lambda_1, \partial] \times \mathbb{P}(\mathrm{B} - \{\ \mathbf{commit}, \mathbf{allow}\ \})\ \}\end{aligned}$$ Eq. 4.81

$$\mathrm{R}_3 \quad \equiv \quad \mathit{Last}(\mathrm{s}) = (\tau_\mathrm{b}, \mathbf{extract}) \quad \Rightarrow \quad \aleph = \aleph_{\mathtt{extr}}$$ Eq. 4.82

$$\begin{aligned}\aleph_{\mathtt{extr}} = \quad & \{\ [\tau_\mathrm{b}, \tau_\mathrm{b}+\lambda_2] \times \mathbb{P}(\mathrm{B})\ \} \cup \\ & \{\ [\tau_\mathrm{b}+\lambda_2, \partial] \times \mathbb{P}(\mathrm{B} - \{\ \mathbf{commit}, \mathbf{allow}\ \})\ \}\end{aligned}$$ Eq. 4.83

$$\mathrm{R}_4 \quad \equiv \quad \mathit{Last}(\mathrm{s}) = (\tau_\mathrm{b}, \mathbf{dstart}) \quad \Rightarrow \quad \aleph = \aleph_{\mathtt{dstart}}$$ Eq. 4.84

$$\begin{aligned}\aleph_{\mathtt{dstart}} = \quad & \{\ [\tau_\mathrm{b}, \tau_\mathrm{b}+\lambda_3] \times \mathbb{P}(\mathrm{B})\ \} \cup \\ & \{\ [\tau_\mathrm{b}+\lambda_3, \partial] \times \mathbb{P}(\mathrm{B} - \{\ \mathbf{abort}\ \})\ \}\end{aligned}$$ Eq. 4.85

Where $\lambda_1, \lambda_2, \lambda_3$ respectively are the times which it takes CON to recover from performing the events **dstop**, **extract** and **dstart**. It is seen that

$$\begin{aligned}&\mathrm{SLIDER}\ \mathbf{sat}\ \mathrm{Q}(\mathrm{s}_1, \aleph_1) \quad \wedge \\ &\mathrm{CON}\ \mathbf{sat}\ \mathrm{R}_1\wedge\mathrm{R}_2\wedge\mathrm{R}_3\wedge\mathrm{R}_4(\mathrm{s}_2, \aleph_2)\end{aligned}$$ Eq. 4.86

By applying the rule for the alphabetized parallel operator in $\mathrm{TM}_\mathrm{F}$, it can be seen that

$$\begin{aligned}&(\mathrm{SLIDER}^\mathrm{T}\ {}_\mathrm{A}\|_\mathrm{B}\ \mathrm{CON}^\mathrm{T}) \\ &\mathbf{sat}\ \mathrm{Q}(\mathrm{s}_1\restriction\mathrm{A}, \aleph_1\cup\aleph_2) \wedge \mathrm{R}_1\wedge\mathrm{R}_2\wedge\mathrm{R}_3\wedge\mathrm{R}_4(\mathrm{s}_2\restriction\mathrm{B}, \aleph_1\cup\aleph_2)\end{aligned}$$ Eq. 4.87

Now, returning to equation 4.74, consider each of disjunct cases in turn

*Case* $(\mathit{Last}(s\upharpoonright A) = (\tau_a, \mathtt{approach}) \wedge s\upharpoonright B = < >)$

Using the fact that $(\mathtt{SLIDER}^{\mathtt{T}}\ {}_{A}\|_{B}\ \mathtt{CON}^{\mathtt{T}})$ satisfies the specifications $\mathtt{Q}(\mathtt{s}_1\upharpoonright \mathtt{A}, \aleph_1 \cup \aleph_2)$ and $\mathtt{R}_1(\mathtt{s}_2\upharpoonright \mathtt{B}, \aleph_1 \cup \aleph_2)$ it can be seen that for all $(\mathtt{s}, \aleph)$ in $(\mathtt{SLIDER}^{\mathtt{T}}\ {}_{A}\|_{B}\ \mathtt{CON}^{\mathtt{T}})'$

$$\mathit{Last}(\mathtt{s}) = (\tau_a, \mathtt{approach}) \Rightarrow \qquad \aleph = \aleph_a \cup \aleph_{< >} \qquad \text{Eq. 4.88}$$

Analysis of the sets $\aleph_a$, $\aleph_{< >}$ shows that there exists at least one time value $\tau'$ $(\tau_a < \tau' < \tau_a + \eta)$ such that $(\tau', \mathtt{commit})$ does not belong to $\aleph_a \cup \aleph_{\mathtt{dstop}}$ provided

$$(\tau_b + \lambda_1) < (\tau_a + \eta) \quad \wedge \qquad (\tau_b + \lambda_1 + \partial) > (\tau_a + \lambda_a) \qquad \text{Eq. 4.89}$$

*Case* $(\mathit{Last}(s\upharpoonright A) = (\tau_a, \mathtt{approach}) \wedge \mathit{Last}(s\upharpoonright B) = (\tau_b, \mathtt{dstop}))$

Using the fact that $(\mathtt{SLIDER}^{\mathtt{T}}\ {}_{A}\|_{B}\ \mathtt{CON}^{\mathtt{T}})$ satisfies the specifications $\mathtt{Q}(\mathtt{s}_1\upharpoonright \mathtt{A}, \aleph_1 \cup \aleph_2)$ and $\mathtt{R}_2(\mathtt{s}_2\upharpoonright \mathtt{B}, \aleph_1 \cup \aleph_2)$ it can be seen that for all $(\mathtt{s}, \aleph)$ in $(\mathtt{SLIDER}^{\mathtt{T}}\ {}_{A}\|_{B}\ \mathtt{CON}^{\mathtt{T}})$

$$\mathit{Last}(\mathtt{s}) = (\tau_a, \mathtt{approach}) \Rightarrow \qquad \aleph = \aleph_a \cup \aleph_{\mathtt{dstop}} \qquad \text{Eq. 4.90}$$

Analysis of the sets $\aleph_a$, $\aleph_{\mathtt{dstop}}$ shows that there exists at least one time value $\tau'$ $(\tau_a < \tau' < \tau_a + \eta)$ such that $(\tau', \mathtt{commit})$ does not belong to $\aleph_a \cup \aleph_{\mathtt{dstop}}$ provided

$$(\tau_b + \lambda_1) < (\tau_a + \eta) \quad \wedge \qquad (\tau_b + \lambda_1 + \partial) > (\tau_a + \lambda_a) \qquad \text{Eq. 4.91}$$

*Case* $(\mathit{Last}(s\upharpoonright A) = (\tau_a, \mathtt{approach}) \wedge \mathit{Last}(s\upharpoonright B) = (\tau_b, \mathtt{extract}))$

Using predicates $\mathtt{Q}$ and $\mathtt{R}_3$ gives that for all $(\mathtt{s}, \aleph)$ in $(\mathtt{SLIDER}^{\mathtt{T}}\ {}_{A}\|_{B}\ \mathtt{CON}^{\mathtt{T}})$

$$\mathit{Last}(\mathtt{s}) = (\tau_a, \mathtt{approach}) \Rightarrow \qquad \aleph = \aleph_a \cup \aleph_{\mathtt{extr}} \qquad \text{Eq. 4.92}$$

Analysis of the sets $\aleph_a$, $\aleph_{\mathtt{extr}}$ shows that there exists at least one time value $\tau'$ $(\tau_a < \tau' < \tau_a + \eta)$ such that $(\tau', \mathtt{commit})$ does not belong to $\aleph_a \cup \aleph_{\mathtt{extract}}$ provided

$$(\tau_b + \lambda_2) < (\tau_a + \eta) \quad \wedge \qquad (\tau_b + \lambda_2 + \partial) > (\tau_a + \lambda_a) \qquad \text{Eq. 4.93}$$

*Case* $(\mathit{Last}(s\upharpoonright A) = (\tau_a, \mathtt{approach}) \wedge \mathit{Last}(s\upharpoonright B) = (\tau_b, \mathtt{dstart}))$

Using predicates $\mathtt{Q}$ and $\mathtt{R}_4$ from equation 4. gives that for all $(\mathtt{s}, \aleph)$ in $(\mathtt{SLIDER}^{\mathtt{T}}\ {}_{A}\|_{B}\ \mathtt{CON}^{\mathtt{T}})$

$$\mathit{Last}(\mathtt{s}) = (\tau_a, \mathbf{approach}) \Rightarrow \qquad \aleph = \aleph_a \cup \aleph_{\mathtt{dstart}} \qquad \text{Eq. 4.94}$$

Analysis of the sets $\aleph_a$, $\aleph_{\mathtt{dstart}}$ shows that there exists at least one time value $\tau'$ ($\tau_a < \tau' < \tau_a+\eta$) such that ($\tau'$, **abort**) does not belong to $\aleph_a \cup \aleph_{\mathtt{dstart}}$ provided

$$(\tau_b + \lambda_3) < (\tau_a + \eta) \quad \wedge \qquad (\tau_b + \lambda_3 + \partial) > (\tau_a + \lambda_a) \qquad \text{Eq. 4.95}$$

Thus for all the cases cited in the consequence of equation 4.74 there is always some suitable time $\tau'$ ($\tau_a < \tau' < \tau_a+\eta$) such that either ($\tau'$, **commit**) or ($\tau'$, **abort**) is not a refusals at the decision point. Therefore it can be concluded that provided the time conditions expressed in equations 4.89, 4.91, 4.93 and 4.95 hold, then the timed liveness specification holds for ($\mathtt{SLIDER}^T\ {}_A\|_B\ \mathrm{CON}^T$). It then follows that the liveness specification holds for ($\mathtt{SLIDER}^T\ {}_A\|_B\ \mathrm{CON}^T$) ${}_{A\cup B}\|_C\ \mathrm{DRUM}^T$.

## CHAPTER FIVE

# CATENARY FUNCTIONS

### 5.1 Introduction.

The previous chapter outlined a verification system which is flexible, comprehensive and rigorous. Its flexibility stems from a well structured mathematical basis which defines a number of inter-related models. It is comprehensive because it has a formal specification language based on predicate calculus. It is rigorous in the sense that there is a sound and complete inference rule associated with each operator for the majority of domains.

However, one sense in which the system is lacking is its inability to present a tangible user interface. Even for the simpler models the inference rules can be complex. In turn this means that proofs for even relatively small systems are liable to rapidly become prohibitively complex.

This is tempered to some extent by the compositional nature of the proof system. Proofs can be carried out in a modular fashion and be syntax directed. However, these procedures often require both an in depth knowledge of the calculus and a high level of familiarity with the process involved.

An obvious benefit would be the ability to replace some of the complex proof obligations with automated methods. Automation provides a more user-friendly interface to formal verification. It has the potential to reduce the tedium of long proofs, which in turn minimises the occurrence of trivial mistakes, and to accelerate verification methods and instil confidence in them.

This chapter has two aims. The first is to study some of the different approaches to automatic verification and testing. The second is to lay the foundations for a new approach to testing based on the concept of an ideal test. Central to the development of this approach is the idea of Catenary functions. A Catenary function is one which permits its fixed point to be represented as the catenation of traces from a certain defined set.

## 5.2 Verification and Ideal Tests.

There are two main approaches to verifying that a system will meet its specification; testing and formal proof. To assess their relevance it is worth describing and contrasting these methods.

### 5.2.1 Testing and Exhaustive Testing.

The procedure of testing a system involves empirically comparing its actual response against its desired response [ Lanski 89 ]. Most often testing is implemented by comparing the actual output obtained from a specific input against the desired output with regard to the specification. By using specific input values testing can show that those particular values do not lead to incorrect behaviour of the system. As a result it is only possible to demonstrate the correctness of a system by testing that every input results in a correct output. That is the system can only be verified by individually investigating every possible behaviour the system may exhibit. This procedure is known as exhaustive testing.

The advantage of exhaustive testing is that it is relatively easy to implement. It adopts a direct approach which does not impose the need for an in-depth understanding of the system upon the verifier. The disadvantage is that it is a cumbersome method which can lead to long, possibly unmanageable, verification procedures.

Exhaustive checking is fully automatable, and is the basis of the concept of model checking described later in this chapter. To exhaustively test a CSP process would involve evaluating the behavioural specification for each and every process observation.

### 5.2.2 Formal Proof.

With a formal proof a system is expressed as a collection of axioms and inference rules. The system specification is formally expressed as a theorem. Correctness is established by showing that the specification theorem is a valid theorem of the formally described system. A valid theorem is one which can be deduced by applying a sequence of inference rules to established axioms. Thus verification is carried out by applying appropriate inference rules to suitable axioms to factorize a proof for the required specification. In order to assure the validity of any formal correctness proof there are three requirements which must be fulfilled

i) There is a complete axiomatization of all aspects of the system at hand; including the code, the operating systems, drivers etc.

ii) A suitable set of initial axioms and a corresponding sequence of inference rules must be found which will lead to the proof of the specification.

iii) Each deductive step in the proof can be shown to be correct.

The advantage of formal proof is that it provides an elegant and concise means of proving correctness. Additionally it possesses general rules which can cope with complex structures such as loops and recursion. It is these same structures which often make exhaustive testing infeasible because of the quantity of behaviours they produce. This gives formal proof a wider scope than exhaustive testing.

Alternatively to effectively employ formal proof often requires both a familiarity with specific techniques and an in depth understanding of the system at hand. In particular the first and second of the above criteria, axiomatising the system and choosing suitable axioms and inference rules to factorize a proof, are formidable procedures to implement.

For all the semantic models of CSP described in Chapter 4 each syntactic operator is fully axiomatized by a sound inference rule. This means that for a syntactically correct CSP process conditions (i) and (iii) will always hold. It is the role of automatic theorem provers such as those described later in the chapter to determine a suitable path to a proof.

### 5.2.3 Comparison of Testing and Proof.

When comparing exhaustive testing with formal proof it should be noted that the objective of testing is in some respects at odds to that of proving. Formal proof aims to show that a system is correct, whereas testing tries to show that certain behaviours are incorrect.

To some extent the advantages and disadvantages of each approach overlap one another. Testing is automatable and has an approachable user interface in that it requires little comprehension of the system on the part of the verifier. But it offers the prospect of a long, sometimes infeasible, verification procedure. Alternatively, formal proof provides a concise and elegant method of verification, but it is difficult to automate and requires more expertise with the formal method and the system concerned by the verifier.

This overlap indicates that there is scope for a compromise between both of these methods. It may be possible to combine the most desirable properties of each and produce a verification method which has a tangible user interface, a flexible scope and is realistic to automate.

This concept was addressed in a seminal paper by Goodenough and Gerhart [ Goodenough 75 ]. They recognize the benefits of testing but also acknowledge the impracticality of exhaustive testing. The paper identifies the essence of testing as being to establish a base proposition for an inductive proof. It sets out to develop a theory of testing based on the following concept of an ideal test.

**Definition 5.1:** (From [ Goodenough 75 ] ) Let $\mathbf{D}$ be the set of all possible inputs available to a system, and let $\mathbf{F}(\mathbf{d})$ be the output of the system corresponding to an input $\mathbf{d}$. The predicate OUT($\mathbf{d}$, $\mathbf{F}(\mathbf{d})$) is defined to be a statement which is true if and only if $\mathbf{F}(\mathbf{d})$ is an acceptable output with respect to the specification of the system.

If $\mathbf{T}$ is a subset of $\mathbf{D}$, then $\mathbf{T}$ is said to be an Ideal Test if and only if

$$(\forall\, \mathbf{t} \in \mathbf{T} \bullet \mathrm{OUT}(\mathbf{t}, \mathbf{F}(\mathbf{t}))) \quad \Rightarrow \quad (\forall\, \mathbf{d} \in \mathbf{D} \bullet \mathrm{OUT}(\mathbf{d}, \mathbf{F}(\mathbf{d}))) \quad \text{Eq. 5.1}$$

■

## 5.3 Automated Verification Techniques.

Automated verification techniques fall broadly into two categories. The first approach concentrates on demonstrating specifications by a method called model checking [ Ramsay 88 ]. With this, labelled transition systems are generated and possible sequences of states or behaviours are explored. Analysis is carried out usually by means of a decision procedure. The advantages of model checking are that it is simple to implement and fully automatic. However, it can only practically be used for reasoning about a finite number of process behaviours, with a small number of branches at each stage. It is susceptible to the problem of state explosion in complex systems.

Implementing model checking for a CSP process involves generating the set of process observations and then comparing them against the behavioural specification. Kourie [ Kourie 87 ] has developed a generator for CSP, written in Prolog, which generates the observations of a process in the traces domain $M_T$. He goes on to indicate that with modification the Prolog code has the ability to pose questions about these traces and thus establish certain properties about the process concerned.

A more rigorous approach to developing a extensive model checker is achieved by Concurrency Workbench [ Cleaveland 90 ]. Developed by researchers at the Laboratory for the Foundations of Computer Science in Edinburgh during the middle to late 80's, this tool is based on Milner's CCS [ Milner 80 ]. Specifications are initially expressed in the user oriented interface logic which is a modal logic known as the mu-calculus (μ-calculus). These specifications are then translated into a logic called the system logic, which is a machine oriented form, and compared against a generated semantic model.

The second approach to automated verification is that of theorem provers [ Ramsay 88 ]. These represent the specification of a system as a set of theorems. The system itself is defined by a collection of axioms and inference rules. A theorem must either be postulated as an axiom or deduced from existing theorems and axioms via inference rules. Thus specifications are established by an automatic process of logical deduction. The advantage of such an approach is that in many instances is affords a more elegant proof than model checkers, and it is not restricted to systems with finite behaviours. It is, however, much more complex to implement. Additionally, formal logics of the order of predicate logic or higher are not fully decidable [ Galton 90 ], and therefore for these theorem proving can never be a fully automatic procedure.

The concepts and strategies involved in theorem proving encompass a wide area in the field of artificial intelligence, and for further discussion the reader is directed to [ Ramsay 88 ]. For the purposes of this text two particular examples of theorem provers are considered because of their application to CSP. The first is the HOL theorem prover, developed by M.J. Gordon at the University of Edinburgh [ Gordon 88, Inverardi 91 ]. It consists of two elements, the Higher Order Logic (HOL), which is an extension of predicate calculus, and a general purpose programming language called ML [ Wilkstrom 87 ] which is used to mechanize the logic. The HOL theorem prover has been used to mechanize both the traces model [ Camilleri 90 ] and the failures divergence model [ Camilleri 91 ] of CSP. It achieves this by adopting a suitable set-based interpretation for each model and then representing the semantic definition of each operator as an inference rule expressed as a program in ML. These programs are then applied to supplied axioms to generate new theories.

The B-Tool [ Abrial, J.R. ] was developed by J.R. Abrial in cooperation with the Programming Research Group at Oxford University and British Petroleum's Research Centre at Sunbury. It specifies processes in a simple language which is an extension of Dijkstra's language of guarded commands [ Dijkstra 76 ]. The tool itself is a Pascal program that acts as a proof assistant by applying inference rules which are supplied to it as theories. By supplying the appropriate inference rules the tool has been used to support alternative formalisms including CSP [ Davies 87 ] and Z.

## 5.4 Motivations.

In the light of the concept of model checking, the following observations were made about the arbor drum example cited in Chapter 4:

- Each of the sequential processes `SLIDER`, `CON` and `DRUM` essentially repeated one of a fixed set of behaviours. That is, on each recursive call the processes perform a trace from this fixed set and then make another recursive call. Here the recursion was seen to be equivalent to a loop. For a specific illustration of this see Example 5.1.

- There was no inherent bound to the behaviours of each of the sequential processes `SLIDER`, `CON` and `DRUM`. Theoretically they would never terminate.

- There existed a compositional proof system in which to prove particular system properties for each of `SLIDER`, `CON` and `DRUM`. In the example cited the inference rule associated with the alphabetized parallel operator was successfully used to derive a correctness proof.

The first of these observations suggests that for at least some CSP functions there is an explicit link between recursion and catenation. That is that the recursive definition of a function may be replaced by a definition based on the successive catenation of traces from a particular set. This in turn suggested that such a link had the potential to be exploited in the verification of a process behaviour, particularly with respect to model checking.

The second property, however, indicates that, in its basic form, model checking would be unsuitable. This is because a theoretically non-terminating process would generate a prohibitively large amount of behaviours. This makes exhaustive testing of the process behaviours infeasible. If it were to be used, therefore, it would be necessary to appropriately modify the principles of model checking.

The third observation indicates that even if it were only possible to establish a verification system for a limited set of functions which behaved in a suitable manner, CSP possesses a compositional proof system with the ability to fill in any gaps in the proof which this would entail.

The chapter now concerns itself with the problem of finding a subclass of processes which repeat the same behaviours on each recursive call. In order to investigate the nature of such processes the text introduces a particular category of CSP functions, namely the Catenary functions. The following example gives an instance of such a function

**Example 5.1:** One particular trait inherent in the control logic of the slider and drum mechanism of the previous chapter was that recursion led to the process repeatedly performing a trace taken from a limited set. Consider the untimed process SLIDER which was defined by the function

$$\begin{aligned} F_{um}(X) = \quad & \mathbf{approach} \rightarrow ((\mathbf{commit} \rightarrow \mathbf{enter} \rightarrow \mathbf{slow} \rightarrow \\ & \mathbf{perform} \rightarrow \mathbf{extract} \rightarrow X) \\ & \square\ (\mathbf{abort} \rightarrow \mathbf{decelerate} \rightarrow \mathbf{allow} \rightarrow \\ & \mathbf{accelerate} \rightarrow \mathbf{enter} \rightarrow \mathbf{slow} \rightarrow \\ & \mathbf{perform} \rightarrow \mathbf{extract} \rightarrow X)) \end{aligned} \qquad \text{Eq. 5.2}$$

Furthermore, consider the two following traces,

$$\begin{aligned} s_1 \quad = \quad & < \mathbf{approach}, \mathbf{commit}, \mathbf{enter}, \mathbf{slow}, \\ & \mathbf{perform}, \mathbf{extract} > \end{aligned} \qquad \text{Eq. 5.3}$$

$$\begin{aligned} s_2 \quad = \quad & < \mathbf{approach}, \mathbf{abort}, \mathbf{decelerate}, \mathbf{allow}, \\ & \mathbf{accelerate}, \mathbf{enter}, \mathbf{slow}, \\ & \mathbf{perform}, \mathbf{extract} > \end{aligned} \qquad \text{Eq. 5.4}$$

When placed outside the environment of the slider/drum mechanism SLIDER may either approach and commit, in which case it performs trace $s_1$, or it may approach and abort, in which case it will perform trace $s_2$. After each of these options SLIDER then recurs and repeats the same behaviour. The result of applying the function $F_{um}$ to some arbitrary process P is that $F_{um}(P)$ will either perform $s_1$ and then behave as P, or perform $s_2$ and behave as P. This is expressed in terms of traces as

$$\begin{aligned} \mathit{Traces}(F_{um}(P)) \quad & = \mathit{Traces}(F_{um}(STOP)) \cup \\ & \{\ s \hat{}\ t \mid (s = s_1 \vee s = s_2) \wedge t \in \mathit{Traces}(P)\ \} \end{aligned} \qquad \text{Eq. 5.5}$$

It is noted that the behaviour of the operand P is not fundamentally altered by the process $F_{um}$ - at some stage in its execution $F_{um}(P)$ will behave like P. Specifically $F_{um}(P)$ behaves like P after engaging in either $s_1$ or $s_2$ . That is

$$\mathtt{F(P)/s_1} = \mathtt{F(P)/s_2} = \mathtt{P} \qquad \text{Eq. 5.6}$$

If P is a process which does not deadlock on its first step then it is possible to derive the equivalence

$$\mathit{Traces}(\mathtt{F_{um}(P)}) = \mathit{Traces}(\mathtt{F_{um}(SKIP)\ ;\ P}) \qquad \text{Eq. 5.7}$$

Because SLIDER is the fixed point of $\mathtt{F_{um}}$ - the limit of repeatedly applying the function to some process - the above equivalence can be used to show that

$$\mathit{Traces}(\mu\mathtt{P.F_{um}(P)}) = \mathit{Traces}(\mathtt{F_{um}(SKIP)\ ;\ F_{um}(SKIP)\ ;\ ..}) \qquad \text{Eq. 5.8}$$

■

Equation 5.7 shows that $\mathtt{F_{um}(P)}$ can be teased into two parts, first the behaviour induced by the functional definition, and then the behaviour exhibited by the operand. Equation 5.8 shows how this leads to recursion being represented as the sequential composition of an infinite number of finite processes.

## 5.5 Relating Recursion to Sequential Composition and Catenation.

Before discussing the link between recursion and catenation it is convenient to introduce here the concept and notation of Set Catenation. This extends the concept of catenation from that given in Chapter 3 to sets of traces and serves to simplify equations which follow in the text. It has the following definition

**Definition 5.2:** For two sets of traces A and B the Set Catenation of A and B is defined as A ^ B, where

$$A \wedge B = \{ \mathtt{s\hat{}t} \mid \mathtt{s} \in A \wedge \mathtt{t} \in B \} \qquad \text{Eq. 5.9}$$

Set Catenation is associative, distributes over set union and intersection and has the empty trace, { <> }, as its unit. The follows rules apply for all sets of traces A, B, C and for any set of events D.

$$\text{i)} \qquad A^{\wedge}\{ <> \} \quad = \quad \{ <> \}^{\wedge} A = A \qquad \text{Eq. 5.10}$$

ii) $A^\wedge(B^\wedge C) \quad = \quad (A^\wedge B)^\wedge C$ Eq. 5.11

iii) $A^\wedge(B \cup C) \quad = \quad (A^\wedge B) \cup (A^\wedge C)$

& $(A \cup B)^\wedge C \quad = \quad (A^\wedge C) \cup (B^\wedge C)$ Eq. 5.12

iv) $A^\wedge(B \cap C) \quad = \quad (A^\wedge B) \cap (A^\wedge C)$

& $(A \cap B)^\wedge C \quad = \quad (A^\wedge C) \cap (B^\wedge C)$ Eq. 5.13

v) $A^\wedge B - A^\wedge C \quad = \quad A^\wedge(B - C)$

& $A^\wedge C - B^\wedge C \quad = \quad (A - B)^\wedge C$ Eq. 5.14

vi) $A\upharpoonright \mathtt{D}^\wedge B\upharpoonright \mathtt{D} \quad = \quad (A^\wedge B)\upharpoonright \mathtt{D}$ Eq. 5.15

vii) $\{\}^\wedge A \quad = \quad \{\} \quad = \quad A^\wedge \{\}$ Eq. 5.16

These results are proved in Appendix B ■

Example 5.1 illustrates that there is at least one function which allows recursion to be expressed in terms of sequential composition and thus catenation. A natural response to this is to address the problem of identifying a broad general class of functions which exhibit similar properties. In order to achieve this it is necessary to specify the qualities which such a class of functions must possess. For a function F and an arbitrary process P these are summed up by the following points.

- Primarily a function must not fundamentally alter the behaviour of its operand. That is for a function F and operand P there must be some stage in the execution of F(P) during which it behaves like P.

- Additionally it must be possible to divide the behaviour of F(P) into two distinct parts. The behaviour induced by the functional definition of F, and the behaviour exhibited by the operand P.

These points give rise to two main problems. First the behaviour induced by the functional definition must be identified and expressed in a convenient fashion. Secondly it is necessary to precisely define the nature of the link between recursion and catenation, and to place it on a mathematical footing which permits analysis.

To address the first of these problems, consider the following theorem

**Theorem 5.1:** Let F be a CSP function. For every process P the process F(P) contains the behaviours of the process F(STOP). That is

$$\forall\, \mathtt{P} \in \mathrm{CSP} \quad \bullet \qquad \mathit{Traces}(\mathtt{F(P)}) \supseteq \mathit{Traces}(\mathtt{F(STOP)}) \qquad \text{Eq. 5.17}$$

Furthermore, if F is a contraction on the metric space ($M_T$, $d_T$), then for every process P the process F(P) will exhibit the same behaviour as F(STOP) for at least its first step. That is

$$\forall\, \mathtt{s} \in \mathit{Traces}(\mathtt{F(P)}) \bullet < \mathit{First}(\mathtt{s}) > \in \mathit{Traces}(\mathtt{F(STOP)}) \qquad \text{Eq. 5.18}$$

■

*Proof*

Appendix A states that for the partial ordering ⊒ on the set of CSP processes the lower bound is STOP and every CSP function is monotonic. Therefore

$$\mathtt{P} \quad \sqsupseteq \quad \mathtt{STOP} \qquad \text{, } \mathit{lower\ bound\ of} \sqsupseteq \qquad \text{Eq. 5.19}$$

$$\therefore \quad \mathtt{F(P)} \quad \sqsupseteq \quad \mathtt{F(STOP)} \qquad \text{, } \mathit{monotonicity\ of}\ \mathtt{F} \qquad \text{Eq. 5.20}$$

$$\therefore \quad \mathit{Traces}(\mathtt{F(P)}) \quad \supseteq \quad \mathit{Traces}(\mathtt{F(STOP)})\text{, } \mathit{definition\ of} \sqsupseteq \qquad \text{Eq. 5.21}$$

If F is a contraction, then for $n_1$, $n_2 \in \mathbb{N}$

$$d_T(\,\mathtt{F(P)}, \mathtt{STOP}\,) = \frac{1}{n_1} \qquad > \qquad \frac{1}{n_2} = d_T(\mathtt{F(P)}, \mathtt{F(STOP)}) \qquad \text{Eq. 5.22}$$

Since $\mathtt{STOP}\upharpoonright 0 = \mathtt{STOP}$ it can be deduced that $n_1 = 0$. As a consequence $n_2$ must be greater then or equal to 1. Hence from the definition of $d_T$

$$\mu\mathtt{P.F(P)}\upharpoonright 1 = \mathtt{F(STOP)}\upharpoonright 1 \qquad \text{Eq. 5.23}$$

$$\therefore \quad \mathtt{s} \in \mathit{Traces}(\mathtt{F(P)}) \Rightarrow < \mathit{First}(\mathtt{s}) > \in \mathit{Traces}(\mathtt{F(STOP)}) \qquad \text{Eq. 5.24}$$

□

What Theorem 5.1 hints at (but does not prove) is that a process of the form F(P), where F is a contraction and P an arbitrary process, will always initially behave in the same way as F(STOP). This in turn suggests that, because F(STOP) is independent of the operand P, this behaviour is a result of the functional definition of F rather than any behaviour of P. Additionally, because the text is only concerned with functions which do not fundamentally alter the behaviour of P, it is reasonable to suppose that the behaviour of F(P) can be informally summed up by the statement

$$\mathrm{F(STOP)}\ \mathit{then}\ \mathrm{P} \qquad \text{Eq. 5.25}$$

In the above, *then* represents some indefinite operation which sequentially joins F(STOP) to P. It is inappropriate to replace *then* with the sequential operator ";", because this makes demands about the successful termination of F(STOP) which it may not be prepared to meet. Instead *then* anticipates that, for F(P), there are certain traces of F(STOP) that can be prefixed to those of P which result in traces of F(P). It therefore identifies such traces and accordingly prefixes only these to traces of P.

Thus the process F(STOP) *then* P is one which behaves like F(STOP), and subsequently, after certain traces of F(STOP), behaves like P. This can be summed up formally as

$$\mathit{Traces}(\mathrm{F(STOP)}\ \mathit{then}\ \mathrm{P}) \equiv \mathit{Traces}(\mathrm{F(STOP)}) \cup ( \mathcal{D}\,^\wedge\ \mathit{Traces}(\mathrm{P})) \qquad \text{Eq. 5.26}$$

where $\mathcal{D}$ is some subset of F(STOP). That is $\mathcal{D}$ is the set of traces from F(STOP) which can be prefixed to traces in P to yield traces in F(P).

## 5.6 Dedicated Events and Catenary Functions.

The concept of a dedicated event is introduced here as a notational tool. Its purpose is to allow the subset $\mathcal{D}$ of F(STOP), referred to above, to be identified in terms of the functional definition of F alone.

**Definition 5.3:** For a function F the dedicated event * is defined to be some event which is not directly or indirectly included in the definition of F. That is it is totally unconnected with F in that it is not in the alphabet of F(STOP), F does not hide * from the environment and F does not involve a change of symbol concerning *. Furthermore DED is defined to be the process which starts, performs the dedicated event * and does nothing more. ■

Informally, the process DED is intended to act as a substitute for the process P in the expression F(P). If a trace s^< * > belongs to F(DED) this implies that a trace s^t belongs to the process F(P), where t ∈ *Traces*(P).

This line of reasoning permits the generation of a set of traces which may be suffixed to *Traces*(P) to result in *Traces*(F(P)). Call this set $\mathcal{D}_{\mathcal{F}}$[†] and define it by

**Definition 5.4:** For a function F the set $\mathcal{D}_{\mathcal{F}}$ is defined by the expression

$$\mathcal{D}_{\mathcal{F}} = (\mathit{Traces}(\mathrm{F(DED)}) - \mathit{Traces}(\mathrm{F(STOP)}))\upharpoonright \{\Sigma - *\} \qquad \text{Eq. 5.27}$$

■

The above definition of the set $\mathcal{D}_{\mathcal{F}}$ now permits the formal concept of a Catenary function to be introduced here.

**Definition 5.5:** A function F is said to be Catenary if for all processes P

$$\mathit{Traces}(\mathrm{F(P)}) = \mathit{Traces}(\mathrm{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}}\ \hat{}\ \mathit{Traces}(\mathrm{P})) \qquad \text{Eq. 5.28}$$

■

Informally a function F is Catenary if, when it is applied to a process P, it results in a process which behaves first as F(STOP) and then as P.

## 5.7 Instances of Catenary Functions.

This section considers each of the individual syntactic operators of CSP and determines in what way they can be used to define Catenary functions.

**Theorem 5.2:** The identity function

$$\mathrm{F(P)} = \mathrm{P} \qquad \text{Eq. 5.29}$$

is a Catenary function ■

*Proof*

F is the identity function, therefore F(STOP) = STOP. Furthermore,

[†] Note: The set description $\mathcal{D}_{\mathcal{F}}$ should not be confused with the metric $d_F$. They are unconnected.

$$\mathcal{D}_{\mathcal{F}} = (\mathit{Traces}(\mathtt{DED}) - \mathit{Traces}(\mathtt{STOP})) \upharpoonright \{\Sigma - *\} = \{ <> \} \qquad \text{Eq. 5.30}$$

$$\therefore \quad \mathit{Traces}(\mathtt{F}(\mathtt{STOP})) \cup ( \mathcal{D}_{\mathcal{F}} \,\hat{}\; \mathit{Traces}(\mathtt{P})) \qquad \text{Eq. 5.31}$$

$$= \{ <> \} \cup ( \{ <> \} \,\hat{}\; \mathit{Traces}(\mathtt{P})) \qquad = \mathit{Traces}(\mathtt{P}) = \mathit{Traces}(\mathtt{F}(\mathtt{P})) \qquad \text{Eq. 5.32}$$

Therefore the identity function is a Catenary Function. □

**Theorem 5.3:** If F is a function which is independent of the operand P, then F is a Catenary function. ■

*Proof*

If F is independent of the operand P then F(P) will be the same for all values of P. That is there is some set of traces A, say, such that.

$$\forall\, \mathtt{P} \in \mathrm{CSP} \bullet \mathit{Traces}(\mathtt{F}(\mathtt{P})) = \mathrm{A} \qquad \text{Eq. 5.33}$$

Consider then the expression

$$\mathit{Traces}(\mathtt{F}(\mathtt{STOP})) \cup (\mathcal{D}_{\mathcal{F}} \,\hat{}\; \mathit{Traces}(\mathtt{P})) \qquad \text{Eq. 5.34}$$

$$= \mathrm{A} \cup (( \mathrm{A} - \mathrm{A} ) \upharpoonright \{\Sigma - *\} \,\hat{}\; \mathit{Traces}(\mathtt{P})) \qquad \text{Eq. 5.35}$$

$$= \mathrm{A} \cup ( \{\} \,\hat{}\; \mathit{Traces}(\mathtt{P})) \qquad = \mathrm{A} \qquad = \mathit{Traces}(\mathtt{F}(\mathtt{P})) \qquad \text{Eq. 5.36}$$

Therefore F is a Catenary function. □

The immediate corollary of Theorem 5.3 is that F(P) = SKIP and F(P) = STOP are Catenary functions.

**Theorem 5.4:** The prefix operator is a Catenary Function. That is for some event **a** the function F defined by

$$\mathtt{F}(\mathtt{P}) = \mathbf{a} \rightarrow \mathtt{P} \qquad \text{Eq. 5.37}$$

is a Catenary Function. ■

*Proof*

From the definition of the prefix operator,

$$Traces\ (\mathrm{F(P)}) = \{\ \langle\rangle\ ,\ \langle\ \mathbf{a}\ \rangle\ \} \cup (\{\ \langle\ \mathbf{a}\ \rangle\ \}\ \hat{}\ Traces(\mathrm{P})) \qquad \text{Eq. 5.38}$$

Consider the expression,

$$Traces(\mathrm{F(STOP)}) \cup$$
$$\big((Traces(\mathrm{F(DED)}) - Traces(\mathrm{F(STOP)})) \restriction \{\Sigma - *\}\ \hat{}\ Traces(\mathrm{P})\big) \qquad \text{Eq. 5.39}$$

It can be seen that,

$$Traces(\mathrm{F(STOP)}) = \{\ \langle\rangle\ ,\ \langle\ \mathbf{a}\ \rangle\ \} \qquad \text{Eq. 5.40}$$

$$Traces(\mathrm{F(DED)}) = \{\ \langle\rangle\ ,\ \langle\ \mathbf{a}\ \rangle\ ,\ \langle\ \mathbf{a},\ *\ \rangle\ \} \qquad \text{Eq. 5.41}$$

Therefore equation 5.39 becomes,

$$\{\ \langle\rangle\ ,\ \langle\ \mathbf{a}\ \rangle\ \} \cup$$
$$((\{\ \langle\rangle\ ,\ \langle\ \mathbf{a}\ \rangle\ ,\ \langle\ \mathbf{a},\ *\ \rangle\ \} - \{\ \langle\rangle\ ,\ \langle\ \mathbf{a}\ \rangle\ \}) \restriction \{\Sigma - *\}\ \hat{}\ Traces(\mathrm{P})) \qquad \text{Eq. 5.42}$$

$$= \{\ \langle\rangle\ ,\ \langle\ \mathbf{a}\ \rangle\ \} \cup (\{\ \langle\ \mathbf{a}\ \rangle\ \}\ \hat{}\ Traces(\mathrm{P})) \quad = Traces(\mathrm{F(P)}) \qquad \text{Eq. 5.43}$$

Thus the prefix operator is a Catenary function. □

**Theorem 5.5:** Both Deterministic Choice and Nondeterministic Choice are Catenary functions. That is if a function F is defined by either expression

$$\mathrm{F(P)} = \mathrm{G}_1(\mathrm{P}) \,\square\, \mathrm{G}_2(\mathrm{P}) \qquad \text{Eq. 5.44}$$

$$\mathrm{F(P)} = \mathrm{G}_1(\mathrm{P}) \sqcap \mathrm{G}_2(\mathrm{P}) \qquad \text{Eq. 5.45}$$

where $\mathrm{G}_1$ and $\mathrm{G}_2$ are Catenary functions, then F is a Catenary function. ■

*Proof*

To prove this result first requires three preliminary results

**Lemma A:** If G is a Catenary function then

$$\mathrm{s} \in (\mathit{Traces}(\mathrm{G(DED)}) - \mathit{Traces}(\mathrm{G(STOP)})) \Rightarrow \mathbf{*}\ \underline{\mathbf{in}}\ \mathrm{s} \qquad \text{Eq. 5.46}$$

■

*Proof of Lemma A*

From the definition of a Catenary function there is a set of traces such that

$$\mathit{Traces}(\mathrm{G(DED)}) = \mathit{Traces}(\mathrm{G(STOP)}) \cup (\mathcal{D}_{\mathcal{G}} \wedge \mathit{Traces}(\mathrm{DED})) \qquad \text{Eq. 5.47}$$

$$\therefore \quad \mathit{Traces}(\mathrm{G(DED)}) - \mathit{Traces}(\mathrm{G(STOP)}) = \mathcal{D}_{\mathcal{G}} \wedge \{ < * > \} \qquad \text{Eq. 5.48}$$

$$\therefore \quad \mathrm{s} \in (\mathit{Traces}(\mathrm{G(DED)}) - \mathit{Traces}(\mathrm{G(STOP)})) \qquad \text{Eq. 5.49}$$

$$\Rightarrow \quad \mathit{Last}(\mathrm{s}) = * \quad \Rightarrow \quad *\ \underline{\mathbf{in}}\ \mathrm{s} \qquad \text{Eq. 5.50}$$

□

**Lemma B:** If $\mathrm{G}_1$ and $\mathrm{G}_2$ are CSP functions and $\mathrm{G}_1$ is a Catenary function then

$$(\mathit{Traces}(\mathrm{G}_1(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}_1(\mathrm{STOP}))) \cap \mathrm{G}_2(\mathrm{STOP}) = \{\} \qquad \text{Eq. 5.51}$$

■

*Proof of Lemma B*

From the definition of DED and the Lemma B

$$\begin{array}{l}(\mathrm{s} \in \mathrm{G}_2(\mathrm{STOP}) \quad \Rightarrow \quad \neg(*\ \underline{\mathbf{in}}\ \mathrm{s})) \\ (\wedge\ \mathrm{s} \in (\mathit{Traces}(\mathrm{G}_1(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}_1(\mathrm{STOP}))) \ \Rightarrow\ *\ \underline{\mathbf{in}}\ \mathrm{s})\end{array} \qquad \text{Eq. 5.52}$$

$$\therefore \quad (\mathit{Traces}(\mathrm{G}_1(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}_1(\mathrm{STOP}))) \cap \mathit{Traces}(\mathrm{G}_2(\mathrm{STOP})) = \{\} \qquad \text{Eq. 5.53}$$

□

**Lemma C:** If A, B, C and D are sets then

$$(\mathrm{A} - \mathrm{B}) \cap \mathrm{D} = \{\} \quad \wedge \quad (\mathrm{C} - \mathrm{D}) \cap \mathrm{B} = \{\}$$

$$\Rightarrow \quad (\mathrm{A} - \mathrm{B}) \cup (\mathrm{C} - \mathrm{D}) = (\mathrm{A} \cup \mathrm{C}) - (\mathrm{B} \cup \mathrm{D}) \qquad \text{Eq. 5.54}$$

■

*Proof of Lemma C*

This result is proved in Appendix B. □

Continuing with the main proof, by definition for processes $\mathtt{X}_1$ and $\mathtt{X}_2$

$$\mathit{Traces}(\mathtt{X}_1 \sqcap \mathtt{X}_2) = \mathit{Traces}(\mathtt{X}_1 \Box \mathtt{X}_2) = \mathit{Traces}(\mathtt{X}_1) \cup \mathit{Traces}(\mathtt{X}_2) \tag*{Eq. 5.55}$$

Expanding the definition of a Catenary function, where $\mathtt{F(P)} = \mathtt{G}_1\mathtt{(P)} \Box \mathtt{G}_2\mathtt{(P)}$

$$\mathit{Traces}(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} \,{}^\wedge\, \mathit{Traces}(\mathtt{P})) \tag*{Eq. 5.56}$$

$$= \mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)} \Box \mathtt{G}_2\mathtt{(STOP)}) \cup ((\mathit{Traces}(\mathtt{G}_1\mathtt{(DED)} \Box \mathtt{G}_2\mathtt{(DED)}) \\ - \mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)} \Box \mathtt{G}_2\mathtt{(STOP)})) \upharpoonright \{\Sigma - *\} \,{}^\wedge\, \mathit{Traces}(\mathtt{P})) \tag*{Eq. 5.57}$$

$$= \mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)}) \cup \mathit{Traces}(\mathtt{G}_2\mathtt{(STOP)}) \cup (((\ \mathit{Traces}(\mathtt{G}_1\mathtt{(DED)}) \cup \\ \mathit{Traces}(\mathtt{G}_2\mathtt{(DED)}))) - (\mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)}) \cup \mathit{Traces}(\mathtt{G}_2\mathtt{(STOP)})))) \upharpoonright \{\Sigma - *\} \\ {}^\wedge\, \mathit{Traces}(\mathtt{P})) \tag*{Eq. 5.58}$$

Using Lemmas B and C,

$$= \mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)}) \cup \mathit{Traces}(\mathtt{G}_2\mathtt{(STOP)}) \cup (((\mathit{Traces}(\mathtt{G}_1\mathtt{(DED)}) - \\ \mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)})) \cup (\mathit{Traces}(\mathtt{G}_2\mathtt{(DED)}) - \\ \mathit{Traces}(\mathtt{G}_2\mathtt{(STOP)}))) \upharpoonright \{\Sigma - *\} \,{}^\wedge\, \mathit{Traces}(\mathtt{P})) \tag*{Eq. 5.59}$$

Expanding this expression using the rules given in Definition 5.2

$$= (\mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)}) \cup ((\mathit{Traces}(\mathtt{G}_1\mathtt{(DED)}) - \mathit{Traces}(\mathtt{G}_1\mathtt{(STOP)})) \upharpoonright \{\Sigma - *\} \\ {}^\wedge\, \mathit{Traces}(\mathtt{P}))) \cup \\ (\mathit{Traces}(\mathtt{G}_2\mathtt{(STOP)}) \cup ((\mathit{Traces}(\mathtt{G}_2\mathtt{(DED)}) - \mathit{Traces}(\mathtt{G}_2\mathtt{(STOP)})) \upharpoonright \{\Sigma - *\} \\ {}^\wedge\, \mathit{Traces}(\mathtt{P}))) \tag*{Eq. 5.60}$$

From the definitions of the Catenary Functions $\mathtt{G}_1$ and $\mathtt{G}_2$ this reduces to

$$= \mathit{Traces}(\mathtt{G}_1\mathtt{(P)}) \cup \mathit{Traces}(\mathtt{G}_2\mathtt{(P)}) \tag*{Eq. 5.61}$$

$$= \mathit{Traces}(\mathtt{G}_1\mathtt{(P)} \Box \mathtt{G}_2\mathtt{(P)}) \qquad = \mathit{Traces}(\mathtt{F(P)}) \tag*{Eq. 5.62}$$

Therefore the function $\mathtt{F}$ is a Catenary function.

The proof for the nondeterministic operator follows in the same way, because it has the same semantic interpretation in the traces model. □

**Theorem 5.6:** For a Catenary function G and a process X the function F defined by

$$\mathrm{F}(\mathrm{P}) = \mathrm{X} \; ; \mathrm{G}(\mathrm{P}), \qquad \textit{where X is independent of P} \qquad \text{Eq. 5.63}$$

is a Catenary function. ■

*Proof*

Let A be the set of all traces in *Traces*(X) which end with the termination event. That is

$$\mathrm{A} = \{\, \mathrm{t} \mid \mathit{Last}(\mathrm{t}) = \checkmark \wedge \mathrm{t} \in \mathit{Traces}(\mathrm{X}) \,\} \qquad \text{Eq. 5.64}$$

Then by the definition of sequential composition [ Hoare 85 ]

$$\mathit{Traces}(\mathrm{X} \; ; \mathrm{G}(\mathrm{P})) = (\mathit{Traces}(\mathrm{X}) - \mathrm{A}) \cup (\mathrm{A} \,\hat{}\, \mathit{Traces}(\mathrm{G}(\mathrm{P}))) \qquad \text{Eq. 5.65}$$

$$= (\mathit{Traces}(\mathrm{X}) - \mathrm{A}) \cup (\mathrm{A} \,\hat{}\, (\mathit{Traces}(\mathrm{G}(\mathrm{STOP})) \cup ((\mathit{Traces}(\mathrm{G}(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}(\mathrm{STOP})))\restriction\{\Sigma - *\} \,\hat{}\, \mathit{Traces}(\mathrm{P})))) \qquad \text{Eq. 5.66}$$

$$= (\mathit{Traces}(\mathrm{X}) - \mathrm{A}) \cup (\mathrm{A} \,\hat{}\, \mathit{Traces}(\mathrm{G}(\mathrm{STOP}))) \cup (\mathrm{A} \,\hat{}\, ((\mathit{Traces}(\mathrm{G}(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}(\mathrm{STOP})))\restriction\{\Sigma - *\} \,\hat{}\, \mathit{Traces}(\mathrm{P}))) \qquad \text{Eq. 5.67}$$

$$= (\mathit{Traces}(\mathrm{X}) - \mathrm{A}) \cup (\mathrm{A} \,\hat{}\, \mathit{Traces}(\mathrm{G}(\mathrm{STOP}))) \cup ((((\mathit{Traces}(\mathrm{X}) - \mathrm{A}) \cup (\mathrm{A} \,\hat{}\, \mathit{Traces}(\mathrm{G}(\mathrm{DED})))) - ((\mathit{Traces}(\mathrm{X}) - \mathrm{A}) \cup (\mathrm{A} \,\hat{}\, \mathit{Traces}(\mathrm{G}(\mathrm{STOP})))))\restriction\{\Sigma - *\} \,\hat{}\, \mathit{Traces}(\mathrm{P})) \qquad \text{Eq. 5.68}$$

$$= \mathit{Traces}(\mathrm{F}(\mathrm{STOP})) \cup \big((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP})))\restriction\{\Sigma - *\} \,\hat{}\, \mathit{Traces}(\mathrm{P})\big) \qquad \text{Eq. 5.69}$$

Which satisfies the definition for a Catenary function. □

**Theorem 5.7:** Interleaving is not a Catenary Function ■

*Proof*

By counter example. Consider the function F and process P defined

$$\mathtt{F(P)} = \mathbf{a} \rightarrow \mathtt{STOP} \;|||\; \mathtt{P}, \qquad \mathtt{P} = \mathbf{b} \rightarrow \mathbf{c} \rightarrow \mathtt{STOP} \qquad \text{Eq. 5.70}$$

$$\begin{aligned} \therefore \quad & Traces(\mathtt{F(P)}) = \{\, ..., \langle \mathbf{b}, \mathbf{a}, \mathbf{c} \rangle ,...\, \} \\ & \wedge\, Traces(\mathtt{F(STOP)}) = \{\, \langle\rangle , \langle \mathbf{a} \rangle \,\} \\ & \wedge\, (Traces(\mathtt{F(DED)}) - Traces(\mathtt{F(STOP)})) \upharpoonright \{\Sigma - *\} = \{\, \langle\rangle , \langle \mathbf{a} \rangle \,\} \qquad \text{Eq. 5.71} \end{aligned}$$

The Catenary definition can be expanded

$$Traces(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} {}^\wedge\, Traces(\mathtt{P})) \qquad \text{Eq. 5.72}$$

$$= \{\, \langle\rangle , \langle \mathbf{a} \rangle \,\} \cup (\{\, \langle\rangle , \langle \mathbf{a} \rangle \,\} {}^\wedge \{\, \langle\rangle , \langle \mathbf{b} \rangle , \langle \mathbf{b}, \mathbf{c} \rangle \,\}) \qquad \text{Eq. 5.73}$$

$$\begin{aligned} \therefore \quad & \langle \mathbf{b}, \mathbf{a}, \mathbf{c} \rangle \notin (Traces(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} {}^\wedge\, Traces(\mathtt{P}))) \\ & \Rightarrow\, Traces(\mathtt{F(P)}) \neq Traces(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} {}^\wedge\, Traces(\mathtt{P})) \qquad \text{Eq. 5.74} \end{aligned}$$

Therefore Interleaving is not Catenary □

**Theorem 5.8:** Change of Symbol is not a Catenary Function ■

*Proof*

By counter example. Consider the function F and process P defined

$$\mathtt{F(P)} = f(\mathtt{P}), \quad \text{where } f\!: \mathbf{a} \rightarrow \mathbf{b} \text{ - } (f \text{ changes event } \mathbf{a} \text{ to event } \mathbf{b}) \qquad \text{Eq. 5.75}$$

$$\mathtt{P} = \mathbf{a} \rightarrow \mathtt{SKIP} \qquad \text{Eq. 5.76}$$

From these definitions

$$\mathcal{D}_{\mathcal{F}} = \{\, \langle\rangle \,\} \wedge Traces(\mathtt{F(P)}) = \{\, \langle\rangle , \langle \mathbf{b} \rangle , \langle \mathbf{b}, \checkmark \rangle \,\} \qquad \text{Eq. 5.77}$$

$$\therefore \quad Traces(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} {}^\wedge\, Traces(\mathtt{P})) \qquad \text{Eq. 5.78}$$

$$= \{\, \langle\rangle \,\} \cup (\, \{\, \langle\rangle \,\} {}^\wedge \{\, \langle\rangle , \langle \mathbf{a} \rangle , \langle \mathbf{a}, \checkmark \rangle \,\}\,) \qquad \text{Eq. 5.79}$$

$$= \{\, \langle\rangle , \langle \mathbf{a} \rangle , \langle \mathbf{a}, \checkmark \rangle \,\} \neq Traces(\mathtt{F(P)}) \qquad \text{Eq. 5.80}$$

Thus Change of Symbol is not Catenary □

**Theorem 5.9:** The hiding operator is not a Catenary function. ■

*Proof*

By counter example. Consider the function F and process P defined

$$F(P) = P \setminus \{ \mathbf{a} \}, \qquad P = \mathbf{a} \rightarrow \mathbf{b} \rightarrow SKIP \qquad \text{Eq. 5.81}$$

From these definitions

$$\mathcal{D}_{\mathcal{F}} = \{ \langle\rangle \} \wedge (\mathit{Traces}(F(P)) = \{ \langle\rangle , \langle \mathbf{b} \rangle , \langle \mathbf{b}, \checkmark \rangle \}) \quad \text{Eq. 5.82}$$

$$\therefore \quad \mathit{Traces}(F(STOP)) \cup (\mathcal{D}_{\mathcal{F}} \,\hat{}\, \mathit{Traces}(P)) \qquad \text{Eq. 5.83}$$

$$= \quad \{ \langle\rangle \} \cup ( \{ \langle\rangle \} \,\hat{}\, \mathit{Traces}(P) ) \qquad \text{Eq. 5.84}$$

$$= \quad \{ \langle\rangle , \langle \mathbf{a} \rangle , \langle \mathbf{a}, \mathbf{b} \rangle , \langle \mathbf{a}, \mathbf{b}, \checkmark \rangle \} \qquad \text{Eq. 5.85}$$

$$\neq \quad \mathit{Traces}(F(P)) \qquad \text{Eq. 5.86}$$

Thus for an arbitrary set of events A the function F = P \ A is not a Catenary function. □

**Theorem 5.10:** If F and G are Catenary functions, then so is the composition of F and G, G ∘ F, ( That is F(G())). ■

*Proof*

Using the rules given by Definition 5.2 (v) and (vi), first establish the result

$$\begin{aligned} &(\mathit{Traces}(F(DED)) - \mathit{Traces}(F(STOP))) \upharpoonright \{\Sigma\text{-}*\} \\ &\;\hat{}\; (\mathit{Traces}(G(DED)) - \mathit{Traces}(G(STOP))) \upharpoonright \{\Sigma - *\} \end{aligned} \qquad \text{Eq. 5.87}$$

$$\begin{aligned} = &((\mathit{Traces}(F(DED)) - \mathit{Traces}(F(STOP))) \,\hat{}\, \mathit{Traces}(G(DED)))\upharpoonright \{\Sigma - *\} \\ &- ((\mathit{Traces}(F(DED)) - \mathit{Traces}(F(STOP))) \,\hat{}\, \mathit{Traces}(G(STOP)))\upharpoonright \{\Sigma - *\} \end{aligned}$$

Eq. 5.88

$$\begin{aligned}
= &((\mathit{Traces}(\mathrm{F}(\mathrm{STOP})) \cup ((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\ \hat{}\ \mathit{Traces}(\mathrm{G}(\mathrm{DED}))))\upharpoonright \{\Sigma - *\} \\
&- (\mathit{Traces}(\mathrm{F}(\mathrm{STOP})) \cup ((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\ \hat{}\ \mathit{Traces}(\mathrm{G}(\mathrm{STOP})))))\upharpoonright \{\Sigma - *\} && \text{Eq. 5.89}
\end{aligned}$$

$$= (\mathit{Traces}(\mathrm{F}(\mathrm{G}(\mathrm{DED}))) - \mathit{Traces}(\mathrm{F}(\mathrm{G}(\mathrm{STOP})))) \upharpoonright \{\Sigma - *\} \qquad \text{Eq. 5.90}$$

Because both F and G are Catenary functions,

$$\begin{aligned}
\mathit{Traces}(\mathrm{F}(\mathrm{G}(\mathrm{P}))) \quad &= \mathit{Traces}(\mathrm{F}(\mathrm{STOP})) \\
&\cup ((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\ \hat{}\ \mathit{Traces}(\mathrm{G}(\mathrm{P}))) && \text{Eq. 5.91}
\end{aligned}$$

$$\begin{aligned}
\mathit{Traces}(\mathrm{G}(\mathrm{P})) \quad &= \mathit{Traces}(\mathrm{G}(\mathrm{STOP})) \\
&\cup ((\mathit{Traces}(\mathrm{G}(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\ \hat{}\ \mathit{Traces}(\mathrm{P})) && \text{Eq. 5.92}
\end{aligned}$$

Therefore,

$$\begin{aligned}
\mathit{Traces}(\mathrm{F}(\mathrm{G}(\mathrm{P}))) \quad &= \mathit{Traces}(\mathrm{F}(\mathrm{STOP})) \\
&\cup ((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\ \hat{}\ (\mathit{Traces}(\mathrm{G}(\mathrm{STOP})) \\
&\cup ((\mathit{Traces}(\mathrm{G}(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\ \hat{}\ \mathit{Traces}(\mathrm{P})))) && \text{Eq. 5.93}
\end{aligned}$$

$$\begin{aligned}
= \quad & \mathit{Traces}(\mathrm{F}(\mathrm{STOP})) \\
&\cup ((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\}\ \hat{}\ \mathit{Traces}(\mathrm{G}(\mathrm{STOP}))) \\
&\cup ((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\ \hat{}\ (\mathit{Traces}(\mathrm{G}(\mathrm{DED})) - \mathit{Traces}(\mathrm{G}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\}\ \hat{}\ \mathit{Traces}(\mathrm{P})) && \text{Eq. 5.94}
\end{aligned}$$

From the definition of a Catenary function,

$$\begin{aligned}
\mathit{Traces}(\mathrm{F}(\mathrm{G}(\mathrm{STOP}))) \quad &= \mathit{Traces}(\mathrm{F}(\mathrm{STOP})) \\
&\cup ((\mathit{Traces}(\mathrm{F}(\mathrm{DED})) - \mathit{Traces}(\mathrm{F}(\mathrm{STOP}))) \upharpoonright \{\Sigma - *\} \\
&\quad \hat{}\ \mathit{Traces}(\mathrm{G}(\mathrm{STOP}))) && \text{Eq. 5.95}
\end{aligned}$$

From equation 5.90,

$$\begin{aligned}&(\mathit{Traces}(\mathtt{F(G(DED))}) - \mathit{Traces}(\mathtt{F(G(STOP))}))) \upharpoonright \{\Sigma - *\}\\ &\qquad = (\mathit{Traces}(\mathtt{F(DED)}) - \mathit{Traces}(\mathtt{F(STOP)})) \upharpoonright \{\Sigma - *\}\\ &\qquad\quad {}^\wedge\ (\mathit{Traces}(\mathtt{G(DED)}) - \mathit{Traces}(\mathtt{G(STOP)})) \upharpoonright \{\Sigma - *\} \qquad \text{Eq. 5.96}\end{aligned}$$

By substituting these into equation 5.95,

$$\begin{aligned}\mathit{Traces}(\mathtt{F(G(P))}) &= \mathit{Traces}(\mathtt{F(G(STOP))})\\ &\quad \cup ((\mathtt{F(G(DED))} - \mathtt{F(G(STOP))}) \upharpoonright \{\Sigma - *\}\\ &\quad {}^\wedge\ \mathit{Traces}(\mathtt{G(STOP)})) \qquad \text{Eq. 5.97}\end{aligned}$$

This is the definition of the Catenary function $\mathtt{G} \circ \mathtt{F}$. Thus $\mathtt{G} \circ \mathtt{F}$ is a Catenary function. □

As a direct result of Theorem 5.10 it is now possible to derive a syntactic definition for a Catenary function.

**Corollary 5.1:** A function F is a Catenary function if it holds for both the following

- F is an expression composed of the following operators: sequential , deterministic choice, nondeterministic choice, prefix.

- For every occurrence of the sequential operator the left hand argument must be independent of the operand.

■

The advantage of this is that, compared to the original semantic definition, a syntactic definition is easier to identify. It comes straight from the syntactic definition of the process and does not require any further analysis.

## 5.8 Parallel Composition.

There is one notable omission to the operators discussed in the previous section, that of the parallel operators. This is because they possess certain properties over recursion which require discussion.

The first point to note concerns the use of the alphabetized parallel operator in a recursive function. In these circumstances it becomes equivalent to the non-alphabetized parallel operator ||, which requires parallel processes to synchronize on every step. This is illustrated by the definition from [ Hoare 85 ] that for all processes P and CSP functions F

$$\text{Alphabet}(F(P)) \quad = \quad \text{Alphabet}(P) \qquad \text{Eq. 5.98}$$

This is illustrated by considering the function F defined by

$$F(P) = G_1(P)\ {}_A||_B\ G_2(P) \qquad \text{Eq. 5.99}$$

It can be seen that since Alphabet($G_1(P)$) = Alphabet(P) and Alphabet($G_2(P)$) = Alphabet(P) it follows that Alphabet($G_1(P)$) = Alphabet($G_2(P)$). That is that A = B. In these circumstances the alphabetized parallel operator becomes equivalent to ||.

The simplification which this assertion affords leads to an important result. That if, for a set of Catenary functions $F_1$, $F_2$, .., $F_n$, the alphabetized parallel operator is used in the definition of a recursive function of the form

$$F_1(P) \,||\, F_2(P) \,||\, .. \,||\, F_n(P) \qquad \text{Eq. 5.100}$$

then the traces of this process are such that

$$\mathit{Traces}(F_1(P) \,||\, .. \,||\, F_n(P)) = \mathit{Traces}(F_1(P)) \cap .. \cap \mathit{Traces}(F_n(P)) \qquad \text{Eq. 5.101}$$

The second point to note is that parallelism is not a Catenary function.

**Theorem 5.11:** The Alphabetized Parallel Operator is not Catenary ■

*Proof*

By counter example. Consider the function F and the process P defined by

$$F(P) = \mathbf{b} \rightarrow P \,||\, \mathbf{b} \rightarrow \mathbf{c} \rightarrow P, \qquad P = \mathbf{c} \rightarrow \mathbf{c} \rightarrow \mathbf{b} \rightarrow \mathrm{SKIP} \qquad \text{Eq. 5.102}$$

Now,

$$\mathit{Traces}(F(\mathrm{DED})) = \mathit{Traces}(F(\mathrm{STOP})) = \{ <> , < \mathbf{b} > \} \qquad \text{Eq. 5.103}$$

It is now possible to use this definition of Weak Catenary functions to develop a rule determining the properties of the parallel operator.

**Theorem 5.13:** If $\mathtt{P}$ is an arbitrary process and $\mathtt{F}$, $\mathtt{G}$ are Weak Catenary functions, then the parallel composition of $\mathtt{F(P)}$ and $\mathtt{G(P)}$, $\mathtt{F(P)} \parallel \mathtt{G(P)}$, is such that

$$Traces(\mathtt{F(P)} \parallel \mathtt{G(P)}) \subseteq Traces(\mathtt{F(STOP)} \parallel \mathtt{G(STOP)}) \wedge Traces(\mathtt{P}) \qquad \text{Eq. 5.110}$$

That is $\mathtt{F(P)} \parallel \mathtt{G(P)}$ is Weak Catenary. ■

*Proof*

By definition,

$$\forall\, \mathtt{P} \bullet Traces(\mathtt{F(P)} \parallel \mathtt{G(P)}) = Traces(\mathtt{F(P)}) \cap Traces(\mathtt{G(P)})$$

$$\subseteq (Traces(\mathtt{F(STOP)}) \wedge Traces(\mathtt{P})) \cap (Traces(\mathtt{G(STOP)}) \wedge Traces(\mathtt{P})) \qquad \text{Eq. 5.111}$$

$$\subseteq (Traces(\mathtt{F(STOP)}) \cap Traces(\mathtt{G(STOP)})) \wedge Traces(\mathtt{P}) \qquad \text{Eq. 5.112}$$

$$\subseteq Traces(\mathtt{F(P)} \parallel \mathtt{G(P)}) \wedge Traces(\mathtt{P}) \qquad \text{Eq. 5.113}$$

Therefore $\mathtt{F(P)} \parallel \mathtt{G(P)}$ is Weak Catenary. □

Theorem 5.13 may be generalised to a set of $\mathtt{n}$ Catenary functions.

**Corollary 5.2:** If the functions $\mathtt{F}_1$, $\mathtt{F}_2$, .., $\mathtt{F}_n$ are either Catenary or Weak Catenary functions, then the parallel composition.

$$\mathtt{F}_1\mathtt{(P)} \parallel \mathtt{F}_2\mathtt{(P)} \parallel .. \parallel \mathtt{F}_n\mathtt{(P)} \qquad \text{Eq. 5.114}$$

is a Weak Catenary Function. That is

$$Traces(\mathtt{F}_1\mathtt{(P)} \parallel \mathtt{F}_2\mathtt{(P)} \parallel .. \parallel \mathtt{F}_n\mathtt{(P)})$$
$$\subseteq Traces(\mathtt{F}_1\mathtt{(P)} \parallel \mathtt{F}_2\mathtt{(P)} \parallel .. \parallel \mathtt{F}_n\mathtt{(P)}) \wedge Traces(\mathtt{P}) \qquad \text{Eq. 5.115}$$

□

If $\mathtt{F}$ is a Weak Catenary function then this implies that the fixed point of $\mathtt{F}$ can be expressed as the catenation of a infinite number of the same set of behaviours.

**Theorem 5.14:** If $\mathtt{F}$ is a Weak Catenary Function then

$$\mathit{Traces}(\mu\mathtt{P.F(P)}) \subseteq \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} .. \qquad \text{Eq. 5.116}$$

■

*Proof*

$\mathtt{F}$ is Weak Catenary, therefore

$$\mathit{Traces}(\mathtt{F(P)}) \subseteq \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} \mathit{Traces}(\mathtt{P}) \qquad \text{Eq. 5.117}$$

Because $\mu\mathtt{P.F(P)}$ is the fixed point,

$$\mu\mathtt{P.F(P)} = \mathtt{F}(\mu\mathtt{P.F(P)}) \qquad \text{Eq. 5.118}$$

$$\therefore \quad \mathit{Traces}(\mathtt{F}(\mu\mathtt{P.F(P)})) \subseteq \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} \mathit{Traces}(\mu\mathtt{P.F(P)})$$

$$\subseteq \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} \mathit{Traces}(\mathtt{P}) \qquad \text{Eq. 5.119}$$

$$\subseteq .. \quad .. \quad ..$$

$$\subseteq \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} \mathit{Traces}(\mathtt{F(STOP)}) \mathbin{\hat{}} .. \qquad \text{Eq. 5.120}$$

□

## 5.9 Relating Catenary Functions to Traces.

Finally in this chapter a result is provided which explicitly illustrates how the fixed point of every Catenary function $\mathtt{F}$ is made up from the catenation of traces from a particular subset, specifically $\mathtt{F(STOP)}$.

**Theorem 5.15:** Let $\mathtt{F}$ be a Catenary function. Let $\mathtt{n}$ be a non-zero positive integer and $\mathtt{r}_1,..,\mathtt{r}_n$ be traces. Then the trace $\mathtt{s}$ is such that

$$\mathtt{s} \in \mathtt{F}^{n}(\mathtt{STOP}) \quad \Leftrightarrow \quad \mathtt{s} = \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_q \qquad \text{Eq. 5.121}$$

where $\mathtt{q} \leq \mathtt{n}$, $\mathtt{r}_1,..,\mathtt{r}_{\mathtt{q-1}} \in \mathcal{D}_{\mathcal{F}}$ and $\mathtt{r}_\mathtt{q} \in \mathit{Traces}(\mathtt{F(STOP)})$ ■

*Proof*

For convenience of notation, let $(\mathcal{D}_{\mathcal{F}}^{\mathtt{n}})$ represent the set catenation of $\mathtt{n}$ sets $\mathcal{D}_{\mathcal{F}}$. That is

$$(\mathcal{D}_{\mathcal{F}}^{\mathtt{n}}) \quad = \quad (\mathcal{D}_{\mathcal{F}} \,^\wedge \, .. \,^\wedge\, \mathcal{D}_{\mathcal{F}}) \qquad \text{Eq. 5.122}$$

From the definition of a Catenary function it can be seen that

$$\mathit{Traces}(\mathtt{F}^{\mathtt{n}}(\mathtt{STOP})) \quad = \quad \mathit{Traces}(\mathtt{F}(\mathtt{F}^{\mathtt{n-1}}(\mathtt{STOP})))$$

$$= \quad \mathit{Traces}(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} \,^\wedge\, \mathit{Traces}(\mathtt{F}^{\mathtt{n-1}}(\mathtt{STOP}))) \qquad \text{Eq. 5.123}$$

$$= \quad \mathit{Traces}(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} \,^\wedge\, (\mathit{Traces}(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} \,^\wedge\, \mathit{Traces}(\mathtt{F}^{\mathtt{n-2}}(\mathtt{STOP}))))) \qquad \text{Eq. 5.124}$$

$$= \quad \mathit{Traces}(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} \,^\wedge\, (\mathit{Traces}(\mathtt{F(STOP)})) \cup (\mathcal{D}_{\mathcal{F}} \,^\wedge\, \mathcal{D}_{\mathcal{F}} \,^\wedge\, \mathit{Traces}(\mathtt{F}^{\mathtt{n-2}}(\mathtt{STOP}))))) \qquad \text{Eq. 5.125}$$

Continuing this expansion results in the expression

$$= \quad \mathit{Traces}(\mathtt{F(STOP)}) \cup (\mathcal{D}_{\mathcal{F}} \,^\wedge\, \mathit{Traces}(\mathtt{STOP})) \cup ((\mathcal{D}_{\mathcal{F}}^{2}) \,^\wedge\, \mathit{Traces}(\mathtt{P})) \cup .. \cup ((\mathcal{D}_{\mathcal{F}}^{\mathtt{n-1}}) \,^\wedge\, \mathit{Traces}(\mathtt{STOP})) \qquad \text{Eq. 5.126}$$

Now, if there is some integer $\mathtt{q} \leq \mathtt{n}$ such that $\mathtt{r}_1,..,\mathtt{r}_{\mathtt{q-1}} \in \mathcal{D}_{\mathcal{F}}$ and $\mathtt{r}_\mathtt{q} \in \mathit{Traces}(\mathtt{F(STOP)})$, then

$$\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_\mathtt{q} \in .(\mathcal{D}_{\mathcal{F}}^{\mathtt{q-1}}) \,^\wedge\, \mathit{Traces}(\mathtt{F(STOP)}) \qquad \text{Eq. 5.127}$$

Therefore comparison with equation 5.126 yields

$$\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_\mathtt{q} \in \mathit{Traces}(\mathtt{F}^{\mathtt{n}}(\mathtt{STOP}))$$

Conversely if $\mathtt{s} \in \mathit{Traces}(\mathtt{F}^{\mathtt{n}}(\mathtt{STOP}))$ then inspection of equation 5.126 shows that $\mathtt{s}$ must belong to one of the unified sets. Therefore

$$\exists\, q \leq n \bullet s \in ((\mathcal{D}_{\mathcal{F}}^{q-1}) \wedge Traces(P)) \qquad \text{Eq. 5.128}$$

$$\therefore \quad s = r_1\hat{\ }..\hat{\ }r_q \qquad \text{Eq. 5.129}$$

where $r_1,..,r_{q-1} \in \mathcal{D}_{\mathcal{F}}$ and $r_q \in Traces(F(STOP))$. □

## 5.10 Summary.

The chapter has set out to categorize a subclass of CSP functions which, when used in recursive definitions, result in processes that repeat the same set of behaviours on each recursive step. The class of functions is called the Catenary functions, and suitable definitions have been provided to allow functions to be categorized as such. To make this categorisation simpler, the chapter has produced a set of syntactic rules which indicate that a function is Catenary.

The functions which remain after the limitations imposed by Catenary functions are the deterministic choice operator, the nondeterministic choice operator, the prefix operator and the sequential operator. However, perhaps a greater insight into the scope of Catenary functions can be gained by considering those functions which do not fall into this category.

The first exception was the hiding operator. This is neither Catenary nor is it contractive on any of the semantic domains $M_T$, $M_F$, $TM_F$. As such its inclusion in any functional definition would require care since it may lead to a divergent process.

The change of symbol operator is a contraction on the semantic domains of CSP, but is not a Catenary function. While its omission is a drawback, there are suggestions that the theory could be modified to include it, particularly if the change of symbol function $f$ is such that there exists some $n$ such that

$$\forall\, P \bullet f^n(P) = P \qquad \text{Eq. 5.130}$$

However, it was felt that such modifications would complicate the work of subsequent chapters for an operator with limited use.

The loss of the interleaving operator is considered to be more serious. It was initially thought that interleaving was a Catenary function but subsequent analysis has shown otherwise. Admittedly the use of the interleaving operator in a functional definition may lead to nondeterministic behaviour since there is little guarantee about the ordering of the subsequent traces. However, the operator has been useful in the modelling of interrupts and timeout mechanisms [ Davies 89a ].

The non Catenary nature of the parallel operator was also seen as a drawback. The development of the idea of a Weak Catenary function was a direct measure to counteract this and to place the parallel operator in a similar league with the other Catenary operators. But with regard to subsequent work on the composition of predicates and the construction of Ideal Test Sets (see next chapter) it was found that the definition of a Weak Catenary function was insufficient to support this theory. There is, however, some question as to the validity of introducing a parallel operator into the definition of a recursive function. It allows the possibly of the recursive process creating an unlimited number of parallel processes, which would invariably cause implementation problems.

Finally it should be pointed out that the nature of Catenary functions is somewhat similar to the concept of tail recursion used in functional languages such as ML.

# CHAPTER SIX

# IDEAL TEST SETS

## 6.1 Introduction.

The previous chapter discussed a subset of recursive processes which repeatedly perform the same fundamental set of behaviours. That is those defined as the fixed point of a Catenary function. By doing so it established that, for Catenary functions, an explicit connection between recursion and catenation can be made. It is the purpose of this chapter to exploit this link between recursion and catenation, and to show how it may be used to develop a theory of testing for CSP processes.

The chapter opens by considering the idea of an ideal test which was also discussed in the previous chapter. This is a procedure by which the properties of an entire system may be inferred by considering only a part of the possible behaviours the system may exhibit. By placing these ideas within the context of CSP the related concept of an Ideal Test Set is developed. An Ideal Test Set is a subset of the behaviours of a CSP process for which the truth of a particular specification over all the behaviours may be determined.

The chapter then moves on to show how Ideal Test Sets can be generated for the recursive fixed points of those Catenary functions which were introduced in Chapter 5. By utilising the link established between recursion and catenation in Catenary functions the chapter introduces four categories that describe the properties which behavioural specifications display over catenation. By a series of definitions and theorems a mathematical theory is built up around these categories. This theory is developed using the traces model $M_T$. Finally this theory is subsequently used to generate and justify Ideal Test Sets.

## 6.2 Extending Ideal Tests to CSP.

The concept of an ideal test as given in Definition 5.1 suggests an attractive means of reducing the obligation for correctness proofs. In this thesis it was considered to be particularly applicable to recursive CSP processes for two reasons.

i) Recursive processes have the potential to produce many, sometimes unlimited, system behaviours. This makes exhaustive testing alone infeasible for verifying them.

ii) There is a well established mathematical theory of recursion, which uses fixed point solutions and induction to prove properties about recursive processes. It was felt that such a theory could be used to provide justification for ideal tests.

The first problem encountered was that of providing a definition of an ideal test which was properly applicable to CSP. The main obstacle to this was the fact that Goodenough's original definition [ Goodenough 75 ] was for state-based systems where specifications were written as predicates on the input/output relationships. However, the semantic models of CSP given in Chapters 2 and 3 represent systems in terms of the events and traces they produce rather than an input/output relationship. These conventional models of CSP deliberately avoid a state transitional approach and thus have no explicit embodiment of state.

Similar problems arise with input/output relationships. Perhaps the closest analogy for defining the input and output of a CSP process is to equate input with the environment of a process and output with the behaviour of that process when placed in that environment. However these definitions do not resolve how ideal tests can be applied to CSP on their own. The main problem perceived by the author is that behavioural specifications do not take the environment of a process into account, and thus are independent of the input. In such circumstances it makes no sense to try and determine a subset of the input over which the truth of a specification can be inferred.

To avoid the problems outlined above, the approach adopted was to preserve the idea of inferring properties of a set from a subset, but to apply it to the behaviours of a particular process rather than the input/output relationships of a particular system. This resulted in the new concept of an Ideal Test Set, presented below.

**Definition 6.1:** Let P be a CSP process and let R be a behavioural specification on P. Let Obs denote the set of all observations which process P may exhibit. The set I is said to be an Ideal Test Set for the pair (P, R) if and only if

i) $\mathrm{Obs} \supseteq \mathrm{I}$ Eq. 6.1

ii) $(\forall t \in \mathrm{I} \bullet \mathrm{R}(t)) \quad \Leftrightarrow \quad (\forall s \in \mathrm{Obs} \bullet \mathrm{R}(s))$ Eq. 6.2

■

Condition (i) is present both to simplify and fortify the definition. If, for example, there is some observation in I for which specification R does not hold then $\mathrm{Obs} \supseteq \mathrm{I}$ implies that R will not hold for all observations in Obs. That is R is true over I if and only if R is true over Obs. Consequently it is possible to infer both the truth *and* the falsehood of predicates with an Ideal Test Set.

In order to develop this concept and to construct methods for the generation of Ideal Test Sets for specific processes, such as those defined by Catenary functions, it is necessary to examine the types of process behaviours for which the approach is valid. In particular, Section 6.3 examines the need to exclude infinite traces and processes with infinite alphabets.

## 6.3 Treatment of Infinite Traces and Alphabets.

An infinite trace is one which has an infinite number of events in it. Likewise a process has an infinite alphabet if it has the potential to perform an infinite set of events. Note that a process with an infinite alphabet is not necessarily the same as a process which has the potential to perform all events.

The treatment of infinite traces requires special consideration because their properties are different to those of finite traces. With respect to the work of this text the main problem that infinite traces present is that it is not possible to catenate an infinite trace with any other trace. This is because an infinite trace effectively has no end onto which another trace can be attached. However, catenation is crucial to the concept of an Ideal Test Set. Therefore, in the following text infinite traces are excluded and subsequently all traces are assumed to be finite, unless otherwise stated.

This assumption is not as restrictive as it may appear. The notion of an infinite trace is very much an abstract concept. The abstract processes described in this thesis are ultimately intended to realise an implementation. No implementation can truly generate an infinite sequence of discrete actions.

One circumstance in which infinite traces may arise is for recursively defined processes. Therefore it is useful to explicitly state what interpretation will be assumed by this text. The statement

$$\mu \mathtt{P.F(P)}\ \mathbf{sat}\ \mathtt{R} \qquad \text{Eq. 6.3}$$

as defined by Hoare [Hoare 85] is interpreted as " The behavioural specification R is true for every *finite* observation of the process $\mu$P.F(P). " More formally, drawing on the fixed point treatment of chapter three

$$\mu \mathtt{P.F(P)}\ \mathbf{sat}\ \mathtt{R} \quad \Leftrightarrow \quad \forall\, \mathtt{n} \in \mathbb{N} \bullet \mathtt{F}^{\mathtt{n}}(\mathtt{STOP})\ \mathbf{sat}\ \mathtt{R} \qquad \text{Eq. 6.4}$$

The presence of an infinite alphabet may lead to the problem of unbound nondeterminism [ Roscoe 88a ]. Unbound nondeterminism arises when a process is able to make a nondeterministic choice between an infinite number of events at any one time. The difficulties encountered with this have forced the language and syntax of CSP to be restricted.

It is recognized that the problem of unbound nondeterminism is an inconvenience and that research has been undertaken to address it by reappraising the semantic definitions of CSP processes [ Roscoe 88a ]. However there is no indication that the presence of either an infinite alphabet or unbound nondeterminism has an adverse effect on any of the catenation rules. Consequently in this chapter there is not thought to be a need for any assumption to be made about alphabets. (However, it will be shown in the next chapter that there are advantages to finite alphabets in increasing the scope of the method used.)

## 6.4 Categorizing Specifications.

There are certain behavioural specifications which exhibit properties over catenation. Take for example the predicate ( **a** **in** s ) which states that a trace s will contain the event **a**. If, for two traces s and t, the event **a** is present in either of them, then the event **a** will also be present in the catenation of s and t. More formally

$$(\,\mathbf{a}\ \underline{\mathbf{in}}\ \mathtt{s}\,) \vee (\,\mathbf{a}\ \underline{\mathbf{in}}\ \mathtt{t}\,) \quad \Rightarrow \quad (\,\mathbf{a}\ \underline{\mathbf{in}}\ \mathtt{s}\hat{\ }\mathtt{t}\,) \qquad \text{Eq. 6.5}$$

This is an example of a predicate which distributes over catenation. Other predicates can be shown to distribute over catenation in other ways. It is the aim of this section to develop four categories which define the properties a predicate may possess over the catenation of traces. Furthermore it will show how these predicate categories behave when operated upon by logical connectives.

### 6.4.1 Notation.

As this chapter progresses some of the expressions will require notational conventions. These are defined and explained below.

**Definition 6.2:** Let $e_i$ be some indexed expression, where $i \in \mathbb{N}$. The logical compositions under conjunction and disjunction of $n$ such expressions, where $n \in \mathbb{N}$, may be abbreviated by the notations

$$\bigwedge_{i=1}^{n} (e_i) \equiv e_1 \wedge e_2 \wedge .. \wedge e_n \qquad \text{Eq. 6.6}$$

$$\bigvee_{i=1}^{n} (e_i) \equiv e_1 \vee e_2 \vee .. \vee e_n \qquad \text{Eq. 6.7}$$

Additionally, for arbitrary predicates $Q$ and $R$, logical conjunction and disjunction may be abbreviated by

$$Q(s) \wedge R(s) \equiv Q{\wedge}R(s) \qquad \text{Eq. 6.8}$$

$$Q(s) \vee R(s) \equiv Q{\vee}R(s) \qquad \text{Eq. 6.9}$$

■

In the subsequent text expressions may be written as the combination of an ordered set of indexed terms under some operator. That is for an operator ⊗ , say, and some set of indexed terms, $e_1$, .., $e_n$ ($n \in \mathbb{N}$), there may be the expression

$$e_1 \otimes e_2 \otimes .. \otimes e_{n-1} \otimes e_n \qquad \text{Eq. 6.10}$$

The convention adopted in this text is that two full stops in an indexed expression indicate the presence of an intermediate sequence of ordered indexed terms. For example

$$e_5 \otimes .. \otimes e_9 = e_5 \otimes e_6 \otimes e_7 \otimes e_8 \otimes e_9 \qquad \text{Eq. 6.11}$$

Sometimes it is necessary to express an ordered sequence of $\mathtt{n}$ terms with the $\mathtt{i}^{\mathtt{th}}$ term missing. The notation chosen for this is

$$\mathtt{e}_1 \otimes .. \otimes \mathtt{e}_{\mathtt{i-1}} \otimes \mathtt{e}_{\mathtt{i+1}} .. \otimes \mathtt{e}_{\mathtt{n}} \qquad \text{Eq. 6.12}$$

To avoid confusion, especially when the index $\mathtt{i}$ takes the value $\mathtt{1}$ or $\mathtt{n}$, when $\mathtt{i=1}$ equation 6.12 is interpreted as having the first term missing, but no term $\mathtt{e}_0$ and when $\mathtt{i} = \mathtt{n}$ equation 6.12 has the last term missing. That is

$$\mathtt{i=1} \;\Rightarrow\; \mathtt{e}_1 \otimes .. \otimes \mathtt{e}_{\mathtt{i-1}} \otimes \mathtt{e}_{\mathtt{i+1}} .. \otimes \mathtt{e}_{\mathtt{n}} = \mathtt{e}_2 \otimes .. \otimes \mathtt{e}_{\mathtt{n}} \qquad \text{Eq. 6.13}$$

$$\mathtt{i=n} \;\Rightarrow\; \mathtt{e}_1 \otimes .. \otimes \mathtt{e}_{\mathtt{i-1}} \otimes \mathtt{e}_{\mathtt{i+1}} .. \otimes \mathtt{e}_{\mathtt{n}} = \mathtt{e}_1 \otimes .. \otimes \mathtt{e}_{\mathtt{n-1}} \qquad \text{Eq. 6.14}$$

## 6.4.2 Categories.

There are four ways in which a predicate may distribute over catenation. The following definition introduces four categories which correspond to these.

**Definition 6.3:** Four categories of predicates which possess properties over the catenation of traces are defined here. They are the sets α, β, γ, δ † such that for traces $\mathtt{s}$, $\mathtt{t}$

$$\alpha = \{\mathtt{R} \mid \forall\, \mathtt{s} \neq \langle\rangle \;\bullet\; \mathtt{R(s)} \Leftrightarrow \mathtt{R(s\hat{}t)}\} \qquad \text{Eq. 6.15}$$

$$\beta = \{\mathtt{R} \mid \forall\, \mathtt{t} \neq \langle\rangle \;\bullet\; \mathtt{R(t)} \Leftrightarrow \mathtt{R(s\hat{}t)}\} \qquad \text{Eq. 6.16}$$

$$\gamma = \{\mathtt{R} \mid \forall\, \mathtt{s,t} \neq \langle\rangle \;\bullet\; \mathtt{R(s)} \vee \mathtt{R(t)} \Leftrightarrow \mathtt{R(s\hat{}t)}\} \qquad \text{Eq. 6.17}$$

$$\delta = \{\mathtt{R} \mid \forall\, \mathtt{s,t} \neq \langle\rangle \;\bullet\; \mathtt{R(s)} \wedge \mathtt{R(t)} \Leftrightarrow \mathtt{R(s\hat{}t)}\} \qquad \text{Eq. 6.18}$$

■

†Note: The categories α, β, γ, δ and the later categories $\overline{\alpha}$, $\overline{\beta}$ and $\delta^{\mathtt{n}}$ are unconnected with the conditions of continuity (α), (β), (γ) and (δ) imposed upon predicates in [Roscoe 88a]. The similarity is purely notational.

## 6.5 General Properties of the Categories.

Having given the above definition for the categories α, β, γ, δ it is useful to be able to determine some general properties about all the categories. Theorem 6.1 illustrates a property which is common to all four categories.

**Theorem 6.1:** Let R be a predicate such that

$$R \in \{ \alpha \cup \beta \cup \gamma \cup \delta \} \qquad \text{Eq. 6.19}$$

For traces s and t, the predicate R is such that

$$\forall s \neq \langle\rangle, t \neq \langle\rangle \quad \bullet \quad R(s) \wedge R(t) \Rightarrow \quad R(s\hat{\ }t) \qquad \text{Eq. 6.20}$$

■

*Proof*

*Case* $R \in \alpha$

$$(\forall s, t \neq \langle\rangle \bullet ( R(s) \wedge R(t) \Rightarrow \quad R(s) \Rightarrow R(s\hat{\ }t) )) \qquad \text{Eq. 6.21}$$

*Case* $R \in \beta$

$$(\forall s, t \neq \langle\rangle \bullet ( R(s) \wedge R(t) \Rightarrow \quad R(t) \Rightarrow R(s\hat{\ }t) )) \qquad \text{Eq. 6.22}$$

*Case* $R \in \gamma$

$$(\forall s, t \neq \langle\rangle \bullet ( R(s) \wedge R(t) \Rightarrow \quad R(s) \vee R(t) \Rightarrow R(s\hat{\ }t) )) \qquad \text{Eq. 6.23}$$

*Case* $R \in \delta$

$$(\forall s, t \neq \langle\rangle \bullet ( R(s) \wedge R(t) \Rightarrow \quad R(s\hat{\ }t) )) \qquad \text{Eq. 6.24}$$

□

It is this particular property and its derivatives which form the fundamental basis of this chapter. It illustrates that the defined categories permit the truth of a composite trace to be deduced from the truth of its constituent parts.

To explain the advantage of this consider a process P which performs a trace t. Let t be formed from the catenation of n non-empty traces $r_1, r_2, .., r_n$ and let R be a predicate belonging to one of the above categories and suppose that R(t) must be established. By repeatedly applying equation 6.20 to t it is possible to show that

$$R(r_1) \wedge R(r_2) \wedge .. \wedge R(r_n) \quad \Rightarrow \quad R(t) \qquad \text{Eq. 6.25}$$

At present there may seem little advantage in determining the truth of the component parts to infer the truth of the composite trace when the truth of t could easily be established on its own. But if n were a very large number, and it was shown that all the traces $r_1, r_2, .., r_n$ belonged to a relatively small set, A say, then in these circumstances it might be easier to establish that R were true over A, and thus infer R(t), rather than showing that R were true for a long trace t. Furthermore, if it could be shown that every possible trace in P was expressible as the catenation of traces which belonged to A, then if R were true for every trace in A, R would be true for the catenation of every trace in A, and thus R would be true for every trace in P.

In the previous chapter it was shown that the fixed point of a Catenary function F could be expressed as a process made up from a set of traces, each of which was in turn made up from the catenation of traces from the limited set *Traces*(F(STOP)). This suggests that, for a predicate R from one of the above categories, if R is true for every trace in *Traces*(F(STOP)), then R is true for every trace in the fixed point of F.

In addition the set *Traces*(F(STOP)) is a subset of *Traces*(μP.F(P)). Therefore the set *Traces*(F(STOP)) fulfils the definition of an Ideal Test Set for the pair (μP.F(P), R). This line of reasoning is more formally summed up by the following theorem.

**Theorem 6.2:** Let F be a Weak Catenary function, and let R be a predicate which belongs to the set { $\alpha \cup \beta \cup \gamma \cup \delta$ }. Then the set of traces

$$\mathit{Traces}(F(STOP)) \qquad \text{Eq. 6.26}$$

is an Ideal Test Set for the pair (μP.F(P), R). That is

$$(\forall\, s \in \mathit{Traces}(F(STOP)) \bullet R(s)) \\ \Leftrightarrow (\forall\, s \in \mathit{Traces}(\mu P.F(P)) \bullet R(s)) \qquad \text{Eq. 6.27}$$

■

*Proof*

Let the trace $s$ be any trace of the process $\mu P.F(P)$. From the definition of $F$ as a Weak Catenary function

$$\mathit{Traces}(\mu P.F(P)) \subseteq \mathit{Traces}(F(STOP)) \mathbin{\hat{}} \mathit{Traces}(F(STOP))\mathbin{\hat{}} \ .. \qquad \text{Eq. 6.28}$$

Therefore, for traces $r_1, r_2, .., r_n \in \mathit{Traces}(F(STOP))$, where $n \in \mathbb{N}$

$$\exists\, n \geq 1 \bullet s = r_1\hat{}r_2\hat{}..\hat{}r_n \qquad \text{Eq. 6.29}$$

Establish by testing that predicate $R$ holds for the process $F(STOP)$. Thus

$$\forall\, s \in \mathit{Traces}(F(STOP)) \bullet R(s) \qquad \text{Eq. 6.30}$$

$$\therefore \quad R(r_1) \wedge R(r_2) \wedge .. \wedge R(r_n) \qquad \text{Eq. 6.31}$$

Now, $R \in \{\ \alpha \cup \beta \cup \gamma \cup \delta\ \}$ and thus, by Theorem 6.1

$$R(r_1) \wedge R(r_2) \Rightarrow R(r_1\hat{}r_2) \qquad \text{Eq. 6.32}$$

$$R(r_1\hat{}r_2) \wedge R(r_3) \Rightarrow R(r_1\hat{}r_2\hat{}r_3) \qquad \text{Eq. 6.33}$$

$$.. \ .. \ ..$$

$$R(r_1\hat{}..\hat{}r_{n-1}) \wedge R(r_n) \Rightarrow R(r_1\hat{}..\hat{}r_n) \qquad \text{Eq. 6.34}$$

Therefore, if $s$ is a finite trace in the process $\mu P.F(P)$, and $R$ is a predicate which is true over the set $F(STOP)$, then $R$ is true for all finite traces $s$. If $Obs$ is the set of all finite behaviours of $\mu P.F(P)$

$$(\forall\, s \in \mathit{Traces}(F(STOP)) \bullet R(s)) \quad \Rightarrow \quad (\forall\, s \in Obs \bullet R(s)) \qquad \text{Eq. 6.35}$$

Additionally,

$$Obs \supseteq \mathit{Traces}(F(STOP)) \qquad \text{Eq. 6.36}$$

$$\therefore \quad (\forall\, s \in Obs \bullet R(s)) \quad \Leftrightarrow \quad (\forall\, s \in \mathit{Traces}(F(STOP)) \bullet R(s)) \qquad \text{Eq. 6.37}$$

Thus by definition *Traces*(`F(STOP)`) is an Ideal Test Set for (`μP.F(P)`, `R`)

□

This theorem captures the essence of how it is possible to generate Ideal Test Sets for all pairs of the form (`μP.F(P)`, `R`), where `F` is a Catenary function and `R` is a predicate in one of the categories $\alpha$, $\beta$, $\gamma$, $\delta$.

## 6.6 Using the Categories to Build Compound Predicates.

In themselves, the four categories so far defined have a limited scope. They represent a somewhat restricted range of predicates, that is those which possess certain well defined properties. As such it would clearly be useful to be able to extend the range of predicates over which the categories $\alpha$, $\beta$, $\gamma$, $\delta$ apply. It is intended to achieve this by permitting predicates from the different categories to be composed under logical operators or connectives.

A language, called RSPEC, is now introduced in which such composed predicates can be expressed. As is seen below a statement in RSPEC is essentially the result of combining a number of predicates from the set $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$ under the logical connectives $\{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$. RSPEC is defined by the following Backus Naur Form

**Definition 6.4**: Let RSPEC be a formal language in which to express predicates. The BNF definition of RSPEC is

```
STATEMENT      :: =   CATEGS |
                      STATEMENT, "∨", STATEMENT |
                      STATEMENT, "∧", STATEMENT |
                      STATEMENT, "⇒", STATEMENT |
                      STATEMENT,  "⇔", STATEMENT |
                      "¬", STATEMENT;

CATEGS         ::=    PREDα | PREDβ | PREDγ | PREDδ ;

PREDα          ::=    ( A predicate in the category α );

PREDβ          ::=    ( A predicate in the category β );
```

PREDγ ::= ( A predicate in the category γ );

PREDδ ::= ( A predicate in the category δ );

■

It can be surmised from the above definition that an informal description of the set CATEGS would be the set of syntax of all predicates which belong to categories α, β, γ, δ. The structure of RSPEC is very similar to that of propositional calculus, with predicates from the set { $\alpha \cup \beta \cup \gamma \cup \delta$ } taking the place of propositions. In fact, this parallel is so strong that it is possible to use the concept of substitution instances [ Hamilton 78 ] to state the following theorem.

**Theorem 6.3**: Let R be a statement in RSPEC which is not a tautology. Then it is possible to write R in the conjunctive normal form as

$$\bigwedge_{i=1}^{n} \left( \bigvee_{j=1}^{m} (Q_{ij}) \right) \qquad \text{Eq. 6.38}$$

where either $Q_{ij}$ or $\neg(Q_{ij})$ is a predicate from the set { $\alpha \cup \beta \cup \gamma \cup \delta$ }.

■

*Proof*

The reader is referred to [ Hamilton 78 ] □

Essentially, a statement in RSPEC is a tautology if it is always true. For the purposes of this text all statements will be assumed not to be tautologies. This is justified on the grounds that it is the aim of this work to prove statements true with regard to some system, and that as a consequence it is unnecessary to prove tautologies true.

Theorem 6.2 showed that given a predicate R from the set { $\alpha \cup \beta \cup \gamma \cup \delta$ } and a Catenary function F, that it was possible to construct a finite Ideal Test Set for the pair (μP.F(P), R). This was possible because the categories related how predicates within them distributed over trace catenation. This suggested that if similar distributive properties could be identified in all the predicates of RSPEC then a similar mechanism may exist for RSPEC. That is, given a predicate R from RSPEC and a Catenary Function F, it should be possible to generate an Ideal Test Set for the pair (μP.F(P), R).

The purpose of the remaining sections of this chapter is to develop this idea and show how Ideal Test Sets are generated for predicates in RSPEC.

## 6.7 Closures.

Certain predicates from the set { $\alpha \cup \beta \cup \gamma \cup \delta$ } when operated upon by connectives can be shown to remain in that set. The advantage of this is that it is possible to show that some compound predicates in RSPEC actually belong to { $\alpha \cup \beta \cup \gamma \cup \delta$ }. For example, if R is a predicate in category $\alpha$, then $\neg$R is also a predicate in category $\alpha$. Category $\alpha$ is therefore said to be closed under logical negation. The following theorem shows how all the categories behave under logical negation.

**Theorem 6.4:** The sets $\alpha$, $\beta$, $(\gamma \cup \delta)$ are each individually closed under logical negation. That is for some predicate R

i) $R \in \alpha \quad \Leftrightarrow \quad \neg R \in \alpha$

ii) $R \in \beta \quad \Leftrightarrow \quad \neg R \in \beta$

iii) $R \in \gamma \quad \Leftrightarrow \quad \neg R \in \delta$ ■

*Proof*

*Case (i)*

$R \in \alpha$ if and only if

$$(R(s) \quad \Leftrightarrow \quad R(s\hat{\ }t)) \qquad \text{Eq. 6.39}$$

$$\text{Eq. 6.39} \Leftrightarrow (\neg R(s) \quad \Leftrightarrow \quad \neg R(s\hat{\ }t)) \qquad \text{Eq. 6.40}$$

Thus $R \in \alpha$ if and only if $\neg R \in \alpha$

*Case(ii)*

The proof is similar to that of Case (i)

*Case(iii)*

For traces s, $t \neq \langle\rangle$, $R \in \gamma$ if and only if

$$(\mathtt{R(s)} \vee \mathtt{R(t)}) \quad \Leftrightarrow \quad \mathtt{R(s\hat{}t)} \qquad \text{Eq. 6.41}$$

$$\text{Eq. 6.41} \Leftrightarrow \neg(\,\mathtt{R(s)} \vee \mathtt{R(t)}\,) \quad \Leftrightarrow \quad \neg\mathtt{R(s\hat{}t)} \qquad \text{Eq. 6.42}$$

Using De Morgan's Rule [ Borowski 89 ]

$$\text{Eq. 6.41} \Leftrightarrow \neg\mathtt{R(s)} \wedge \neg\mathtt{R(t)}\,) \quad \Leftrightarrow \quad \neg\mathtt{R(s\hat{}t)} \qquad \text{Eq. 6.43}$$

Thus $\mathtt{R} \in \gamma$ if and only if $\neg\mathtt{R} \in \delta$ □

Theorem 6.4 is a useful and deliberate result. Not only are the categories specifically closed under the negation connective $\neg$ in the way described, they are also closed as a whole. That is if $\mathtt{R}$ is a predicate in the set $\{\alpha \cup \beta \cup \gamma \cup \delta\}$, then it can be deduced that $\neg\mathtt{R}$ is also a predicate in the set $\{\alpha \cup \beta \cup \gamma \cup \delta\}$. Therefore, referring back to equation 6.38, it is possible to deduce that every statement in RSPEC can now be expressed in the form

$$\bigwedge_{i=1}^{n} \left( \bigvee_{j=1}^{m} (\mathtt{Q}_{ij}) \right) \qquad \text{Eq. 6.44}$$

where $\mathtt{Q}_{ij}$ is a predicate from the set $\{\alpha \cup \beta \cup \gamma \cup \delta\}$.

The next operator, that of logical disjunction (the 'or' operator) is less straightforward. It does possess a range of certain useful closure properties which are given by the following theorem.

**Theorem 6.5:** The sets $\alpha$, $\beta$, $\gamma$ are specifically closed under logical disjunction. That is for predicates $\mathtt{R}_1$, $\mathtt{R}_2$

i) $\mathtt{R}_1 \in \alpha \;\wedge\; \mathtt{R}_2 \in \alpha \;\Rightarrow\; (\mathtt{R}_1 \vee \mathtt{R}_2) \in \alpha$

ii) $\mathtt{R}_1 \in \beta \;\wedge\; \mathtt{R}_2 \in \beta \;\Rightarrow\; (\mathtt{R}_1 \vee \mathtt{R}_2) \in \beta$

iii) $\mathtt{R}_1 \in \gamma \;\wedge\; \mathtt{R}_2 \in \gamma \;\Rightarrow\; (\mathtt{R}_1 \vee \mathtt{R}_2) \in \gamma$ ■

*Proof*

*Case (i)*

By definition

$$R_1(s) \vee R_2(s) \Leftrightarrow R_1(s\hat{\ }t) \vee R_2(s\hat{\ }t) \qquad \text{Eq. 6.45}$$

$$\therefore \quad R_1 \vee R_2(s) \Leftrightarrow R_1 \vee R_2(s\hat{\ }t) \qquad \text{Eq. 6.46}$$

Thus $R_1 \vee R_2 \in \alpha$

*Case (ii)*

The proof is similar to that of Case (i)

*Case (iii)*

By definition

$$(R_1(s) \vee R_2(s)) \vee (R_1(t) \vee R_2(t)) \Leftrightarrow R_1(s\hat{\ }t) \vee R_2(s\hat{\ }t) \qquad \text{Eq. 6.47}$$

$$\therefore \quad R_1 \vee R_2(s) \vee R_1 \vee R_2(t) \Leftrightarrow R_1 \vee R_2(s\hat{\ }t) \qquad \text{Eq. 6.48}$$

Thus $R_1 \vee R_2 \in \gamma$ □

Each of the categories $\alpha$, $\beta$, $\gamma$ are therefore closed under logical disjunction but category $\delta$ is not so closed. However category $\delta$ does preserve some properties under logical disjunction. That is for predicates $Q$, $R \in \delta$ the predicate $Q \vee R$ can be shown to exhibit certain properties over the catenation of traces. To illustrate these it is necessary to extend the definition of category $\delta$.

**Definition 6.5:** Category $\delta$ may be extended to the general indexed category $\delta^n$, where $n \in \mathbb{N}$. This is defined by

$$\delta^n = \{ R \mid \forall r_1, .., r_{n+1} \neq \langle\rangle \bullet \bigwedge_{i=1}^{n+1} R(r_1\hat{\ }..\hat{\ }r_{i-1}\hat{\ }r_{i+1}\hat{\ }..\hat{\ }r_{n+1}) \Leftrightarrow R(r_1\hat{\ }..\hat{\ }r_{n+1}) \} \qquad \text{Eq. 6.49}$$

■

Contrast equation 6.49 with the definition of category $\delta$ given by equation 6.18. Note that when $n = 1$ equation 6.49 is equivalent to equation 6.18. Category $\delta$ showed how a predicate true over a set of distinct traces could be shown to be equivalent to a predicate true on the ordered catenation of those traces. The general category $\delta^n$

illustrates how a predicate true over a set of overlapping traces may be shown equivalent to a particular composite trace.

The relevance of category $\delta^n$ is that it allows a theorem to be developed which relates the properties of category $\delta$ over disjunction.

**Theorem 6.6:** Consider $n$ predicates, $R_1$, $R_2$, .., $R_n$, which belong to category $\delta$. Then the logical disjunction of these is such that

$$R_1 \vee R_2 \vee .. \vee R_n \in \delta^n \qquad \text{Eq. 6.50}$$

■

*Proof*

To prove this result the following Lemma is required

**Lemma 6.6(a):** For positive integers $i$, $j$ let $Q_j^i$ be a statement in propositional calculus. For every positive integer $m$ the following equivalence holds:

$$\bigwedge_{i=1}^{m} \left( \bigvee_{j=1}^{m-1} (Q_1^j \wedge .. \wedge Q_{i-1}^j \wedge Q_{i+1}^j \wedge .. \wedge Q_m^j) \right)$$

$$\Leftrightarrow \quad \bigvee_{j=1}^{m-1} ( Q_1^j \wedge Q_2^j \wedge .. \wedge Q_m^j ) \qquad \text{Eq. 6.51}$$

■

*Proof (of Lemma 6.6(a))*

This Lemma is proved in Appendix C. □

Turning to the main proof let $r_1$, $r_2$, .. ,$r_{n+1}$ be non-empty traces. Represent the logical disjunction of predicates $R_1$, $R_2$,.., $R_n$ by

$$\Re = R_1 \vee R_2 \vee .. \vee R_n \qquad \text{Eq. 6.52}$$

Consider the expression

$$\bigwedge_{i=1}^{n+1} \Re(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i-1}\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.53}$$

$$\Leftrightarrow \bigwedge_{i=1}^{n+1} (R_1(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i-1}\hat{}..\hat{}r_{n+1}) \vee R_2(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i-1}\hat{}..\hat{}r_{n+1}) \vee .. \vee R_n(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i-1}\hat{}..\hat{}r_{n+1})) \qquad \text{Eq. 6.54}$$

By definition, for all predicates $R$ in category $\delta$

$$R(r_1\hat{}..\hat{}r_{n+1}) \Leftrightarrow R(r_1) \wedge R(r_2) \wedge .. \wedge R(r_{n+1}) \qquad \text{Eq. 6.55}$$

Therefore

$$\begin{aligned} \text{Eq. 6.54} \Leftrightarrow \bigwedge_{i=1}^{n+1} & ((R_1(r_1) \wedge .. \wedge R_1(r_{i-1}) \wedge R_1(r_{i+1}) \wedge .. \wedge R_1(r_{n+1})) \\ & \vee (R_2(r_1) \wedge .. \wedge R_2(r_{i-1}) \wedge R_2(r_{i+1}) \wedge .. \wedge R_2(r_{n+1})) \\ & \vee .. \vee (R_n(r_1) \wedge .. \wedge R_n(r_{i-1}) \wedge R_n(r_{i+1}) \wedge .. \wedge R_n(r_{n+1}))) \end{aligned} \qquad \text{Eq. 6.56}$$

By making the substitution $Q_j^i = R_i(r_j)$

$$\begin{aligned} \text{Eq. 6.54} \Leftrightarrow \bigwedge_{i=1}^{n+1} & ((Q_1^1 \wedge .. \wedge Q_{i-1}^1 \wedge Q_{i+1}^1 \wedge .. \wedge Q_{n+1}^1) \\ & \vee (Q_1^2 \wedge .. \wedge Q_{i-1}^2 \wedge Q_{i+1}^2 \wedge .. \wedge Q_{n+1}^2) \\ & \vee .. \vee (Q_1^n \wedge .. \wedge Q_{i-1}^n \wedge Q_{i+1}^n \wedge .. \wedge Q_{n+1}^n)) \end{aligned} \qquad \text{Eq. 6.57}$$

Using Lemma 6.6(a) for the value $m = n+1$ yields

$$\begin{aligned} \text{Eq. 6.54} \Leftrightarrow & ( Q_1^1 \wedge Q_2^1 \wedge .. \wedge Q_{n+1}^1 ) \vee ( Q_1^2 \wedge Q_2^2 \wedge .. \wedge Q_{n+1}^2 ) \\ & \vee .. \vee ( Q_1^n \wedge Q_2^n \wedge .. \wedge Q_{n+1}^n ) \end{aligned} \qquad \text{Eq. 6.58}$$

Re-substituting $R_i(r_j) = Q_j^i$ gives

$$\begin{aligned} \text{Eq. 6.54} \Leftrightarrow & (R_1(r_1) \wedge R_1(r_2) \wedge .. \wedge R_1(r_{n+1})) \\ & \vee (R_2(r_1) \wedge R_2(r_2) \wedge .. \wedge R_2(r_{n+1})) \\ & \vee .. \vee (R_n(r_1) \wedge R_n(r_2) \wedge .. \wedge R_n(r_{n+1})) \end{aligned} \qquad \text{Eq. 6.59}$$

$$\Leftrightarrow R_1(r_1\hat{}r_2\hat{}..\hat{}r_{n+1}) \vee R_2(r_1\hat{}r_2\hat{}..\hat{}r_{n+1}) \vee .. \vee R_n(r_1\hat{}r_2\hat{}..\hat{}r_{n+1})$$

Eq. 6.60

$$\Leftrightarrow \quad \Re(\mathtt{r}_1\hat{}\mathtt{r}_2\hat{}..\hat{}\mathtt{r}_{n+1}) \qquad \text{Eq. 6.61}$$

Therefore the compound predicate $\Re$ is such that

$$\Re \in \delta^n \qquad \text{Eq. 6.62}$$

□

## 6.8 Logical Disjunction across the Categories.

The previous text has shown that it is possible to come up with an expression which relates the behaviour of certain predicates $\mathtt{R}_1\vee\mathtt{R}_2\vee..\vee\mathtt{R}_n$ over catenation. That is, consider a general statement $\Re$

$$\Re \quad \equiv \quad \mathtt{R}_1\vee\mathtt{R}_2\vee..\vee\mathtt{R}_n \qquad \text{Eq. 6.63}$$

where $\mathtt{R}_1, \mathtt{R}_2, .., \mathtt{R}_n$ are predicates and ( $\mathtt{R}_1, \mathtt{R}_2, .., \mathtt{R}_n \in \alpha$ ) or ( $\mathtt{R}_1, \mathtt{R}_2, .., \mathtt{R}_n \in \beta$ ) or ( $\mathtt{R}_1, \mathtt{R}_2, .., \mathtt{R}_n \in \gamma$ ) or ( $\mathtt{R}_1, \mathtt{R}_2, .., \mathtt{R}_n \in \delta$ ). Then it is possible to show that $\Re$ belongs to one of $\alpha$, $\beta$, $\gamma$ or $\delta^n$. Thus properties of $\Re$ over catenation are known. However, it has not yet been determined to what extent the predicate categories are closed amongst themselves. That is, what properties can be inferred about a statement of the form

$$\mathtt{R}_1\vee\mathtt{R}_2\vee..\vee\mathtt{R}_n \wedge (\forall\ \mathtt{i} \bullet \mathtt{R}_\mathtt{i} \in \{\alpha \cup \beta \cup \gamma \cup \delta\}) \qquad \text{Eq. 6.64}$$

where $\mathtt{R}_1, \mathtt{R}_2, .., \mathtt{R}_n$ are predicates

The categories so far introduced have all stipulated that a predicate will satisfy some composite trace *if and only if* it satisfies some set of lesser constituent traces. The *if and only if* condition is required to make the categories strong enough to be closed under logical negation. The same condition, however, serves to restrict the scope of closures over logical disjunction. It was seen from Theorem 6.1 that weakening the if and only if condition resulted in equation 6.20 which encompassed all the defined categories. This idea is extended by the following definition.

**Definition 6.6:** The category $\sigma^n$ is such that

$$\sigma^n = \{ \mathtt{R} \mid \forall\, \mathtt{r}_1,..,\mathtt{r}_{n+1} \neq \langle\rangle \quad \bullet \bigwedge_{i=1}^{n+1} \mathtt{R}(\mathtt{r}_1\text{^}..\text{^}\mathtt{r}_{i-1}\text{^}\mathtt{r}_{i+1}\text{^}..\text{^}\mathtt{r}_{n+1}) \Rightarrow \mathtt{R}(\mathtt{r}_1\text{^}..\text{^}\mathtt{r}_{n+1}) \} \qquad \text{Eq. 6.65}$$

■

Note how category $\sigma^n$ is equivalent to equation 6.20 when $\mathtt{n} = 1$. Equation 6.20 described a property of catenation which was common to each predicate in the set $\{\alpha \cup \beta \cup \gamma \cup \delta\}$. Category $\sigma^n$ has been introduced here with the aim of providing a general statement about the catenation properties of predicates which lie outside this set. Specifically those predicates represented by equation 6.64. This constitutes a relatively broad category of predicates in comparison to the set to which equation 6.20 could be applied. The rest of this section outlines the justification for stating that $\sigma^n$ has this scope. To achieve this it is first necessary to investigate some of the properties of $\sigma^n$.

**Theorem 6.7:** For the definition of the set $\sigma^n$, the following properties hold

i) $\alpha, \beta, \gamma, \delta$ are all subsets of $\sigma^1$ Eq. 6.66

ii) $\delta^n \subseteq \sigma^n$ Eq. 6.67

■

*Proof*

(i) By definition, for traces $\mathtt{r}_1, \mathtt{r}_2$

$$\sigma^1 \equiv \{\mathtt{R} \mid \forall\, \mathtt{r}_1, \mathtt{r}_2 \neq \langle\rangle \bullet \mathtt{R}(\mathtt{r}_1) \wedge \mathtt{R}(\mathtt{r}_2) \Rightarrow \mathtt{R}(\mathtt{r}_1\text{^}\mathtt{r}_2)\} \qquad \text{Eq. 6.68}$$

The result follows from Theorem 6.1

(ii) This follows directly from the definitions of $\delta^n$ and $\sigma^n$. □

Consider now the disjunction of two predicates, one in category $\alpha$, the other in category $\beta$. The following theorem shows how he definition of $\sigma^n$ can determine some of the properties of this disjunction over catenation.

**Theorem 6.8:** Let $\mathtt{Q}$ and $\mathtt{R}$ be predicates such that:

$$\mathtt{Q} \in \alpha \qquad \& \qquad \mathtt{R} \in \beta \qquad \text{Eq. 6.69}$$

The logical disjunction of Q and R, Q∨R, is such that

$$\mathrm{Q{\vee}R} \in \sigma^2 \qquad \text{Eq. 6.70}$$

■

*Proof*

Let $r_1$, $r_2$, $r_3$ be non-empty traces and consider the expression

$$\bigwedge_{i=1}^{3} \mathrm{Q{\vee}R}(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_3) \qquad \text{Eq. 6.71}$$

$$\Leftrightarrow \quad \mathrm{Q{\vee}R}(r_2\hat{}r_3) \wedge \mathrm{Q{\vee}R}(r_1\hat{}r_3) \wedge \mathrm{Q{\vee}R}(r_1\hat{}r_2) \qquad \text{Eq. 6.72}$$

$$\Leftrightarrow \quad (\mathrm{Q}(r_2\hat{}r_3) \vee \mathrm{R}(r_2\hat{}r_3)) \;\wedge\; (\mathrm{Q}(r_1\hat{}r_3) \vee \mathrm{R}(r_1\hat{}r_3)) \;\wedge\; (\mathrm{Q}(r_1\hat{}r_2) \vee \mathrm{R}(r_1\hat{}r_2)) \qquad \text{Eq. 6.73}$$

Because $\mathrm{Q} \in \alpha$ and $\mathrm{R} \in \beta$, the middle term in equation 6.73 is such that

$$\text{Eq. 6.71} \Rightarrow \quad (\mathrm{Q}(r_1\hat{}r_3) \vee \mathrm{R}(r_1\hat{}r_3)) \quad \Leftrightarrow \quad \mathrm{Q}(r_1) \vee \mathrm{R}(r_3) \qquad \text{Eq. 6.74}$$

$$\Leftrightarrow \quad \mathrm{Q}(r_1\hat{}r_2\hat{}r_3) \vee \mathrm{R}(r_1\hat{}r_2\hat{}r_3) \quad \Leftrightarrow \quad \mathrm{Q{\vee}R}(r_1\hat{}r_2\hat{}r_3) \qquad \text{Eq. 6.75}$$

Therefore, from the definition of $\sigma^n$ when $n = 2$, $\mathrm{Q{\vee}R} \in \sigma^2$ □

Theorems 6.9 and 6.10 show how a predicate from the general category $\delta^n$ can be disjoined with a predicate from another category.

**Theorem 6.9:** Let Q and R be predicates such that:

$$\mathrm{Q} \in \delta^n \qquad \& \qquad \mathrm{R} \in \alpha. \qquad \text{Eq. 6.76}$$

The logical disjunction of Q and R, Q∨R, is such that

$$\mathrm{Q{\vee}R} \in \sigma^{n+1} \qquad \text{Eq. 6.77}$$

■

*Proof*

By the definition of the predicates Q and R, for all non-empty traces $r_1, .., r_{n+2}, t$

$$\bigwedge_{i=1}^{n+1} Q(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \quad \Leftrightarrow \quad Q(r_1\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.78}$$

$$\forall t \quad \bullet \quad R(r_1) \quad \Leftrightarrow \quad R(r_1\hat{}t) \qquad \text{Eq. 6.79}$$

Consider the expansion of the below expression

$$\bigwedge_{i=1}^{n+2} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.80}$$

The proof is split into two cases.

*Case $R(r_1)$ is true*

From equation 6.79 the truth of $R(r_1)$ implies the truth of $R(r_1\hat{}..\hat{}r_{n+2})$, and thus

$$R(r_1) \wedge \bigwedge_{i=1}^{n+2} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \Rightarrow Q{\vee}R(r_1\hat{}..\hat{}r_{n+2}) \quad \text{Eq. 6.81}$$

*Case $R(r_1)$ is false*

The expansion of equation 6.80 is such that

$$\text{Eq. 6.80} \Leftrightarrow (Q(r_2\hat{}..\hat{}r_{n+2}) \vee R(r_2)) \wedge \bigwedge_{i=2}^{n+2} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.82}$$

$$\text{Eq. 6.80} \Rightarrow \bigwedge_{i=2}^{n+2} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.83}$$

By making the substitution $s_1 = r_1\hat{}r_2$ it is seen that

$$\text{Eq. 6.80} \Rightarrow (Q(r_1\hat{}r_3\hat{}..\hat{}r_{n+2}) \vee R(r_1)) \wedge \bigwedge_{i=3}^{n+2} Q{\vee}R(s_1\hat{}r_3\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.84}$$

By applying equation 6.78 it is possible to deduce that

$$Q(r_1\hat{}r_3\hat{}..\hat{}r_{n+2}) \quad \Rightarrow \quad Q(r_3\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.85}$$

Thus

$$\text{Eq. 6.80} \Rightarrow (Q(r_3\hat{}..\hat{}r_{n+2}) \vee R(r_1)) \\ \wedge \bigwedge_{i=3}^{n+2} Q{\vee}R(s_1\hat{}r_3\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.86}$$

By making the substitutions $s_2 = r_3$, $s_3 = r_4$, ..., $s_{n+1} = r_{n+2}$ and assuming that $R(r_1)$ is false, equation 6.86 becomes

$$\text{Eq. 6.80} \Rightarrow \bigwedge_{i=1}^{n+1} Q(s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. 6.87}$$

Applying equation 6.78 and re-substituting,

$$\text{Eq. 6.80} \Rightarrow Q(s_1\hat{}..\hat{}s_{n+1}) \Leftrightarrow Q(r_1\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.88}$$

Therefore

$$(\neg R(r_1)) \wedge \bigwedge_{i=1}^{n+2} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \\ \Rightarrow Q{\vee}R(r_1\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.89}$$

Resolving equations 6.81 and 6.89 yields

$$\bigwedge_{i=1}^{n+2} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \Rightarrow Q{\vee}R(r_1\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. 6.90}$$

Thus by definition $Q{\vee}R \in \sigma^{n+1}$ □

**Theorem 6.10:** Let $Q$ and $R$ be predicates such that:

$$Q \in \delta^n \qquad \& \qquad R \in \beta \qquad \text{Eq. 6.91}$$

The logical disjunction of $Q$ and $R$, $Q{\vee}R$, is such that

$$Q \vee R \in \sigma^{n+1} \qquad \text{Eq. 6.92}$$

■

*Proof*

The proof of this theorem follows in a similar manner as that of the previous theorem and is presented in Appendix C. □

Theorem 6.11 shows how predicates from three distinct classes can be logically disjoined to form a predicate in the category $\sigma^n$.

**Theorem 6.11:** Let Q, R and S be predicates such that:

$$Q \in \delta^n \quad \& \quad R \in \alpha \quad \& \quad S \in \beta \qquad \text{Eq. 6.93}$$

The logical disjunction of Q, R and S, $Q \vee R \vee S$, is such that

$$Q \vee R \vee S \in \sigma^{n+2} \qquad \text{Eq. 6.94}$$

■

*Proof*

The proof of this theorem is similar to that for Theorems 6.9 and 6.10 and is presented in the Appendix C. □

Theorem 6.12 shows that the category $\gamma$ is so weak that a predicate from it can be disjoined to one in the category $\sigma^n$ ($n \geq 1$) and result in a predicate which is still in category $\sigma^n$.

**Theorem 6.12:** Let Q and R be predicates such that:

$$Q \in \sigma^n \quad \& \quad R \in \gamma \qquad \text{Eq. 6.95}$$

The logical disjunction of Q and R, $Q \vee R$, is such that

$$Q \vee R \in \sigma^n \qquad \text{Eq. 6.96}$$

■

*Proof*

Let $\mathtt{r}_1, \mathtt{r}_2, .., \mathtt{r}_{n+1}$ be arbitrary non-empty traces. By definition predicates Q and R are such that

$$\bigwedge_{i=1}^{n+1} Q(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \quad \Rightarrow \quad Q(r_1\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.97}$$

$$R(r_1\hat{}r_2\hat{}..\hat{}r_{n+1}) \quad \Leftrightarrow \quad R(r_1) \vee R(r_2) \vee .. \vee R(r_{n+1}) \qquad \text{Eq. 6.98}$$

To prove that $Q \vee R \in \sigma^n$ the proof itself is split into two cases

*Case $R(r_1\hat{}..\hat{}r_{n+1})$ is true*

Therefore

$$R(r_1\hat{}r_2\hat{}..\hat{}r_{n+1}) \wedge Q\vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \Rightarrow Q\vee R(r_1\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.99}$$

*Case $R(r_1\hat{}..\hat{}r_{n+1})$ is false*

From equation 6.98

$$\neg R(r_1\hat{}r_2\hat{}..\hat{}r_{n+1}) \quad \Leftrightarrow \quad \neg R(r_1) \wedge \neg R(r_2) \wedge .. \wedge \neg R(r_{n+1}) \qquad \text{Eq. 6.100}$$

From equation 6.100 and the definition of category $\gamma$

$$\forall\ s \in \{r \mid \bigvee_{i=1}^{n+1} (r = r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1})\} \bullet \neg R(s) \qquad \text{Eq. 6.101}$$

Consider the expansion of the expression

$$\bigwedge_{i=1}^{n+1} Q\vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.102}$$

$$\Rightarrow \quad \bigwedge_{i=1}^{n+1} (Q(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1})) \qquad \text{Eq. 6.103}$$

Using equation 6.101

$$\text{Eq. 6.102} \Rightarrow \bigwedge_{i=1}^{n+1} Q(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.104}$$

Using equation 6.97

$$\text{Eq. 6.104} \Rightarrow Q(r_1\hat{}..\hat{}r_{n+1}) \Rightarrow \quad Q{\vee}R(r_1\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.105}$$

Therefore

$$R(r_1\hat{}r_2\hat{}..\hat{}r_{n+1}) \wedge \bigwedge_{i=1}^{n+1} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \\ \Rightarrow Q{\vee}R(r_1\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.106}$$

Combining equations 6.99 and 6.106

$$\bigwedge_{i=1}^{n+1} Q{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \Rightarrow Q{\vee}R(r_1\hat{}..\hat{}r_{n+1}) \qquad \text{Eq. 6.107}$$

And thus $Q{\vee}R \in \sigma^n$ □

The result of the last four theorems taken as a whole is twofold. First it shows that the logical disjunction of a finite number of predicates from the set $\alpha \cup \beta \cup \gamma \cup \delta$ lies within the category $\sigma^n$ for some value $n$. Thus such a disjunction can be shown to have certain properties over catenation. Secondly the theorems are precise enough to allow the calculation of a lowest possible value $x$ such that the predicate lies in the class $\sigma^x$. This is expressed more formally by the following equation.

**Theorem 6.13:** For all integers $i$, let $R_i^\alpha$, $R_i^\beta$, $R_i^\gamma$, $R_i^\delta$, be predicates such that

$$R_i^\alpha \in \alpha \qquad R_i^\beta \in \beta \qquad R_i^\gamma \in \gamma \qquad R_i^\delta \in \delta \qquad \text{Eq. 6.108}$$

Consider the expression

$$\Re = \bigvee_{i=1}^{a} R_i^{\alpha} \vee \bigvee_{i=1}^{b} R_i^{\beta} \vee \bigvee_{i=1}^{c} R_i^{\gamma} \vee \bigvee_{i=1}^{n} R_i^{\delta} \qquad \text{Eq. 6.109}$$

where a, b, c, n are non-negative integers. Let the expression x be defined by

$$x = n + [\, a \neq 0 \,] + [\, b \neq 0 \,] + [\, c \neq 0 \wedge a = 0 \wedge b = 0 \wedge n = 0 \,] \qquad \text{Eq. 6.110}$$

(where for a statement Q the expression [Q] has the value 1 if Q is true and the value 0 if Q is false). The compound predicate $\Re$ is such that

$$\Re \in \sigma^{x} \qquad \text{Eq. 6.111}$$

■

*Proof*

By Theorem 6.5, the sets α, β and γ are closed under logical disjunction. Therefore, for all values a, b, c, n ≥ 1 it can be seen that

$$\bigvee_{i=1}^{a} R_i^{\alpha} \in \alpha \quad \& \quad \bigvee_{i=1}^{b} R_i^{\beta} \in \beta \quad \& \quad \bigvee_{i=1}^{c} R_i^{\gamma} \in \gamma \qquad \text{Eq. 6.112}$$

Additionally, from Theorem 6.6

$$\bigvee_{i=1}^{n} R_i^{\delta} \in \delta^{n} \qquad \text{Eq. 6.113}$$

The theorem is proved by showing that $\Re \in \sigma^{x}$ for every possible case where one or more of the variables a, b, c, n is zero.. There are fifteen cases in all, excluding the trivial instance when a = b = c = n = 0. Two cases are considered here, the rest follow in a similar manner.

*Case $a \neq 0$, $b \neq 0$, $c \neq 0$, $n \neq 0$*

Evaluating x for this yields

$$x = n + [\text{TRUE}] + [\text{TRUE}] + [\text{FALSE}] = n{+}2 \qquad \text{Eq. 6.114}$$

By Theorem 6.11

$$\bigvee_{i=1}^{a} R_i^{\alpha} \vee \bigvee_{i=1}^{b} R_i^{\beta} \vee \bigvee_{i=1}^{n} R_i^{\delta} \in \sigma^{n+2} \qquad \text{Eq. 6.115}$$

And by Theorem 6.12

$$\bigvee_{i=1}^{a} R_i^{\alpha} \vee \bigvee_{i=1}^{b} R_i^{\beta} \vee \bigvee_{i=1}^{n} R_i^{\delta} \vee \bigvee_{i=1}^{c} R_i^{\gamma} \in \sigma^{n+2} \qquad \text{Eq. 6.116}$$

Therefore $\Re \in \sigma^{x}$.

*Case* $a = b = n = 0$*, and* $c \neq 0$
Evaluating x for this yields

$$x = 0 + [\text{FALSE}] + [\text{FALSE}] + [\text{TRUE}] = 1 \qquad \text{Eq. 6.117}$$

By inspection $\Re$ is in category $\gamma$. By Theorem 6.5 category $\gamma$ is closed under logical disjunction. Thus for all $c \neq 0$

$$\bigvee_{i=1}^{c} R_i^{\gamma} \in \sigma^{1} \qquad \text{Eq. 6.118}$$

Therefore $\Re \in \sigma^{x}$.

The remaining cases show that for all values of a, b, c, n the predicate $\Re$ is such that

$$\Re \in s^{(n + [a \neq 0] + [b \neq 0] + [c \neq 0 \wedge a = 0 \wedge b = 0 \wedge n = 0])} \qquad \text{Eq. 6.119}$$

□

The result of Theorem 6.13 is that given a predicate $\Re$ which is formed from the logical disjunction of a number of predicates from the set $\{\alpha \cup \beta \cup \gamma \cup \delta\}$ it is always possible to calculate an integer value x such that $\Re \in \sigma^{x}$. In turn this means that for all predicates of the type $\Re$ there always exists an equation which relates to the distribution of $\Re$ over trace catenation.

Taking this finding to the language RSPEC means that it can now be seen that every predicate in RSPEC can be written as

$$\bigwedge_{i=1}^{k} Q_i \qquad \text{Eq. 6.120}$$

where k is an integer and $Q_i$ is a compound predicate such that, for every i, there exists a value $x_i$ ( which can be calculated from Theorem 6.13) such that

$$Q_i \in \sigma^{x_i} \qquad \text{Eq. 6.121}$$

## 6.9 Using the General Category $\sigma^n$ to Construct an Ideal Test Set.

Theorem 6.2 illustrated how the category $\sigma^1$ could be used to construct an Ideal Test Set for a process recursively defined by a Catenary function. In turn it is possible to show that the general category $\sigma^n$ can be used to construct an Ideal Test Set for a similar process. This is illustrated by the following theorem.

**Theorem 6.14:** Let F be a Catenary function and let R be a predicate such that

$$R \in \sigma^n \qquad \text{Eq. 6.122}$$

where n is a non-zero positive integer. The set of traces defined by

$$\mathit{Traces}(F^n(STOP)) \qquad \text{Eq. 6.123}$$

is an Ideal Test Set for the pair (μP.F(P), R). ■

*Proof*

This proof requires the definition of $\mathcal{D}_{\mathcal{F}}$ from Chapter 5. The theorem is proved by mathematical induction.

INDUCTIVE STEP

Assume that there exists some positive, non-zero integer m, where m ≥ n, such that the predicate R is true for all traces t of $F^m(STOP)$. That is

$$\forall\, t \in \mathit{Traces}(F^m(STOP)) \bullet R(t) \qquad \text{Eq. 6.124}$$

Let s be any trace in the set $\mathit{Traces}(F^{m+1}(STOP))$. Then there is some non-zero positive integer $p \le m+1$ such that

$$s = r_1\hat{}..\hat{}r_p \qquad \text{Eq. 6.125}$$

where $r_1,..,r_{p-1} \in \mathcal{D}_{\mathcal{F}}$ and $r_p \in Traces(F(STOP))$. The proof divides into two cases

*Case* $p < m+1$

If $p < m+1$ then $p \leq m$. By Theorem 5.15 $s \in Traces(F^m(STOP))$. Therefore $R(s)$ holds from the initial assumption.

*Case* $p = m+1$

If $p = m+1$ then

$$s = r_1\hat{}r_2\hat{}\,..\hat{}r_{m+1} \qquad \text{Eq. 6.126}$$

By substituting

$$s_1 = r_1\hat{}..\hat{}r_{(m-n)+1} \quad s_2 = r_{(m-n)+2}\ .. \qquad s_{n+1} = r_{m+1} \qquad \text{Eq. 6.127}$$

It is seen that

$$s = s_1\hat{}s_2\hat{}\,..\hat{}s_{n+1} \qquad \text{Eq. 6.128}$$

Additionally, by Theorem 5.15

$$\forall\ 1 \leq i \leq n+1 \bullet s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n+1} \in Traces(F^m(STOP)) \qquad \text{Eq. 6.129}$$

$$\therefore \quad \forall\ 1 \leq i \leq n+1 \bullet R(s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. 6.130}$$

$R \in \sigma^n$, therefore by definition

$$\forall\ s_1,.., s_{n+1} \neq \langle\rangle \bullet \bigwedge_{i=1}^{n+1} R(s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n+1}) \Rightarrow R(s_1\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. 6.131}$$

The antecedent of equation 6.131 holds from equation 6.130. Therefore

$$R(s_1\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. 6.132}$$

Therefore predicate R is true for every trace in *Traces*($F^{m+1}$(STOP)). Therefore if R is true over the set *Traces*($F^m$(STOP)) then R is true over the set *Traces*($F^{m+1}$(STOP)). This establishes the inductive step.

BASE CASE

The base case for $m = n$ can be established by testing the Ideal Test Set *Traces*($F^n$(STOP)).

Therefore if R is true for every trace in *Traces*($F^n$(STOP)), then by induction it can be seen that for all values $i \in \mathbb{N}$

$$\forall s \in \mathit{Traces}(F^i(\mathrm{STOP})) \bullet R(s) \qquad \text{Eq. 6.133}$$

Thus *Traces*($F^n$(STOP)) is an Ideal Test Set for the pair ($\mu P.F(P)$, R) □

## 6.10 Introducing Logical Conjunction.

Up to this point it has been shown that an Ideal Test Set may be generated for all predicates of the form

$$\bigvee_{j=1}^{m} (Q_{ij}) \qquad \text{Eq. 6.136}$$

where every $Q_{ij}$ or $\neg(Q_{ij})$ is a predicate from the set $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$. The Ideal Test Sets produced for predicates of the form of equation 6.136 shall be termed provisional Ideal Test Sets in this thesis, since they represent only a part of the ultimate Ideal Test Set. However the aim of this work is to generate an Ideal Test Set for all predicates in conjunctive normal form. To achieve this it is necessary to consider the effect of combining predicates under the conjunction connective.

Predicates were combined under disjunction by developing a set of rules which related to the behaviour of composite predicates over catenation. In turn this information was used to construct Ideal Test Sets. However, with conjunction there is no need to develop similar rules concerning the distribution of predicates over catenation. Instead it has been found that the structure of conjunction allows the Ideal Test Sets themselves to be manipulated. This is illustrated by the following theorem.

**Theorem 6.16:** Let $\mathtt{P}$ be a CSP process and let $\mathtt{Q}$, $\mathtt{R}$ be predicates on that process. Let the sets $\mathtt{I}_1$, $\mathtt{I}_2$ be Ideal Test Sets for the pairs ($\mathtt{P}$, $\mathtt{Q}$) and ($\mathtt{P}$, $\mathtt{R}$) respectively. If

$$\mathtt{I}_1 \supseteq \mathtt{I}_2 \qquad \text{Eq. 6.137}$$

then an Ideal Test Set for the pair ($\mathtt{P}$, $\mathtt{Q}{\wedge}\mathtt{R}$) is $\mathtt{I}_1$. Conversely if $\mathtt{I}_1 \subseteq \mathtt{I}_2$ an Ideal Test Set for the pair ($\mathtt{P}$, $\mathtt{Q}{\wedge}\mathtt{R}$) is $\mathtt{I}_2$. ■

*Proof*

Let $\mathtt{Obs}$ be the set of all behaviours exhibited by process $\mathtt{P}$ and assume that $\mathtt{I}_1 \supseteq \mathtt{I}_2$. From the definition of an Ideal Test Set,

$$\mathtt{Obs} \supseteq \mathtt{I}_1 \supseteq \mathtt{I}_2 \qquad \text{Eq. 6.138}$$

$\mathtt{I}_1$ and $\mathtt{I}_2$ are Ideal Test Sets for ($\mathtt{P}$, $\mathtt{R}$) and ($\mathtt{P}$, $\mathtt{Q}$) respectively, therefore

$$(\forall\, \mathtt{t} \in \mathtt{I}_1 \bullet \mathtt{R(t)}) \quad \Leftrightarrow \quad (\forall\, \mathtt{u} \in \mathtt{Obs} \bullet \mathtt{R(u)}) \qquad \text{Eq. 6.139}$$

$$(\forall\, \mathtt{t} \in \mathtt{I}_2 \bullet \mathtt{Q(t)}) \quad \Leftrightarrow \quad (\forall\, \mathtt{u} \in \mathtt{Obs} \bullet \mathtt{Q(u)}) \qquad \text{Eq. 6.140}$$

From equation 6.138, $\mathtt{I}_1 \supseteq \mathtt{I}_2$ and $\mathtt{Obs} \supseteq \mathtt{I}_1$ thus

$$(\forall\, \mathtt{t} \in \mathtt{I}_1 \bullet \mathtt{Q(t)}) \quad \Rightarrow \quad (\forall\, \mathtt{t} \in \mathtt{I}_2 \bullet \mathtt{Q(t)}) \qquad \text{Eq. 6.141}$$

$$(\forall\, \mathtt{u} \in \mathtt{Obs} \bullet \mathtt{Q(u)}) \quad \Rightarrow \quad (\forall\, \mathtt{t} \in \mathtt{I}_1 \bullet \mathtt{Q(t)}) \qquad \text{Eq. 6.142}$$

Therefore from equations 6.139 - 6.141,

$$(\forall\, \mathtt{t} \in \mathtt{I}_1 \bullet \mathtt{Q(t)}) \quad \Leftrightarrow \quad (\forall\, \mathtt{u} \in \mathtt{Obs} \bullet \mathtt{Q(u)}) \qquad \text{Eq. 6.143}$$

$$\therefore \quad (\forall\, \mathtt{t} \in \mathtt{I}_1 \bullet \mathtt{R(t)}) \wedge (\forall\, \mathtt{t} \in \mathtt{I}_1 \bullet \mathtt{Q(t)})$$
$$\Leftrightarrow (\forall\, \mathtt{u} \in \mathtt{Obs} \bullet \mathtt{R(u)}) \wedge (\forall\, \mathtt{u} \in \mathtt{Obs} \bullet \mathtt{Q(u)}) \qquad \text{Eq. 6.144}$$

$$\therefore \quad (\forall\, \mathtt{t} \in \mathtt{I}_1 \bullet \mathtt{R}{\wedge}\mathtt{Q(t)}) \Leftrightarrow (\forall\, \mathtt{u} \in \mathtt{Obs} \bullet \mathtt{R}{\wedge}\mathtt{Q(u)}) \qquad \text{Eq. 6.145}$$

So $\mathtt{I}_1$ is an Ideal Test Set for ($\mathtt{P}$, $\mathtt{R}{\wedge}\mathtt{Q}$). The proof for the case $\mathtt{I}_1 \subseteq \mathtt{I}_2$ follows in a similar manner. □

It can be seen from this theorem how the need to know the properties of a predicate over catenation is obviated when considering the conjunction connective. The corollary of this theorem is that if F is a Catenary function *and* the sets $I_1,..., I_n$ are Ideal Test Sets for the pairs ($\mu$P.F(P), $R_1$),..,($\mu$P.F(P), $R_n$) respectively *and* one of the sets $I_1,..., I_n$ is a superset of all the others, then that set is an Ideal Test Set.

Provided Ideal Test Sets are written in the general form

$$Traces(F^u(STOP) \qquad \text{Eq. 6.146}$$

where u is a positive integer, then it is possible to show from the partial ordering given in Appendix A that for any two Ideal Test Sets $I_1$, $I_2$ one is always a superset of the other. As a consequence for any such set $I_1,..,I_n$ of Ideal Test Sets there will always be at least one which is a superset of all the others. That is

$$\exists m \in \mathbb{N} \bullet (m \leq n \ \wedge \ (\forall 0 \leq i \leq n \bullet I_m \supseteq I_i )) \qquad \text{Eq. 6.147}$$

The theory presented in this chapter is now sufficiently mature to show how an Ideal Test Set can be generated for every pair ($\mu$P.F(P)), $\mathfrak{R}$), where F is a Catenary function and $\mathfrak{R}$ is a predicate in RSPEC. The following theorem describes the mechanism for generating an Ideal Test Set.

**Theorem 6.17:** Let F be a Catenary function. For all non-zero positive integers i, j let

$$Rj_i^{\alpha} \in \alpha \qquad Rj_i^{\beta} \in \beta \qquad Rj_i^{\gamma} \in \gamma \qquad Rj_i^{\delta} \in \delta \qquad \text{Eq. 6.148}$$

Furthermore, let $\mathfrak{R}_j$ be a compound predicate such that

$$\mathfrak{R}_j = \bigvee_{i=1}^{aj} Rj_i^{\alpha} \vee \bigvee_{i=1}^{bj} Rj_i^{\beta} \vee \bigvee_{i=1}^{cj} Rj_i^{\gamma} \vee \bigvee_{i=1}^{nj} Rj_i^{\delta} \qquad \text{Eq. 6.149}$$

where, for all non-zero positive integers j, the values $a_j$, $b_j$, $c_j$, $n_j$ are positive integers. Let $x_j$ be defined by

$$\mathtt{x}_\mathtt{j} = \mathtt{n}_\mathtt{j} + [\mathtt{a}_\mathtt{j} \neq 0] + [\mathtt{b}_\mathtt{j} \neq 0] + [\mathtt{c}_\mathtt{j} \neq 0 \wedge \mathtt{a}_\mathtt{j} = 0 \wedge \mathtt{b}_\mathtt{j} = 0 \wedge \mathtt{c}_\mathtt{j} = 0] \qquad \text{Eq. 6.150}$$

Let the predicate $\Re$ be defined by

$$\Re \equiv \Re_1 \wedge \Re_2 \wedge .. \Re_\mathtt{k} \qquad \text{Eq. 6.151}$$

where $\mathtt{k}$ is an integer. Note that equation 6.151 is in conjunctive normal form and that every predicate in RSPEC can be expressed in this form. An Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, $\Re$) is

$$\mathit{Traces}(\mathtt{F}^{\mathtt{xmax}}(\mathtt{STOP})), \qquad \mathtt{xmax} = \max\{\mathtt{x}_\mathtt{j} \mid \mathtt{j} = 1 \text{ to } \mathtt{k}\} \qquad \text{Eq. 6.152}$$

■

*Proof*

By Theorem 6.13, for all predicates of the form $\Re_\mathtt{j}$ there is a value $\mathtt{x}_\mathtt{j}$ such that

$$\Re_\mathtt{j} \in \sigma^{\mathtt{x}_\mathtt{j}} \qquad \text{Eq. 6.153}$$

where $\mathtt{x}_\mathtt{j}$ is given by the equation

$$\mathtt{x}_\mathtt{j} = \mathtt{n}_\mathtt{j} + [\mathtt{a}_\mathtt{j} \neq 0] + [\mathtt{b}_\mathtt{j} \neq 0] + [\mathtt{c}_\mathtt{j} \neq 0 \wedge \mathtt{a}_\mathtt{j} = 0 \wedge \mathtt{b}_\mathtt{j} = 0 \wedge \mathtt{c}_\mathtt{j} = 0] \qquad \text{Eq. 6.154}$$

Consequently, by Theorem 6.14, for every $\mathtt{j}$ ( $0 \leq \mathtt{j} \leq \mathtt{k}$ ) the set $\mathit{Traces}(\mathtt{F}^{\mathtt{x}_\mathtt{j}}(\mathtt{STOP}))$ is an Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, $\Re_\mathtt{j}$).

Consider the set of Ideal Test Sets given by

$$\{\mathit{Traces}(\mathtt{F}^{\mathtt{x}_1}(\mathtt{STOP})), .., \mathit{Traces}(\mathtt{F}^{\mathtt{x}_\mathtt{k}}(\mathtt{STOP}))\} \qquad \text{Eq. 6.155}$$

Let $\mathtt{xmax}$ be the maximum element of the set $\{\mathtt{x}_1, .., \mathtt{x}_\mathtt{k}\}$. It is possible to show from the partial ordering over CSP processes given in Appendix A that

$$\forall\, \mathtt{u}, \mathtt{v} \in \mathbb{N} \bullet (\mathtt{u} \geq \mathtt{v} \Rightarrow \mathit{Traces}(\mathtt{F}^{\mathtt{u}}(\mathtt{STOP})) \supseteq \mathit{Traces}(\mathtt{F}^{\mathtt{v}}(\mathtt{STOP}))) \qquad \text{Eq. 6.156}$$

It follows from the definition of $\mathtt{xmax}$ that

$$\forall\, 1 \leq \mathtt{j} \leq \mathtt{k} \bullet (\mathit{Traces}(\mathtt{F}^{\mathtt{xmax}}(\mathtt{STOP})) \supseteq \mathit{Traces}(\mathtt{F}^{\mathtt{j}}(\mathtt{STOP}))) \qquad \text{Eq. 6.157}$$

Theorem 6.16 and its corollaries may now be used to show that $\mathtt{F}^{\mathtt{xmax}}(\mathtt{STOP})$ is an Ideal Test Set for the pair

$$(\mu\mathtt{P}.\mathtt{F}(\mathtt{P}), \mathfrak{R}) \qquad \text{Eq. 6.158}$$

where

$$\mathfrak{R} \equiv \mathfrak{R}_1 \wedge \mathfrak{R}_2 \wedge .. \mathfrak{R}_k \qquad \text{Eq. 6.159}$$

□

## 6.11 Summary.

The main points developed in this chapter are as follows

i) The concept of ideal tests proposed by Goodenough and Gerhart [ Goodenough 75 ] for state-transition systems has been translated into the new concept of Ideal Test Sets for CSP. A formal definition of an Ideal Test Set has been provided by Definition 6.1.

ii) It has been shown that the theoretical foundations for generating an Ideal Test Set can be established by exploiting the link between recursion and catenation exhibited by Catenary functions which was discussed in the previous chapter.

iii) The chapter has introduced the four categories $\alpha$, $\beta$, $\gamma$, $\delta$ into which certain predicates may be classified by virtue of their properties over catenation. The nature of these categories has been utilised to determine rules concerning how compound predicates distribute over catenation.

iv) A predicate language RSPEC has been proposed to facilitate the generation of Ideal Test Sets. It has been shown that the language RSPEC is such that for every pair $(\mu\mathtt{P}.\mathtt{F}(\mathtt{P}),\mathfrak{R})$, where $\mathfrak{R}$ is a predicate of RSPEC and $\mathtt{F}$ is a Catenary function, there exists a well defined procedure for generating an Ideal Test Set.

CHAPTER SEVEN

# A SYNTAX OF RSPEC

## 7.1 Introduction.

The previous chapter proposes a syntax, RSPEC, for describing particular behavioural specifications of CSP processes. It was shown how, given a suitable process defined by a Catenary function, there exists a general procedure for establishing the correctness of behavioural specifications in RSPEC.

However, there remain a number of points which the previous chapter has not fully addressed. First there is the question of providing a precise syntactic definition of RSPEC. Recall that RSPEC relied on the definition of CATEGS as being " The set of syntax of all predicates which belong to categories $\alpha$, $\beta$, $\gamma$, $\delta$.". Although this definition may be sufficient to understand the nature of CATEGS it still does not constitute a precise syntactic definition. Secondly, the expressibility of RSPEC is limited. That is the scope of the behavioural specifications permitted by the composition of predicates from the categories $\alpha$, $\beta$, $\gamma$, $\delta$ is somewhat restricted.

This Chapter has two main objectives. The first is to create a well defined syntax for RSPEC which can be used to capture and prove the correctness of behavioural specifications. To achieve this the Chapter opens by addressing the problems caused by the initial loose definition of the syntactic set CATEGS. It is recognised that in order to provide a proper syntactic definition of RSPEC it will first be necessary to delineate a more precise definition for CATEGS. It is shown how a general set of syntactic rules to indicate inclusion in each of the categories $\alpha$, $\beta$, $\gamma$, $\delta$ can be factorized by studying specific examples of predicates. This leads to a partial syntactic definition of CATEGS and, in turn, to a well defined syntax for RSPEC. The applications of

RSPEC and its role in generating Ideal Test Sets for suitable processes are then demonstrated by a number of examples.

The second aim is to critically analyse the scope that RSPEC has to represent specifications and to subsequently propose and implement an extension to RSPEC which constitutes an improvement. Attention is focused on the scope and limitations of RSPEC in representing behavioural specifications for CSP processes. By discussion and examples it is seen that, while RSPEC is able to capture a number of interesting and useful properties, there is room for improving the method. With this in mind a strategy is proposed for extending the syntax of RSPEC to that of ERSPEC, which is shown to be more expressive.

The Chapter then turns to the specific problem of defining a particular syntax for the extension ERSPEC and resolving this extension with the theory outlined in Chapter Six. The basis for the extension is the use of monotonic trace endomorphisms. By a series of definitions and theorems it is demonstrated how every behavioural specification in ERSPEC can be used to generate an Ideal Test Set for a process defined as the fixed point of a Catenary function.

Finally the chapter summarises these results and draws conclusions about their applications.

## 7.2 Syntax and Semantics of CATEGS.

In the previous chapter, CATEGS was defined as being the set of syntax of all those predicates which fall into one of the four categories $\alpha$, $\beta$, $\gamma$, $\delta$. However, such a loose definition leads to problems in the subsequent development of a syntax for RSPEC. RSPEC is intended to provide a purely syntactic definition for behavioural specifications which can be used for the generation of Ideal Test Sets. It is based directly on the syntax set CATEGS. Inclusion of a predicate in CATEGS depends on membership of one of the categories $\alpha$, $\beta$, $\gamma$, $\delta$. In turn membership of these categories is dependent on the interpretation given to a particular predicate when it is evaluated over all traces. Thus although the set CATEGS is one of syntax, the criteria for inclusion is semantic.

Now, suppose there were a set of syntactic rules equivalent to saying that a predicate belonged to CATEGS. That is a set Z, say, of rules such that a predicate R satisfies Z if and only if R belongs to CATEGS. Z is, as a consequence, also equivalent to the semantic rules which indicate inclusion in the set $\{\ \alpha \cup \beta \cup \gamma \cup \delta\ \}$, such that

$$\mathrm{R} \text{ satisfies } \mathrm{Z} \quad \Leftrightarrow \quad \mathrm{R} \in \{\ \alpha \cup \beta \cup \gamma \cup \delta\ \} \qquad \text{Eq. 7.1}$$

Also, it would be possible to use Z to manufacture a syntactic definition for CATEGS. This gives rise to the following question. Is it feasible to develop a set of syntactic rules Z which are equivalent to the semantic criteria for inclusion in CATEGS?

There are two factors which direct the answer to this question, expediency and compromise. To find Z would involve drawing comparisons between two complementary theories, one of syntax and one of semantics. Such theories already exist for CSP and are well documented [ Roscoe 82, Olderog 86]. However, these specifically relate to an axiomatic proof system for CSP. It is not known how the concepts would translate to the verification system described in the previous chapter, which is predominantly based on the properties of catenation. As a consequence, to have confidence in any results it would be necessary to carry out a detailed investigation into the relationships between the syntax and semantics of this new system. The literature [ Revesz 83, Watt 91 ] readily concedes that the general relationship between syntax and semantics is complex and, as a consequence, analysis would prove difficult and time consuming.

Therefore, a lateral approach to the problem was adopted that yielded an alternative solution which, although not as comprehensive, offered a compromise. This was based on finding a set Z' of syntactic rules which, when true, imply that a predicate belonged to CATEGS.

These rules can be compared with Z. They are such that a predicate R satisfies Z' only if the predicate belongs to CATEGS. That is

$$\mathrm{R} \text{ satisfies } Z' \quad \Rightarrow \quad \mathrm{R} \in \{ \alpha \cup \beta \cup \gamma \cup \delta \} \qquad \text{Eq. 7.2}$$

Note that Z' is a much weaker conceptual set of rules than Z. With Z it was possible to take any predicate R, see if it satisfied Z and thus determine whether or not R was a predicate in { $\alpha \cup \beta \cup \gamma \cup \delta$ }. With Z' it is only possible to take a predicate R and deduce that it belongs to CATEGS if it satisfies Z'. If it does not satisfy Z' then it is unknown whether or not the predicate is in CATEGS.

To summarise, the search for syntactic rules which completely define the set CATEGS was not initiated because it was felt to be prohibitively complex and a feasible alternative had been proposed. This alternative involves finding a weaker set of rules and this proved easier to generate.

The problems of finding such a set of rules Z' are addressed by the following two sections. The tactic adopted is to consider, for each category, specific examples of member predicates. The properties of these member predicates are then analysed and used to generalise a syntax for a class of related predicates.

## 7.3 Syntactically Defining Predicates in Categories α and β.

### 7.3.1 Category α.

Consider the first category, category α. This details a class in which a predicate holds for some non-empty trace s if and only if the predicate also holds for all traces with s as a prefix. An example in this category is provided by the predicate which determines whether the first event of a trace s is the event **a**. This may be written as R'(s) where

$$R'(s) \equiv \quad \mathcal{F}irst(s) = \mathbf{a} \qquad \text{Eq. 7.3}$$

Provided predicate R' holds for s it is reasonable to suppose that if any trace were suffixed onto s the resulting trace would also begin with event **a**. That is

$$\forall\, s, t \bullet \mathcal{F}irst(s) = \mathbf{a} \ \Rightarrow \quad \mathcal{F}irst(s\hat{\ }t) = \mathbf{a} \qquad \text{Eq. 7.4}$$

The trace s has a first event if and only if it is non-empty. Therefore it is also reasonable to suppose that if a trace satisfies R', then any non-empty prefix of that trace will also satisfy R'. That is

$$\forall\, s \neq <> \bullet \mathcal{F}irst(s\hat{\ }t) = \mathbf{a} \quad \Rightarrow \quad \mathcal{F}irst(s) = \mathbf{a} \qquad \text{Eq. 7.5}$$

By combining equations 7.4 and 7.5 the following result is obtained.

$$\forall\, s \neq <> \bullet \mathcal{F}irst(s) = \mathbf{a} \quad \Leftrightarrow \quad \mathcal{F}irst(s\hat{\ }t) = \mathbf{a} \qquad \text{Eq. 7.6}$$

$$\equiv \quad \forall\, s \neq <> \bullet R'(s) \quad \Leftrightarrow \quad R'(s\hat{\ }t) \qquad \text{Eq. 7.7}$$

By comparing this against the original definition given by Definition 6.3 it can be seen that the predicate R' is indeed a member of category α.

Predicate R' is by no means an isolated example. Other predicates with a similar syntax can also be shown to possess similar semantic properties, and thus belong to the category α. For example

$$(\ \mathcal{F}irst(s) = \mathbf{b}\ ) \in \alpha, \qquad (\ \mathcal{F}irst(t) = \mathbf{up}\ ) \in \alpha \qquad \text{Eq. 7.8}$$

In fact, it is possible to generalise the predicate `R'` to form a whole class of syntactically similar predicates by developing the following definition.

**Definition 7.1:** Let a set of predicates be syntactically defined by PRED$\alpha$. This is represented as follows

$$\text{PRED}\alpha \quad ::= \quad \text{“}\mathcal{F}\mathit{irst}\text{(”}, \mathit{tracevar}, \text{“) = ”}, \mathit{event}\,; \qquad \text{Eq. 7.9}$$

where *event* is the syntactic representation of an event (e.g. "**a**" or "**startit**") and *tracevar* is the syntactic representation of a trace variable (e.g. "s" or "t"). (These definitions of *event* and *tracevar* shall be used throughout the text.) ■

Because all the predicates in PRED$\alpha$, can be shown to possess similar properties to those of `R'`, it is likewise possible to show that all the predicates in PRED$\alpha$ belong to the category $\alpha$. This leads to a useful and interesting conclusion, namely that the definition of PRED$\alpha$ provides syntactic rules to establish a predicate's inclusion in the category $\alpha$. That is if `Q` is a predicate syntactically defined by PRED$\alpha$, then `Q` belongs to category $\alpha$. Note, however, that the reverse is not always so. If `Q` is a predicate in category $\alpha$ then `Q` is not necessarily syntactically defined by PRED$\alpha$. PRED$\alpha$ does not purport to be a complete syntax for the category $\alpha$.

### 7.3.2 Category β.

A similar syntactic definition can be arrived at for the second category. Category $\beta$ details a class of predicates which are such that a predicate holds for some non-empty trace `s` if and only if the predicate also holds for all traces with `s` as a suffix. Consider the following example predicate `Q'`

$$Q' \quad \equiv \quad \mathcal{L}\mathit{ast}(s) = \mathbf{a} \qquad \text{Eq. 7.10}$$

By analysis it can be shown that the predicate `Q'` is a member of category $\beta$. The syntax of `Q'` can be generalised to produce the following definition

**Definition 7.2:** Consider the set of predicates syntactically defined by PRED$\beta$

$$\text{PRED}\beta \quad ::= \quad \text{“}\mathcal{L}\mathit{ast}\text{(”}, \mathit{tracevar}, \text{“) = ”}, \mathit{event}\,; \qquad \text{Eq. 7.11}$$

■

Again, by exploiting the similarity of predicates in PREDβ to the predicate $Q'$, and observing that $Q'$ belongs to category β, it is possible to conclude that PREDβ provides syntactic rules to establish a predicate's inclusion in the category β. That is if a predicate belongs to PREDβ it also belongs to category β.

## 7.4 Syntactically Defining Predicates in Categories γ and δ.

### 7.4.1 Category δ.

Category δ details a class of predicates such that a predicate holds for both of two traces if and only if the predicate holds for the catenation of the two traces. To illustrate an example of a predicate in this category let $R''$ state that an event **a** does not belong to a trace $s$. This is written

$$R''(s) \equiv \neg(\mathbf{a}\ \underline{\mathbf{in}}\ s) \qquad \text{Eq. 7.12}$$

By undertaking analysis of $R''$ it is possible to show that for any traces $s$, $t$

$$\neg(\mathbf{a}\ \underline{\mathbf{in}}\ s) \wedge \neg(\mathbf{a}\ \underline{\mathbf{in}}\ t) \quad \Leftrightarrow \quad \neg(\mathbf{a}\ \underline{\mathbf{in}}\ s\hat{\ }t) \qquad \text{Eq. 7.13}$$

$$\therefore \quad \forall\, s, t \neq \langle\,\rangle \bullet R''(s) \wedge R''(t) \quad \Leftrightarrow \quad R''(s\hat{\ }t) \qquad \text{Eq. 7.14}$$

Referring back to the Definition 6.3 it can be seen that the predicate $R''$ is a member of the category δ. Now consider the predicate which states that an event **a** belongs to a trace $u$ when $u$ has been restricted to a set of events $A$. This can be written $Q''$ where

$$Q''(s) \equiv \neg(\mathbf{a}\ \underline{\mathbf{in}}\ u{\upharpoonright}A) \qquad \text{Eq. 7.15}$$

By making the substitutions $s = u{\upharpoonright}A$ and $t = v{\upharpoonright}A$ into equation 7.13, and noting that restriction distributes over catenation, it is seen that

$$\neg(\mathbf{a}\ \underline{\mathbf{in}}\ u{\upharpoonright}A) \wedge \neg(\mathbf{a}\ \underline{\mathbf{in}}\ v{\upharpoonright}A) \quad \Leftrightarrow \quad \neg(\mathbf{a}\ \underline{\mathbf{in}}\ (u{\upharpoonright}A\ \hat{\ }\ v{\upharpoonright}A)) \qquad \text{Eq. 7.16}$$

$$\Leftrightarrow \quad \neg(\mathbf{a}\ \underline{\mathbf{in}}\ (u\hat{\ }v{\upharpoonright}A)) \qquad \text{Eq. 7.17}$$

Again, by analysis, it is possible to deduce that $Q''$ is also a member of category δ. In the same manner as for section 7.2, these examples of predicates can be generalised into a syntactic definition.

**Definition 7.3:** Let a set of predicates be syntactically defined by PREDδ. This is represented as follows

PREDδ = "¬(", *event*, " **in** ", *tracevar*,")"
| "¬(", *event*, " **in** ", *tracevar*, "↾ ", *eventset*, ")"; Eq. 7.18

where *eventset* represents the syntax of any set of events (e.g. { a, b, c }, This definition of *eventset* will be used throughout the text.) ■

By noting the similarity of predicates in PREDδ to the examples R" and Q" it is possible to draw the conclusion that every predicate in PREDδ is also a predicate in the category δ.

### 7.4.2 Category γ.

The previous chapter showed that the logical negation of a predicate in category δ resulted in a predicate in category γ. Utilising this property leads to the following definition

**Definition 7.4:** Consider the set of predicates syntactically defined by PREDγ

PREDγ = *event*, " **in** ", *tracevar*
| *event*, " **in** ", *tracevar*, "↾", *eventset* ; Eq. 7.19

■

The definition of category γ (Definition 6.3) implies that if P' is a predicate in PREDγ, then (¬P') is equivalent to a predicate in PREDδ. As a result (¬P') will be in category δ, and so P' ( = ¬(¬P') ) will be in category γ. Therefore it can be concluded that if P' is a predicate in PREDγ, P' is also a predicate in category γ.

The rules and definitions espoused by Sections 7.3 and 7.4 now permit the following statement which summarises the relationship between the semantic definition of CATEGS and the syntaxes of PREDα, PREDβ, PREDγ and PREDδ.

**Statement 7.1:** If R is a predicate defined by one of PREDα, PREDβ, PREDγ or PREDδ, then R is also a member of the set { $\alpha \cup \beta \cup \gamma \cup \delta$ }. ■

*Justification*

See Sections 7.3 and 7.4 □

## 7.5 A Well Defined Syntax RSPEC.

The rules and definitions provided in Sections 7.3 and 7.4 can now be used to develop a full syntactic definition for the set CATEGS. But first it is important to comment on the original definition and make adjustments. In the previous chapter CATEGS was defined as " The syntax of all predicates which belong to categories $\alpha$, $\beta$, $\gamma$, $\delta$.". The decision was made in Section 7.2 to find rules which would imply a predicates inclusion in the set $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$ ( that is Z') rather than rules which would be equivalent to that inclusion (that is Z). Any subsequent syntax of CATEGS may not contain every predicate in $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$. As a consequence it was decided to re-define CATEGS as being " A syntax of predicates which belong to categories $\alpha$, $\beta$, $\gamma$, $\delta$.". Since the previous chapter only assumes that predicates in CATEGS are also predicates in $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$, adjusting this definition does not affect the theory.

Recall the previous definition of RSPEC.

```
STATEMENT ::=     TRUE | FALSE | CATEGS |
                  STATEMENT, "∨", STATEMENT |
                  STATEMENT, "∧", STATEMENT |
                  STATEMENT, "⇒", STATEMENT |
                  STATEMENT, "⇔", STATEMENT |
                  "¬", STATEMENT ;                          Eq. 7.20
```

By employing the definitions and rules given so far, it is now possible to generalise the above into a more complete definition of RSPEC, by including the formal definition of a syntax for CATEGS.

**Definition 7.5:** The syntax of RSPEC is defined in BNF form as

```
STATEMENT ::=     TRUE | FALSE | CATEGS |
                  STATEMENT, "∨", STATEMENT |
                  STATEMENT, "∧", STATEMENT |
                  STATEMENT, "⇒", STATEMENT |
                  STATEMENT, "⇔", STATEMENT |
```

"¬", STATEMENT ;

CATEGS ::= PREDα | PREDβ | PREDγ | PREDδ

PREDα ::= " first(", *tracevar*, ") = ", *event* ;

PREDβ ::= " *Last*(", *tracevar*, ") = ", *event* ;

PREDγ ::= *event*, " **in** ", *tracevar*
| *event*, " **in** ", *tracevar*, "↾", *eventset* ;

PREDδ ::= "¬(", *event*, " **in** ", *tracevar*, ")"
| "¬(", *event*, " **in** ", *tracevar*, "↾", *eventset*, ")" ;

*event* ::= "a" | "b" | .. ** list of event names **

*eventset* ::= "{", *event* , ",", *event*, .., *event* "}"

*tracevar* ::= "s" | "t" | .. ** list of trace variable names **

■

RSPEC is essentially a syntax for representing behavioural specifications of CSP processes. The motivation behind it is to provide behavioural specifications which can be always be used to generate Ideal Test Sets in conjunction with Catenary functions. The syntax provided above is aimed at behavioural specifications ranging over the simplest semantic model, the traces model.

The conclusion of Statement 7.1 was that all predicates in CATEGS ( as defined by the syntax of PREDα, PREDβ, PREDγ, PREDδ ) were also predicates in the set { $\alpha \cup \beta \cup \gamma \cup \delta$ }. The conclusion of Theorem 6.17 was that if predicates from the set { $\alpha \cup \beta \cup \gamma \cup \delta$ } were composed together under the logical operators ($\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, $\neg$ ), then the resulting behavioural specification had particular properties which, in combination with a suitable process, allowed the generation of an Ideal Test Set. Together, these conclusions lead to the following statement.

**Statement 7.2:** Let R be a (finite defined) behavioural specification from RSPEC and let F be a (finite defined) Catenary function. Then there exists a (finite) set I which is an Ideal Test Set for the pair ($\mu$P.F(P), R). Furthermore there is a defined procedure for generating I. ■

*Justification*

The basis for this statement lies in the work of this and previous chapters. It can be said to capture the theme of this part of the thesis. That is that there exists an automatic procedure in the form of an Ideal Test Set for establishing the correctness of certain processes against certain specifications, and that furthermore the limits of the acceptable processes and specifications can be bound by a suitable syntax. In this case that syntax is provided by RSPEC and by the syntactic definition of Catenary functions. □

## 7.6 Testing for the Empty Trace.

Sometimes when composing behavioural specifications it is useful to be able to test for the presence of an empty trace. This is represented by the predicate

$$s = < > \qquad \text{Eq. 7.21}$$

The problem with this predicate is that it does not belong to any of the four defined categories. Recall that these categories specifically exclude empty traces and as a consequence they exclude the empty trace predicate. To overcome this problem this thesis develops a procedure which allows the empty trace predicate to be included in behavioural specifications defined by RSPEC. Introduce a further syntactic class $\text{PRED}_{< >}$ defined by

$$\text{PRED}_{< >} \quad ::= \quad \textit{tracevar}, \text{“} = < > \text{”} \mid \textit{tracevar}, \text{“} \neq < > \text{”}; \qquad \text{Eq. 7.22}$$

This is incorporated into RSPEC by replacing the definition of CATEGS with

$$\text{CATEGS} \quad ::= \text{PRED}\alpha \mid \text{PRED}\beta \mid \text{PRED}\gamma \mid \text{PRED}\delta \mid \text{PRED}_{< >} \qquad \text{Eq. 7.23}$$

To distinguish the revised version of RSPEC from the original, call it RSPEC+. That is RSPEC+ is simply RSPEC with the inclusion of the empty trace predicate. Now let Q be some behavioural specification syntactically defined with RSPEC+ and let F be a Catenary function. To verify that Q satisfies the process $\mu$P.F(P) the following procedure is adopted

i) Given the behavioural specification Q in RSPEC+, replace all occurrences of "s = < >" with the proposition FALSE. (Note that all occurrences of " ¬(s = < >)" or "s ≠ < >" should be accordingly replaced by the proposition TRUE). This results in a new Q′, which is a behavioural specification in RSPEC.

ii) Using the procedures outlined, generate an Ideal Test Set I for the pair ( μP.F(P), Q′ ) in the conventional way.

iii) Test Q′ against the set I - { < > }

iv) Test Q against the set { < > }

v) Q satisfies the process μP.F(P) if and only if tests (iii) and (iv) are successful.

## 7.7 Some Examples of Using RSPEC and RSPEC+.

This section contains four examples of natural language specifications and shows their equivalent representation in RSPEC. The first three examples are intended to illustrate the scope of RSPEC and give an indication of how it is used to capture behavioural specifications. The fourth example is more detailed. Given a suitable Catenary function and behavioural specification in RSPEC+ it describes how an Ideal Test Set is generated to establish correctness.

**Example 7.1:** Let P be some process. Consider the natural language requirement "P will always perform the event **shiver** before the event **shake**". This can be expressed by the behavioural specification

$$\mathcal{Last}(\mathtt{s}) = \mathbf{shake} \quad \Rightarrow \quad \mathbf{shiver}\ \underline{\mathbf{in}}\ \mathtt{s} \qquad \text{Eq. 7.24}$$

The conjunctive normal form is

$$\neg\ (\mathcal{Last}(\mathtt{s}) = \mathbf{shake}) \quad \vee \quad (\mathbf{shiver}\ \underline{\mathbf{in}}\ \mathtt{s}) \qquad \text{Eq. 7.25}$$

■

**Example 7.2:** Let P be some process. Consider the natural language requirement "If the first event P performed was **shake**, then when it performs **roll** it will previously have performed **rattle**". This may be expressed

$$(\mathcal{L}ast(\mathrm{s}) = \mathbf{roll} \quad \wedge \quad \mathcal{F}irst(\mathrm{s}) = \mathbf{shake}) \Rightarrow \quad \mathbf{rattle}\ \underline{\mathbf{in}}\ \mathrm{s} \qquad \text{Eq. 7.26}$$

The conjunctive normal form is

$$\neg(\mathcal{L}ast(\mathrm{s}) = \mathbf{roll}) \vee \neg(\mathcal{F}irst(\mathrm{s}) = \mathbf{shake}) \vee (\mathbf{rattle}\ \underline{\mathbf{in}}\ \mathrm{s}) \qquad \text{Eq. 7.27}$$

■

**Example 7.3:** Let P be some process. Consider the natural language requirement "Once P has performed the event **wobble** it cannot perform the event **roll**". This may be expressed

$$\mathbf{wobble}\ \underline{\mathbf{in}}\ \mathrm{s} \quad \Rightarrow \quad \neg(\mathcal{L}ast(\mathrm{s}) = \mathbf{roll}) \qquad \text{Eq. 7.28}$$

The conjunctive normal form is

$$\neg(\ \mathbf{wobble}\ \underline{\mathbf{in}}\ \mathrm{s}\ ) \ \vee \quad \neg(\mathcal{L}ast(\mathrm{s}) = \mathbf{roll}) \qquad \text{Eq. 7.29}$$

■

**Example 7.4:** Let F be some Catenary function such that

$$\mathrm{F(X)} = \mathbf{a} \rightarrow \mathbf{b} \rightarrow \mathrm{X} \Box \mathbf{a} \rightarrow \mathbf{c} \rightarrow \mathbf{b} \rightarrow \mathrm{X} \qquad \text{Eq. 7.30}$$

and let P be the process defined by the fixed point of F, μP.F(P). A requirement of P is that the first event performed must always be **a** and prior to performing the event **c** process P must not have performed event **b**. This can be expressed by the behavioural specification R

$$\mathrm{R(s)} \equiv (\mathcal{F}irst(\mathrm{s}) = \mathbf{a} \vee \mathrm{s} = < >) \wedge ((\mathcal{L}ast(\mathrm{s}) = \mathbf{c} \vee \mathrm{s} = < >) \Rightarrow \neg(\ \mathbf{b}\ \underline{\mathbf{in}}\ \mathrm{s}\ )) \qquad \text{Eq. 7.31}$$

Note that this compound behavioural specification contains the empty trace predicate. As a consequence the appropriate procedure must be followed. Replacing " $\mathtt{s} = \langle\ \rangle$ " with FALSE yields the new behavioural specification $\mathtt{R'}$

$$\mathtt{R'(s)} \equiv (\mathit{First}(\mathtt{s}) = \mathbf{a}) \wedge ((\mathit{Last}(\mathtt{s}) = \mathbf{c}) \Rightarrow \neg(\mathbf{b}\ \underline{\mathbf{in}}\ \mathtt{s})) \qquad \text{Eq. 7.31}$$

To generate an Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, $\mathtt{R'}$) it is necessary to express $\mathtt{R'}$ in conjunctive normal form. This is

$$(\mathit{First}(\mathtt{s}) = \mathbf{a}) \wedge (\neg(\mathit{Last}(\mathtt{s}) = \mathbf{c}) \vee \neg(\mathbf{b}\ \underline{\mathbf{in}}\ \mathtt{s})) \qquad \text{Eq. 7.32}$$

The above may be written as two behavioural specifications under logical conjunction.

$$\mathtt{R'} \quad \equiv \quad \mathtt{R'}_1(\mathtt{s}) \wedge \mathtt{R'}_2(\mathtt{s}) \qquad \text{Eq. 7.33}$$

$$\mathtt{R'}_1(\mathtt{s}) \quad \equiv \quad (\mathit{First}(\mathtt{s}) = \mathbf{a}) \qquad \text{Eq. 7.34}$$

$$\mathtt{R'}_2(\mathtt{s}) \quad \equiv \quad (\neg(\mathit{Last}(\mathtt{s}) = \mathbf{c}) \vee \neg(\mathbf{b}\ \underline{\mathbf{in}}\ \mathtt{s})) \qquad \text{Eq. 7.35}$$

Because $\mathtt{R'}_1 \in \alpha$, by Theorem 6.2 an Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, $\mathtt{R'}_1$) is *Traces*($\mathtt{F(STOP)}$). $\mathtt{R'}_2$ is the logical disjunction of two predicates in two separate categories, one in $\beta$ and the other in $\delta$. Therefore it can be seen from Theorem 6.10 that $\mathtt{R'}_2 \in \sigma^2$.

By Theorem 6.14 an Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, $\mathtt{R'}_2$) is *Traces*($\mathtt{F}^2\mathtt{(STOP)}$). Theorem 6.16 now gives that an Ideal Test Set the pair ($\mu\mathtt{P.F(P)}$, $\mathtt{R'}$) $\equiv$ ($\mu\mathtt{P.F(P)}$, $\mathtt{R'}_1 \wedge \mathtt{R'}_2$) is *Traces*($\mathtt{F}^2\mathtt{(STOP)}$). Recalling the overall procedure which is being followed, first test $\mathtt{R}$ against the empty trace

$$\begin{aligned} \mathrm{R}(\langle\ \rangle) \equiv\ & (\mathit{First}(\langle\ \rangle) = \mathbf{a} \vee \langle\ \rangle = \langle\ \rangle) \\ & \wedge ((\mathit{Last}(\langle\ \rangle) = \mathbf{c} \vee \langle\ \rangle = \langle\ \rangle) \Rightarrow \neg(\mathbf{b}\ \underline{\mathbf{in}}\ \langle\ \rangle)) \end{aligned}$$

$$\equiv \quad (\text{FALSE} \vee \text{TRUE}) \wedge ((\text{FALSE} \vee \text{TRUE}) \Rightarrow \text{TRUE}) \equiv \text{TRUE} \qquad \text{Eq. 7.37}$$

Now $\mathtt{R'}$ must be tested against the Ideal Test Set

$$\begin{aligned} & (\mathit{Traces}(\mathtt{F}^2(\mathtt{STOP}) - \{\ \langle\ \rangle\ \}) \\ = \quad & \langle\,\mathbf{a}\,\rangle, \langle\,\mathbf{a,b}\,\rangle, \langle\,\mathbf{a,c}\,\rangle, \langle\,\mathbf{a,c,b}\,\rangle, \langle\,\mathbf{a,b,a}\,\rangle, \langle\,\mathbf{a,b,a,c}\,\rangle, \end{aligned}$$

$$< a, b, a, c, b >, \quad < a, b, a, b >, \quad < a, c, b, a >,$$
$$< a, c, b, a, b >, \quad < a, c, b, a, c >, \quad < a, c, b, a, c, b >$$

Eq. 7.38

The predicate R' does not satisfy all the above traces. Therefore it can be concluded that the predicate R does not hold for the process $\mu$P.F(P). Note, however, that the predicate R' does hold for the traces of F(STOP), given by the first four traces in the expansion of equation 7.38. Thus to establish the fact that R did not satisfy $\mu$P.F(P) it was necessary to analyse the second recursion of function F.

■

## 7.8 The Expressibility of RSPEC and a Strategy for Improving it.

The examples cited illustrate that there exist processes and behavioural specifications for which RSPEC can provide a method of generating an Ideal Test Set. However, it was also noted that the present scope of RSPEC is somewhat limited; there are a number of important and desirable properties which it is unable to capture. For example, it cannot represent requirements such as "The length of a trace must be more than/less than/equal to 5 events." or "The fifth event in a trace must be **a**.".

An overview of the scope and capabilities of RSPEC suggested that it could be extended. It was decided to undertake a detailed analysis of RSPEC in terms of the behavioural specifications which it could capture. This would provide an insight into the nature of RSPEC and, in turn, hopefully suggest a strategy by which it could be extended and improved.

By its structure, RSPEC relies on the combination of predicates from the four categories to which it is bound. Every behavioural specification in RSPEC can be decomposed into a number of predicates. Each of these predicates belongs to one of the four categories $\alpha$, $\beta$, $\gamma$, $\delta$. From the definition of these categories it is possible to assert that each decomposed predicate can be used for at most one of the following.

i) To directly reason about the first event in a trace. That is it can define the position and nature of the first event.

ii) To directly reason about the last event of a trace.

iii) To indirectly reason about the rest of the trace. That is it can determine whether or not a particular event is in a trace, but it cannot define the position of that event.

It was found that as a behavioural specification in RSPEC is composed of predicates which are limited by the properties i) - iii), so all behavioural specifications in RSPEC are also limited to these properties. As a result of experience gained in decomposing specifications, an interesting property of RSPEC surfaced. It was seen that there exist certain traces between which RSPEC is unable to differentiate. That is there exist pairs of non-equal traces $t_1$, $t_2$ such that there is no expression R in RSPEC for which

$$R(t_1) \Leftrightarrow \neg R(t_2) \qquad \text{Eq. 7.39}$$

For example there is no expression in RSPEC which can distinguish between the following traces

$$< \mathbf{a, c, d, e, b} > \quad \& \quad < \mathbf{a, d, e, c, b} > \qquad \text{Eq. 7.40}$$

This property, which for convenience will be called the non-differential property of RSPEC, is postulated. It is not formally proved in this text but there is strong evidence to suggest that it is both true and provable.

The realisation that RSPEC is both non-differential and limited in its expressibility leads to the hypothesis that the two properties are in some way related. If this is so, then a further conjecture could be that an extension to RSPEC which is differential, in that it can always distinguish between two non-equal traces, would also prove to be more expressive. Although these concepts are unproven and based on intuition rather than deductive reasoning, they can be seen to serve a useful purpose. Previously it was recognised that an extension to RSPEC was needed but there was no clear strategy for developing it. If, however, expressibility is related to the non-differential property then this suggests that a productive strategy may be to extend RSPEC so that it is differential.

In summary, the lack of a tangible strategy by which to develop an extension for RSPEC led to analysis. This analysis showed that RSPEC was unable to differentiate between certain traces and this in turn suggested a favourable strategy. This now draws attention to how it is possible to extend RSPEC to a differential form and how to resolve any extension to the existing theory. The nest section addresses these points and introduces a fundamental factor in their solution, monotonic endomorphic functions.

## 7.9 Endomorphic Functions.

### 7.9.1 Definition of Endomorphisms.

An endomorphic function is a function which maps any particular space onto itself [ Borowski 89 ]. That is if f is a function mapping from the set A to the set A then f is said to be endomorphic or an endomorphism. A preliminary result is that the composition of two endomorphisms is also an endomorphism. That is

**Theorem 7.1:** Let f, g be endomorphic functions and A be some set such that

$$f : A \rightarrow A, \qquad g : A \rightarrow A \qquad \text{Eq. 7.41}$$

Then the composition of f, $f \circ g$, is also endomorphic in that

$$f \circ g : A \rightarrow A \qquad \text{Eq. 7.42}$$

■

*Proof*

For a proof of Theorem 7.1 the reader is directed to [ Herstein 75 ]. □

A function which takes the set of traces as its domain can be loosely termed a trace function. By applying the general concept of endomorphisms to trace functions a new definition is arrived at, that of the endomorphic trace function or the trace endomorphism

**Definition 7.6:** A trace endomorphism is a function which has the set of all traces, $\Sigma^*$, as both its domain and its codomain. ■

### 7.9.2 The Role of Endomorphic Functions.

Suppose it were required to explicitly reason about a particular event, $\mathbf{p_e}$ say, whose position in a trace was known, but is not the first or last event in that trace. In these circumstances RSPEC is insufficient. It only has the capability to explicitly reason about the first and last events of a trace. However, suppose there was some function which, when applied to a trace, always returned a trace in which $\mathbf{p_e}$ was the first or last event, then it would be possible to use RSPEC to reason about the function of the trace and thus about the event explicitly.

For example, suppose a requirement dictates that the third event in a trace s must always be event **a**. This is not expressible in RSPEC. Let f represent a function which removes the first two events in a trace. Then, provided s is of three events or more, the first event of f(s) is also the third event of s. The following RSPEC expression Q can be used to determine that the first event of f(s) is event **a**

$$Q(f(s)) \quad \equiv \quad \mathcal{First}(f(s)) = \mathbf{a} \qquad \text{Eq. 7.43}$$

Note that to apply Q to a function of s means that the value of that function should be of the same type as its argument. This is so with endomorphic functions.

This example provides an insight into the use of endomorphic functions to extend RSPEC. The scope of an expression Q from RSPEC has been extended by taking as its subject a function of a trace variable rather than just the trace variable itself. In a similar manner it is proposed to allow behavioural specifications in RSPEC to take both trace variables and functions of trace variables as their subjects. Because such behavioural specifications relate to the traces of a process it is appropriate that any function of a trace should also be a trace. For this reason the functions used are restricted to endomorphisms.

To extend RSPEC in this way involves two main stages. First the syntax must be extended with suitable rules that allow the correct definition of expressions containing endomorphic functions. Once this has been achieved it will then be necessary to resolve this extended syntax with the existing theory developed for generating Ideal Test Sets. These two stages are the subject of the following sections.

### 7.9.3 Instances of Endomorphisms.

The concept of an endomorphic function has so far only been described in general terms. It is now appropriate to provide some specific examples of trace endomorphisms and investigate their characteristics.

#### 7.9.3.1 Trace Restriction Function.

The trace restriction function has already been described in the text in Chapter Three. It is the function which restricts a trace to a particular set of events. Strictly speaking it is not an endomorphism because it has the domain $\mathbb{P}(\Sigma) \times \Sigma^*$ and the codomain $\Sigma^*$. However, if the restricting set of events is bound then the function can be treated as endomorphic. For a set of events A the restriction function is defined by

$$s \upharpoonright A : \Sigma^* \rightarrow \Sigma^* \qquad \text{Eq. 7.44}$$

Examples of this function are

$$< a, c, d, b > \upharpoonright \{ a, b \} = < a, b > \qquad \text{Eq. 7.45}$$

$$< > \upharpoonright \{ d, e \} = < > \qquad \text{Eq. 7.46}$$

### 7.9.3.2 The Head Function.

The head function takes as its argument any trace s of arbitrary length n and returns the trace containing the first n-1 events of s in order. Effectively *Head*(s) removes the last event in a trace. In the exceptional case of the empty trace as argument the function returns the empty trace as its value. The head function is endomorphic, it maps from the set of traces to the set of traces.

$$\mathcal{H}ead(s) : \Sigma^* \rightarrow \Sigma^* \qquad \text{Eq. 7.47}$$

Some examples of the use of this function are.

$$\mathcal{H}ead( < a, b, c, d, e > ) = < a, b, c, d > \qquad \text{Eq. 7.48}$$

$$\mathcal{H}ead( < d > ) = < > \qquad \text{Eq. 7.49}$$

$$\mathcal{H}ead( < > ) = < > \qquad \text{Eq. 7.50}$$

### 7.9.3.3 The Tail Function.

The tail function takes as its argument any trace s of arbitrary length n and returns the trace containing the last n-1 events of s in order. Effectively *Tail*(s) removes the first event in a trace. In the exceptional case of the empty trace argument the function returns the empty trace as its value. The tail function is endomorphic, it maps from the set of traces to the set of traces.

$$\mathcal{T}ail(s) : \Sigma^* \rightarrow \Sigma^* \qquad \text{Eq. 7.51}$$

Examples of the use of the tail function are

$$\mathcal{T}ail( < a, b, c, d, e > ) = < b, c, d, e > \qquad \text{Eq. 7.52}$$

$$\mathit{Tail}(\langle \mathbf{d} \rangle) \quad = \quad \langle\,\rangle \qquad \text{Eq. 7.53}$$

$$\mathit{Tail}(\langle\,\rangle) \quad = \quad \langle\,\rangle \qquad \text{Eq. 7.54}$$

(It is noted from equations 7.49, 7.50, 7.53 and 7.54 that because the *Head* and *Tail* operators map more than one trace to the empty trace then they are not isomorphic, and thus endomorphisms rather than automorphisms [ Borowski 89 ].)

### 7.9.4 Other Definitions of Head and Tail.

The notations for the *Head* and *Tail* operators are introduced in this thesis as specific examples of endomorphic functions. These functions are based on the trace operators of the same name described in [ Hoare 85 ]. There are, however, two major differences. Primarily the head of a trace $\mathrm{s}$ is denoted by Hoare as $\mathrm{s}_0$, and is interpreted as the first event in a non-empty trace, rather than the result of removing the last event. The tail of a trace is denoted by Hoare as $\mathrm{s}'$, having the same interpretation as $\mathit{Tail}(\mathrm{s})$ except that $\mathrm{s}'$ is not defined for the empty trace. To draw parallels with Hoare's notation the following equivalences are formulated

$$\begin{aligned} \mathit{Tail}(\mathrm{s}) &= \mathrm{s}' &&, \mathrm{s} \neq \langle\,\rangle \\ &= \langle\,\rangle &&, \mathrm{s} = \langle\,\rangle \end{aligned} \qquad \text{Eq. 7.55}$$

$$\begin{aligned} \mathit{Head}(\mathrm{s}) &= \overline{(\overline{\mathrm{s}})'} &&, \mathrm{s} \neq \langle\,\rangle \\ &= \langle\,\rangle &&, \mathrm{s} = \langle\,\rangle \end{aligned} \qquad \text{Eq. 7.56}$$

where a bar over the top of a trace indicates the reverse of that trace, that is the reversing of the order of events in the trace.

### 7.9.5 Developing a Syntax of Endomorphic Functions.

If endomorphisms are to be included in any extension to the syntax of RSPEC then it stands to reason that they themselves must be able to be expressed in a syntactic form. That is there should be some syntax *function*, say, such that if $\mathrm{f}$ is an expression in *function* then $\mathrm{f}$ is endomorphic.

Syntactic rules which indicate that an expression is endomorphic can be developed along the same lines as those which govern inclusion in the set $\{\ \alpha \cup \beta \cup \gamma \cup \delta\ \}$. Specifically that a number of examples are studied, and their syntax is generalised.

To achieve this the text restricts itself to the three functions already mentioned, namely the restriction, head and tail functions. It is noted that, as endomorphic functions, the composition of any of these functions is also endomorphic. This realisation allows the following class of endomorphic functions called $\mathcal{HTR}$ functions to be introduced here.

**Definition 7.7:** A function f is said to be an $\mathcal{HTR}$ (head-tail-restriction) function if it is exclusively composed of head, tail or restriction functions. That is if f is such that

$$f(s) \equiv g_1 \circ .. \circ g_n(s) \qquad \text{Eq. 7.57}$$

where for all i such that $1 \leq i \leq n$

$$g_i(s) = \mathcal{Head}(s) \ \vee \ g_i(s) = \mathcal{Tail}(s) \ \vee \ (g_i(s) = s\upharpoonright A \wedge A \subseteq \mathbb{P}(\Sigma)) \qquad \text{Eq. 7.58}$$

An $\mathcal{HTR}$ function is said to be primitive if it is either the head, tail or restriction function. ■

The result that all $\mathcal{HTR}$ functions are endomorphisms follows naturally.

**Theorem 7.2:** All $\mathcal{HTR}$ functions are endomorphic ■

*Proof*

All $\mathcal{HTR}$ functions are the composition of primitive $\mathcal{HTR}$ functions. These are all endomorphic and so, by Theorem 7.1, the composition of them is also endomorphic.

□

The general syntactic form of each of the primitive $\mathcal{HTR}$ functions can be quickly derived. In turn these can be combined to yield the following precise definition of $\mathcal{HTR}$ functions in BNF.

**Definition 7.8:** A function f is an $\mathcal{HTR}$ function if and only if it has the following BNF syntactic definition

```
function    ::=    tracevar |
                   | "(", function, "↾", eventset, ")"
                   | "Head(", function, ")"
                   | "Tail(", function, ")";                    Eq. 7.59
```

■

Note that $\mathcal{HTR}$ functions only constitute a subset of all endomorphic trace functions. One given reason for identifying $\mathcal{HTR}$ functions is the convenient way in which they syntactically define endomorphisms.

Another less apparent, but equally important, reason for restricting the definition of $\mathcal{HTR}$ functions to the composition of the given primitives was to assure that the resulting functions were also monotonic. To understand this first introduce the following definition of monotonic.

**Definition 7.9:** Define two partial orders on the set of traces. The first is suffix partial ordering, $\underset{s}{\geq}$, defined as

$$u \underset{s}{\geq} v \quad \Leftrightarrow \quad \exists\, r \bullet u = r\hat{\ }v \qquad \text{Eq. 7.60}$$

where u, v, r are traces. The second is prefix partial ordering, $\underset{p}{\geq}$, defined as

$$u \underset{p}{\geq} v \quad \Leftrightarrow \quad \exists\, r \bullet u = v\hat{\ }r \qquad \text{Eq. 7.61}$$

where u, v, r are traces. A function is said to be monotonic if it preserves a partial order. For the purposes of this text, a traces function is defined to be monotonic if and only if it preserves both the prefix and suffix partial orders. That is f is monotonic if and only if

$$\begin{aligned} \forall\, u, v \bullet \; ( \quad & u \underset{p}{\geq} v \;\Rightarrow\; f(u) \underset{p}{\geq} f(v) \quad ) \\ \wedge \; ( \quad & u \underset{s}{\geq} v \;\Rightarrow\; f(u) \underset{s}{\geq} f(v) \quad ) \end{aligned} \qquad \text{Eq. 7.62}$$

■

## 7.10 Properties of Trace Endomorphisms.

The importance of a function being monotonic will become more apparent as the text proceeds. For the present it is stressed that if a function is not monotonic, then it cannot be resolved with the subsequent theory. Consequently it is necessary to establish that all the functions defined by the syntax *function* are indeed monotonic. This is achieved in Theorem 7.4. However, prior to this it is useful to take stock of some of the more general and specific properties of the trace endomorphisms which are being used. These properties are valuable both in terms of understanding more about the nature of trace functions as well as furnishing the subsequent proofs with some prerequisite results. The following Theorem lays out some of the properties which trace functions possess.

**Theorem 7.3:** The following properties hold for arbitrary traces $\mathtt{s}, \mathtt{t}$

i) $\mathit{Last}(\mathit{Tail}(\mathtt{s})) = \mathit{Last}(\mathtt{s}) \quad \Leftrightarrow \quad \mathit{Tail}(\mathtt{s}) \neq <\,>$ Eq. 7.63

ii) $\mathit{First}(\mathit{Head}(\mathtt{s})) = \mathit{First}(\mathtt{s}) \quad \Leftrightarrow \quad \mathit{Head}(\mathtt{s}) \neq <\,>$ Eq. 7.64

iii) $\mathit{Last}(\mathtt{s}\hat{}\mathtt{t}) = \mathit{Last}(\mathtt{t}) \quad \Leftrightarrow \quad \mathtt{t} \neq <\,>$ Eq. 7.65

iv) $\mathit{First}(\mathtt{s}\hat{}\mathtt{t}) = \mathit{First}(\mathtt{s}) \quad \Leftrightarrow \quad \mathtt{s} \neq <\,>$ Eq. 7.66

v) $\mathit{Tail}(\mathtt{s}\hat{}\mathtt{t}) = \mathit{Tail}(\mathtt{s})\hat{}\mathtt{t} \quad \Leftrightarrow \quad \mathtt{s} \neq <\,>$ Eq. 7.67

vi) $\mathit{Head}(\mathtt{s}\hat{}\mathtt{t}) = \mathtt{s}\hat{}\mathit{Head}(\mathtt{t}) \quad \Leftrightarrow \quad \mathtt{t} \neq <\,>$ Eq. 7.68

vii) $\mathtt{s} = \mathit{First}(\mathtt{s})\,\hat{}\,\mathit{Tail}(\mathtt{s})$ Eq. 7.69

viii) $\mathtt{s} = \mathit{Head}(\mathtt{s})\,\hat{}\,\mathit{Last}(\mathtt{s})$ Eq. 7.70

ix) $\mathit{Head}(\mathit{Tail}(\mathtt{s})) = \mathit{Tail}(\mathit{Head}(\mathtt{s}))$ Eq. 7.71

x) $\mathtt{s}\upharpoonright\mathtt{A}\,\hat{}\,\mathtt{t}\upharpoonright\mathtt{A} = \mathtt{s}\hat{}\mathtt{t}\upharpoonright\mathtt{A}$ Eq. 7.72

■

*Proof*

Each rule is proved by inspection. □

**Theorem 7.4:** All $\mathcal{HTR}$ functions are monotonic. ■

*Proof*

To prove Theorem 7.4 it is first necessary to demonstrate that each of the primitive $\mathcal{HTR}$ functions are monotonic. To achieve this let $\mathtt{r}, \mathtt{s}, \mathtt{t}$ be traces such that $\mathtt{r} \underset{\mathtt{p}}{\geq} \mathtt{s}$ and $\mathtt{r} \underset{\mathtt{s}}{\geq} \mathtt{t}$. Therefore there are arbitrary traces $\mathtt{u}, \mathtt{v}$ such that $\mathtt{r} = \mathtt{s}\hat{}\mathtt{v}$ and $\mathtt{r} = \mathtt{u}\hat{}\mathtt{t}$.

*Case Head(r)*

Taking the head function for both sides of the equality $\mathtt{r} = \mathtt{u}\hat{}\mathtt{t}$ yields

$$\mathit{Head}(\mathtt{r}) \quad = \quad \mathit{Head}(\mathtt{u}\hat{}\mathtt{t}) \qquad \text{Eq. 7.73}$$

$$= \quad \mathtt{u}\hat{}\mathit{Head}(\mathtt{t}) \qquad \text{from eq. 7.68 if } \mathtt{t} \neq < > \qquad \text{Eq. 7.74}$$

$$\therefore \quad \mathit{Head}(\mathtt{r}) \quad \underset{s}{\geqq} \quad \mathit{Head}(\mathtt{t}) \qquad \textit{if } \mathtt{t} \neq < > \qquad \text{Eq. 7.75}$$

Additionally, from equations 6.78 and 6.70

$$\mathit{Head}(\mathtt{u}\hat{}\mathtt{t}) \quad = \quad \mathit{Head}(\mathtt{u})\hat{}\mathit{Last}(\mathtt{u})\hat{}\mathit{Head}(\mathtt{t}) \qquad \textit{if } \mathtt{t} \neq < > \qquad \text{Eq. 7.76}$$

$$\therefore \quad \mathit{Head}(\mathtt{r}) \quad \underset{p}{\geqq} \quad \mathit{Head}(\mathtt{t}) \qquad \textit{if } \mathtt{t} \neq < > \qquad \text{Eq. 7.77}$$

If $\mathtt{t} = < >$ then it follows trivially that $\mathit{Head}(\mathtt{r}) \underset{s}{\geqq} \mathit{Head}(\mathtt{t})$ and $\mathit{Head}(\mathtt{r}) \underset{s}{\geqq} \mathit{Head}(\mathtt{t})$. Therefore, it is seen that the function *Head* is monotonic.

*Case Tail(r)*

By a similar argument as for *Head* it is possible to show that for *Tail*

$$\begin{array}{lll} & \mathtt{r} \underset{s}{\geqq} \mathtt{t} \; \Rightarrow & \mathit{Tail}(\mathtt{r}) \underset{s}{\geqq} \mathit{Tail}(\mathtt{t}) \\ \wedge & \mathtt{r} \underset{p}{\geqq} \mathtt{s} \; \Rightarrow & \mathit{Tail}(\mathtt{r}) \underset{p}{\geqq} \mathit{Tail}(\mathtt{s}) \end{array} \qquad \text{Eq. 7.78}$$

*Case Restriction*

Restriction is distributive over trace catenation. For a set of events $\mathtt{A}$ the general restriction function $\upharpoonright\mathtt{A}$ over the equality $\mathtt{r} = \mathtt{u}\hat{}\mathtt{t}$ yields

$$\mathtt{r}\upharpoonright\mathtt{A} \quad = \quad (\mathtt{u}\hat{}\mathtt{t})\upharpoonright\mathtt{A} \qquad \text{Eq. 7.79}$$

$$= \quad (\mathtt{u}\upharpoonright\mathtt{A})\hat{}(\mathtt{t}\upharpoonright\mathtt{A}) \qquad \text{Eq. 7.80}$$

$$\therefore \quad \mathtt{r}\upharpoonright\mathtt{A} \quad \underset{s}{\geqq} \quad \mathtt{t}\upharpoonright\mathtt{A} \qquad \text{Eq. 7.81}$$

The case for prefix ordering is shown in a similar way. Thus restriction is a monotonic function.

It now remains to show that if f and g are monotonic functions, then so is f ∘ g. Let s, t be traces such that

$$r \underset{s}{\geq} t \qquad \text{Eq. 7.82}$$

$$\therefore \quad f(r) \underset{s}{\geq} f(t) \qquad \textit{f is monotonic} \qquad \text{Eq. 7.83}$$

$$\therefore \quad g(f(r)) \underset{s}{\geq} g(f(t)) \qquad \textit{g is monotonic} \qquad \text{Eq. 7.84}$$

Again the case for the prefix ordering follows in a similar way. Therefore f ∘ g is monotonic. All *HTR* functions consist of the composition of a finite number of the primitive *HTR* functions. The primitive *HTR* functions are monotonic and thus so is their composition. Therefore all *HTR* functions are monotonic. □

## 7.11 Extending the Syntax of RSPEC.

Having given a syntax for *HTR* functions, it is now appropriate to address the question of how it may be incorporated into an extension of RSPEC. This extension will be called ERSPEC. It is broadly similar to RSPEC in overall structure, but the syntactic definition of predicates given by CATEGS in RSPEC has been extended. Recall that in RSPEC the set CATEGS was defined as

$$\text{CATEGS} \quad ::= \text{PRED}\alpha \mid \text{PRED}\beta \mid \text{PRED}\gamma \mid \text{PRED}\delta \qquad \text{Eq. 7.85}$$

It is proposed to add two more syntactic groups to this definition. The first is the syntax of all those predicates with the general form

$$\mathcal{F}irst(f(s)) = a \qquad \text{Eq. 7.86}$$

where f is an *HTR* function and a represents some arbitrary event. The form of expressions such as equation 7.86 can be generalised by the following BNF syntax.

$$\text{PRED}\bar{\alpha} \quad ::= \quad \text{“}\mathcal{F}irst(\text{”}, \textit{function}, \text{“}) = \text{”}, \textit{event}\ ; \qquad \text{Eq. 7.87}$$

The second group is the set of all those predicates with the general form

$$\mathcal{L}ast(f(s)) = a \qquad \text{Eq. 7.88}$$

where f is an $\mathcal{HTR}$ function and **a** represents some arbitrary event. Again it is possible to generalise expressions such as equation 7.88 into the following BNF syntax

PRED$\bar{\beta}$ ::= "*Last*(", *function*, ") = ", *event* ; Eq. 7.89

Extending the existing CATEGS by including the definitions PRED$\bar{\beta}$ and PRED$\bar{\beta}$ yields a new set of syntax NEWCATEGS given by the definition.

NEWCATEGS ::= PRED$\alpha$ | PRED$\beta$ | PRED$\gamma$ | PRED$\delta$
| PRED$\bar{\alpha}$ | PRED$\bar{\beta}$ | PRED$\varepsilon$ Eq. 7.90

The extra syntax PRED$\varepsilon$ is included here to afford a more complete definition of NEWCATEGS and thus ERSPEC. It will be described in more detail in Section 7.16. The method of generating an Ideal Test Set for a behavioural specification in RSPEC relies upon being able to determine the properties over catenation of every predicate in CATEGS. This is assured by the assertion that every predicate in CATEGS belongs to $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$, and thus its properties over catenation are known. However, this is no longer the case for NEWCATEGS. If a predicate is in NEWCATEGS then it does not necessarily imply that that predicate belongs to $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$. This is illustrated by the following example

**Example 7.5:** Let R(s) be a predicate defined by PRED$\bar{\alpha}$ where

$$\mathrm{R(s)} \equiv \mathcal{F}irst(\mathcal{T}ail(\mathrm{s})) = \mathbf{b} \qquad \text{Eq. 7.91}$$

Then for the traces <**a**> and <**b**, **c**> the predicate R is such that

$$\mathrm{R}(\langle\mathbf{a}\rangle\hat{\ }\langle\mathbf{b},\mathbf{c}\rangle) \not\Leftrightarrow \mathrm{R}(\langle\mathbf{a}\rangle) \qquad \text{Eq. 7.92}$$

$$\mathrm{R}(\langle\mathbf{a}\rangle\hat{\ }\langle\mathbf{b},\mathbf{c}\rangle) \not\Leftrightarrow \mathrm{R}(\langle\mathbf{b},\mathbf{c}\rangle) \qquad \text{Eq. 7.93}$$

$$\mathrm{R}(\langle\mathbf{a}\rangle\hat{\ }\langle\mathbf{b},\mathbf{c}\rangle) \not\Leftrightarrow \mathrm{R}(\langle\mathbf{a}\rangle) \vee \mathrm{R}(\langle\mathbf{b},\mathbf{c}\rangle) \qquad \text{Eq. 7.94}$$

$$\mathrm{R}(\langle\mathbf{a}\rangle\hat{\ }\langle\mathbf{b},\mathbf{c}\rangle) \not\Leftrightarrow \mathrm{R}(\langle\mathbf{a}\rangle) \wedge \mathrm{R}(\langle\mathbf{b},\mathbf{c}\rangle) \qquad \text{Eq. 7.95}$$

Thus R does not belong to any of the categories $\alpha$, $\beta$, $\gamma$, $\delta$. ■

Building an Ideal Test Set for a behavioural specification in the manner of Chapter Six requires knowledge about how that specification behaves over trace catenation. This knowledge can be derived from understanding how the individual predicates which make up the specification behave over catenation. For RSPEC this means being able to attribute a property over catenation to each and every predicate in CATEGS. For an extension to RSPEC built around the set NEWCATEGS this will involve being able to attribute a property of catenation to each and every predicate in NEWCATEGS.

Example 7.5 showed that not every predicate in PRED$\overline{\alpha}$ belongs to the set { $\alpha \cup \beta \cup \gamma \cup \delta$ }. The same is true for PRED$\overline{\beta}$. Because of this and the importance of being able to attribute a property over catenation to every predicate in NEWCATEGS it proved necessary to introduce two new categories $\overline{\alpha}, \overline{\beta}$ which could attribute such qualities to predicates in PRED$\overline{\alpha}$ and PRED$\overline{\alpha}$. This is achieved in the next two theorems.

**Theorem 7.5:** Let f be an $\mathcal{HTR}$ function and let $\Re$ be a predicate defined by the expression

$$\Re(s) \equiv \mathit{Last}(f(s)) = \mathbf{a} \tag*{Eq. 7.96}$$

Then $\Re$ is such that

$$\forall\, f(t) \neq \langle\, \rangle \bullet \quad \Re(t) \Leftrightarrow \Re(s\hat{\ }t) \tag*{Eq. 7.97}$$

■

*Proof*

Let s, t be traces such that $f(t) \neq \langle\, \rangle$. By the definition of the partial ordering on traces

$$s\hat{\ }t \underset{s}{\geq} t \tag*{Eq. 7.98}$$

The function f is an $\mathcal{HTR}$ function and thus by Theorem 7.4 is monotonic. Therefore

$$f(s\hat{\ }t) \underset{s}{\geq} f(t) \tag*{Eq. 7.99}$$

Thus, by definition, there exists some trace u such that

$$f(s\hat{\ }t) = u\hat{\ }f(t) \tag*{Eq. 7.100}$$

$$\therefore \quad \mathit{Last}(\mathtt{f(s\hat{}t)}) \;=\; \mathit{Last}(\mathtt{u\hat{}f(t)}) \qquad \text{Eq. 7.101}$$

Now, since $\mathtt{f(t)} \neq \langle\;\rangle$ it is seen from Theorem 7.3 (iii)

$$\mathit{Last}(\mathtt{u\hat{}f(t)}) \;=\; \mathit{Last}(\mathtt{f(t)}) \qquad \text{Eq. 7.102}$$

$$\therefore \quad \forall\, \mathtt{f(s)} \neq \langle\;\rangle \bullet \mathit{Last}(\mathtt{f(s\hat{}t)}) \;=\; \mathit{Last}(\mathtt{u\hat{}f(t)}) \qquad \text{Eq. 7.103}$$

$$\therefore \quad \forall\, \mathtt{f(s)} \neq \langle\;\rangle \bullet \mathit{Last}(\mathtt{f(t)}) = \mathbf{a} \;\Leftrightarrow\; \mathit{Last}(\mathtt{u\hat{}f(s\hat{}t)}) = \mathbf{a} \qquad \text{Eq. 7.104}$$

□

**Theorem 7.6:** Let f be an $\mathcal{HTR}$ function and let $\Re$ be a predicate defined by the expression

$$\Re(\mathtt{s}) \;\equiv\; \mathit{First}(\mathtt{f(s)}) = \mathbf{a} \qquad \text{Eq. 7.105}$$

Then $\Re$ is such that

$$\forall\, \mathtt{f(s)} \neq \langle\;\rangle \bullet \quad \Re(\mathtt{s}) \;\Leftrightarrow\; \Re(\mathtt{s\hat{}t}) \qquad \text{Eq. 7.106}$$

■

*Proof*

The proof of Theorem 7.6 is similar to that of Theorem 7.5 □

Theorems 7.5 and 7.6 indicate that the predicates formed under the syntax PRED$\bar{\alpha}$ and PRED$\bar{\beta}$ possess similar but different semantic properties to predicates formed under PRED$\alpha$ and PRED$\beta$. These properties are espoused by equations 7.97 and 7.106. The Theorems now allow the semantic properties of predicates in PRED$\bar{\alpha}$ and PRED$\bar{\beta}$ to be made clear by the following definitions.

**Definition 7.10:** Let M be some condition on s. Then a predicate R belongs to the category $\bar{\alpha}$ provided

$$\forall\, \mathtt{M(s), t} \bullet \quad \mathtt{R(s)} \;\Leftrightarrow\; \mathtt{R(s\hat{}t)} \qquad \text{Eq. 7.107}$$

■

**Definition 7.11:** Let M be some condition on s. Then a predicate R belongs to the category $\bar{\beta}$ provided

$$\forall\, M(t), s \bullet \quad R(t) \quad \Leftrightarrow \quad R(s\hat{}t) \qquad \text{Eq. 7.108}$$

■

These new categories allow the properties over catenation of every predicate formed from the syntax NEWCATEGS to be made explicit. This is summed up by the statement

**Statement 7.3:** Let R be a predicate in the syntactic set NEWCATEGS. Then R belongs to one of the categories $\alpha$, $\beta$, $\gamma$, $\delta$, $\bar{\alpha}$, $\bar{\beta}$.

■

*Justification*

If R is a predicate in NEWCATEGS then R is either also in CATEGS or it belongs to the set { PRED$\bar{\alpha}$ $\cup$ PRED$\bar{\beta}$ $\cup$ PRED$\varepsilon$ }. If it is in CATEGS then from Statement 7.1 that R is also in the set { $\alpha \cup \beta \cup \gamma \cup \delta$ }. If it belongs to the set { PRED$\bar{\alpha}$ $\cup$ PRED$\bar{\beta}$ } then it is seen from Theorem 7.5 and 7.6 that R possess the semantic properties to be included in the set { $\bar{\alpha} \cup \bar{\beta}$ }. For the present it is simply assumed without qualification that if R belongs to PRED$\varepsilon$ then R is also in { $\bar{\alpha} \cup \bar{\beta}$ }. This is qualified in section 7.16 □

Theorems 7.5 and 7.6 also highlight the importance of the earlier stricture that functions such as f must be both endomorphic and monotonic. Theorem 7.5 showed that the monotonicity of the function f was a requirement for an expression in PRED$\bar{\alpha}$ to belong to the category $\bar{\alpha}$. Later sections reinforce this point by demonstrating that all behavioural specifications in the extended syntax ERSPEC need to be decomposed into predicates which belong to the categories $\alpha$, $\beta$, $\gamma$, $\delta$, $\bar{\alpha}$, $\bar{\beta}$. This is only feasible if it can be guaranteed that Statement 7.3 is valid, and this is dependent on the monotonicity of participating functions.

As an aside, it is worth noting that if a predicate belongs to category $\alpha$, then it also belongs to category $\bar{\alpha}$. To illustrate this substitute the condition "s is non-empty" for the expression M(s) in equation 7.107 in the definition of $\bar{\alpha}$. In a similar way it can be shown that if a predicate belongs to category $\beta$ it also belongs to category $\bar{\beta}$. It can thus be concluded that

$$\{ \bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta \} \quad \supseteq \quad \{ \alpha \cup \beta \cup \gamma \cup \delta \} \qquad \text{Eq. 7.109}$$

This section concludes with a result which determines a normal form for behavioural specifications in ERSPEC. This follows the fashion of Chapter Six where a normal form was given for expressions in RSPEC. It standardises the approach to dealing with behavioural specifications in ERSPEC.

**Theorem 7.7:** Let R be a statement in ERSPEC which is not a tautology. Then it is possible to write R in the conjunctive normal form as

$$\bigwedge_{i=1}^{\phi} \left( \bigvee_{j=1}^{\varphi} (Q_{ij}) \right) \qquad \text{Eq. 7.110}$$

where either $Q_{ij}$ or $\neg(Q_{ij})$ is a predicate from the set $\{ \bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta \}$ and $\phi$, $\varphi$ are integers.

■

*Proof*

Theorem 7.7 can be proved by using the concept of substitution instances given in [ Hamilton 78 ]. □

## 7.12 Closures of $\bar{\alpha}$, $\bar{\beta}$, $\gamma$, $\delta$.

In the previous chapter it is shown how an Ideal Test Set is generated for behavioural specifications of RSPEC. The procedure adopted is to take an expression in normal form and use reasoning to determine the properties over catenation of those parts of the expression in the disjunctive form $\bigvee_{j=1}^{\phi} (Q_{ij})$. These properties are then used to develop provisional Ideal Test Sets. The union of the provisional Ideal Test Sets is used to formulate an overall Ideal Test Set for the behavioural specification given by $\bigwedge_{i=1}^{\phi} \bigvee_{j=1}^{\varphi} (Q_{ij})$. This chapter approaches the generation of Ideal Test Sets for behavioural specifications of ERSPEC in a similar way. The following sections are concerned with generating provisional Ideal Test Sets for a general expression of the form

$$\mathtt{S} \quad = \quad \bigvee_{j=1}^{\varphi} (\mathtt{Q'}_j) \qquad \text{Eq. 7.111}$$

where each $\mathtt{Q'}_j$ or $\neg(\mathtt{Q'}_j)$ is a predicate from the set $\{\ \bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta\ \}$. Specifically, it is required to generate an Ideal Test Set for the pair ($\mu\mathtt{P}.\mathtt{F}(\mathtt{P})$, $\mathtt{S}$), where $\mathtt{F}$ is some Catenary function. To achieve this it will be necessary to establish by reasoning the properties that $\mathtt{S}$ possesses over catenation. Thus it is important initially to establish closures over the set $\{\ \bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta\ \}$. That is to determine those predicates which, when composed under logical disjunction, preserve certain properties relating to catenation.

### 7.12.1 Closures of $\bar{\alpha}$ and $\bar{\beta}$ under Logical Negation and Disjunction.

Attention is first focused on those two new categories $\bar{\alpha}$ and $\bar{\beta}$. The following theorems establish important results for closures on these categories.

**Theorem 7.8:** The category $\bar{\alpha}$ is closed under

i) logical negation

ii) logical disjunction ■

*Proof*

*Case i) negation*

Let $\mathtt{R}$ be a predicate in $\bar{\alpha}$ such that

$$\forall\, \mathtt{M(s)} \bullet \mathtt{R(s)} \quad \Leftrightarrow \quad \mathtt{R(s\hat{}t)} \qquad \text{Eq. 7.112}$$

$$\therefore \quad \forall\, \mathtt{M(s)} \bullet \neg\mathtt{R(s)} \quad \Leftrightarrow \quad \neg\mathtt{R(s\hat{}t)} \qquad \text{Eq. 7.113}$$

$$\therefore \quad \neg\mathrm{R} \in \bar{\alpha} \qquad \text{Eq. 7.114}$$

*Case ii) disjunction*

Let $\mathtt{Q}$, $\mathtt{R}$ be arbitrary predicates in $\bar{\alpha}$ with conditions $\mathtt{M}$, $\mathtt{N}$ respectively such that

$$\forall\, \mathtt{M(s)} \bullet \mathtt{Q(s)} \ \Leftrightarrow \quad \mathtt{Q(s\hat{}t)} \qquad \text{Eq. 7.115}$$

$$\forall\, \mathtt{N(s)} \bullet \mathtt{R(s)} \ \Leftrightarrow \quad \mathtt{R(s\hat{}t)} \qquad \text{Eq. 7.116}$$

$$\therefore \quad \forall\, \mathtt{M(s)} \wedge \mathtt{N(s)} \bullet \mathtt{Q(s)} \vee \mathtt{R(s)} \;\Leftrightarrow\; \mathtt{Q(s\hat{\ }t)} \vee \mathtt{R(s\hat{\ }t)} \qquad \text{Eq. 7.117}$$

Let $\mathtt{E(s)}$ be equivalent to $\mathtt{M(s)} \wedge \mathtt{N(s)}$.

$$\therefore \quad \forall\, \mathtt{E(s)} \bullet \mathtt{Q{\vee}R(s)} \Leftrightarrow \mathtt{Q{\vee}R(s\hat{\ }t)} \qquad \text{Eq. 7.118}$$

Thus $\mathtt{Q{\vee}R}$ belongs to the category $\bar{\alpha}$ and so $\bar{\alpha}$ is closed under logical disjunction.

□

**Theorem 7.9:** The category $\bar{\beta}$ is closed under

i) logical negation

ii) logical disjunction

■

*Proof*

Theorem 7.9 follows in a similar manner as Theorem 7.8. □

Taken in conjunction with Theorem 6.4 it is now possible to use these results to show that the set { $\bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta$ } is closed under logical negation (just as the set { $\alpha \cup \beta \cup \gamma \cup \delta$ } is closed under logical negation). The corollary of this is that the expression $\mathtt{S}$ can be rewritten as

$$\mathtt{S} \quad = \quad \bigvee_{j=1}^{\varphi} (\mathtt{Q''}_j) \qquad \text{Eq. 7.119}$$

where each $\mathtt{Q''}_j$ is a predicate from the set { $\bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta$ }. By employing Theorem 7.8 (ii) all the predicates in the expansion of $\mathtt{S}$ which belong to $\bar{\alpha}$ can be composed into one expression $\mathtt{R}_{\bar{\alpha}}$ which also belongs to $\bar{\alpha}$. Similarly, by Theorem 7.9 (ii), all predicates belonging to $\bar{\beta}$ can be composed into some predicate $\mathtt{R}_{\bar{\beta}} \in \bar{\beta}$. Using Theorem 6.5 and Theorem 6.6 it is likewise possible to show that all expressions in category $\gamma$ can be composed into a predicate $\mathtt{R}_\gamma \in \gamma$ and the $\mathtt{n}$ remaining expressions in category $\delta$ can be composed into an expression $\mathtt{R}_{\delta\mathtt{n}} \in \delta^{\mathtt{n}}$.

The result of these simplifications to $\mathtt{S}$ is that, provided there exists an expression $\mathtt{Q''}_j$ in each of the defined categories, an expression of the form $\mathtt{S}$ can now be written

$$\mathtt{S} \quad \equiv \quad R_{\bar{\alpha}} \vee R_{\bar{\beta}} \vee R_{\gamma} \vee R_{\delta^n} \qquad \text{Eq. 7.120}$$

where $R_{\bar{\alpha}} \in \bar{\alpha}$, $R_{\bar{\beta}} \in \bar{\beta}$, $R_{\gamma} \in \gamma$, $R_{\delta^n} \in \delta^n$. If however, as is often the case, there is not an expression $\mathtt{Q}''_{\mathtt{j}}$ in every category, then the expression $\mathtt{S}$ can be written in one of fourteen other ways.

$$\begin{array}{llll} & \mathtt{S} & \equiv & R_{\bar{\alpha}} \vee R_{\bar{\beta}} \vee R_{\gamma} \\ \vee & \mathtt{S} & \equiv & R_{\bar{\alpha}} \vee R_{\bar{\beta}} \vee R_{\delta^n} \\ \vee & \ldots & & \\ \vee & \mathtt{S} & \equiv & R_{\delta^n} \end{array} \qquad \text{Eq. 7.121}$$

That is there are in total fifteen different ways in which predicates of the form $\mathtt{S}$ can be expressed as the disjunction of one or more of the predicates from the set { $R_{\bar{\alpha}}$, $R_{\bar{\beta}}$, $R_{\gamma}$, $R_{\delta^n}$ }.

## 7.13 Ideal Test Sets for Predicates in Categories $\bar{\alpha}$ and $\bar{\beta}$.

Having simplified the expression $\mathtt{S}$ by demonstrating closures within the individual categories, it is now appropriate to consider the effect of logical disjunction between the categories. However, prior to this there is another important consideration.

It was seen in Chapter Six that every predicate $\mathtt{Q}$ from the set { $\alpha \cup \beta \cup \gamma \cup \delta$ } is such that *Traces*(F(STOP)) is an Ideal Test Set for the pair ($\mu$P.F(P), Q). This particular property is instrumental in the theory of Chapter Six. For instance, recall the set defined by $\sigma^n$. One of the fundamental results is that for an expression

$$\mathfrak{R} \quad = \quad \bigvee_{\mathtt{j}=1}^{\varphi} (Q_{\mathtt{j}}) \qquad \text{Eq. 7.122}$$

where each $Q_{\mathtt{j}} \in \{ \alpha \cup \beta \cup \gamma \cup \delta \}$ it is always possible to generate an integer $\mathtt{n}$ such that $\mathfrak{R} \in \sigma^n$. This property is then utilised to generate an Ideal Test Set. The ability to place all such $\mathfrak{R}$ in $\sigma^n$ is only possible because all the constituent predicates of $\mathfrak{R}$ are related to the same Ideal Test Set.

However, the existence of Ideal Test Sets for predicates in the categories $\bar{\alpha}$ and $\bar{\beta}$ is not as straightforward as it was for categories $\alpha$ and $\beta$. For a Catenary function F it is not possible to state that for all $\mathtt{R} \in \{ \bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta \}$ there is an equivalent Ideal Test Set for all the pairs ($\mu$P.F(P), R). As a direct result the method of placing all predicates into some indexed classification such as $\sigma^n$ is no longer tenable. A different method for generating provisional Ideal Test Sets is required.

The approach adopted for ERSPEC is to develop theorems to explicitly determine the nature of the provisional Ideal Test Sets. That is given an expression of the form S it will be possible to generate an Ideal Test Set associated with S by applying one or more of a number of relevant theorems. These theorems are given in the next section.

However there remains one obstacle. The change in the definition of the categories $\alpha$, $\beta$ to $\overline{\alpha}$, $\overline{\beta}$ means that Theorem 6.1 no longer applies. That is for a Catenary function F and predicate $R \in \{ \overline{\alpha} \cup \overline{\beta} \cup \gamma \cup \delta \}$ the set *Traces*(F(STOP)) is no longer necessarily an Ideal Test Set for the pair ($\mu$P.F(P), R). Yet, just as in Chapter Six, the theorems to be developed for composite predicates are dependent on the Ideal Test Sets of the constituent predicates. Thus it is necessary to have knowledge of these Ideal Test Sets. Consequently results for Ideal Test Sets of predicates from the set $\{ \overline{\alpha} \cup \overline{\beta} \}$ are required.

Before this is undertaken, the following definition will prove useful in subsequent work

**Definition 7.12:** Let M(s) be some condition on s. Then the order of the pair (F, M) is defined as the minimum integer x such that

$$\forall\, s \in \{ \mathit{Traces}(F^{x}(STOP)) - \mathit{Traces}(F^{x-1}(STOP)) \} \bullet M(s) \qquad \text{Eq. 7.123}$$

Alternatively, the order of the pair (F, M) can be related to the set $\mathcal{D}_F$ which was introduced in Chapter Five. If y is the minimum integer such that the condition M holds for every trace in ($\mathcal{D}_F{}^{y}$) then y+1 is a value for the order of (F, M).

■

The rationale behind this definition is not easier with the hindsight of subsequent work. Its essence is that it provides a means of simplifying notation. Definition 7.12 now enables the following theorems which provide a means of generating Ideal Test Sets for predicates in the categories $\overline{\alpha}$ and $\overline{\beta}$.

**Theorem 7.10:** Let M be a condition on traces and R be a predicate in $\overline{\alpha}$ that is

$$\forall\, M(s), t \bullet \quad R(s) \quad \Leftrightarrow \quad R(s\hat{\ }t) \qquad \text{Eq. 7.124}$$

Let F be a Catenary function. Then the set

$$\mathit{Traces}(\mathtt{F}^{\mathtt{x}}(\mathtt{STOP})) \qquad \text{Eq. 7.125}$$

is an Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, $\mathtt{R}$), where $\mathtt{x}$ is order of ($\mathtt{F}$, $\mathtt{M}$). ■

*Proof*

Assume that $\mathtt{R}$ has tested positive against the set $\mathit{Traces}(\mathtt{F}^{\mathtt{x}}(\mathtt{STOP}))$. Let $\mathtt{r}$ be an arbitrary finite trace of the process $\mu\mathtt{P.F(P)}$. Then, by Theorem 5.15, there is some integer $\mathtt{n}$ such that $\mathtt{r}$ can be expressed

$$\mathtt{r} \quad = \quad \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_n \qquad \text{Eq. 7.126}$$

where $\mathtt{r}_1, .., \mathtt{r}_{n-1} \in \mathcal{D}_F$ and $\mathtt{r}_n \in \mathit{Traces}(\mathtt{F(STOP)})$. To prove that predicate $\mathtt{R}$ always holds for the trace $\mathtt{r}$ consider the two following cases

*Case* $\mathtt{n} \leq \mathtt{x}$

If $\mathtt{n} \leq \mathtt{x}$ then, by Theorem 5.15, the trace $\mathtt{r}$ belongs to the set $\mathit{Traces}(\mathtt{F}^{\mathtt{x}}(\mathtt{STOP}))$ and its truth against predicate $\mathtt{R}$ follows from testing.

*Case* $\mathtt{n} > \mathtt{x}$

If $\mathtt{n} > \mathtt{x}$ then $\mathtt{r}$ can be expressed

$$\mathtt{r} \quad = \quad \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}..\hat{}\mathtt{r}_n \qquad \text{Eq. 7.127}$$

From Theorem 5.15 it is seen that

$$\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x \in \mathit{Traces}(\mathtt{F}^{\mathtt{x}}(\mathtt{STOP})) \wedge \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x \notin \mathit{Traces}(\mathtt{F}^{\mathtt{x-1}}(\mathtt{STOP})) \qquad \text{Eq. 7.128}$$

$$\therefore \quad \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x \in \{\ \mathit{Traces}(\mathtt{F}^{\mathtt{x}}(\mathtt{STOP})) - \mathit{Traces}(\mathtt{F}^{\mathtt{x-1}}(\mathtt{STOP}))\ \} \qquad \text{Eq. 7.129}$$

$$\therefore \quad \mathtt{M}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \qquad \text{,\textit{definition of order} x} \qquad \text{Eq. 7.130}$$

Therefore from the initial definition of $\mathtt{R}$

$$\forall\, \mathtt{t} \bullet\ \mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \quad \Leftrightarrow \quad \mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}\mathtt{t}) \qquad \text{Eq. 7.131}$$

Let $\mathtt{t} = \mathtt{r}_{x+1}\hat{}..\hat{}\mathtt{r}_n$

$$\therefore \quad \mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \quad \Leftrightarrow \quad \mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}\mathtt{r}_{x+1}\hat{}\mathtt{r}_{x+2}\hat{}..\hat{}\mathtt{r}_n) \qquad \text{Eq. 7.132}$$

By Theorem 5.15 the trace $\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x$ belongs to the set *Traces*($\mathtt{F}^x$(STOP)) and its truth against R follows from testing.

Thus provided R holds over *Traces*($\mathtt{F}^x$(STOP)) then R(r) is true for all traces r. Therefore *Traces*($\mathtt{F}^x$(STOP)) is an Ideal Test Set for the pair (μP.F(P), R).

□

**Theorem 7.11:** Let M be a condition on traces and R be a predicate in $\bar{\beta}$ such that

$$\forall \mathtt{M(t), s} \bullet \quad \mathtt{R(t)} \quad \Leftrightarrow \quad \mathtt{R(s\hat{}t)} \qquad \text{Eq. 7.133}$$

Let F be a Catenary function. Then the set

$$\textit{Traces}(\mathtt{F}^x(\mathtt{STOP})) \qquad \text{Eq. 7.134}$$

is an Ideal Test Set for the pair (μP.F(P), R), where x is order of (F, M). ■

*Proof*

It is possible to prove Theorem 7.11 by studying Theorem 7.10. □

## 7.14 Generating Provisional Ideal Test Sets.

In Section 7.12 it was shown that there are fifteen possible ways of expressing S as the disjunction of predicates $\mathtt{R}_{\bar{\bar{\alpha}}}$, $\mathtt{R}_{\bar{\bar{\alpha}}}$, $\mathtt{R}_\gamma$, $\mathtt{R}_{\delta n}$, where $\mathtt{R}_a \in \bar{\alpha}$, $\mathtt{R}_{\bar{\beta}} \in \bar{\beta}$, $\mathtt{R}_\gamma \in \gamma$ and $\mathtt{R}_{\delta n} \in \delta^n$. Therefore the task of generating an Ideal Test Set for the pair (μP.F(P), S), where F is a Catenary function, is equivalent to the problem of finding an Ideal Test Set for each of the fifteen combinations of S.

By individually treating all fifteen combinations of S it would be possible to generate an Ideal Test Set for each, and then use these to factorize an algorithm for generating an Ideal Test Set for the general expression S. However, such an exhaustive method is clumsy and can be circumvented.

Instead consider initially only those combinations of S which do not contain any predicates in category γ. This reduces the number of combinations to seven. The justification for removing category γ will be given later. Those seven combinations are

i) $S = R_{\bar{\alpha}}$ ii) $S = R_{\bar{\beta}}$

iii) $S = R_{\delta n}$ iv) $S = R_{\bar{\alpha}} \vee R_{\bar{\beta}}$

v) $S = R_{\bar{\alpha}} \vee R_{\delta n}$ vi) $S = R_{\bar{\beta}} \vee R_{\delta n}$

vii) $S = R_{\bar{\alpha}} \vee R_{\bar{\beta}} \vee R_{\delta n}$

Theorems have been developed which cover these seven cases and show how Ideal Test Sets are generated for each. These are given in the next section.

### 7.14.1 Theorems for Deriving Ideal Test Sets.

The means of generating Ideal Test Sets for the pairs ($\mu P.F(P)$, $S$) in cases i), ii) and iii) are given by Theorems 7.10, 7.11 and 6.6 respectively. It remains to develop theorems which can be used to determine values for iv), v), vi) and vii).

The following theorem provides a treatment for case iv). Here two predicates, one each from category $\bar{\alpha}$ and $\bar{\beta}$, are composed under logical disjunction.

**Theorem 7.12:** Let $Q$, $R$ be predicates from the categories $\bar{\alpha}, \bar{\beta}$ respectively and $M$, $N$ be conditions such that

$$\forall M(s) \bullet \quad Q(s) \Leftrightarrow Q(s\hat{}t) \qquad \text{Eq. 7.135}$$

$$\forall N(t) \bullet \quad R(t) \Leftrightarrow R(s\hat{}t) \qquad \text{Eq. 7.136}$$

Let $F$ be a Catenary function. Let $x$, $y$ be the orders of the pairs ($F$, $M$) and ($F$, $N$) respectively. Then an Ideal Test Set for the pair ($\mu P.F(P)$, $Q \vee R$) is

$$\mathit{Traces}(F^{x+y}(STOP)) \qquad \text{Eq. 7.137}$$

■

*Proof*

It is seen from Theorem 7.10 and 7.11 that the sets $\mathit{Traces}(F^{x}(STOP))$, $\mathit{Traces}(F^{y}(STOP))$ are Ideal Test Sets for the pairs ($\mu P.F(P)$, $Q$), ($\mu P.F(P)$, $R$) respectively.

Assume that the predicate $Q \vee R$ has tested positive over the set $\mathit{Traces}(F^{x+y}(STOP))$. Let $r$ be an arbitrary finite trace from the process $\mu P.F(P)$. Then by Theorem 5.15 there is some integer $n$ such that $r$ can be expressed

$$\mathtt{r} \quad = \quad \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_n \qquad \text{Eq. 7.138}$$

where $\mathtt{r}_1, .., \mathtt{r}_{n-1} \in \mathcal{D}_F$ and $\mathtt{r}_n \in$ *Traces*(F(STOP)). To show that Q∨R always holds for r consider the two cases.

*Case* $x+y \geq n$

If $x+y \leq n$ then, by Theorem 5.15, r is a member of the set *Traces*($\mathrm{F}^{x+y}$(STOP)), and so Q∨R(r) holds by testing.

*Case* $x+y < n$

The trace r can be written as

$$\mathtt{r} \quad = \quad \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}\mathtt{r}_{x+1}\hat{}..\hat{}\mathtt{r}_{n-y}\hat{}\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n \qquad \text{Eq. 7.139}$$

From the definitions of x and y it can be shown that

$$\mathrm{M}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \quad \wedge \quad \mathrm{N}(\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n) \qquad \text{Eq. 7.140}$$

$$\therefore \quad \forall\, \mathtt{t} \bullet \; \mathrm{Q}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \quad \Leftrightarrow \quad \mathrm{Q}(\mathtt{r}_1\hat{}\mathtt{r}_2\hat{}..\hat{}\mathtt{r}_x\hat{}\mathtt{t}) \qquad \text{Eq. 7.141}$$

$$\& \quad \forall\, \mathtt{s} \bullet \; \mathrm{R}(\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n) \quad \Leftrightarrow \quad \mathrm{R}(\mathtt{s}\hat{}\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n) \qquad \text{Eq. 7.142}$$

By Theorem 5.15 the traces $\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x$ and $\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n$ both belong to *Traces*($\mathrm{F}^{x+y}$(STOP)). Thus

$$\mathrm{Q}(\mathtt{r}_1\hat{}\mathtt{r}_2\hat{}..\hat{}\mathtt{r}_x) \quad \wedge \quad \mathrm{R}(\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n) \qquad \text{Eq. 7.143}$$

Consider the predicate Q∨R

$$\mathrm{Q{\vee}R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_n) \quad \Leftrightarrow \quad \mathrm{Q}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}..\hat{}\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n)$$
$$\vee\, \mathrm{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}..\hat{}\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n) \qquad \text{Eq. 7.144}$$

From equations 7.141 and 7.142

$$\mathrm{Q{\vee}R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_n) \quad \Leftrightarrow \quad \mathrm{Q}(\mathtt{r}_1\hat{}\mathtt{r}_2\hat{}..\hat{}\mathtt{r}_x) \vee \mathrm{R}(\mathtt{r}_{1+n-y}\hat{}..\hat{}\mathtt{r}_n) \qquad \text{Eq. 7.145}$$

Equation 7.143 yields that Q∨R always holds for r

Since these cases cover all possible values for $\mathtt{r}$, it is deduced that if $\mathtt{Q} \vee \mathtt{R}$ holds for *Traces*($\mathtt{F}^{\mathtt{x+y}}$(STOP)) then $\mathtt{Q} \vee \mathtt{R}$ holds over $\mu$P.F(P). Therefore *Traces*($\mathtt{F}^{\mathtt{x+y}}$(STOP)) is an Ideal Test Set for the pair ($\mu$P.F(P), $\mathtt{Q} \vee \mathtt{R}$) □

Note the similarities between Theorem 7.12 and Theorem 6.8. To deal with case v) another theorem in a similar style is required.

**Theorem 7.13:** Let Q and R be predicates from the categories $\delta^n$, $\overline{\alpha}$ respectively and M be a condition such that

$$\forall\, \mathtt{M(s)} \bullet \mathtt{R(s)} \Leftrightarrow \mathtt{R(s\hat{}t)} \qquad \text{Eq. 7.146}$$

Let F be a Catenary function and let x be the order of the pair (F, M). Then an Ideal Test Set for the pair ($\mu$P.F(P), $\mathtt{Q} \vee \mathtt{R}$) is

$$\mathtt{F}^{\mathtt{n+x}}(\mathtt{STOP}) \qquad \text{Eq. 7.147}$$

■

*Proof*

This theorem is proved by induction

INDUCTIVE STEP

Assume that there is some integer m ( >n+x ) such that $\mathtt{Q} \vee \mathtt{R}$ holds over

$$\textit{Traces}(\mathtt{F}^{\mathtt{m}}(\mathtt{STOP})) \qquad \text{Eq. 7.148}$$

Now let r be an arbitrary trace of the process

$$\mathtt{F}^{\mathtt{m+1}}(\mathtt{STOP}) \qquad \text{Eq. 7.149}$$

and consider the following cases

*Case* $r \in$ *Traces*($F^m$(*STOP*))

$\mathtt{Q} \vee \mathtt{R}$ holds for r because $\mathtt{Q} \vee \mathtt{R}$ holds over *Traces*($\mathtt{F}^{\mathtt{m}}$(STOP)).

*Case* $r \notin$ *Traces*($F^m$(*STOP*))

In this case it can be shown that r is such that

$$\mathtt{r} = \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_{m+1} \qquad \text{Eq. 7.150}$$

where $\mathtt{r}_1,.., \mathtt{r}_m \in \mathcal{D}_F$ and $\mathtt{r}_{m+1} \in \mathtt{F(STOP)}$. Consider the two following subcases

*Subcase* $R(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x)$ *holds*

If x is the order of (F, M), then by definition $\mathtt{M}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x)$ holds. From the definition of R

$$\mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \quad \Leftrightarrow \quad \mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}\mathtt{t}) \qquad \text{Eq. 7.151}$$

$$\therefore \quad \mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \quad \Leftrightarrow \quad \mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}\mathtt{r}_{x+1}\hat{}..\hat{}\mathtt{r}_{m+1}) \qquad \text{Eq. 7.152}$$

Therefore $\mathtt{Q} \vee \mathtt{R}$ holds for trace r

*Subcase* $\neg R(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x)$ *holds*

Consider the set of traces given by

$$\Omega \quad = \quad \bigcap_{i=1}^{m+1} (\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_{i-1}\hat{}\mathtt{r}_{i+1}\hat{}..\hat{}\mathtt{r}_{m+1}) \qquad \text{Eq. 7.153}$$

$\mathtt{Q} \vee \mathtt{R}$ holds over $\Omega$ because for all $1 \leq i \leq m+1$ the trace $\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_{i-1}\hat{}\mathtt{r}_{i+1}\hat{}..\hat{}\mathtt{r}_{m+1}$ belongs to the set *Traces*($\mathtt{F}^m$(STOP)), and $\mathtt{Q} \vee \mathtt{R}$ holds over this. If x is the order of (F, M), then by definition $\mathtt{M}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x)$ holds. From the definition of R

$$\forall\, \mathtt{t} \bullet \neg\mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x) \Leftrightarrow \neg\mathtt{R}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}\mathtt{t}) \qquad \text{Eq. 7.154}$$

That is R will not hold for all those traces with $\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x$ as a prefix. Because $\mathtt{Q} \vee \mathtt{R}$ holds for all traces in $\Omega$ it can be deduced that Q must hold for all the traces in $\Omega$ which have the prefix $\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x$. This can be expressed by

$$\bigwedge_{i=x+1}^{m+1} \mathtt{Q}(\mathtt{r}_1\hat{}..\hat{}\mathtt{r}_x\hat{}..\hat{}\mathtt{r}_{i-1}\hat{}\mathtt{r}_{i+1}\hat{}..\hat{}\mathtt{r}_{m+1}) \qquad \text{Eq. 7.155}$$

The following substitutions are made

$$\mathtt{s}_1 = \mathtt{r}_1\hat{}..\hat{}\mathtt{r}_{m+1-n} \qquad \text{Eq. 7.156}$$

$$s_2 = r_{m+2-n} \qquad \text{Eq. 7.157}$$

.. ..

$$s_{n+1} = r_{m+1} \qquad \text{Eq. 7.158}$$

Equation 7.155 becomes

$$\bigwedge_{i=x+1}^{m-n} Q(r_1\hat{}..\hat{}r_x\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m-n}\hat{}s_2\hat{}..\hat{}s_{n+1})$$
$$\wedge \bigwedge_{i=2}^{n+1} Q(s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n-1}) \qquad \text{Eq. 7.159}$$

By definition Q is such that

$$\forall\, u_j \neq <> \bullet \bigwedge_{i=1}^{n+1} Q(u_1\hat{}..\hat{}u_{i-1}\hat{}u_{i+1}\hat{}..\hat{}u_{n+1}) \Leftrightarrow Q(u_1\hat{}..\hat{}u_{n+1}) \qquad \text{Eq. 7.160}$$

Equation 7.160 can be used to show that

$$\text{Eq. 7.159} \quad \Rightarrow \quad Q(s_2\hat{}..\hat{}s_{n+1}) \wedge \bigwedge_{i=2}^{n+1} Q(s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. 7.161}$$

$$\Rightarrow \quad \bigwedge_{i=1}^{n+1} Q(s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. 7.162}$$

Again equation 7.160 shows that

$$\text{Eq. 7.162} \quad \Rightarrow \quad Q(s_1\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. 7.163}$$

Therefore Q∨R holds for r

Thus if $r \in$ *Traces*($F^{m+1}$(STOP)) then Q∨R(r). The inductive step is

$$F^{m}(\text{STOP})\ \textbf{sat}\ Q\vee R \quad \Rightarrow \quad F^{m+1}(\text{STOP})\ \textbf{sat}\ Q\vee R \qquad \text{Eq. 7.164}$$

BASE CASE

When $m = n+x$ the predicate Q∨R holds over the set $F^{n+x}$(STOP) by testing.

Therefore for all finite integers m, $F^{m}$(STOP) **sat** Q∨R. As a consequence it is seen that $F^{n+x}$(STOP) is an Ideal Test Set for the pair (μP.F(P), Q∨R). □

Again, note the similarities between Theorem 7.13 and Theorem 6.9. The next Theorem develops a similar result for the disjunction of predicates from categories $\bar{\beta}$ and $\delta^n$ for case vi).

**Theorem 7.14:** Let Q and R be predicates from the categories $\delta^n, \bar{\beta}$ respectively and M be a condition such that

$$\forall\, M(s) \bullet R(t) \Leftrightarrow \quad R(s\hat{\ }t) \qquad \text{Eq. 7.165}$$

Let F be a Catenary function and let x be the order of the pair (F, M). Then an Ideal Test Set for the pair (μP.F(P), Q∨R) is

$$F^{n+x}(STOP) \qquad \text{Eq. 7.166}$$

■

*Proof*

Theorem 7.14 follows in a similar manner as that for Theorem 7.13. □

Finally a Theorem for the disjunction of predicates from all three categories to cater for case vii).

**Theorem 7.15:** Let Q, R and S be predicates from the categories $\delta^n, \bar{\alpha}, \bar{\beta}$ respectively and M, N be conditions such that

$$\forall\, M(s) \bullet R(s) \Leftrightarrow R(s\hat{\ }t) \qquad \text{Eq. 7.167}$$

$$\forall\, N(s) \bullet S(t) \Leftrightarrow S(s\hat{\ }t) \qquad \text{Eq. 7.168}$$

Let F be a Catenary function. Let x, y be the orders of the pairs (F, M) and (F, N) respectively. Then an Ideal Test Set for the pair (μP.F(P), Q∨R∨S) is

$$F^{n+x+y}(STOP) \qquad \text{Eq. 7.169}$$

■

*Proof*

Theorem 7.15 is proved in a similar manner as Theorems 7.13 and 7.14. A full proof is provided in Appendix D □

## 7.14.2 Including Category γ.

Section 7.14.1 shows how Ideal Test Sets can be generated for the seven combinations of S which excluded the category γ. The following result illustrates how category γ can be incorporated.

**Theorem 7.16:** Let $\Re$ be a predicate that can be expressed in the form

$$\Re \quad = \quad \bigvee_{j=1}^{\varphi} A_j \qquad \text{Eq. 7.170}$$

where $A_j$ is any predicate from the set $R_{\bar{\alpha}} \cup R_{\bar{\beta}} \cup R_{\delta n}$ and $\varphi$ is a non-zero positive integer . Let F be a Catenary function and suppose that the set I is an Ideal Test Set for the pair $(\mu P.F(P), \Re)$. Then I is also an Ideal Test Set for the pair

$$(\mu P.F(P), \Re \vee R_{\gamma}) \qquad \text{Eq. 7.171}$$

■

*Proof*

The proof for Theorem 7.16 follows in a similar manner from the proof of Theorems 7.15, 7.14, 7.13 and 6.12. □

Of the remaining eight combinations of S yet to be treated, seven are equivalent to the seven combinations already treated but with a predicate from category γ under disjunction. That is they are the cases

viii) $S = R_{\bar{\alpha}} \vee R_{\gamma}$ ix) $S = R_{\bar{\beta}} \vee R_{\gamma}$

x) $S = R_{\delta n} \vee R_{\gamma}$ xi) $S = R_{\bar{\alpha}} \vee R_{\bar{\beta}} \vee R_{\gamma}$

xii) $S = R_{\bar{\alpha}} \vee R_{\delta n} \vee R_{\gamma}$ xiii) $S = R_{\bar{\beta}} \vee R_{\delta n} \vee R_{\gamma}$

xiv) $\mathtt{S} = R_{\bar{\alpha}} \vee R_{\bar{\beta}} \vee R_{\delta n} \vee R_{\gamma}$

From Theorem 7.16 it can be seen that Ideal Test Sets for cases viii) to xiv) are identical to those for cases i) to vii). The final case is given by

xv) $\mathtt{S} = R_{\gamma}$

An Ideal Test Set for this is given by Theorem 6.2. Thus it has been shown how to generate an Ideal Test Set for each of the fifteen combinations of S.

## 7.15 Generating Ideal Test Sets for ERSPEC.

The previous sections have shown how it is possible to generate an Ideal Test Set for the pair ($\mu$P.F(P), S). The results can be summed up by the statement

**Statement 7.4:** For every expression from ERSPEC of the form S (see below) and Catenary function F there exists an integer m such that

$$\mathit{Traces}(\mathtt{F}^{\mathtt{m}}(\mathtt{STOP})) \qquad \text{Eq. 7.171}$$

is an Ideal Test Set for the pair ($\mu$P.F(P), S). To calculate a value for m, express S in the form

$$\mathtt{S} \quad = \quad \bigvee_{j=1}^{p_1} R_j^{\bar{\alpha}} \vee \bigvee_{j=1}^{p_2} R_j^{\bar{\beta}} \vee \bigvee_{j=1}^{p_3} R_j^{\gamma} \vee \bigvee_{j=1}^{p_4} R_j^{\delta} \qquad \text{Eq. 7.172}$$

where $p_1,.., p_4$ are integers and $\forall\, j \bullet R_j^{\bar{\alpha}} \in \bar{\alpha}, R_j^{\bar{\beta}} \in \bar{\beta}, R_j^{\gamma} \in \beta, R_j^{\delta} \in \beta$.

If $p_1 > 0$ then let $m_1$ be the order of $(\mathtt{F}, \bigvee_{j=1}^{p_1} R_j^{\bar{\alpha}})$ otherwise let $m_1$ be zero.

If $p_2 > 0$ then let $m_2$ be the order of $(\mathtt{F}, \bigvee_{j=1}^{p_2} R_j^{\bar{\beta}})$ otherwise let $m_2$ be zero.

If $p_1+p_2+p_4 = 0$ then let $m_3 = 1$ otherwise let $m_3 = 0$.

Let $\mathtt{m}_4 = \mathtt{n}$.

Then a value for $\mathtt{m}$ is $\mathtt{m} = \mathtt{m}_1+\mathtt{m}_2+\mathtt{m}_3+\mathtt{m}_4$ ■

*Justification*

The justification of this statement lies in the work of preceding sections. □

Statement 7.4 provides a general means for generating provisional Ideal Test Sets. Now that this has been accomplished it is possible to focus attention on the overall Ideal Test Set. This is captured by the following statement and justification.

**Statement 7.5:** Let $\mathtt{R}$ be a behavioural specification in ERSPEC and $\mathtt{F}$ be a Catenary function. Then there exists a method for generating an Ideal Test Set for the pair ($\mu\mathtt{P}.\mathtt{F}(\mathtt{P})$, $\mathtt{R}$). ■

*Justification*

Let $\mathtt{F}$ be a Catenary function. From Theorem 7.7 every expression $\mathtt{R}$ in ERSPEC can be expressed in conjunctive normal form as

$$\mathtt{R} = \bigwedge_{i=1}^{\phi} \left( \bigvee_{j=1}^{\varphi} (\mathtt{Q}_{ij})\right) \qquad \text{Eq. 7.173}$$

where each $\mathtt{Q}_{ij}$ is a predicate from the set $\{ \bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta \}$ and $\phi$, $\varphi$ are integers. By definition each $\bigvee_{j=1}^{\varphi} (\mathtt{Q}_{ij})$ is an expression of the form $\mathtt{S}$ as given in Statement 7.4. Therefore from Statement 7.4 there exists a set of integers $\mathtt{q}_1,.., \mathtt{q}_n$ such that

$$\mathit{Traces}(\mathtt{F}^{\mathtt{q}_1}(\mathtt{STOP})),.., \mathit{Traces}(\mathtt{F}^{\mathtt{q}_n}(\mathtt{STOP})) \qquad \text{Eq. 7.174}$$

are respectively Ideal Test Sets for the pairs

$$(\mu\mathtt{P}.\mathtt{F}(\mathtt{P}), \bigvee_{j=1}^{\varphi} (\mathtt{Q}_{ij})),..,(\mu\mathtt{P}.\mathtt{F}(\mathtt{P}), \bigvee_{j=1}^{\varphi} (\mathtt{Q}_{\phi j})) \qquad \text{Eq. 7.175}$$

Let $\mathtt{q}$ be the maximum value from the set $\{ \mathtt{q}_1,.., \mathtt{q}_n \}$. Then from the partial ordering of processes in the traces domain given in Appendix A the set $\mathit{Traces}(\mathtt{F}^{\mathtt{q}}(\mathtt{STOP}))$ contains all the other sets in equation 7.174. From Theorem 6.16

it is seen that the set *Traces*($\mathtt{F}^{q}(\mathtt{STOP})$) is an Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, R) where R is the logical conjunction of all $\bigvee_{j=1}^{\varphi}(\mathtt{Q_{ij}})$. That is

$$Traces(\mathtt{F}^{q}(\mathtt{STOP})) \qquad \text{Eq. 7.176}$$

is an Ideal Test Set for the pair ($\mu\mathtt{P.F(P)}$, R) □

## 7.16 Empty Trace Predicates in ERSPEC.

To complete the treatment of ERSPEC it is necessary to address the problem of expressing predicates which determine whether or not a particular trace is empty. The approach to this was simplified in RSPEC by the fact that there are no trace-to-trace functions present in the syntax and so there is only one trace, the subject trace, which a predicate can show to be the empty trace. However, in ERSPEC there are trace-to-trace functions in the form of $\mathcal{HTR}$ functions. This means that a predicate may required to show that functions of the subject trace are the empty trace. That is, for some $\mathcal{HTR}$ function f let a predicate R be defined as

$$\mathtt{R(s)} \quad \equiv \quad \mathtt{f(s)} = <\,> \qquad \text{Eq. 7.177}$$

The problem is how can predicates such as R be incorporated into the syntax of ERSPEC?

The first step in answering this is to develop a general syntax for predicates such as R. This is given by the following definition

**Definition 7.13:** Let PREDε be defined in BNF as

PREDε ::= *function*, "(", *tracevar*, ") = < >"
| *function*, "(", *tracevar*, ") ≠ < >"; Eq. 7.178

■

Reference to Section 7.11 shows that PREDε has already been included in NEWCATEGS. Statement 7.3 made the unqualified assumption that every predicate in PREDε was also a member of the set { $\bar{\alpha} \cup \bar{\beta} \cup \gamma \cup \delta$ }. Since Statement 7.3 embodies the assumptions made by the semantic theory about the syntax of ERSPEC it is necessary to qualify that assumption.

The following Theorem reveals an interesting property of predicates from the set PREDε. It was expected that, as they contained monotonic endomorphisms, they would be somehow linked to one of the semantic categories $\bar{\alpha}$ or $\bar{\beta}$. In truth it was seen that any predicate in PREDε could be shown to belong to not one but both of the categories $\bar{\alpha}$ or $\bar{\beta}$.

**Theorem 7.17:** Let f be an $\mathcal{HTR}$ function and let R be a predicate defined by

$$\mathtt{R(s)} \quad \equiv \quad \mathtt{f(s)} = \langle\, \rangle \qquad \text{Eq. 7.179}$$

Then R is such that

$$\forall\, \mathtt{f(s)} \neq \langle\, \rangle \bullet \mathtt{R(s)} \quad \Leftrightarrow \quad \mathtt{R(s\hat{\ }t)} \qquad \text{Eq. 7.180}$$

$$\forall\, \mathtt{f(t)} \neq \langle\, \rangle \bullet \mathtt{R(t)} \quad \Leftrightarrow \quad \mathtt{R(s\hat{\ }t)} \qquad \text{Eq. 7.181}$$

That is R belongs to both category $\bar{\alpha}$ and category $\bar{\beta}$. ■

*Proof*

By Theorem 7.4 the function f is monotonic over prefix ordering and therefore for traces s, t

$$\mathtt{f(s\hat{\ }t)} \quad \underset{p}{\geq} \quad \mathtt{f(s)} \qquad \text{Eq. 7.182}$$

$$\therefore \quad \mathtt{f(s)} \neq \langle\, \rangle \quad \Rightarrow \quad \mathtt{f(s\hat{\ }t)} \neq \langle\, \rangle \qquad \text{Eq. 7.183}$$

$$\therefore \quad \forall\, \mathtt{f(s)} \neq \langle\, \rangle \bullet \quad \neg\mathtt{R(s)} \wedge \quad \neg\mathtt{R(s\hat{\ }t)} \qquad \text{Eq. 7.184}$$

$$\therefore \quad \forall\, \mathtt{f(s)} \neq \langle\, \rangle \bullet \quad \mathtt{R(s)} \quad \Leftrightarrow \quad \mathtt{R(s\hat{\ }t)} \qquad \text{Eq. 7.185}$$

Furthermore f is also monotonic over suffix ordering.

$$\mathtt{f(s\hat{\ }t)} \quad \underset{s}{\geq} \quad \mathtt{f(t)} \qquad \text{Eq. 7.186}$$

$$\therefore \quad \mathtt{f(t)} \neq \langle\, \rangle \quad \Rightarrow \quad \mathtt{f(s\hat{\ }t)} \neq \langle\, \rangle \qquad \text{Eq. 7.187}$$

$$\therefore \quad \forall\, \mathtt{f(t)} \neq \langle\, \rangle \bullet \quad \neg\mathtt{R(t)} \wedge \quad \neg\mathtt{R(s\hat{\ }t)} \qquad \text{Eq. 7.188}$$

$$\therefore \quad \forall\, \mathtt{f(t)} \neq \langle\,\rangle \bullet \qquad \mathtt{R(t)} \quad \Leftrightarrow \quad \mathtt{R(s\hat{\ }t)} \qquad \text{Eq. 7.189}$$

Thus equations 7.185 and 7.189 demonstrate that R belongs to both category $\bar{\alpha}$ and category $\bar{\beta}$. □

**Corollary 7.1:** If R is a predicate in PREDε then R belongs to both category $\bar{\alpha}$ and category $\bar{\beta}$.

This corollary implies that Statement 7.3 is fully justified.

## 7.17 Some Final Considerations.

To draw the main body of this chapter to a close it is worth taking up two points which, up to now, have not been treated. The first concerns the hypothesis that the expressibility of RSPEC was in some way related to its non-differential nature. Recall that the strategy for improving RSPEC was to find a differential extension. This consideration was instrumental in the structuring of ERSPEC and it is now useful to show that ERSPEC is in fact differential. This is achieved by the following Theorem.

**Theorem 7.18:** ERSPEC is differential. That is for every two finite non-equal traces s, t there always exists a predicate Q in ERSPEC such that

$$\mathtt{Q(s)} \quad \not\Leftrightarrow \quad \mathtt{Q(t)} \qquad \text{Eq. 7.190}$$

■

*Proof*

Let the expression s(i) represent the $\mathtt{i}^{\mathtt{th}}$ event of a trace s. Two traces are equivalent if they have the same set of events in the same order. That is traces s, t are equivalent if

$$\forall\, \mathtt{i} \bullet \mathtt{s(i)} = \mathtt{t(i)} \qquad \text{Eq. 7.191}$$

Now, let s and t be any non equivalent traces. Then there must exist some integer value q such that

$$\mathtt{s(q)} \neq \mathtt{t(q)} \qquad \text{Eq. 7.192}$$

The $\mathtt{q}^{\mathtt{th}}$ element of a trace can also be given by the function $\mathit{Last}(\mathit{Tail}^{\mathtt{q}-1}(\mathtt{s}))$ (See next chapter). Thus the expression $\mathit{Last}(\mathit{Tail}^{\mathtt{q}-1}(\mathtt{s}))$ is such that

$$\mathcal{L}ast(\mathcal{T}ail^{q-1}(\mathtt{s})) = \mathbf{a} \quad \Leftrightarrow \quad \neg(\mathcal{L}ast(\mathcal{T}ail^{q-1}(\mathtt{t}))) = \mathbf{a} \qquad \text{Eq. 7.193}$$

Therefore for any two traces s, t there exists a predicate of the form $\mathcal{L}ast(\mathcal{T}ail^{q-1}(\mathtt{s}))$ is in ERSPEC which can differentiate between s and t. Therefore ERSPEC is differential. □

## 7.18 Summary.

The main points of this chapter can be summarised as follows

• It was recognised that the definition provided for RSPEC in Chapter Six was not strong enough for RSPEC to be defined totally in terms of a syntax. This was overcome by developing a set of syntactic rules which restricted RSPEC to those predicates which were compatible with the existing theory for generating Ideal Test Sets. This resulted in a precisely defined syntax for RSPEC which, in tandem with a suitable process, could be used as a means for generating Ideal Test Sets and thus as a verification method.

• The suitability of RSPEC as a specification language and its use as a means for generating Ideal Test Sets was illustrated by a number of examples

• Although RSPEC is a useful syntax in its own right, it was noted that there were certain modifications which could be made to improve it. Therefore an extension to RSPEC, namely ERSPEC was suggested. It was shown how ERSPEC could also be expressed solely in terms of a syntax and that by restructuring the theory developed in Chapter Six it was demonstrated that an Ideal Test Set existed for every pair ($\mu$P.F(P), R), where F is a Catenary function and R a behavioural specification of ERSPEC. Thus ERSPEC was shown to be a more powerful verification tool than RSPEC.

• The extension of RSPEC was built up around the use of a particular class of trace functions. It was found that restricting these trace functions to monotonic endomorphisms enabled the subsequent theory and led to a number of elegant proofs.

• Finally a hypothesis was suggested for measuring the expressibility of the extended syntax ERSPEC against the original RSPEC, namely the quality of nondifferentiation. It was stated that, although this hypothesis was unproved, it was able to provide a strategy for improving RSPEC which was successful.

CHAPTER EIGHT

# CONCLUSIONS

## 8.1 Introduction.

This chapter sums up the work of previous chapters. It opens by providing a complete formal definition of the language ERSPEC which was introduced in Chapter 7 and is used to form the basis of the method of Ideal Test Sets. The chapter then moves on to show through a number of examples how ERSPEC can be used to capture different system requirements. By using the example of the Arbor Drum introduced in Chapter 4 a fully worked example of the use of ERSPEC and the method of Ideal Test Sets is provided which demonstrates the verification of a control logic. The chapter then assesses the suitability of the methods employed and draws some conclusions concerning their advantages and disadvantages. It also makes suggestions for possible improvements and future work which would extend the scope of the methods.

## 8.2 The Syntactic Definition of ERSPEC.

As stated in previous chapters, ERSPEC is the syntax which was proposed as an extension to the original syntax RSPEC. It has the same fundamental structure as RSPEC except that it has three extra syntactic definitions incorporated into it. These are PRED$\overline{\alpha}$, PRED$\overline{\beta}$ and PRED$\varepsilon$. Recall from Chapter 7 that predicates expressed in these syntaxes were shown to belong to one or more of the semantic categories $\overline{\alpha}$, $\overline{\beta}$. The full definition of ERSPEC is as follows

**Definition 8.1:** The syntax of ERSPEC is defined in BNF as

STATEMENT::= TRUE | FALSE | NEWCATEGS |
STATEMENT, “∨”, STATEMENT |
STATEMENT, “∧”, STATEMENT |
STATEMENT, “⇔”, STATEMENT |
STATEMENT, “⇒”, STATEMENT |
“¬”, STATEMENT ;

NEWCATEGS::= PREDα | PREDβ | PREDγ | PREDδ
| PRED$\bar{\alpha}$ | PRED$\bar{\beta}$ | PREDε

PREDα ::= “*First*(”, *tracevar*, “) = ”, *event* ;

PREDβ ::= “*Last*(”, *tracevar*, “) = ”, *event* ;

PREDγ ::= *event*, “ **in** ”, *tracevar*, |
*event*, “ **in** ”, *tracevar*, “↾”, *eventset* ;

PREDδ ::= “¬(”, *event*, “ **in** ”, *tracevar*, “)”
|“¬(”, *event*, “ **in** ”, *tracevar*, “↾ ”, *eventset*, “)”;

PRED$\bar{\alpha}$ ::= “*First*(”, *function*, “) = ”, *event* ;

PRED$\bar{\beta}$ ::= “*Last*(”, *function*, “) = ”, *event* ;

PREDε ::= *function*, “(”, *tracevar*, “) = < >”
| *function*, “(”, *tracevar*, “) ≠ < >”;

*function* ::= *tracevar* | “(”, *function*, “↾”, *eventset*, “)”
| “*Head*(”, *function*, “)” | “*Tail*(”, *function*, “)”;

*event* ::= “**a**” | “**b**” | .. ** list of event names **

*tracevar* ::= “s” | “t” | .. ** list of trace variable names **

*eventset* ::= “{”, *event* , “,”, .., *event* “}” ■

## 8.3 Expressing Specifications in ERSPEC.

This section illustrates how ERSPEC can be used to capture different properties for processes in CSP. This is achieved by a series of specific examples which demonstrate how certain requirements are shown to be equivalent to behavioural specifications in ERSPEC.

The most straightforward properties which ERSPEC can capture are those which are simply predicates in one of the syntaxes PRED$\bar{\alpha}$, PRED$\bar{\beta}$, PRED$\gamma$ or PRED$\delta$. To demonstrate these consider the following example

**Example 8.1:** Suppose that a requirement of a process `P` is that the third event is always the event **`swing`**. This is equivalent to demonstrating that the first event of any trace which has its first two events removed is the event **`swing`**. That is

$$\mathcal{First}(\mathcal{Tail}(\mathcal{Tail}(\mathtt{s}))) \quad = \quad \mathtt{swing} \qquad \text{Eq. 8.1}$$

■

Note that equation 8.1 is an expression in PRED$\bar{\alpha}$ and as such belongs to ERSPEC. The reasoning employed in Example 8.1 can be quickly extended to cover the general case. That is showing that the $\mathtt{n}^{\mathtt{th}}$ event (where $\mathtt{n} \in \mathbb{N}$) of a trace is some event, say `a`. This can be written as

$$\mathcal{First}(\mathcal{Tail}^{\mathtt{n-1}}(\mathtt{s})) \quad = \quad \mathtt{a} \qquad \text{Eq. 8.2}$$

Likewise, consider the requirement "The fourth event from the end of a trace must be **`rattle`**". This can be demonstrated by removing three events from the end of the trace and comparing the remaining last event with **`rattle`**. That is

$$\mathcal{Last}(\mathcal{Head}^{4}(\mathtt{s})) \quad = \quad \mathtt{rattle} \qquad \text{Eq. 8.3}$$

Again this can be extended to the general case of the $\mathtt{n}^{\mathtt{th}}$ event from the end of a trace by the expression

$$\mathcal{Last}(\mathcal{Head}^{\mathtt{n}}(\mathtt{s})) \quad = \quad \mathtt{rattle} \qquad \text{Eq. 8.4}$$

From equations 8.6 and 8.8 this is seen to be equivalent to

$$\#\mathtt{s} = \mathtt{n} \quad \Leftrightarrow \quad \mathcal{Head}^{\mathtt{n}-1}(\mathtt{s}) \neq \langle\rangle \wedge \mathcal{Head}^{\mathtt{n}}(\mathtt{s}) = \langle\rangle \qquad \text{Eq. 8.10}$$

Now consider the following example

**Example 8.3:** Suppose that it is required for some process that a trace either satisfies some property Q or that trace is < **roger**, **over**, **out** >. This requirement would conventionally be expressed

$$\mathtt{Q}(\mathtt{s}) \quad \vee \quad \mathtt{s} = \langle\, \mathbf{roger}, \mathbf{over}, \mathbf{out} \,\rangle \qquad \text{Eq. 8.11}$$

The right hand side expression of equation 8.11 is not in the syntax of ERSPEC and thus neither is the whole equation. However, the statement s = < **roger**, **over**, **out** > is equivalent to saying that the first event of s must be **roger**, the second event must be **over**, the third event must be **out** and that the trace must be of length 3. Using previous results this can be expressed

$$\begin{aligned}&\mathcal{First}(\mathtt{s}) = \mathbf{roger} \;\wedge\; \mathcal{First}(\mathcal{Tail}(\mathtt{s})) = \mathbf{over} \;\wedge\\ &\mathcal{First}(\mathcal{Tail}^2(\mathtt{s})) = \mathtt{out} \;\wedge\; \mathcal{Head}^2(\mathtt{s}) \neq \langle\rangle \;\wedge\\ &\mathcal{Head}^3(\mathtt{s}) = \langle\rangle \end{aligned} \qquad \text{Eq. 8.12}$$

■

Examples 8.2 and 8.3 demonstrate how it is possible to use an understanding of ERSPEC to express requirements as predicates in ERSPEC. By making the assumption that the alphabet of a process is finite it is possible to express other types of requirement in ERSPEC. Consider the following example

**Example 8.5:** A group of children encounter a chocolate machine which vends **flake**, **bounty**, **twirl** and **crunchie** bars. While it does not matter which bar comes out first, in the interests of a peaceful afternoon it is required that all the bars are the same as the first.

On first inspection this requirement can be expressed as

$$\mathcal{First}(\mathtt{s}) = \mathcal{Last}(\mathtt{s}) \qquad \text{Eq. 8.13}$$

However, equation 8.13 is not an expression in ERSPEC. Suppose it is assumed that the chocolate machine is restricted to the following four events, each corresponding to the machine delivering a chocolate bar.

$$\mathrm{A} = \{\mathbf{flake}, \mathbf{bounty}, \mathbf{twirl}, \mathbf{crunchie}\} \qquad \text{Eq. 8.14}$$

Then it is seen that the process has a finite alphabet consisting of the above four events A. The terms *First*(s), *Last*(s) therefore can each have one of five possible values, namely $\emptyset$, **flake**, **bounty**, **twirl** or **crunchie**. These terms are equal when they both have the same value. That is they are equal if the following holds

$$\begin{array}{ll} & (\mathcal{First}(\mathrm{s}) = \mathbf{flake} \wedge \mathcal{Last}(\mathrm{s}) = \mathbf{flake}) \\ \vee & (\mathcal{First}(\mathrm{s}) = \mathbf{bounty} \wedge \mathcal{Last}(\mathrm{s}) = \mathbf{bounty}) \\ \vee & (\mathcal{First}(\mathrm{s}) = \mathbf{twirl} \wedge \mathcal{Last}(\mathrm{s}) = \mathbf{twirl}) \\ \vee & (\mathcal{First}(\mathrm{s}) = \mathbf{crunchie} \wedge \mathcal{Last}(\mathrm{s}) = \mathbf{crunchie}) \qquad \text{Eq. 8.15} \end{array}$$

Thus, provided the alphabet of the process concerned is as above, it can be seen that equation 8.15 is equivalent to the expression *Last*(s) = *First*(s). ■

By restricting processes to those with finite alphabets, it is also possible to express other predicates in a similar form to that of equation 8.15. Examples are

$$\mathrm{s} \upharpoonright \mathrm{A} = \mathrm{s} \qquad \text{Eq. 8.16}$$

$$\mathcal{First}(\mathrm{f}(\mathrm{s})) = \mathcal{Last}(\mathrm{g}(\mathrm{s})), \text{ where f, g are } \mathcal{HTR} \text{ functions} \qquad \text{Eq. 8.17}$$

## 8.4 A Worked Example.

This section illustrates the use of Ideal Test Sets as developed in Chapter 6 and Chapter 7 for verification. It presents an overview of the techniques employed and applies them to a specific example to illustrate their use.

For clarity, and to emphasise the use of the technique, the procedure is abstracted from the work presented in previous chapters. It was considered that a informal description, given in point form, would serve to augment the understanding of the method of Ideal Test Sets.

### 8.4.1 A Procedure for Generating Ideal Test Sets.

For a process P and a behavioural specification R the following procedure for verification by the method of Ideal Test Sets is presented

i) Determine whether or not the process/specification pair (P, R) is a suitable candidate for verification. That is, ensure that the process P can be expressed as the fixed point of a Catenary function and that the behavioural specification R is in ERSPEC.

ii) Express the behavioural specification in conjunctive normal form as

$$R \equiv \bigwedge_{i=1}^{m} S_i \qquad \text{Eq. 8.18}$$

where each $S_i$ is the disjunction of predicates from NEWCATEGS. That is

$$\forall i \exists n \bullet ( S_i \equiv \bigvee_{j=1}^{n} Q_{ij}) \qquad \text{Eq. 8.19}$$

where $Q_{ij}$ are predicates from NEWCATEGS. The expressions $S_1,..,S_m$ which represent the conjunction of predicates in NEWCATEGS are termed the provisional predicates of R.

iii) Categorize each of the disjoint predicates $Q_{ij}$ from the provisional predicates of R into one of the categories α, β, γ, δ, $\bar{\alpha}$, $\bar{\beta}$. This is achieved by determining to which of the syntaxes PREDα, PREDβ, PREDγ, PREDδ, PRED$\bar{\alpha}$, PRED$\bar{\beta}$ or PREDε they belong.

iv) For each provisional predicate $S_i$ of R use the theorems given in Chapter 6 and Chapter 7 to generate Ideal Test Sets for the pair (P, $S_i$). This results in a number of Ideal Test Sets, termed the provisional Ideal Test Sets.

v) Using Theorem 6.16 the intersection of all provisional Ideal Test Sets yields an Ideal Test Set I for the pair (P, R).

vi) It finally remains to determine the correctness of process P relative to specification R by testing R against all the behaviours in I.

### 8.4.2 The Arbor Drum

Consider the Arbor Drum mechanism given in Chapter 4. This concerned a CSP process which modelled the control logic of a high speed rotating drum with a slider which periodically inserted into an aperture on the edge of the drum [ Clarke 92a ]. The motivation behind choosing this particular problem is to provide a means of demonstrating that the method of Ideal Test Sets can successfully be applied to an industrial problem. The CSP model of the controller and the formal representation of the system requirements were constructed before the main body of the theory of Ideal Test Sets was developed. As such they are more likely to represent a 'real' problem rather than an example specifically tailored for the theory. In addition it cites an example which has already been solved by the existing method of axiomatic compositional proof.

In order to establish the hazard free operation of the mechanism it was necessary to prove that the slider and drum would never collide. This was formulated as a safety property and expressed as a behavioural specification on the untimed traces model. Its formal representation was

$$\begin{array}{ll} (\texttt{SLIDER}\ {}_{A}\|_{B}\ \texttt{CON})\ {}_{A\cup B}\|_{C}\ \texttt{DRUM} & \\ \textbf{sat} & \\ \mathit{Last}(s\upharpoonright\{\texttt{enter}, \texttt{extract}\}) = \texttt{enter} & \\ \Rightarrow\quad \mathit{Last}(s\upharpoonright\{\texttt{dstart}, \texttt{dstop}\}) \neq \texttt{dstart} & \text{Eq. 8.20} \end{array}$$

In Chapter 4 it was shown how equation 8.20 could be established by first determining that certain behavioural specifications held for each of the three processes SLIDER, CON and DRUM. Once established these specifications were composed under an inference rule associated with the alphabetized parallel operator to show the correctness of the process (SLIDER $_{A}\|_{B}$ CON) $_{A\cup B}\|_{C}$ DRUM.

This example is concerned with establishing the first of these criteria. That is showing that certain specifications hold for each of the processes SLIDER, CON and DRUM. The requirement for SLIDER was that it satisfy the predicate given as $\Xi_1$. More formally that the following statement was true

$$\begin{array}{ll} \texttt{SLIDER}\ \textbf{sat}\ \Xi_1 & \\ \Xi_1(s) \equiv \mathit{Last}(s\upharpoonright\{\texttt{enter}, \texttt{extract}\}) = \texttt{enter} & \\ \Rightarrow (\mathit{Last}(s\upharpoonright\{\texttt{commit}, \texttt{allow}, \texttt{extract}\}) = \texttt{commit} & \\ \vee\ \mathit{Last}(s\upharpoonright\{\texttt{commit}, \texttt{allow}, \texttt{extract}\}) = \texttt{allow}) & \text{Eq. 8.21} \end{array}$$

To establish this using the method of Ideal Test Sets the procedure outlined in Section 8.4.1 will be pursued. Stage (i) requires that the suitability of the candidate process and specification is shown. This is so if `SLIDER` is the fixed point of a Catenary function and if $\Xi_1$ is expressible in ERSPEC. By definition the process `SLIDER` is the fixed point of the function `F`, where

$$\begin{aligned}\texttt{F(P)} = \texttt{approach} \rightarrow\ & ((\,\texttt{commit} \rightarrow \texttt{enter} \rightarrow \texttt{slow} \rightarrow \\ & \;\texttt{perform} \rightarrow \texttt{extract} \rightarrow \texttt{P}\,) \\ & \Box \\ & (\,\texttt{abort} \rightarrow \texttt{decelerate} \rightarrow \texttt{allow} \\ & \;\rightarrow \texttt{accelerate} \rightarrow \texttt{enter} \rightarrow \texttt{slow} \\ & \;\rightarrow \texttt{perform} \rightarrow \texttt{extract} \rightarrow \texttt{P}\,))\end{aligned}$$

Eq. 8.22

`F` is composed entirely of prefix and deterministic choice operators. Thus, by reference to Corollary 5.1, it is a Catenary function. It was also established in Chapter 4 that `F` was a contraction on $M_T$. Therefore `SLIDER` can be deduced to be the unique fixed point of a Catenary function.

By comparing the behavioural specification $\Xi_1$ against the syntax of ERSPEC it can also be seen that it obeys the syntactic rules and thus is an expression of ERSPEC.

Having established the suitability of the pair ( `SLIDER`, $\Xi_1$ ) as candidates for the method of Ideal Test Sets, stage (ii) requires the behavioural specification $\Xi_1$ to be expressed in conjunctive normal form.

$$\begin{aligned}\Xi_1(s) \equiv\ & \mathit{Last}(s\upharpoonright\{\texttt{enter}, \texttt{extract}\}) = \texttt{enter} \\ & \Rightarrow (\mathit{Last}(s\upharpoonright\{\texttt{commit}, \texttt{allow}, \texttt{extract}\}) = \texttt{commit} \\ & \vee \mathit{Last}(s\upharpoonright\{\texttt{commit}, \texttt{allow}, \texttt{extract}\}) = \texttt{allow})\end{aligned}$$

Eq. 8.23

$$\begin{aligned}\equiv\ & \neg(\mathit{Last}(s\upharpoonright\{\texttt{enter}, \texttt{extract}\}) = \texttt{enter}\,) \\ & \vee\ \mathit{Last}(s\upharpoonright\{\texttt{commit}, \texttt{allow}, \texttt{extract}\}) = \texttt{commit} \\ & \vee\ \mathit{Last}(s\upharpoonright\{\texttt{commit}, \texttt{allow}, \texttt{extract}\}) = \texttt{allow}\end{aligned}$$

Eq. 8.24

At this point, for convenience, define the predicates $\Re_1$, $\Re_2$, $\Re_3$ as

$$\Re_1 \equiv \neg(\mathit{Last}(s\upharpoonright\{\texttt{enter}, \texttt{extract}\}) = \texttt{enter}\,)$$ Eq. 8.25

$$\mathfrak{R}_2 \quad \equiv \quad \mathcal{L}ast(s\restriction\{\mathtt{commit, allow, extract}\}) = \mathtt{commit}$$

Eq. 8.26

$$\mathfrak{R}_3 \quad \equiv \quad \mathcal{L}ast(s\restriction\{\mathtt{commit, allow, extract}\}) = \mathtt{allow}$$

Eq. 8.27

so that

$$\Xi_1(\mathtt{s}) \quad \equiv \quad (\mathfrak{R}_1 \vee \mathfrak{R}_2 \vee \mathfrak{R}_3)$$ Eq. 8.28

Stage (iii) requires that each of the predicates $\mathfrak{R}_1$, $\mathfrak{R}_2$, $\mathfrak{R}_3$ be placed into one of the defined categories. An inspection of each of $\mathfrak{R}_1$, $\mathfrak{R}_2$, $\mathfrak{R}_3$ reveals that they all belong to the syntax of PRED$\bar{\beta}$. Therefore, by Statement 7.3, they also all belong to the category $\bar{\beta}$. The specific conditions under which they belong to $\bar{\beta}$ are given by

$$\mathfrak{R}_1 \in (\mathtt{R} \mid \forall\ \mathtt{s} \bullet \mathtt{s}\restriction\{\mathtt{enter, extract}\} \neq \langle\rangle \quad \Rightarrow \quad (\mathtt{R(t)} \Leftrightarrow \mathtt{R(s\hat{\ }t)}))$$ Eq. 8.29

$$\mathfrak{R}_2, \mathfrak{R}_3 \in (\mathtt{R} \mid \forall\ \mathtt{s} \bullet \mathtt{s}\restriction\{\mathtt{commit, allow, extract}\} \neq \langle\rangle \quad \Rightarrow \quad (\mathtt{R(t)} \Leftrightarrow \mathtt{R(s\hat{\ }t)}))$$ Eq. 8.30

Let $\mathtt{M(s)}$ represent the condition

$$\mathtt{s}\restriction\{\mathtt{commit, allow, extract}\} \neq \langle\rangle \quad \wedge \quad \mathtt{s}\restriction\{\mathtt{enter, extract}\} \neq \langle\rangle$$ Eq. 8.31

Applying Theorem 7.9 gives

$$\mathfrak{R}_1 \vee \mathfrak{R}_2 \vee \mathfrak{R}_3 \in (\mathtt{R} \mid \forall\ \mathtt{s} \bullet \mathtt{M(s)} \Rightarrow \mathtt{R(t)} \Leftrightarrow \mathtt{R(s\hat{\ }t)})$$ Eq. 8.32

Stage (iv) requires the generation of the provisional Ideal Test Sets. There is no logical conjunction in the normal form of $\Xi_1$ and so there is only one provisional Ideal Test Set for the pair ($\mathtt{SLIDER}$, $\mathfrak{R}_1 \vee \mathfrak{R}_2 \vee \mathfrak{R}_3$). By Theorem 7.10 this is given by

$$Traces(\mathtt{F}^{\mathtt{x}}(\mathtt{STOP}))$$ Eq. 8.33

where $\mathtt{x}$ is the order of the pair ($\mathtt{F}$, $\mathfrak{R}_1 \vee \mathfrak{R}_2 \vee \mathfrak{R}_3$). To establish the order $\mathtt{x}$ first employ Definition 5.3 to determine the set $\mathcal{D}_{\mathcal{F}}$ as follows

$$\mathcal{D}_{\mathcal{F}} = \{\langle \mathtt{approach}, \mathtt{commit}, \mathtt{enter}, \mathtt{slow}, \mathtt{perform}, \mathtt{extract} \rangle, \langle \mathtt{approach}, \mathtt{abort}, \mathtt{decelerate}, \mathtt{allow}, \mathtt{accelerate}, \mathtt{enter}, \mathtt{slow}, \mathtt{perform}, \mathtt{extract} \rangle \} \qquad \text{Eq. 8.34}$$

By comparing each of the two traces in $\mathcal{D}_{\mathcal{F}}$ against M it is seen that M holds for both. Using the definition of the order of (F, M) it can be seen that since M holds for all traces in ($\mathcal{D}_{\mathcal{F}}^1$), then the order of (F, M) is 2. Therefore an Ideal Test Set I for the pair (SLIDER, $\mathfrak{R}_1 \vee \mathfrak{R}_2 \vee \mathfrak{R}_3$) is

$$\mathrm{I} = \mathit{Traces}(\mathrm{F}^2(\mathrm{STOP})) \qquad \text{Eq. 8.35}$$

It finally remains to compare the behavioural specification $\Xi_1$ against each of the specific traces in $\mathrm{F}^2$(STOP). There are 43 traces in total and each was successfully tested against $\Xi_1$. This leads to the final conclusion that since I is an Ideal Test Set for the pair ($\mu$P.F(P), $\Xi_1$), and $\Xi_1$ holds for I, then

$$\mathrm{SLIDER}\ \mathbf{sat}\ \Xi_1 \qquad \text{Eq. 8.36}$$

By adopting the same procedure it is possible to determine that the following also holds

$$\begin{aligned}
&\mathrm{CON}\ \mathbf{sat}\ \psi_1,\\
&\psi_1(\mathrm{s}) \equiv\\
&(\mathit{Last}(\mathrm{s}\restriction\{\ \mathtt{commit}, \mathtt{allow}, \mathtt{extract}\ \}) = \mathtt{commit}\\
&\vee\ \mathit{Last}\ (\mathrm{s}\restriction\{\ \mathtt{commit}, \mathtt{allow}, \mathtt{extract}\ \}) = \mathtt{allow})\\
&\Rightarrow\\
&(\mathit{Last}\ (\mathrm{s}\restriction\{\ \mathtt{dstart}, \mathtt{dstop}\ \}) \neq \mathtt{dstart}) \qquad \text{Eq. 8.37}
\end{aligned}$$

Having established that equations 8.36 and 8.37 are valid, the alphabetized parallel operator can now be applied to establish the truth of equation 8.20.

### 8.4.3 Comparisons.

Two methods have now been used to establish the hazard free operation of the slider and drum mechanism with respect to the specifications and models given in Chapter 3. This permits some comparisons to be made on the different methods employed.

It was noted that where the method of Ideal Test Sets was applied it proved to be more procedural and well defined than the axiomatic techniques. In Chapter 4 the truths of the statements SLIDER **sat** $\Xi_1$ and CON **sat** $\psi_1$ were established by an intuitive analysis of their process defining functions. This analysis was also helped by the fact that the processes were relatively simple. In contrast the method of Ideal Test Sets has an algorithmic approach which does not rely on intuition or the simplicity of the model to establish such predicates.

However, it was also noted that the method of Ideal Test Sets was not sufficient to cope with the alphabetized parallel operator, and thus could not establish the correctness of the mechanism as a whole. This is in contrast to the complete approach of the axiomatic verification. This inabililty to treat certain models was recognised at an early stage in the development of these techniques. From its inception the method of Ideal Test Sets has been structured so as to be compatible with the axiomatic approach. Thus when difficulties such as those presented by the inclusion of the alphabetized parallel operator are encountered the underlying axiomatic proof system is always available to provide a solution.

## 8.5 Conclusions.

This thesis presents a method for verifying the correctness of a specific class of CSP processes, namely the fixed point of Catenary functions, relative to a particular set of specifications, namely those defined by the syntax ERSPEC. This method is based on the concept of Ideal Test Sets. An Ideal Test Set is a finite subset of the behaviours of a process over which the relative correctness of a particular specification can be determined.

The main perceived advantage of the method of Ideal Test Sets is that it reduces the proof obligation. It has been demonstrated that the correctness of a particular specification relative to a recursive process is equivalent to the correctness of that same specification relative to a simpler and finite process. This removes much of the complexity of verification. Indeed for suitable process/specification pairs it removes the need for the inference rule connected with the recursion operator.

Another advantage to the technique is that the method of Ideal Test Sets has developed a procedural approach to verification coupled with a precise syntactic definition of suitable process/specification pairs. These factors suggest that the method lends itself well to automated verification techniques such as those described in Chapter 5. Model checking in particular suggests itself since the proposed method generates a finite set of process behaviours over which the truth for the whole process can be determined.

There are strong indications that the original approach of Chapter 6 which used the syntax RSPEC and the categories $\alpha$, $\beta$, $\gamma$, $\delta$ is fully automatable. It must, however, be noted that there are at present reservations about a fully automated version for ERSPEC specifications introduced in Chapter 7. This is based on a realization that for certain process/specification pairs ($\mu$P.F(P), R), where the predicate $R \in \{ \bar{\alpha} \cup \bar{\beta} \}$ and N is the corresponding condition on R as defined in Definition 7.12, there are difficulties in determining the order of the pair (F, N).

One of the disadvantages of the method is the fact that it is restricted to a particular class of processes as well as a particular set of predicates and thus cannot be freely applied. This is in comparison to the axiomatic approach of the compositional proof system described in Chapter 3 which is complete for the semantic domains $M_T$, $M_F$ and $TM_F$ and applies to all behavioural specifications [ Blamey 91, Schneider 90 ].

With respect to these constraints this thesis has emphasised the importance of being able to clearly and efficiently define the limits of the method of Ideal Test Sets. It was considered to be important that there existed a procedure for determining the suitability of a particular process/specification pair for verification with Ideal Test Sets. For this purpose the syntactic definition of Catenary functions and ERSPEC were developed. Provided a process/specification pair meets these syntactic requirements then the pair will also conform to the semantic requirements necessary for structuring an Ideal Test Set. The advantage of the syntactic definitions are that they provide a concise set of rules which require no interpretation.

Another observation made of the method of Ideal Test Sets is that it does not truly address the concepts of concurrency for which CSP was originally developed. This is seen as the result of two factors. First, the alphabetized parallel operator was shown in Chapter 5 to be not Catenary, and secondly the method is restricted to the fixed point of a single valued CSP function.

A response to the realization that the alphabetized parallel operator was not Catenary was the introduction of the concept of Weak Catenary functions to which alphabetized parallelism belonged. It was originally hoped that weakening the requirement of Catenary functions to that of Weak Catenary functions would still permit a viable theory of Ideal Test Sets. However it ultimately only proved possible to establish limited results for Weak Catenary functions. It was found that if R were a predicate such that $R \in \sigma^n$ and F were a Weak Catenary function, then

$$(\forall\, s \in \underbrace{(\mathit{Traces}(F(STOP)\ \hat{}\,..\ \hat{}\,\mathit{Traces}(F(STOP))}_{n\ terms} \bullet R(s)\,) \;\Rightarrow\; \mu P.F(P)\ \mathbf{sat}\ R$$

Eq. 8.38

The problem with this result is that it was also found that there were many correct process/specification pairs for which the antecedent of equation 8.38 did not hold. Such a method would only be able to prove the truth of a limited number of specifications and the falsehood of none. Thus the method would only be partially effective.

The difficulties associated with the application of the method to processes which were not fixed point Catenary solutions is addressed either by decomposing the process into suitable fixed point solutions and then using the existing axiomatic proof system to complete the proof (as was used in Section 8.4) or by efforts to establish a process's equivalence to a fixed point solution as discussed in Section 8.6.3.

In conclusion, it is the opinion of this thesis that although the method of Ideal Test Sets is restricted in terms of the properties it can verify and the processes to which it may be applied, its strength lies in the fact that these limitations are known and are clearly defined by the semantic and syntactic constraints set upon it. These limits mean that suitable process/specification pairs can be identified and treated accordingly, and that any remaining proof obligation can be addressed by the compositional axiomatic proof system. Because of this its role is seen as that of an aid to verification which may be used in conjunction with existing axiomatic proof methods.

## 8.6 Suggestions for Further Work.

### 8.6.1 Extending the Theory of Ideal Test Sets to the Failures Model.

The theory presented in this thesis has concentrated on generating Ideal Test Sets for the simplest semantic representation of CSP processes, that of the traces model. However, as was noted in Chapter 3, the traces model has certain shortcomings in that it is unable to capture liveness properties and it cannot semantically distinguish between deterministic and nondeterministic choice. Therefore it would be useful if it were possible to extend the theory of Ideal Test Sets into the failures domain $M_F$.

The main difficulty perceived in achieving this is the fact that the theory developed has been heavily based on the concept of catenation and, unlike for $M_T$, there is no proper definition of the catenation of behaviours over $M_F$. That is the behaviours of $M_T$ are traces and they can be directly concatenated, however the behaviours of $M_F$ are trace-refusal pairs, which cannot be directly concatenated.

To overcome this it would be necessary to develop a concept of catenation for $M_F$. That is to develop for this thesis a rule by which any two behaviours of $M_F$ can be joined together as if one immediately followed the other. From an understanding of the principles of CSP the following rule for catenation over $M_F$ is proposed

**Definition 8.2:** Let $(\mathtt{s}, \aleph_1)$ and $(\mathtt{t}, \aleph_2)$ be two behaviours in the semantic domain $M_F$. Then the trace/refusals pair which represents the behaviour of $(\mathtt{s}, \aleph_1)$ followed by the behaviour of $(\mathtt{t}, \aleph_2)$ is given by

$$(\mathtt{s}, \aleph_1)^{\neg}(\mathtt{t}, \aleph_2) \quad = \quad (\mathtt{s}\hat{\ }\mathtt{t}, \aleph_2), \text{ provided } \mathit{First}(\mathtt{t}) \notin \aleph_1 \qquad \text{Eq. 8.39}$$

The symbol $^{\neg}$ is introduced here as failures catenation. ■

Using such a definition of failures catenation as a basis rather than traces catenation, it is then possible to form categories $\alpha_F$, $\beta_F$, $\gamma_F$, $\delta_F$ for $M_F$ which correspond to the categories $\alpha$, $\beta$, $\gamma$, $\delta$ introduced for $M_T$.

**Definition 8.3:** The categories $\alpha_F$, $\beta_F$, $\gamma_F$, $\delta_F$, are defined as sets of predicates such that

$$\alpha_F = \{\mathtt{R} \mid \forall\ \mathtt{s} \neq \langle\rangle \bullet \mathtt{R}(\mathtt{s}, \aleph_1) \Leftrightarrow \mathtt{R}((\mathtt{s}, \aleph_1)^{\neg}(\mathtt{t}, \aleph_2))\ \} \qquad \text{Eq. 8.40}$$

$$\beta_F = \{\mathtt{R} \mid \forall\ \mathtt{t} \neq \langle\rangle \bullet \mathtt{R}(\mathtt{t}, \aleph_2) \Leftrightarrow \mathtt{R}((\mathtt{s}, \aleph_1)^{\neg}(\mathtt{t}, \aleph_2))\ \} \qquad \text{Eq. 8.41}$$

$$\begin{aligned}\gamma_F = \{&\mathtt{R} \mid \forall\ \mathtt{s}, \mathtt{t} \neq \langle\rangle \bullet \\ &\quad \mathtt{R}(\mathtt{s}, \aleph_1) \vee \mathtt{R}(\mathtt{t}, \aleph_2) \Leftrightarrow \mathtt{R}((\mathtt{s}, \aleph_1)^{\neg}(\mathtt{t}, \aleph_2))\ \}\end{aligned} \qquad \text{Eq. 8.42}$$

$$\begin{aligned}\delta_F = \{&\mathtt{R} \mid \forall\ \mathtt{s}, \mathtt{t} \neq \langle\rangle \bullet \\ &\quad \mathtt{R}(\mathtt{s}, \aleph_1) \wedge \mathtt{R}(\mathtt{t}, \aleph_2) \Leftrightarrow \mathtt{R}((\mathtt{s}, \aleph_1)^{\neg}(\mathtt{t}, \aleph_2))\ \}\end{aligned} \qquad \text{Eq. 8.43}$$

■

These categories could then be used to formulate a corresponding theory of Ideal Test Sets for pairs ($\mu\mathtt{P}.\mathtt{F}(\mathtt{P})$, $\mathtt{R}$) where $\mathtt{R}$ was a behavioural specification on $M_F$.

In terms of the scope of such an extended theory it can be seen that because the failures catenation rule preserves the structure of trace catenation, then a predicate in $\{ \alpha \cup \beta \cup \gamma \cup \delta \}$ is also a predicate in $\{ \alpha_F \cup \beta_F \cup \gamma_F \cup \delta_F \}$. Thus the extended theory would at least have the expressibility of RSPEC. Added to this would be the ability to represent predicates on failures. Consider the example predicate

$$\mathtt{R}(\mathtt{s}, \aleph) \equiv \mathtt{a} \in \aleph \qquad \text{Eq. 8.44}$$

Inspection shows that R belongs to the category $\beta_F$ as defined above. In fact it is possible to see that all predicates which take only the refusals as their subject would belong to the category $\beta_F$, because failures catenation simply preserves the refusals of the suffix behaviour.

However this still suggests a syntax which treats predicates over traces and predicates over refusals separately. In the opinion of this thesis the true challenge to developing a method of Ideal Test Sets for $M_F$ would lie in structuring a syntax and corresponding theory which allowed predicates that range over both traces and refusals simultaneously.

### 8.6.2 A Timed Failures Extension.

Just as for the failures model, extending the theory of Ideal Test Sets to the Timed Failures domain would necessitate a definition of the catenation of two behaviours of $TM_F$. One definition suggested here is

**Definition 8.4:** Let $(s, \aleph_1)$ and $(t, \aleph_2)$ be two behaviours of a TCSP process in the timed failures domain $TM_F$. Furthermore let the expression $r+\tau$ indicate the trace r which has the time value $\tau$ added to each of its timed events, and let the expression $\aleph+\tau$ indicate the set which adds a time $\tau$ to every timed event in the set of refusals $\aleph$. That is

$$\aleph+\tau = ((t+\tau, e)|(t, e) \in \aleph) \qquad \text{Eq. 8.45}$$

Then the timed trace/timed refusals pair which represents the behaviour of $(s, \aleph_1)$ followed by the behaviour of $(t, \aleph_2)$ is given by

$$(s, \aleph_1)\_(t, \aleph_2)$$
$$= (s\hat{\ }(t+\phi), \aleph_2+\phi) \quad \text{(provided } (\mathcal{First}(t), \mathit{start}(t)) \notin \aleph_1 \qquad \text{Eq. 8.46}$$

where $\phi$ is some value such that

$$\phi > \mathit{end}(s) - \mathit{start}(t) \qquad \text{Eq. 8.47}$$

and *start*(t) is the time at which trace t performs its first event.

The symbol _ is introduced here as timed failures catenation. ■

### 8.6.3 Catenary Functions and Fixed Points.

Chapter 5 and Chapter 6 demonstrated that an integral part of the method of Ideal Test Sets was that in its stated form it could only be applied to processes which were the fixed points of Catenary functions.

However, it is also noted that there exist processes which, while not explicitly represented as the fixed point of a Catenary function, can be shown to be equivalent to one. For example consider the process given by

$$\mathtt{A} = \{\ \mathbf{a}, \mathbf{b}, \mathbf{c}\ \} \qquad \& \qquad \mathtt{B} = \{\ \mathbf{a}, \mathbf{c}, \mathbf{d}\ \} \qquad \text{Eq. 8.48}$$

$$\mathtt{F(P)} = \mathbf{a} \rightarrow \mathbf{b} \rightarrow \mathbf{c} \rightarrow \mathtt{P} \qquad \& \qquad \mathtt{G(P)} = \mathbf{a} \rightarrow \mathbf{d} \rightarrow \mathbf{c} \rightarrow \mathtt{P} \qquad \text{Eq. 8.49}$$

$$\mathtt{PROC1} \quad = \quad \mu\mathtt{P.F(P)}\ {}_{\mathtt{A}}\|_{\mathtt{B}}\ \mu\mathtt{P.G(P)} \qquad \text{Eq. 8.50}$$

PROC1 is not defined as the fixed point of a CSP function, rather as the alphabetized parallel combination of processes $\mu$P.F(P) and $\mu$P.G(P) which are themselves fixed points. However, analysis of PROC1 reveals that PROC1 is equivalent to $\mu$ P.H(P), where

$$\mathtt{H(P)} = \mathbf{a} \rightarrow (\mathbf{b} \rightarrow \mathtt{SKIP} \sqcap \mathbf{c} \rightarrow \mathtt{SKIP})\ ;\ \mathbf{d} \rightarrow \mathtt{P} \qquad \text{Eq. 8.51}$$

The statement that PROC1 is equivalent to $\mu$ P.H(P) is made on the basis that

$$\mathit{Failures}(\mathtt{PROC1}) = \mathit{Failures}(\mu\ \mathtt{P.H(P)}) \qquad \text{Eq. 8.52}$$

Establishing this equivalence suggests that another related area of further study would be to investigate the use of a system of algebraic rules to transform processes into those which are the fixed point of a Catenary function. There already exist published algebraic laws for Dijkstra's language of guarded commands [ Hoare 87 ] and a similar set of laws for occam [ Roscoe 88b ] which is derived from those of CSP [ Hoare 85 ]. It is envisaged that it would be algebraic systems such as these which would form the basis of an approach to demonstrate the equivalence of certain processes to fixed point Catenary solutions.

Finally it is noted that the current semantic definition of a Catenary function given by Definition 5.5 is derived only for the untimed failures model $M_T$. In order to extend the method of Ideal Test Sets to higher semantic models as proposed in Section 8.6.1 and 8.6.2 it would be necessary to formulate new semantic definitions for Catenary functions with respect to the proposed new definitions of failures and timed failures catenation.

[Berthomieu 83] Berthomieu B., Menascre M., An Enumeration Approach for Analysing Time Petri Nets, *Proceedings of the IFIP Congress*, Paris, Pages 41 - 46. 1983.

[Bjørner 78] Bjørner D., Jones C.B., The Vienna Development Method: The Meta Language., *Springer Verlag Lecture Notes in Computer Science*, Volume 61, 1978.

[Blamey 89] Blamey S., TCSP Processes as Predicates, *Oxford University Programming Research Group Technical Report*, Draft Copy, December, 1989.

[Blamey 91] Blamey S., The Soundness and Completeness of Axioms for CSP Processes, *Topology and Category Theory in Computer Science*, Edited by Reed G.M., Roscoe A.W. & Wachter R.F., Pages 29 - 56, Clarendon Press, 1991.

[Borowski 89] Borowski E.J., Borwein J.M., *Collins Dictionary of Mathematics*, Collins UK, 1989.

[Boucher 87] Boucher A., Gerth R., A Timed Model for Extended Communicating Sequential Processes, Proceedings of ICALP '87, *Springer Verlag Lecture Notes in Computer Science* Volume 267 Pages 95 - 113, 1987.

[Brookes 83] Brookes, S.D., On the Relationship of CCS and CSP, *Springer Verlag Lecture Notes in Computer Science* Volume 154, Pages 83 - 96, 1983.

[Brookes 84] Brookes S.D., Hoare C.A.R., Roscoe A.W., A Theory of Communicating Sequential Processes, *Journal of The ACM* N° 31 (7), Pages 560 - 599, 1984.

[Brookes 85] Brookes S.D., Roscoe A.W., An Improved Failures Model for Communicating Sequential Processes, Proceedings NSF-SERC Seminar on Concurrency, *Springer Verlag Lecture Notes in Computer Science* Volume 197 Pages 17 - 43, 1985.

[Bryant 85] Bryant V., *Metric Spaces - Iteration and Application*, Cambridge University Press, 1985.

[Camilleri 90] Camilleri A.J., Mechanizing CSP Trace Theory in Higher Order Logic, *IEEE Transactions on Software Engineering*, Volume SE-16 N° 9, Pages 993-1004, 1990.

[Camilleri 91] Camilleri A.J., A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics, *Proceedings of the 4th Higher Order Workshop, Banff 1990,* Edited by G. Birtwistle, Workshops in Computing, Springer Verlag, Banff, Canada 10-14 September 1990, Pages 123-150, 1991.

[Clarke 92a] Clarke P.J., Holding D.J., The Specification, Design and Verification of Real-Time Embedded Control Logic using CSP and TCSP. *Proceedings of the IFAC International Workshop on Real-Time Programming WRTP '92*. Bruges, Belgium, IFAC, 23 - 26 June 1992.

[Clarke 92b] Clarke P.J., Holding D.J., The Verification of a CSP Process by Constructing an Ideal Test Set, Proceedings of the 4th IMA International Conference on Control: Modelling, Computation, Information, Manchester U.K., IMA & IEEE, Sept 2-4 1992

[Cleaveland 90] Cleaveland R., Parrow J., Steffen B., The Concurrency Workbench, *Springer Verlag Lecture Notes in Computer Science*, Volume 407, Pages 24- 37.

[Cohen 86] Cohen B., Harwood W.T., Jackson M.I., *The Specification of Complex Systems*, Addison Wesley, 1986.

[Darringer 78] Darringer J.A., King J.C., Application of Symbolic Execution to Program Testing, *Computer*, Volume 11 N° 4, Pages 34 - 40, 1978.

[Davies 87] Davies J.W., *Assisted Proofs for Communicating Sequential Processes*, M.Sc. Dissertation, Oxford University Computer Laboratory, September 1987.

[Davies 89a] Davies J.W., Schneider S.A., An Introduction to Timed CSP, *Oxford University Programming Research Group Monograph*, PRG - 75, Pages 1 - 35, 1989.

[Davies 89b] Davies J.W., Schneider S.A., Factorizing Proofs in Timed CSP, *Oxford University Programming Research Group Monograph*, PRG - 75, Pages 36 - 70, 1989.

[Davies 92] Davies J.W., Jackson M., Schneider S.A., A Brief History of Timed CSP, *Oxford University Programming Research Group Monograph*, PRG - 91 Pages1 - 35, 1992.

[de Roever 85] de Roever W.P., The Quest for Compositionality - A Survey of Assertion Based Proof Systems for Concurrent Programs, *The Role of Abstract Models in Computer Science*, Edited by E.J. Neuhold, Pages 181 - 206, 1985.

[Dijkstra 76] Dijkstra E.W., *A Discipline of Programming*, Prentice Hall International, 1976.

[Dijkstra 81] Dijkstra E.W., Why Correctness must be a Mathematical Concern, *The Correctness Problem in Computer Science*, Edited by R.S. Boyer and J.S. Moore, International Lecture Series in Computer Science, Academic Press UK, Pages 1 - 9, 1981.

[Edwards 91] Edwards J., Lawson P., The Advancement of Transputers and Occam, *Occam and the Transputer - Current Developments*, Proceedings of the 14th World Occam and Transputer User Group Technical Meeting, 16-18th September 1991, Loughborough, UK, Edited by J. Edwards, Pages 1 - 12, IOS Press, 1991

[Flon 81] Flon L., Suzuki N., The Total Correctness of Parallel Programs, *Siam Journal on Computing*, Volume 10, N° 2, pages 227 - 246, 1981.

[Floyd 67] Floyd R., Assigning Meaning to Programs, *Mathematical Aspects of Computer Science*, XIX American Mathematical Society, Pages 19 - 32, 1967.

[Froome 88] Froome P., Monahan B., The Role of Mathematically Formal Methods in the Development and Assessment of Safety Critical Systems, *Microprocesors and Microsystems* Volume 12 N° 10, Pages 539 - 546, December 1988.

[Galton 90] Galton A., *Logic for Information Technology*, Wiley Press UK, 1990.

[Glendinning 89] Glendinning I., What makes Occam so Interesting?, *Occam User Group Newsletter*, N° 11, Pages 29 - 32, July 1989.

[Goodenough 75] Goodenough J.B., Gerhart S., Towards a Theory of Test Data Selection, *IEEE Transactions on Software Engineering*, Volume SE-1 N° 2, 1975.

[Gordon 88] Gordon M.J.C., A Proof Generating System for Higher Order Logic, *VLSI Specification, Verification and Synthesis*, Edited by G. Birtwistle and P.A. Subrahmanyam, Kluwer Academic Publishers, Boston, 1988.

[Gries 81] Gries, D., *The Science of Programming*, Springer Verlag, 1981.

[Hamilton 78] Hamilton A. G., *Logic For Mathematicians* , Cambridge Press, 1978.

[Hayes 87] Hayes I.J. (Editor), *Specification Case Studies*, Prentice Hall International, 1987.

[Herstein 75] Herstein I., *Topics in Algebra*, Xerox College Publications, 1975.

[Hoare 69] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *Communications of the ACM*, Volume 12 N° 10, Pages 576 - 583, October 1969.

[Hoare 78] Hoare C.A.R., Communicating Sequential Processes, *Communications of the ACM*, Volume 21 N° 8, Pages 666 - 677, August 1978.

[Hoare 81] Hoare C.A.R., A Calculus of Total Correctnes for Communicating Processes, *Oxford University Programming Research Group Technical Monograph PRG-23*, 1981.

[Hoare 85] Hoare C.A.R., *Communicating Sequential Processes*, Prentice Hall International, 1985.

[Hoare 87] Hoare C.A.R., Hayes I.J., He J., Morgan C.C., Roscoe A.W., Sanders J.W., Sorenson I.H., Spivey, J.M., Sufrin B.A., The Laws of Programming, *Communications of the ACM*, Volume 30 N° 8, Pages 672 - 686, 1987.

[Hoare 91] Hoare C.A.R., Personal Electronic Mail Communication to clarkepj@aston.ac.uk, November 1991.

[Hull 86] Hull M.E.C., Implementations of the CSP Notation for Concurrent Systems, *Computer Journal*, Volume 29 N° 6, Pages 500 - 505, 1986.

[Inmos 88a] Inmos Ltd., *The Transputer Instruction Set,* Prentice Hall International, 1988.

[Inmos 88b] Inmos Ltd., *Occam2 Reference Manual*, Prentice Hall International, 1988.

[Inverardi 91] Inverardi P., Camilleri A., Nesi M., Combining Interaction and Automation in Process Algebra Verification, Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2, Edited by S. Abramsky and T. Maibaum, *Springer Verlag Lecture Notes in Computer Science* Volume 494, Pages 283 - 296, Brighton, 6 - 13 April 1991.

[ISO 91] ISO SC22/WG19(1991). VDM Specification Language - Proto-Standard, *British Standardisation Institute ISO Standard,* 1991.

[Jackson 89] Jackson D.M., *The Specification of Aircraft Engine Control Software Using Timed CSP*, M.Sc. Thesis, Oxford University Programming Research Group, 1989.

[Jazayeri 80] Jazayeri M., CSP/80: A Language for Communicating Sequential Processes, *Proceedings of the Fall IEEE COMPCON 80*, IEEE Press New York, Pages 736 - 740, 1980.

[Jones 86] Jones, C.B., *Systematic Software Development using VDM*, Prentice Hall International, 1986.

[Joseph 89] Joseph M., Goswami A., Formal Description of Realtime Systems: A Review, *Information and Software Technology*, Volume 31, N° 2, pages 67 - 76, March 1989.

[Kieburtz 79] Kieburtz R.B., Silberschatz A., Comments on "Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems*, Volume 1 N° 2, Pages 218 - 225, October 1979.

[King 90] King S., Z and the Refinement Calculus, *Oxford University Programming Research Group Technical Monograph N° 79*, Febuary 1990.

[Kourie 87] Kourie D.G., The Design and Use of a Prolog Trace Generator for CSP, *Software - Practice and Experience*, Volume 17 N° 7, Pages 423 - 438, 1987.

[Lamport 77] Lamport L., Proving the Correctness of Multiprocess Programs, *IEEE Transactions on Software Engineering*, Volume SE-3 N° 2, Pages 125 - 143, March 1977.

[Lamport 84] Lamport L., An Axiomatic Semantics of Programming Languages, in *Logics and Models of Concurrent Systems* edited by K.R. Apt, Proceedings of the NATO Advanced Studies Institute on Logic and Models of Concurrent Systems held at.La Colle-sur-Loup, France, October 1984.

[Lanski 89] Lanski J., Testing in the Program Development Cycle, *Software Engineering Journal*, Volume 4 N° 2, Pages 95-106, 1989.

[Leveson 93] Leveson N.G., Introduction to Special Issue on Software for Critical Systems, IEEE Transactions on Software Engineering, Volume SE-19 Nº 1, Page 1, 1993.

[Levin 81] Levin G.M., Gries D., A Proof Technique For Communicating Sequential Processes, *Acta Informatica* Volume 15, Pages 281-302, 1981.

[Lightfoot 91] Lightfoot D., *Formal Specification using Z*, Macmillan Computer Sciences Series, Macmillan UK, 1991.

[Manna 88] Manna Z., Pnueli A., Temporal Verification of Concurrent Programs, *The Correctness Problem in Computer Science*, Wiley Press, Pages 215 - 273, 1988.

[Merlin 76] Merlin P.M., Farber D.J., Recoverability of Communication Protocols, Implications of a Theoretical Study, *IEEE Transactions on Communications*, Volume COM-24 Nº 9, Pages 1036-1043, 1976.

[Meyer 90] Meyer B., *Introduction to the Theory of Programming Languages*, Prentice Hall International, 1990.

[Milner 80] Milner R., A Calculus of Communicating Systems, *Springer Verlag Lecture Notes in Computer Science*, Volume 90, 1980.

[Milner 83] Milner R., Calculi for Synchrony and Asynchrony, *Theoretical Computer Science*, Volume 25, Pages 267-310, 1983.

[Milner 89] Milner R., *Communication and Concurrency*, Prentice Hall, 1989.

[Misra 81] Misra J., Chandy K.M., Proofs of Networks of Processes, *IEEE Tranactions on Software Engineering*, Volume SE-7 Nº 4, Pages417 - 426, July 1981.

[Mitchell 90] Mitchell D.A.P., Thompson J.A., Manson G.A., Brookes G.R., *Inside the Transputer*, Blackwell Scientific Publications, 1990.

[Murata 89] Murata T., Petri-Nets: Properties, Analysis and Applications, *Proccedings of the IEEE*, Volume 77 Nº 4, Pages 541 - 578, April 1989.

[Murtagh 87] Murtagh T.P., Redundant Proofs of Non-interference in Levin-Gries CSP Program Proofs, *Acta Informatica*, Volume 24 Nº 2, Pages 145 - 156, 1987

[New Sci 89] The Times when Software could Kill, *New Scientist*,11 Febuary 1989 Pages 53-56.

[Nielson 92] Nielson H.R., Nielson F., *Semantics With Applications, a Formal Introduction*, Wiley Press UK, 1992.

[Olderog 86] Olderog E.R., Hoare C.A.R., Specification-Oriented Semantics for Communicating Processes, *Acta Informatica* Volume 23 Nº 1, Pages 9 - 66, April 1986.

[Owicki 76] Owicki S.S., Gries D., Verifying Properties of Parallel Programs: An Axiomatic Approach, *Communications of the ACM*, Volume 19 N° 5, Pages 279 - 285, May 1976.

[Pagan 81] Pagan F.G., *Formal Specification of Programming Languages*, Prentice Hall, 1981.

[Peterson 81] Peterson, J.L., *Petri-Net Theory and the Modelling of Systems*, Prentice Hall International, 1981.

[Petri 66] Petri C., Communication With Automata, *Technical Report RADC-TR-65-377*, Volume 1, Supplement 1, Applied Data Research, Princeton, New Jersey. Translation of Kommunikation mit Automaten. *Bonn: Institut für Instrumentelle Mathematik*, Schriften des IIm N° 2, 1966.

[Pnueli 86] Pnueli A., Applications of Temporal Logic to the Specification and Verification of Reactive systems: A Survey Of Current Trends, in *Current Trends in Concurrency: Overviews and Tutorials*, edited by de Roever W. P., Rozenberg G., *Springer Verlag Lecture Notes in Computer Science*, Volume 224, Pages 510-584, 1986.

[Pountain 88] Pountain R., May D., *A Tutorial Introduction to Occam Programming*, Blackwell Scientific Publications, 1988.

[Prasad 84] Prasad B., Interference Freedom in Proofs of CSP Programs. *Proceedings of the 4th IEEE International Conference on Distributed Computer Systems*, San Francisco, Pages 79 - 86, May 1984.

[Ramsay 88] Ramsay A., *Formal Methods in Artificial Intelligence*, Cambridge University Press, 1988.

[Reed 86] Reed G.M., Roscoe A.W., A Timed Model for Communicating Sequential Processes, *Theoretical Computer Science* 58, Pages 249 - 261, 1988.

[Reed 87] Reed G.M., Roscoe A.W., Metric Spaces as Models for Real-Time Concurrency, Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics, *Springer Verlag Lecture Notes in Computer Science*, Volume 298 , Pages 331 - 343, 1987.

[Reed 90] Reed G.M., A Hierarchy of Domains for Real-Time Distributed Computing, Mathematical Foundations of Programming Semantics, *Springer Verlag Lecture Notes in Computer Science* Volume 442, Pages 80 - 128, 1990.

[Revesz 83] Revesz G., *Introduction to Formal Languages*, McGraw-Hill Computer Science Series, 1983.

[Roper 81] Roper T.J., Barter C.J., A Communicating Sequential Processes Language and Implementation, *Software - Practice and Experience* Volume 11 N° 11, Pages 1215 - 1234, 1981.

[Roscoe 82] Roscoe A.W., A Mathematical Theory of Communicating Sequential Processes, D.Phil. Dissertation, Bodelian Library, Oxford University, UK,1982.

[Roscoe 88a] Roscoe A.W., Two Papers on CSP, *Oxford University Programming Research Group Monograph* N° 93, 1988.

[Roscoe 88b] Roscoe A.W., Hoare C.A.R., The Laws of Occam Programming, *Theoretical Computer Science*, Volume 60 Issue 2, Pages 177-229, September 1988.

[Sagoo 90] Sagoo J.S., Holding D.J., The Specification and Design of Hard Real-Time Systems Using Timed and Temporal Petri Nets, *Microprocessing and Microprogramming* 30, Pages 389 - 396 1990.

[Schneider 90] Schneider S.A., Correctness and Communication in Real-Time Systems, *Oxford University PRG Technical Monograph N° 84*, March 1990.

[Scholefield 90] Scholefield D.J. The Formal Development of Real - Time Systems: A Review, *University of York - Department of Computer Science Report* YCS 145 (1990), 1990

[Silberschatz 79] Silberschatz, A., Communication and Synchronization in Distributed Systems, *IEEE Transactions on Software Engineering*, Vol SE-5 N° 6, Pages 542 - 546.

[Silberschatz 81] Silberschatz A., Port Directed Communication, *Computer Journal* Volume 24 N° 1 Pages 78 - 82, 1981

[Soundararajan 84] Soundararajan N., Axiomatic Semantics of Communicating Sequential Processes, *ACM Transactions on Programming Languages and Systems*, Volume 6, Pages 647 - 662, 1984.

[Stoy 77] Stoy J.E., *Denotational Semantics*, MIT Press, 1977.

[Sutherland 75] Sutherland W.A., Introduction to Metric and Topological Spaces, Open University Press, 1975.

[Toetenel 92] Toetenel H., VDM + CCS + Time = MOSCA, *Proceedings of the IFAC International Workshop on Real-Time Programming WRTP '92*. Bruges, Belgium 23 - 26 June 1992 IFAC.

[Tofts 90] Tofts C., Moller F., A Temporal Calculus of Communicating Systems,Proceedings of CONCUR '90, Edited by J.C.M. Baeten and J.W. Klop, *Springer Verlag Lecture Notes in Computer Science* Volume 458, Pages 401 - 415, 1990

[VDM 87] Proceedings of the 1987 Workshop on Vienna Design Method, *Springer Verlag Lecture Notes in Computer Science* Volume 252, 1987.

[Watt 91] Watt D.A., *Programming Language Syntax and Semantics*, Prentice Hall International, 1991.

[Wilkstrom 87] Wilksrom A., *Functional Programming Using Standard ML*, Prentice Hall International, 1987.

[Wilson 84] Wilson R., *Graph Theory*, Open University Press, 1984.

[Wing 90] Wing J. M. , A Specifiers Introduction to Formal Methods, *IEEE Computer Journal*, Volume 23 N° 9, Pages 8-24, September 1990.

[Woodcock 87] Woodcock, J.C.P., Sorensen, I.H., Mathematics for Specification and Design: The Problem with Lifts..., *Proceedings of the Fourth International Workshop on Software Specification and Design.*, Monterey CA, USA , IEEE Computer Society, April 3-4, 1987.

[Woodcock 88] Woodcock J., Loomes M., *Software Engineering Mathematics*, Pitman Publishing, 1988.

[Zave 82] Zave P., An Operational Approach to Requirement Specifications for Embedded systems, *IEEE Transactions on Software Engineering*, Volume 8 N° 3, Pages 250 - 269, May 1982.

[Zave 86] Zave P., Schell W., Salient Features of an Executable Specification Language and its Environment, *IEEE Transactions on Software Engineering,* Volume 12 N° 3, Pages 312 - 325, May 1986.

[Zave 91] Zave P., An Insiders Evaluation of PAISLey, *IEEE Transactions on Software Engineering*, Volume 17 N° 3, Pages 212 - 225, March 1991.

[Zwarico 85] Zwarico A., Lee I., Proving a Network of Real-Time Processes Correct, *Proceedings of the Real-Time Systems Symposium*, Dec. 1985, California, Pages 169-177, IEEE Press, 1985.

# APPENDIX A

**Definition A.1 (3.6):** Let U be a set and let d be a function such that $d : U \times U \rightarrow \mathbb{R}$, ( $\mathbb{R}$ is the set of real numbers ). Let x, y, z be members of U. Then the pair ( U, d ) is said to be a metric space if and only if the following hold

i) $\forall x, y \in U \quad \bullet \quad d(x, y) \geq 0$ Eq. A.1

ii) $\forall x, y \in U \quad \bullet \quad d(x, y) = 0 \quad \Leftrightarrow \quad x = y$ Eq. A.2

iii) $\forall x, y \in U \quad \bullet \quad d(x, y) = d(y, x)$ Eq. A.3

iv) $\forall x, y, z \in U \bullet \quad d(x, y) + d(y, z) \geq d(x, z)$ Eq. A.4

■

**Definition A.2:** Let ( U, d ) be a metric space and let $a_1, a_2, a_3, ..$ be a sequence of elements from the set U. The sequence is said to be a Cauchy sequence if and only if

$d(a_m, a_n) \rightarrow 0$ as $m, n \rightarrow \infty$ Eq. A.5

The set $A \subseteq U$ is said to be complete in the metric space ( U, d ) if every Cauchy sequence in A converges to a limit in A. ■

**Theorem A.1:** Let f, g be mappings of the complete metric space ( U, d ) onto itself. ( f, g: $U \rightarrow U$ ). Then the following rules hold

i) If f and g are contractions, then so is $f \circ g$.

ii) If f and g are nonexpansions, then so is $f \circ g$.

iii) If f is a contraction and g is a nonexpansion, then $f \circ g$ is a contraction.

iv) If f is a nonexpansion and g is a contraction, then $f \circ g$ is a contraction.

*Proof*

From the definitions of contractions and non-expansions for all functions $f$ and $g$ there exist positive real values $k_1$ and $k_2$ such that

$$d(x, y) \leq k_1.d(f(x), f(y)) \quad \text{Eq. A.6}$$

$$d(f(x), f(y)) \leq k_2.d(g(f(x)), g(f(y))) \quad \text{Eq. A.7}$$

$$\therefore \quad d(x, y) \leq k_1.k_2.d(f \circ g(x), f \circ g(y))$$

Eq. A.8

The corresponding cases are

*Case i)*
$k_1 < 1$ and $k_2 < 1$, therefore $k_1.k_2 < 1$, thus $f \circ g$ is a contraction.

*Case ii)*
$k_1 \leq 1$ and $k_2 \leq 1$, therefore $k_1.k_2 \leq 1$, thus $f \circ g$ is a nonexpansion.

*Case iii)*
$k_1 < 1$ and $k_2 \leq 1$, therefore $k_1.k_2 < 1$, thus $f \circ g$ is a contraction.

*Case iv)*
$k_1 \leq 1$ and $k_2 < 1$, therefore $k_1.k_2 < 1$, thus $f \circ g$ is a contraction.

□

**Theorem A.2:** (*Contraction Mapping Theorem* ) Let $F: U \rightarrow U$ be a contraction of the complete metric space ( $U$, $d$ ). Then $F$ has a unique fixed point. Furthermore, if $x_1$ is any member of $U$ the sequence,

$$x_1, \quad x_2 = F(x_1), \quad x_3 = F(x_2) = F(F(x_1)), \quad \ldots \quad \text{Eq. A.9}$$

converges to that unique fixed point. ■

*Proof*

Consider the sequence

$$x_i: x_1, x_2 = F(x_1), .., x_{i+1} = F(x_i), .. \qquad \text{Eq. A.10}$$

The metric between two consecutive elements of $x_i$ is given by

$$d(x_n, x_{n+1}) = d(F(x_{n-1}), F(x_n)) \qquad \text{Eq. A.11}$$

$F$ is a contraction and thus there is some $k < 1$ such that

$$d(F(x_{n-1}), F(x_n)) \leq k.d(x_{n-1}, x_n) \qquad \text{Eq. A.12}$$

$$\leq k.d(F(x_{n-2}), F(x_{n-1})) \qquad \text{Eq. A.13}$$

$$\leq k^2.d(x_{n-2}, x_{n-1}) \qquad \text{Eq. A.14}$$

$$:::: \quad ::::$$

$$\leq k^{n-2}.d(F(x_1), F(x_2)) \qquad \text{Eq. A.15}$$

$$\leq k^{n-1}.d(x_1, x_2) \qquad \text{Eq. A.16}$$

Thus $d(x_n, x_{n+1}) \leq k^{n-1}.d(x_1, x_2)$. This result can be generalised to $d(x_n, x_m)$ as follows. Using the triangle rule (Definition A.1(iv))

$$d(x_n, x_m) \leq d(x_n, x_{n+1}) + d(x_{n+1}, x_{n+2}) + .. + d(x_{m-1}, x_m) \qquad \text{Eq. A.17}$$

$$\leq k^{n-1}.d(x_1, x_2) + k^n.d(x_1, x_2) + ... + k^{m-2}.d(x_1, x_2) \qquad \text{Eq. A.18}$$

$$\leq (k^{n-1}(1 + k + k^2 + ...)).d(x_1, x_2) \qquad \text{Eq. A.19}$$

$$\leq \frac{k^{n-1}}{1-k}.d(x_1, x_2) \qquad \text{Eq. A.20}$$

Equation A.20 tends towards zero as $n$ tends towards infinity. Additionally $F$ is a mapping from $U$ to $U$. Thus the sequence $x_i$ is a Cauchy sequence. Now since $U$ is complete the sequence $x_i$ has its limit in $U$. Let $x$ be the limit of $x_i$.

$$x_i \rightarrow x \qquad \text{Eq. A.21}$$

Since $\forall\, n \bullet x_{n+1} = F(x_n)$ it is possible to see that the sequence generated by $F(x_i)$ will tend to $F(x)$. The sequences $x_i$ and $F(x_i)$ tend towards the same limit and so

$$x = F(x) \qquad \text{Eq. A.22}$$

Thus the limit of $x_i$ is a fixed point of $F$. To show that this is unique let $x_a$ and $x_b$ be fixed points of the contraction $F$. Assume $x_a \neq x_b$. Then by definition

$$d(x_a, x_b) \quad = \quad d(F(x_a), F(x_b)) \quad \neq 0 \qquad \text{Eq. A.23}$$

Therefore there is no real number $k < 1$ such that

$$k.d(x_a, x_b) \geq \quad d(F(x_a), F(x_b)) \qquad \text{Eq. A.24}$$

Thus $F$ is not a contraction over $(U, d)$. The initial assumption that $x_a \neq x_b$ must be false and so

$$x_a = x_b \qquad \text{Eq. A.25}$$

Since there is only one fixed point of $F$ and the limit of $x_i$ is a fixed point, it follows that the limit of $x_i$ is a unique fixed point.

□

**Definition A.3:** Let $P_1 = (A_1, T_1)$ and $P_2 = (A_2, T_2)$ be two processes in $M_T$ ( See Definition 3.5). The partial ordering $\sqsubseteq$ over $M_T$ is defined as follows

$$(A_1, T_1) \sqsubseteq (A_2, T_2) \quad \equiv \quad (A_1 = A_2 \wedge T_1 \subseteq T_2) \qquad \text{Eq. A.26}$$

For processes $X_1$, $X_2$ and $X_3$ this ordering satisfies the rules

i) $X_1 \sqsubseteq X_1$ Eq. A.27

ii) $(X_1 \sqsubseteq X_2 \wedge X_2 \sqsubseteq X_1) \Rightarrow X_1 = X_2$ Eq. A.28

iii) $(X_1 \sqsubseteq X_2 \wedge X_2 \sqsubseteq X_3) \Rightarrow X_1 \sqsubseteq X_3$ Eq. A.29

Furthermore, a function is monotonic if it preserves some partial ordering [ Lipschitz 69 ]. That is the function $F$ is monotonic if for processes $X_1$ and $X_2$

$$X_1 \sqsubseteq X_2 \quad \Rightarrow \quad F(X_1) \sqsubseteq F(X_2)$$ Eq. A.30

The CSP operators of hiding, deterministic choice, nondeterministic choice, interleaving, prefix, change of symbol, parallel, alphabetized parallel and sequential are all monotonic over the above partial ordering on $M_T$.

■

# APPENDIX B

**Definition 5.1 :** The proofs for each of the rules are as follows

i) Follows from the result that

$$\forall\, s \bullet s\hat{\ }\langle\rangle = \langle\rangle\hat{\ }s = s \qquad \text{Eq. B.1}$$

ii) Follows from the associativity of catenation, i.e.

$$\forall\, s, t, u \bullet s\hat{\ }(t\hat{\ }u) = (s\hat{\ }t)\hat{\ }u \qquad \text{Eq. B.2}$$

iii) By definition

$$B \cup C = \{\, t \mid t \in B \vee t \in C \,\} \qquad \text{Eq. B.3}$$

$$\therefore \quad A\hat{\ }(B \cup C) \;=\; \{\, s\hat{\ }t \mid s \in A \wedge (t \in B \vee t \in C) \,\} \qquad \text{Eq. B.4}$$

$$= \{\, s\hat{\ }t \mid (s \in A \wedge t \in B) \vee (s \in A \wedge t \in C) \,\} \qquad \text{Eq. B.5}$$

$$= (A\hat{\ }B) \cup (A\hat{\ }C) \qquad \text{Eq. B.6}$$

the second result follows from the associativity of ^.

iv) By definition

$$B \cap C = \{\, t \mid t \in B \wedge t \in C \,\} \qquad \text{Eq. B.7}$$

$$\therefore \quad A\hat{\ }(B \cap C) \;=\; \{\, s\hat{\ }t \mid s \in A \wedge (t \in B \wedge t \in C) \,\} \qquad \text{Eq. B.8}$$

$$= \{\, s\hat{\ }t \mid (s \in A \wedge t \in B) \wedge (s \in A \wedge t \in C) \,\} \qquad \text{Eq. B.9}$$

$$= (A\hat{\ }B) \cap (A\hat{\ }C) \qquad \text{Eq. B.10}$$

the second result follows from the associativity of ^.

v) By definition

$A\hat{}B - A\hat{}C$

$= \quad \{ s\hat{}t \mid s \in A \wedge t \in B \} - \{ s\hat{}t \mid s \in A \wedge t \in C \}$ Eq. B.11

$= \quad \{ s\hat{}t \mid s \in A \wedge t \in (B - C) \} \quad = \quad A\hat{}(B - C)$ Eq. B.12

second result follows from the associativity of ^.

vi) Follows from the rule that for a set of events A

$\forall\, s \bullet (s \upharpoonright A)\hat{}(t \upharpoonright A) = (s\hat{}t) \upharpoonright A$ Eq. B.13

vii) By definition

$\{\}\ \hat{}\ A = \{ s\hat{}t \mid s \in \{\} \wedge t \in A \}$ Eq. B.14

However, $s \in \{\}$ can never be true since s must always be at least the empty trace. Thus the expression $(s \in \{\} \wedge t \in A)$ is always false. The result follows.

□

**Theorem 5.6, Lemma C:** If A, B, C and D are sets then

$(A - B) \cap D = \{\} \quad \wedge \quad (C - D) \cap B = \{\}$

$\Rightarrow \quad (A - B) \cup (C - D) = (A \cup C) - (B \cup D)$ Eq. B.15

■

*Proof*

To prove this some initial results are stated from [ Lipshitz 69 ]. For all sets A, B the following equivalence holds

$A - B \quad = \quad A \cap B^C$ Eq. B.16

where $B^C$ is the complement of B. Additionally, for all sets A, B

$A \cap B = \{\} \quad \Leftrightarrow \quad A \cap B^C = A$ Eq. B.17

Now consider the expression

$$((A - B) \cup (C - D)) \cap (B \cup D) \qquad \text{Eq. B.18}$$

This expands to

$$((A - B) \cap B)) \cup ((A - B) \cap D) \cup ((C - D) \cap B) \cup ((C - D) \cap D) \qquad \text{Eq. B.19}$$

And using equation B.16

$$\begin{aligned} = \quad & (A \cap B^C \cap B) \cup ((A - B) \cap D) \cup \\ & ((C - D) \cap B) \cup (C \cap D^C \cap D) \qquad \text{Eq. B.20} \end{aligned}$$

The initial assumption in Lemma C was that

$$(A - B) \cap D = \{\} \quad \wedge \quad (C - D) \cap B = \{\} \qquad \text{Eq. B.21}$$

Using this and the fact that a set intersected with its complement yields the empty set it is seen that equation B.20 is equivalent to { }. Thus it follows from equation B.17 that

$$((A - B) \cup (C -D)) \cap (B \cup D)^C \quad = \quad ((A - B) \cup (C -D)) \qquad \text{Eq. B.22}$$

Now consider the expression

$$(A \cup C) - (B \cup D) \quad = \quad (A \cup C) \cap (B \cup D)^C \qquad \text{Eq. B.23}$$

$$= \quad (A \cup C) \cap (B^C \cap D^C) \qquad \text{Eq. B.24}$$

$$= \quad (A \cap B^C \cap D^C) \cup ( C \cap B^C \cap D^C) \qquad \text{Eq. B.25}$$

$$= \quad ((A - B) \cap D^C) \cup ((C - D) \cap B^C) \qquad \text{Eq. B.26}$$

$$\begin{aligned} = \quad & ((A - B) \cup (C -D)) \cap ((A-B) \cup B^C) \\ & \qquad \cap ((C-D) \cup D^C) \cap (D^C \cup B^C) \qquad \text{Eq. B.27} \end{aligned}$$

$$= \quad ((A - B) \cup (C -D)) \cap B^C \cap D^C \cap (D^C \cup B^C) \qquad \text{Eq. B.28}$$

$$= \quad ((A - B) \cup (C - D)) \cap (B \cup D)^{C} \qquad \text{Eq. B.29}$$

$$= \quad ((A - B) \cup (C - D)) \qquad \text{Eq. B.30}$$

Thus the equality is established □

# APPENDIX C

**Lemma 6.6(a) :** For positive integers $i, j$ let $Q_j^i$ be a statement in propositional calculus. Then the following equivalence holds:

$$((Q_1^1 \wedge Q_2^1 \wedge .. \wedge Q_n^1) \vee .. \vee (Q_1^{n-1} \wedge Q_2^{n-1} \wedge .. \wedge Q_n^{n-1}))$$

$$\Leftrightarrow \bigwedge_{i=1}^{n} ((Q_1^1 \wedge .. \wedge Q_{i-1}^1 \wedge Q_{i+1}^1 \wedge .. \wedge Q_n^1) \vee ..$$

$$\vee (Q_1^{n-1} \wedge .. \wedge Q_{i-1}^{n-1} \wedge Q_{i+1}^{n-1} \wedge .. \wedge Q_n^{n-1})) \qquad \text{Eq. C.1}$$

■

*Proof*

The following notation is introduced for convenience. For positive integers $i, j, n$ let the expression $\mathfrak{R}_{i,n}^j$ be such that

$$\mathfrak{R}_{i,n}^j = (Q_1^j \wedge .. \wedge Q_{i-1}^j \wedge Q_{i+1}^j \wedge .. \wedge Q_n^j) \qquad \text{Eq. C.2}$$

$$\mathfrak{I}_n^j = (Q_1^j \wedge Q_2^j \wedge \wedge Q_n^j) \qquad \text{Eq. C.3}$$

From this it can be seen that equation C.1 can be re-expressed as

$$\mathfrak{I}_n^1 \vee .. \vee \mathfrak{I}_n^{n-1}$$

$$\Leftrightarrow \bigwedge_{i=1}^{n} (\mathfrak{R}_{i,n}^1 \vee \mathfrak{R}_{i,n}^2 \vee .. \vee \mathfrak{R}_{i,n}^{n-1}) \qquad \text{Eq. C.4}$$

□

To establish this result two further lemmas are required.

**Lemma A :** For arbitrary propositions $R$ and $S$ the following equivalence holds

$$R \Leftrightarrow R \vee (R \wedge S) \qquad \text{Eq. C.5}$$

■

*Proof*

This can be shown by the following truth table

| $R$ | $S$ | $R \vee (R \wedge S)$ |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

Eq. C.6

□

**Lemma B:** For integers $j$, $p$, $q$, $n$ such that $p \neq q$:

$$\mathfrak{R}^{j}_{p,n} \wedge \mathfrak{R}^{j}_{q,n} \quad \Leftrightarrow \quad \mathfrak{I}^{j}_{n} \tag{Eq. C.7}$$

■

*Proof*

By definition

$$\mathfrak{R}^{j}_{p,n} \quad \Leftrightarrow \quad (Q^{j}_{1} \wedge .. \wedge Q^{j}_{p-1} \wedge Q^{j}_{p+1} \wedge .. \wedge Q^{j}_{n}) \tag{Eq. C.8}$$

Also, since $p \neq q$

$$\mathfrak{R}^{j}_{q,n} \quad \Leftrightarrow \quad (Q^{j}_{1} \wedge .. \wedge Q^{j}_{p} \wedge .. \wedge Q^{j}_{n}) \tag{Eq. C.9}$$

$$\therefore \quad \mathfrak{R}^{j}_{p,n} \wedge \mathfrak{R}^{j}_{q,n} \quad \Leftrightarrow \quad (Q^{j}_{1} \wedge Q^{j}_{2} \wedge .. \wedge Q^{j}_{n}) \tag{Eq. C.10}$$

□

Returning to the main proof, by expanding the right hand side of equation C.4 it can be seen that

$$\bigwedge_{i=1}^{n} (\mathfrak{R}^{1}_{i,n} \vee \mathfrak{R}^{2}_{i,n} \vee .. \vee \mathfrak{R}^{n-1}_{i,n})$$

$$\Leftrightarrow \quad (\mathfrak{R}^{1}_{1,n} /\vee \mathfrak{R}^{2}_{1,n} \vee .. \vee \mathfrak{R}^{n-1}_{1,n}) \wedge (\mathfrak{R}^{1}_{2,n} \vee \mathfrak{R}^{2}_{2,n} \vee .. \vee \mathfrak{R}^{n-1}_{2,n}) \wedge ..$$

$$\wedge (\mathfrak{R}^{1}_{n,n} \vee \mathfrak{R}^{2}_{n,n} \vee .. \vee \mathfrak{R}^{n-1}_{n,n}) \qquad \text{Eq. C.11}$$

The further expansion of equation C.11 can be expressed as

$$\bigvee \{ (\mathfrak{R}^{x_1}_{1,n} \wedge \mathfrak{R}^{x_2}_{2,n} \wedge .. \wedge \mathfrak{R}^{x_n}_{n,n}) \mid x_1,.., x_n \in \{ 1, .., n \} \} \qquad \text{Eq. C.12}$$

Equation C.12 can be partitioned as $S_1 \vee S_2$, where

$$S_1 \quad = \quad (\bigvee \{ (\mathfrak{R}^{x_1}_{1,n} \wedge \mathfrak{R}^{x_2}_{2,n} \wedge .. \wedge \mathfrak{R}^{x_n}_{n,n}) \mid x_1,.., x_n \in \{ 1, .., n-1 \}$$

$$\wedge\ x_1 = x_2 = .. = x_n \}) \qquad \text{Eq. C.13}$$

$$S_2 \quad = \quad (\bigvee \{ (\mathfrak{R}^{x_1}_{1,n} \wedge \mathfrak{R}^{x_2}_{2,n} \wedge .. \wedge \mathfrak{R}^{x_n}_{n,n}) \mid x_1,.., x_n \in \{ 1, .., n-1 \}$$

$$\wedge\ \neg(x_1 = x_2 =..= x_n) \}) \qquad \text{Eq. C.14}$$

From Lemma B it is possible to deduce that

$$S_1 \quad = \quad \mathfrak{I}^{1}_{n} \vee \mathfrak{I}^{2}_{n} \vee .. \vee\ \mathfrak{I}^{n-1}_{n} \qquad \text{Eq. C.15}$$

The $n$ variables $x_1,.., x_n$ each take one of $n-1$ integer values. Therefore there must always be at least one pair $x_a, x_b$ $(a, b \in \{ 1, .., n-1 \})$ such that $x_a = x_b$. Thus in each expression $(\mathfrak{R}^{x_1}_{1,n} \wedge \mathfrak{R}^{x_2}_{2,n} \wedge .. \wedge \mathfrak{R}^{x_n}_{n,n})$ of $S_2$ there is always a pair $\mathfrak{R}^{x_a}_{1,n}\ \mathfrak{R}^{x_b}_{1,n}$ which conjoin to form $\mathfrak{I}^{x_a}_{n}$. It can be seen by following this reasoning and analysis of $S_2$ that

$$S_2 \quad = \quad (\mathfrak{I}^{1}_{n} \wedge R_1) \vee (\mathfrak{I}^{2}_{n} \wedge R_2) \vee .. \vee\ (\mathfrak{I}^{n-1}_{n} \wedge R_{n-1}) \qquad \text{Eq. C.16}$$

where $R_1, .., R_{n-1}$ are simply some predicates which need not be determined.

The conjunction of $S_1$ and $S_2$ yields

$$\begin{aligned}&(\Im_n^1 \vee \Im_n^2 \vee .. \vee \ \Im_n^{n-1})\\ &\vee ((\Im_n^1 \wedge R_1) \vee (\Im_n^2 \wedge R_2) \vee .. \vee \ (\Im_n^{n-1} \wedge R_{n-1}))\end{aligned} \tag{Eq. C.17}$$

$$\Leftrightarrow$$

$$(\Im_n^1 \vee (\Im_n^1 \wedge R_1)) \vee .. \vee \ (\Im_n^{n-1} \vee (\Im_n^{n-1} \wedge R_{n-1})) \tag{Eq. C.18}$$

Using Lemma A

$$\text{Eq. C.4} \qquad \Leftrightarrow \qquad (\Im_n^1 \vee \Im_n^2 \vee .. \vee \ \Im_n^{n-1}) \tag{Eq. C.19}$$

Thus the theorem holds. □

**Theorem 6.10:** Let $Q$ and $R$ be predicates such that

$$Q \in \delta^n \qquad \& \qquad R \in \beta. \tag{Eq. C.20}$$

The logical disjunction of $Q$ and $R$, $Q \vee R$, is such that

$$Q \vee R \in \sigma^{n+1} \tag{Eq. C.21}$$

■

*Proof*

By the definition of the predicates $Q$ and $R$, for all non-empty traces $r_1, .., r_{n+2}, t$

$$\bigwedge_{i=1}^{n+1} Q(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+1}) \quad \Leftrightarrow \quad Q(r_1\hat{}..\hat{}r_{n+1}) \tag{Eq. C.22}$$

$$\forall t \ \bullet \quad R(r_{n+2}) \quad \Leftrightarrow \quad R(t\hat{}r_{n+2}) \tag{Eq. C.23}$$

Consider the expansion of the below expression

$$\bigwedge_{i=1}^{n+2} Q \vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. C.24}$$

The proof is split into two cases.

*Case $R(r_{n+2})$ is true*

The truth of $R(r_{n+2})$ implies the truth of $R(r_1\hat{}..\hat{}r_{n+2})$, and thus

$$R(r_{n+2}) \wedge \bigwedge_{i=1}^{n+2} Q \vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2})$$
$$\Rightarrow \quad Q \vee R(r_1\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. C.25}$$

*Case $R(r_{n+2})$ is false*

The expansion of equation C.24 is such that

$$\text{Eq. C.24} \Leftrightarrow \bigwedge_{i=1}^{n+1} Q \vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2})$$
$$\wedge\, (Q(r_1\hat{}..\hat{}r_{n+1}) \vee R(r_{n+1})) \qquad \text{Eq. C.26}$$

$$\text{Eq. C.24} \Rightarrow \bigwedge_{i=1}^{n+1} Q \vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. C.27}$$

By making the substitution $s_{n+1} = r_{n+1}\hat{}r_{n+2}$ it is seen that

$$\text{Eq. C.24} \Rightarrow (Q(r_1\hat{}..\hat{}r_n\hat{}r_{n+2}) \vee R(r_{n+2}))$$
$$\wedge \bigwedge_{i=1}^{n} Q \vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_n\hat{}s_{n+1}) \qquad \text{Eq. C.28}$$

By applying equation C.22 it is possible to deduce that

$$Q(r_1\hat{}..\hat{}r_n\hat{}r_{n+2}) \quad \Rightarrow \quad Q(r_1\hat{}..\hat{}r_n) \qquad \text{Eq. C.29}$$

Thus

$$\text{Eq. C.24} \Rightarrow (Q(r_1\hat{}..\hat{}r_n) \vee R(r_{n+2})) \wedge$$
$$\bigwedge_{i=1}^{n} Q \vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_n\hat{}s_{n+1}) \qquad \text{Eq. C.30}$$

By making the substitutions $s_1 = r_1$, $s_2 = r_2$, ...,$s_n = r_n$ and by assuming that $R(r_{n+2})$ is false, equation C.30 becomes

$$\text{Eq. C.24} \Rightarrow \bigwedge_{i=1}^{n+1} Q(s_1\hat{}..\hat{}s_{i-1}\hat{}s_{i+1}\hat{}..\hat{}s_{n+1}) \qquad \text{Eq. C.31}$$

Applying equation C.22 and re-substituting,

$$\text{Eq. C.24} \Rightarrow Q(s_1\hat{}..\hat{}s_{n+1}) \Leftrightarrow \quad Q(r_1\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. C.32}$$

Therefore

$$(\neg R(r_{n+2})) \wedge \bigwedge_{i=1}^{n+2} Q\vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2})$$
$$\Rightarrow Q\vee R(r_1\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. C.33}$$

Resolving equations C.25 and C.33 yields

$$\bigwedge_{i=1}^{n+2} Q\vee R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{n+2}) \Rightarrow \quad Q\vee R(r_1\hat{}..\hat{}r_{n+2}) \qquad \text{Eq. C.34}$$

Thus by definition $Q\vee R \in \sigma^{n+1}$ □

**Theorem 6.11:** Let Q, R and S be predicates such that:

$$Q \in \delta^n \qquad \& \qquad R \in \alpha \qquad \& \qquad S \in \beta \qquad \text{Eq C.35}$$

The logical disjunction of Q, R and S, Q∨R∨S, is such that

$$Q\vee R\vee S \in \sigma^{n+2} \qquad \text{Eq C.36}$$

■

*Proof*

Let W represent Q∨S and let $m = n+1$. It is seen from Theorem 6.10 that

$$W \in \sigma^m \qquad \text{Eq. C.37}$$

Let $u_1, .., u_{m+1}$ be a set of non-empty traces. Using the definitions of predicates $Q$, $S$ it can be deduced that

$$W(u_1\hat{}..\hat{}u_{m+1}) \Leftrightarrow \quad Q(u_1\hat{}..\hat{}u_{m+1}) \vee S(u_1\hat{}..\hat{}u_{m+1}) \qquad \text{Eq. C.38}$$

$$\Rightarrow \quad Q(u_2\hat{}..\hat{}u_{m+1}) \vee S(u_2\hat{}..\hat{}u_{m+1}) \qquad \text{Eq. C.39}$$

$$\Rightarrow \quad W(u_2\hat{}..\hat{}u_{m+1}) \qquad \text{Eq. C.40}$$

Now consider the following expression

$$\bigwedge_{i=1}^{m+2} W{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m+2}) \qquad \text{Eq. C.41}$$

The proof is split into two cases.

*Case $R(r_1)$ is true*

The truth of $R(r_1)$ implies the truth of $R(r_1\hat{}..\hat{}r_{m+2})$, and thus

$$R(r_1) \wedge \bigwedge_{i=1}^{m+2} W{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m+2})$$
$$\Rightarrow \quad W{\vee}R(r_1\hat{}..\hat{}r_{m+2}) \qquad \text{Eq. C.42}$$

*Case $R(r_1)$ is false*

The expansion of equation C.41 is such that

$$\text{Eq. C.41} \Leftrightarrow \quad (W(r_2\hat{}..\hat{}r_{m+2}) \vee R(r_2))$$
$$\wedge \bigwedge_{i=2}^{m+2} W{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m+2}) \qquad \text{Eq. C.43}$$

$$\text{Eq. C.41} \Rightarrow \quad \bigwedge_{i=2}^{m+2} W{\vee}R(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m+2}) \qquad \text{Eq. C.44}$$

By making the substitution $s_1 = r_1\hat{}r_2$ it is seen that

$$\text{Eq. C.41} \Rightarrow \quad (W(r_1\hat{}r_3\hat{}..\hat{}r_{m+2}) \vee R(r_1))$$

$$\wedge \bigwedge_{i=3}^{m+2} \mathtt{W \vee R}(s_1\hat{\ }r_3\hat{\ }..\hat{\ }r_{i-1}\hat{\ }r_{i+1}\hat{\ }..\hat{\ }r_{m+2}) \qquad \text{Eq. C.45}$$

By applying equation C.40 it is possible to deduce that

$$W(r_1\hat{\ }r_3\hat{\ }..\hat{\ }r_{m+2}) \quad \Rightarrow \quad W(r_3\hat{\ }..\hat{\ }r_{m+1}) \qquad \text{Eq. C.46}$$

Thus,

$$\text{Eq. C.41} \Rightarrow (W(r_3\hat{\ }..\hat{\ }r_{m+1}) \vee R(r_1)) \wedge \bigwedge_{i=3}^{m+2} \mathtt{W \vee R}(s_1\hat{\ }r_3\hat{\ }..\hat{\ }r_{i-1}\hat{\ }r_{i+1}\hat{\ }..\hat{\ }r_{m+2}) \qquad \text{Eq. C.47}$$

By making the substitutions $s_2 = r_3$, $s_3 = r_4$, ...,$s_{m+1} = r_{m+2}$ and assuming that $R(r_1)$ is false, equation C.47 becomes

$$\text{Eq. C.41} \Rightarrow \bigwedge_{i=1}^{m+1} W(s_1\hat{\ }..\hat{\ }s_{i-1}\hat{\ }s_{i+1}\hat{\ }..\hat{\ }s_{m+1}) \qquad \text{Eq. C.48}$$

Using the fact that $W \in \sigma^m$, and re-substituting,

$$\text{Eq. C.41} \Rightarrow W(s_1\hat{\ }..\hat{\ }s_{m+1}) \Leftrightarrow \quad W(r_1\hat{\ }..\hat{\ }r_{m+1}) \qquad \text{Eq. C.49}$$

Therefore,

$$(\neg R(r_1)) \wedge \bigwedge_{i=1}^{m+2} \mathtt{W \vee R}(r_1\hat{\ }..\hat{\ }r_{i-1}\hat{\ }r_{i+1}\hat{\ }..\hat{\ }r_{m+2}) \Rightarrow \mathtt{W \vee R}(r_1\hat{\ }..\hat{\ }r_{m+2}) \qquad \text{Eq. C.50}$$

Resolving equations C.42 and C.50 yields

$$\bigwedge_{i=1}^{m+2} \mathtt{W \vee R}(r_1\hat{\ }..\hat{\ }r_{i-1}\hat{\ }r_{i+1}\hat{\ }..\hat{\ }r_{m+2}) \Rightarrow \quad \mathtt{W \vee R}(r_1\hat{\ }..\hat{\ }r_{m+2}) \qquad \text{Eq. C.51}$$

Thus by definition $\mathtt{W \vee R} \in \sigma^{m+1}$. Since $m = n+1$ it is seen that

$$\mathtt{Q \vee R \vee S} \in \sigma^{n+2} \qquad \text{Eq. C.52}$$

□

# APPENDIX D

**Theorem 7.15:** Let Q, R and S be predicates from the categories $\delta^n, \bar{\alpha}, \bar{\beta}$ respectively and M, N be conditions such that

$$\forall M(s) \bullet R(s) \Leftrightarrow R(s\hat{\ }t) \qquad \text{Eq. D.1}$$

$$\forall N(s) \bullet S(t) \Leftrightarrow S(s\hat{\ }t) \qquad \text{Eq. D.2}$$

Let F be a Catenary function. Let x, y be the orders of the pairs (F, M) and (F, N) respectively. Then an Ideal Test Set for the pair ($\mu$P.F(P), Q∨R∨S) is

$$F^{n+x+y}(STOP) \qquad \text{Eq. D.3}$$

■

*Proof*

This theorem is proved by induction

INDUCTIVE STEP

Assume that there is some integer m such that Q∨R∨S holds over

$$Traces(F^{m}(STOP)) \qquad \text{Eq. D.4}$$

Now let r be an arbitrary trace of the process

$$F^{m+1}(STOP) \qquad \text{Eq. D.5}$$

and consider the following cases

*Case* $r \in Traces(F^m(STOP))$

Q∨R∨S holds for r because by definition Q∨R∨S holds over *Traces*($F^m$(STOP)).

*Case* $r \notin Traces(F^m(STOP))$

In this case it can be shown that r is such that

$$r = r_1\hat{\ }..\hat{\ }r_{m+1} \qquad \text{Eq. D.6}$$

where $\mathtt{r}_1,..,\mathtt{r}_\mathtt{m} \in \mathcal{D}_\mathrm{F}$ and $\mathtt{r}_{\mathtt{m}+1} \in \mathtt{F(STOP)}$. Consider the two following subcases

*Subcase* $R(r_1\hat{}..\hat{}r_x) \vee S(r_{m+2-y}\hat{}..\hat{}r_{m+1})$ *holds*

If $\mathtt{x}$ is the order of $\mathtt{(F, M)}$ and $\mathtt{y}$ is the order of $\mathtt{(F, N)}$ then, by definition, $\mathtt{M(r_1\hat{}..\hat{}r_{x+y})}$ and $\mathtt{N(r_1\hat{}..\hat{}r_{x+y})}$ hold. From the definition of $\mathtt{R}$ and $\mathtt{S}$

$$\mathtt{R(r_1\hat{}..\hat{}r_x)} \quad \Leftrightarrow \quad \mathtt{R(r_1\hat{}..\hat{}r_x\hat{}r_{x+1}\hat{}..\hat{}r_{m+1})} \qquad \text{Eq. D.7}$$

$$\mathtt{S(r_{m+2-y}\hat{}..\hat{}r_{m+1})} \quad \Leftrightarrow \quad \mathtt{S(r_1\hat{}..\hat{}r_x\hat{}r_{x+1}\hat{}..\hat{}r_{m+1})} \qquad \text{Eq. D.8}$$

Therefore, since at least one of the antecedents of equations D.7 and D.8 is true $\mathtt{Q{\vee}R{\vee}S(r_1\hat{}..\hat{}r_{m+1})}$ holds.

*Subcase* $\neg R(r_1\hat{}..\hat{}r_x) \vee \neg S(r_{m+2-y}\hat{}..\hat{}r_{m+1})$ *holds*

Consider the set of traces given by

$$\Omega \quad = \quad \bigcap_{\mathtt{i=1}}^{\mathtt{m+1}} \mathtt{(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m+1})} \qquad \text{Eq. D.9}$$

$\mathtt{Q{\vee}R{\vee}S}$ holds over $\Omega$ because for all $\mathtt{1 \le i \le m+1}$ the trace $\mathtt{r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m+1}}$ belongs to the set *Traces*($\mathtt{F^m(STOP)}$), and $\mathtt{Q{\vee}R{\vee}S}$ holds over this. If $\mathtt{x}$ is the order of $\mathtt{(F, M)}$, then by definition $\mathtt{M(r_1\hat{}..\hat{}r_x)}$ holds. From the definition of $\mathtt{R}$

$$\forall\, \mathtt{t} \bullet \neg\mathtt{R(r_1\hat{}..\hat{}r_x)} \Leftrightarrow \neg\mathtt{R(r_1\hat{}..\hat{}r_x\hat{}t)} \qquad \text{Eq. D.10}$$

That is $\mathtt{R}$ will not hold for all those traces with $\mathtt{r_1\hat{}..\hat{}r_x}$ as a prefix. Additionally, the predicate $\mathtt{S}$ will not hold for all those traces with the trace $\mathtt{r_{m+2-y}\hat{}..\hat{}r_{m+1}}$ as a suffix. This is because if $\mathtt{y}$ is the order of $\mathtt{(F, N)}$ then by definition $\mathtt{N(r_{m+2-y}\hat{}..\hat{}r_{m+1})}$ holds and consequently

$$\forall\, \mathtt{s} \bullet \neg\mathtt{S(r_{m+2-y}\hat{}..\hat{}r_{m+1})} \Leftrightarrow \neg\mathtt{S(s\hat{}r_{m+2-y}\hat{}..\hat{}r_{m+1})} \qquad \text{Eq. D.11}$$

Because $\mathtt{Q{\vee}R{\vee}S}$ holds for all traces in $\Omega$ it can be deduced that $\mathtt{Q}$ must hold for all the traces in $\Omega$ which have the prefix $\mathtt{r_1\hat{}..\hat{}r_x}$ or the suffix $\mathtt{r_{m+2-y}\hat{}..\hat{}r_{m+1}}$. This can be expressed by

$$\bigwedge_{\mathtt{i=x+1}}^{\mathtt{m+1-y}} \mathtt{Q(r_1\hat{}..\hat{}r_{i-1}\hat{}r_{i+1}\hat{}..\hat{}r_{m+1})} \qquad \text{Eq. D.12}$$

The following substitutions are made

$$s_1 = r_1 \hat{}..\hat{} r_{m+1-n-y} \qquad \text{Eq. D.13}$$

$$s_2 = r_{m+2-n-y} \qquad \text{Eq. D.14}$$

$$..\ .$$

$$s_n = r_{m-y} \qquad \text{Eq. D.15}$$

$$s_{n+1} = r_{m+1-y} \hat{}..\hat{} r_{m+1} \qquad \text{Eq. D.16}$$

Using these substitutions equation D.12 becomes

$$\begin{aligned} &\bigwedge_{i=x+1}^{m-n-y+1} Q(r_1 \hat{}..\hat{} r_x \hat{}..\hat{} r_{i-1} \hat{} r_{i+1} \hat{}..\hat{} r_{m-n-y+2} \hat{} s_2 \hat{}..\hat{} s_{n+1}) \\ \wedge &\bigwedge_{i=2}^{n} Q(s_1 \hat{}..\hat{} s_{i-1} \hat{} s_{i+1} \hat{}..\hat{} s_{n+1}) \\ \wedge\ & Q(s_1 \hat{}..\hat{} s_n \hat{} r_{m+2-y} \hat{}..\hat{} r_{m+1}) \end{aligned} \qquad \text{Eq. D.17}$$

$$\Rightarrow \quad v_1 \hat{} s_2 \hat{}..\hat{} s_{n+1} \wedge \bigwedge_{i=2}^{n} Q(s_1 \hat{}..\hat{} s_{i-1} \hat{} s_{i+1} \hat{}..\hat{} s_{n+1}) \wedge s_2 \hat{}..\hat{} s_{n+1} \hat{} v_2 \qquad \text{Eq. D.18}$$

where $v_1$, $v_2$ are some traces. By definition $Q$ is such that

$$\forall u_j \neq \langle\,\rangle \bullet \bigwedge_{i=1}^{n+1} Q(u_1 \hat{}..\hat{} u_{i-1} \hat{} u_{i+1} \hat{}..\hat{} u_{n+1}) \Leftrightarrow Q(u_1 \hat{}..\hat{} u_{n+1}) \qquad \text{Eq. D.19}$$

Equation D.19 can be used to show that

$$\text{Eq. D.18} \quad \Rightarrow \quad Q(s_2 \hat{}..\hat{} s_{n+1}) \wedge \bigwedge_{i=2}^{n+1} Q(s_1 \hat{}..\hat{} s_{i-1} \hat{} s_{i+1} \hat{}..\hat{} s_{n+1}) \wedge Q(s_2 \hat{}..\hat{} s_{n+1}) \qquad \text{Eq. D.20}$$

$$\Rightarrow \quad \bigwedge_{i=1}^{n+1} Q(s_1 \hat{}..\hat{} s_{i-1} \hat{} s_{i+1} \hat{}..\hat{} s_{n+1}) \qquad \text{Eq. D.21}$$

Again equation D.19 shows that

Eq. D.21 $\Rightarrow$ $Q(s_1\hat{}..\hat{}s_{n+1})$ Eq. D.22

Therefore $Q \vee R \vee S$ holds for $r$

Thus if $r \in$ *Traces*($F^{m+1}(STOP)$) then $Q \vee R \vee S(r)$. The inductive step is

$$F^{m}(STOP) \textbf{ sat } Q \vee R \vee S \quad \Rightarrow \quad F^{m+1}(STOP) \textbf{ sat } Q \vee R \vee S \qquad \text{Eq. D.23}$$

BASE CASE

When $m = n+x+y$ the predicate $Q \vee R \vee S$ holds over the set $F^{n+x+y}(STOP)$ by testing.

Therefore for all finite integers $m$, $F^{m}(STOP)$ **sat** $Q \vee R \vee S$. As a consequence it is seen that $F^{n+x+y}(STOP)$ is an Ideal Test Set for the pair ($\mu P.F(P)$, $Q \vee R \vee S$). □