Specifying distributed applications: the limits of formality

Thesis

How to cite:

For guidance on citations see FAQs.

© 1989 Edward Joseph Fergus

https://creativecommons.org/licenses/by-nc-nd/4.0/

Version: Version of Record

Link(s) to article on publisher’s website:
http://dx.doi.org/doi:10.21954/ou.ro.0000fc46

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online’s data policy on reuse of materials please consult the policies page.
Specifying distributed applications: the limits of formality

by

Edward Joseph Fergus

Submitted for the Degree of Doctor of Philosophy

Author's number: M7021201
Date of submission: 30th October 1989
Date of award: 27th March 1990
Ms. A. Barrington
Higher Degrees Office
The Open University
PO Box 49
Milton Keynes
MK7 6AD

30/Apr/90

Dear Ms. Barrington

I have received your letter ref. M7021201 dated 24/Apr/90 informing me that the title page is missing from both copies of my recent thesis. I apologise for this oversight, and I would be very grateful if the library staff would insert the missing page.

Yours sincerely,

Edward FERGUS

Edward FERGUS
Specifying distributed applications: the limits of formality.

Abstract

This investigation considers the use of formal specification in achieving demonstrably correct programs in the application area of distributed communication protocols. A major consideration is the factors (both theoretical and practical) which limit the value of the formal approach.

A formal specification consists of abstract mathematical objects. It can be build up systematically by assembling several smaller specification components into an overall specification, and is amenable to rigorous analysis. Three approaches to formal specification are considered: state-space, predicate calculus, and process algebra. These approaches are represented by the finite-state, Z and CSP techniques respectively. These approaches are compared in their description of protocol properties and in their amenability to rigorous analysis.

The principal conclusions are: (1) the concept of state is fundamental to communication protocols, and the finite-state approach to protocol specification is as powerful as predicate calculus and process algebra; (2) a novel technique is developed to enumerate global-state trees economically; (3) modal logic is necessary in a Z specification which addresses questions of liveness and eventuality; (4) the semantic completeness of some CSP constructions is questionable; (5) the concept of fairness in finite descriptions of behaviour is given a rigorous definition; (6) a system specification should be operational in character; (7) a formal specification technique is essentially functional in the narrow mathematical sense; (8) non-functional requirements are invalid elements of a demonstrable specification.
HIGHER DEGREES OFFICE

LIBRARY AUTHORISATION FORM

STUDENT: FERGUS, EDWARD

DEGREE: Ph. D.

TITLE OF THESIS: SPECIFYING DISTRIBUTED APPLICATIONS: THE LIMITS OF FORMALITY

SERIAL NO: M702120

I confirm that I am willing that my thesis be made available to readers and maybe photocopied, subject to the discretion of the Librarian.

SIGNED: _________________________ DATE: 26/Sept/89
Declaration

This thesis is the original work of Edward FERGUS (B.Sc. Keele 1976, M.Sc. Manchester 1980). It is submitted in 1989 to the Open University in application for the degree of Ph.D.. No part of it has previously been submitted to this or to any other institution in application for any other degree.

Acknowledgements

I record my thanks to my research supervisors for their support and guidance during this study: Prof. D.C. Ince of the Dept. of Computing, Open University, and Prof. M.A. Hennell of the Dept. of Statistics and Computational Maths., Liverpool University. I thank Prof. Ince for early guidance in defining precisely the topic of investigation, for his later guidance in academic rigour and support during the investigation, and for his valuable editorial comments on the presentation. I thank Prof. Hennell for showing me the value of unorthodox questioning of established approaches, and for providing a sympathetic environment during the course of the investigation. Both have contributed significantly in their different ways to this thesis. I also recall my late father, who contributed more than he probably realised to some of my attitudes to academic study and other matters.
## Contents

1 - Introduction - a consideration of correct systems

1.1 - The need for correct systems
   1.1.1 - safety-critical and high-integrity systems
   1.1.2 - complementary functionality
   1.1.3 - social necessity

1.2 - Distributed systems

1.3 - Commitment protocols

1.4 - A consideration of specification
   1.4.1 - types of specification
   1.4.2 - using a specification for validation

1.5 - Formal specification, and the limits of formality

2 - Review of literature

2.1 - Abstraction in specification
   2.1.1 - the need for abstraction
   2.1.2 - a choice of terminology

2.2 - Formality in specifications
   2.2.1 - the elements of a formal specification
   2.2.2 - the value of formality
   2.2.3 - some limitations of formality
   2.2.4 - the credibility of rigorous reasoning

2.3 - Completeness and ambiguity
   2.3.1 - the formalists' view
   2.3.2 - the pragmatists' view
   2.3.3 - changing specifications and incompleteness
   2.3.4 - incompleteness and exception conditions

2.4 - Functional and non-functional requirements
   2.4.1 - functional requirements and environment models
   2.4.2 - functional specification of embedded systems
   2.4.3 - non-functional requirements
   2.4.4 - coupling of functional and non-functional requirements

2.5 - Classifications of specification techniques
   2.5.1 - a general taxonomy
   2.5.2 - a taxonomy of formal techniques
   2.5.3 - pertinent specification concepts
3 - Research objectives
3.1 - The application area: distributed applications
3.2 - Definition of terms: formal specification concepts
3.3 - Specific research objectives
  3.3.1 - the formal specification of required behaviour
  3.3.2 - purity of specification
  3.3.3 - the role of the environment
  3.3.4 - functional and non-functional behaviour
  3.3.5 - an excluded question: system realisation
3.4 - Specific research objectives
  3.4.1 - the formal specification of required behaviour
  3.4.2 - purity of specification
  3.4.3 - the role of the environment
  3.4.4 - functional and non-functional behaviour
  3.4.5 - an excluded question: system realisation

4 - Finite state analysis of protocols
4.1 - Definition of two communication protocols
  4.1.1 - master/slave protocol (2-phase commit)
  4.1.2 - a user/server protocol
4.2 - Indeterminacy in the finite-state specification
  4.2.1 - algebraic definition of 2-phase commit
  4.2.2 - three sources of protocol indeterminacy
4.3 - Removing protocol indeterminacy
  4.3.1 - removing indeterminacy - types 1 and 2
    4.3.1.1 - 2-phase commit with no type 1 or 2 indeterminacy
    4.3.1.2 - user/server with no type 1 or 2 indeterminacy
  4.3.2 - removing indeterminacy - type 3
4.4 - Global state analysis
  4.4.1 - enumerating global states
4.5 - Observations and conclusions
  4.5.1 - memory limitations
  4.5.2 - state explosions
  4.5.3 - correctness criteria are informal
  4.5.4 - protocol complexity is not obvious
    4.5.4.1 - unobservable states in Fig.4-3
    4.5.4.2 - complexity of events in Fig.4-4
4.6 - Conclusions on finite-state protocol specifications
5 - The Z specification of communications protocols

5.1 - Introduction to Z
5.1.1 - basic concepts of Z
5.1.2 - indeterminacy in Z
5.1.2.1 - protocol indeterminacy
5.1.2.2 - Z indeterminacy mechanisms
5.1.3 - appropriate levels of rigour and abstraction
5.1.4 - choice of notation
5.2 - The fundamental role of state in certain applications
5.3 - The 2-phase commit protocol in Z
5.3.1 - basic sets, variables, static invariant
5.3.2 - dynamics - substitutions as state changes
5.3.3 - dynamics - removing decision ambiguities
5.3.4 - dynamics - the 2-phase commit master and slave
5.4 - Z facilities for stating protocol properties
5.4.1 - the enumerative statement of properties
5.4.1.1 - separating statement from proof - temporal logic
5.4.1.2 - a brief summary of temporal logic
5.4.1.3 - relation of modal logic to protocol specification
5.4.1.4 - a general procedure
5.4.1.5 - correctness of the 2-phase commit protocol
5.4.1.6 - liveness of the 2-phase commit protocol
5.4.1.7 - other properties of the 2-phase commit protocol
5.4.2 - Z and some fundamental finite-state problems
5.4.2.1 - Z specification of 1-master-many-slaves
5.4.2.2 - correctness and liveness
5.5 - A clean example: the user/server protocol in Z
5.5.1 - basic sets, variables, static invariant
5.5.2 - dynamics - operation of the user/server protocol
5.5.3 - correctness of the user/server protocol
5.5.4 - liveness of the user/server protocol
5.6 - Conclusions on the use of Z
6 - The CSP specification of communications protocols 81
6.1 - Introduction to CSP 81
6.1.1 - basic concepts of CSP 82
6.1.2 - choice and determinacy 84
6.1.3 - process traces 86
6.1.4 - interaction and deadlock of processes 87
6.1.4.1 - mutual recursion 87
6.1.4.2 - parallel composition 87
6.1.4.3 - message passing 89
6.1.4.4 - sequential composition 90
6.1.5 - the operating environment of a process 91
6.1.6 - a formal definition of determinacy 91
6.2 - The 2-phase commit protocol in CSP 92
6.2.1 - the master operating environment 94
6.2.2 - some other approached to defining ME 96
6.2.2.1 - parallel composition 96
6.2.2.2 - interleaving 97
6.2.2.3 - selective failure 97
6.2.2.4 - angelic nondeterminism 97
6.2.3 - the remaining 2-phase commit processes 100
6.2.3.1 - the master process M 100
6.2.3.2 - the slave process S 101
6.2.3.3 - the slave environment SE 101
6.3 - Stating protocol properties 102
6.3.1 - legal interaction, liveness, deadlock 102
6.3.2 - temporal properties 103
6.3.2.1 - liveness, fairness 114
6.3.2.2 - perverse choice 106
6.4 - The user/server protocol in CSP 106
6.4.1 - the server environment SVE 107
6.4.2 - the server SV 107
6.4.3 - the user environment UE 107
6.4.4 - the user U 108
6.5 - Conclusions on the use of CSP 108
6.5.1 - expressive power 109
6.5.2 - modular development of a specification 109
6.5.3 - temporal properties 109
6.5.4 - abstraction and semantics 110
7 - Summary and conclusions

7.1 - Objectives and limitations of the study
7.2 - Achievements of the study
7.3 - The fundamental role of "state" in certain applications
7.4 - A comparison of finite-state, Z and CSP
7.4.1 - finite-state indeterminacy, bandwidth-limited state trees
7.4.2 - indeterminacy, state, and modal logic in Z
7.4.2.1 - Z indeterminacy
7.4.2.2 - states in Z
7.4.2.3 - modal logic
7.4.3 - CSP expressiveness, analysis, protocol properties
7.4.3.1 - liveness, fairness and perverse choice
7.5 - The need for operational specifications
7.5.1 - impossible specifications
7.5.2 - implicit specifications
7.5.3 - operational specifications: avoiding impossible behaviour
7.5.3.1 - impossible finite-state behaviour
7.5.3.2 - impossible CSP behaviour
7.5.3.3 - impossible Z behaviour
7.5.4 - implementing implicit behaviour
7.6 - Modelling the environment: functional behaviour
7.7 - Some observations on formal specification techniques
7.7.1 - the value of formality
7.7.2 - instantaneous events, time resolution
7.7.3 - non-functional behaviour

8 - Further research

8.1 - Non-functional behaviour
8.2 - Stating and analysing criticality levels
8.3 - Probabilistic behaviour
8.4 - Graph-theoretic analysis
8.5 - Development of automatic tools

References

Appendix A: computability, languages and automata
Appendix B: validity of transformation on finite-state automata
Appendix C: global state analysis of a 2-partner protocol
Appendix D: global state analysis of the 2-phase commit protocol
Appendix E: global state analysis of the user/server protocol
Chapter 1 - Introduction: a consideration of correct systems

1.1 - The need for correct systems

The correct operation of a computer system is of great - sometimes crucial - importance in many applications. One can distinguish the following broad application areas:

1.1.1 - Safety-critical and high-integrity systems

An increasing number of industrial processes which are safety-critical or reliability-critical are designed for computer control.

A large chemical plant or a nuclear reactor has too many components which interact in too many ways and at too great a speed for human monitoring and control to be a realistic prospect. In such systems, the human operators can only determine the operating set-points, while a computer automatically monitors the plant and provides the low-level direct control needed to maintain those demanded set-points. The importance of the control system’s correct operation is in proportion to the disastrous effects of a plant malfunction. Some types of plant (e.g. a nuclear reactor, or a weapons control system) are simply not feasible without a control system which is either demonstrably correct or which at least inspires a very high degree of confidence.

1.1.2 - Complementary functionality

A computer control system sometimes provides functionality which complements that of an existing system and thus increases the overall functionality.

For example, the most recent fly-by-wire civil aircraft and many military aircraft are open-loop unstable in flight. This makes them very maneuverable and responsive, but their flying surfaces must be continually adjusted. The aircraft’s reaction to aerodynamic forces is
much too fast and varied to allow the human pilot to react to events. Instead, a computer is in direct control of the plane’s flying surfaces. If a correct computer monitoring and control system cannot be constructed to complement the basic aerodynamic and mechanical engineering of the aircraft, then the plane simply cannot be flown.

1.1.3 - Social necessity

Much human activity and social organisation relies on the methodical organisation of economic resources and the provision of financial services (in the broadest sense of the term).

Industrial societies increasingly depend on computers to provide these services. The use of computers both provides more effective resource management than the traditional clerical methods, and facilitates valuable services which were not previously feasible e.g. large scale economic modelling.

In such applications, the manifestation of failure may be an undramatic degradation over time of overall system performance. In other cases the failure may be very dramatic. For example, the international stock market crisis in October 1987 was widely blamed (in part) on the use of automatic share-dealing programs whose operation was determined by some simplistic market conditions (e.g. "sell everything when the SE100 index drops to X").

Unpredictable behaviour may arise from (1) faulty hardware, or (2) from a misunderstanding of the tasks which the system must accomplish, or (3) from an incorrectly written program. The production of trustworthy hardware and the techniques of systems analysis are well-developed topics which will not be considered further here. This study is concerned with the correct specification of requirements, which is a necessary precondition for the production of correct programs.
1.2 - Distributed systems

An area where correct computer operation is of great interest is that of distributed systems. The essential feature of a distributed system is that the distinct components (at least two) must communicate with each other to achieve the desired result. The extent of the communication depends heavily on the particular distributed application: components might have to co-operate closely and intricately (therefore needing much communication), or merely enough to allocate responsibility for a task (therefore needing minimum communication).

At either of these two extremes it is possible for the communication to go wrong. In a physically distributed system a message between two components A and B might get lost or become corrupted. Whether physically distributed or not, A cannot be certain that B will send the appropriate message in all circumstances.

Consider a transaction which consists of a sequence of stages. Each stage is *atomic* in the sense that once its execution has begun, it will not be interrupted and the stage will always complete. Thus, the progress of the transaction is always well defined. It is well understood how to apply this approach within a single component [BES81] but it is much more difficult in a distributed environment where N distinct co-operating components are concurrently handling N transactions. Clearly, the N components must be able to determine when they are at mutually compatible stages of the distributed transaction. The common solution is to base the communication on a *commitment protocol*.

1.3 - Commitment protocols

A commitment protocol synchronises the operations of N distinct communicating components. The N-component protocol studied in this thesis is *asymmetric*: one of the N components acts as a co-ordinating master while the other N-1 components act as slaves. The aim is to ensure that all N components arrive at a consistent result: either (1) all succeed in completely executing the transaction, or (2) all fail to execute it and all discard any partial results.
When each component receives a transaction to process, it uses its local data to decide whether to accept or reject the transaction. If it decides to reject, it must abort the transaction and inform all other components. To abort a transaction means to discard all partial results: the effect is as if the transaction request had never been received.

If the component decides to accept the transaction, it will process the transaction and arrive at a commitment point beyond which the processing is irreversible. It then informs all the other components that it is willing to commit the transaction, but it cannot do so until all other components are also willing to commit. The offer to commit cannot be revoked, and the offering component can abort the transaction only if at least one other component rejects the transaction. Such a rejection is broadcast to all the offering components, and it releases them from their offer. When all N components have offered to commit, then all N must commit the transaction.

The essence of the protocol is that the commitment point synchronises the progress of all the components. The offer to commit and the refusal to commit are both irrevocable. The result is either that all N complete the transaction, or that all N abandon the transaction and return to their situation immediately before the transaction request arrived. The protocol therefore preserves the consistency of the transaction in the distributed environment.

1.4 - A consideration of specification

Much of the effort in developing a computer program is spent in validating the program. In the most general sense, a validation is a measurement of the program’s quality in terms of some desired characteristics and behaviour. A prerequisite for such a measurement is a statement of the required characteristics and behaviour of the program - the specification. It is axiomatic that a program can be validated only with respect to a given specification.
Definition

A validated (or correct) program is one which has been demonstrated to possess the desired characteristics and behaviour set out in the [system] specification.

1.4.1 - Types of specification

It is useful to distinguish between the requirements specification, the system specification, and the design specification.

The starting point of system development is an informal statement of required behaviour and other characteristics (e.g. observable behaviour, resource limits, etc.) - the requirements specification. The system specification is a precise and abstract account of the user's perception of his needs in terms of the externally visible characteristics and behaviour of a solution. This statement is independent of any particular solution. Finally, the design specification describes the logical and physical structure of a particular solution which satisfies those requirements (e.g. construction techniques, use of particular resources, etc.).

In the remainder of this thesis, the term specification should be taken to mean system specification unless otherwise stated.

A design specification cannot reasonably be taken as an implicit system specification. Some aspects of function and behaviour are relatively easy to state but are very difficult to implement: they often give rise to complex designs which then obscure the intention. To see this, one need only consider the design implications of a requirement for high performance or reliability, even when these are stated in a quantified way.
1.4.2 - Using a specification for validation

When a detailed description (such as a design or an implemented program) is used as an implicit requirements specification, it is usually very difficult to distinguish between (1) the details which result from requirements and (2) the details which result from design or implementation constraints. This distinction is clearly important when validating a program, when one must ensure that every requirement is adequately addressed. Further, when a program is maintained, it is usually the requirements which have changed over time. To successfully maintain a program one must trace the current details to the current requirements.

It is debatable whether the effort of validation is better served by considering primarily the system specification or the design specification. Much depends on the development model (or life cycle) by which one describes the whole development process.

In all life cycles there is a progression from informal requirements through design to executable source program. The traditional life cycle emphasises the later stages and carries out almost all validation, optimisation and maintenance on the executable program. Here the requirements specification is largely ignored after the initial development stages, and the system specification is more important. Validation is largely a matter of demonstrating the mutual consistency of the system specification and the executable program.

A radically different approach focuses attention on a formal system specification and derives all subsequent stages (design, executable code, etc.) by automatic transformation. All validation and maintenance is carried out on the system specification, while the design specification is merely another intermediate stage. Here, validation is concerned with the self-consistency of the system specification and the fidelity of the specification’s account of the user’s needs.
Developments of recent years have led to an interest in formal specifications i.e. based to some significant degree on rigorous mathematical concepts. There is considerable speculation that a formal specification can establish program quality in a demonstrable fashion. A major consequence has been the drawing up of the quality-assurance guidelines "Defence Standard 0055", which demands the use of formal methods in the procurement of safety critical military systems.

A formal specification is constructed using notational devices (e.g. numbers, sets, predicates, etc.) which are amenable to systematic and rigorous analysis. That is, a formal specification is mathematical in nature. It is the possibility of rigorous analysis that makes a formal specification useful, rather than any appeal to classically elegant mathematical notations.

But this is not to say that all interesting analyses can be carried out, even in principle:

1. there are well known fundamental limitations on the analysis of formal systems (e.g. Goedel's and Church's arguments on the absence of decision procedures for first-order logical systems)
2. there are practical limitations arising from the amount of data that such analyses can be allowed to generate while still remaining useful. For example, one can rigorously describe the procedures of applications such as artificial intelligence and theorem proving, but these procedures result in very large solution spaces which can be searched effectively only by adopting branch-and-bound heuristics (e.g. the alpha-beta search algorithm).

A formal description is simply an algebraic form: there can be no rigorous demonstration that the form accurately represents any particular informal concept. The consequence is that a specification which is formally correct might be semantically incomplete: a program may be formally correct (and proven so) and still fail in practical use. For example, an on-line ticket reservation system appears correct after formal analysis but fails in practice because it does not permit
reservations to be cancelled. Cancellation is forbidden simply because the requirements author did not understand the application well enough.

In practice, informal specification is the rule and precise specification is the exception in expressing non-functional requirements such as robustness and user-friendliness. Here the requirement is really an amalgam of several more primitive requirements, or is subjective to some extent (and note carefully that these two circumstances are quite distinct). Such requirements are usually specified operationally. For example, (1) a compiler is usually considered to be correct if it can successfully parse a set of paradigm programs within acceptable limits of time, memory, etc., or (2) an interactive program is considered user-friendly if it generates a particular set of screen displays.
Chapter 2 - Review of literature

This chapter supplies background information to put into context the principal issues of specification addressed by this investigation.

1. the accepted interpretations of the term *specification* are explored and classified. The aim is to justify the use of abstraction in a specification, and to establish concepts and clarify terminology.

2. the role of *formality* in specification is considered.

3. some supporters of formal specification argue that a specification should be complete and unambiguous: this argument is considered.

4. a practical specification technique should address both functional and non-functional aspects of behaviour. In any case, there is some evidence that even the strictly functional aspects are influenced by non-functional requirements. This coupling of interests is examined.

5. finally, some conventional classifications of specification techniques are examined.

2.1 - Abstraction in specification

The basic notion of a *specification* is that it states the intended behaviour and characteristics of a system or program [CHA78]. It is axiomatic that a program can be validated only with respect to a given specification. This description of behaviour and characteristics consists of two parts [LIS80]: the tasks which must be performed, and the constraints which must be observed when performing those tasks.

2.1.1 - The need for abstraction

It is argued [GRE84] [DEN84] [HEI84] that the required behaviour of some computation should contain only those details which are necessarily present in every model of the computation. (For example, [PEP84] specifies the correctness of the communications within a distributed database by requiring the distributed system to be equivalent to a non-distributed one.) [BAL79] states the principle that a specification of behaviour should separate *function* from *implementation*: it should describe what is desired, rather than how that
requirement is to be achieved. [PAR77] also uses specification in this abstract sense but emphasises that abstract must not become a euphemism for vague.

The distinction between the abstract specification of requirements and those specifications which relate to the later design and implementation stages, and the insistence on abstraction in the system specification, are justified in very similar terms in [BAL83] and [ROM85].

"Requirements specifications facilitate understanding of the problem, while design requirements faithfully render physical and logical structures that implement the requirements" [ROM85]

A program's design or implementation cannot reasonably be taken as an implicit specification of behaviour. Even a well understood functional requirement (see below for a discussion of functional and non-functional) such as sorting can give rise to complex and non-obvious designs, while it is usually impossible to deduce the existence of a non-functional requirement such as high reliability from a design alone. Several AI workers [MAN75] [ULR77] [DER81] have attempted to make explicit the information concerning intent which is implicit in the program details. For example, one would observe the analogies between the recursive and iterative versions of several applications and abstract a more general method of converting recursive to iterative. This new method would then be applied to other recursive programs. However, these attempts have been very limited in scope (thus, [DER81] confines his attention to the functional input/output behaviour) and have achieved only modest success.

An attempt to gain an overall understanding of the system (e.g. to validate or maintain the implementation) requires a statement of requirements which is both explicit and distinct from the design. This attempt is more likely to succeed if the requirements statement contains only those details which are necessarily present in every design or implementation. That is, the requirements statement is abstract.
[BAL83] argues that the inclusion of non-abstract details necessarily introduces constraints and decisions which arise from design or implementation considerations (e.g. a particular queue discipline, or a particular data representation). These constraints and decisions then become indistinguishable from the requirements.

The value of a good abstraction has been examined in depth [GER75a] [DUN79] has been demonstrated in theorem-proving applications [PLA80].

2.1.2 - A choice of terminology

The emphasis on abstraction in a specification provides the key to understanding how this relates to several other types of specification. Thus [GUT81] defines both local and system specification (in principle the same, but in practice the latter is combinatorially more complex than the former) as the observable abstract behaviour of a program or system, while the structural specification is a program description which is high-level but not abstract (i.e. a design specification). Similarly [WIN79] uses result specification to mean a process-independent (i.e. abstract) relationship between the input and output of a computation, and program specification to mean the formal structures which can be interpreted in terms of the instructions of a particular machine.

To clarify the concepts and terminology, we distinguish between the requirements specification, the system specification, and the design specification as follows:

1. the starting point of system development is an informal statement of required behaviour and other characteristics (e.g. observable behaviour, resource limits, etc.) - the requirements specification.

2. the system specification is a precise and abstract account of the user’s perception of his needs in terms of the externally visible characteristics and behaviour of a solution. This statement is independent of any particular solution.

3. Finally, the design specification describes the logical and physical structure of a particular solution which satisfies those
requirements (e.g. construction techniques, use of particular resources, etc.).

2.2 - Formality in specifications

*Formal specification* is generally taken [MEL86] to mean that approach which maps the specification concepts of interest into mathematical concepts or objects whose semantics are defined and upon which rigorous mathematical operations can be performed.

Note that one prominent reviewer [LIS80] restricts the use of *mathematical objects* to a narrower class of specifications known as *abstract models*. However it seems advantageous to use the concept much more widely and define *formal* to mean *based on any well-defined mathematical concept*.

2.2.1 - The elements of a formal specification

The mathematical operations on formal specifications constitute a series of transformations [GREN84] [JAC84] which convert real-world requirements into an executable implementation. Thus, one school of thought emphasises algebraic and similar specification [GOG79a] [GOG79b] based on sets [ABR87], sequences [HOA85], lattices [SCO76], functions and relations [HEN83]. Another school emphasises logic and bases specification on predicate calculus [HOA69] [MAN78], using modal logic [OWI76] to handle such non-classical features as explicit time and concurrency. A third school adopts state-space methods [GER80] [CHO78] [ROB77], while a fourth makes use of formal language theory [SCH84] [REV85] and automata [PET77] [ZUB80].

2.2.2 - The value of formality

The use of formal notations for the specification of program behaviour inspires strong opinions which range from avid support [MEY85] [LIS80] to deep scepticism [DEM79] and open hostility [CUL83].
Indeed, the debate is sometimes conducted in anecdotal and somewhat personal terms [CUL83] [THO83].

In support, [MEY85] argues that an informal specification contains too many ambiguities, contradictions and omissions to be useful. He cites the unsuccessful attempts by [G0077] to correct an informal specification of the well known telegram problem [NAU69] and describes several types of specification fault likely to result from the informal approach: \textit{noise, silence, overspecification, contradiction, ambiguity, forward reference, wishful thinking}. [MEL86] argues more moderately that informal notations are unsatisfactory because their interpretation requires a great deal of accompanying context information, whereas formal notations are much more self-contained although very difficult to instill with semantics. (In this connection, [CHAD85] argues that there is a trade-off between power of expression and amenability to rigorous analysis). [THO83] points out that the well-known proponents of formality all advise that it should be used appropriately rather than indiscriminately.

Among the avid supporters of formality are those who advocate the use of \textit{wide spectrum} languages [POT84] [BAU82] which they claim to be appropriate for all stages from abstract specification to detailed implementation.

But in general the formal approach is well exemplified by [LON80] who argues that programs should be constructed using special specification and design languages (e.g. Clu [LIS77], Alphard [WUL76]) and axiomatically defined programming languages (e.g. Pascal [HOA73], Euclid [LAM77]) which can be treated rigorously.

2.2.3 - Some limitations of formality

Some limitations of formality are recognised. In the context of formal languages [REV85] observes that the formal syntactic structures must be interpreted ultimately and that all formal manipulations must therefore preserve the semantics. The meaning of a formal construct may be derived either by using grammatical categories to encode context
information (top down), or by synthesising the meaning of grammatical categories from the meaning of their components (bottom up).

But in all cases the development of theories of formal semantics [JON80] significantly lags behind the development of formal syntax. That is, current knowledge can handle form much better than content. Supporters of the formal approach usually sidestep the issue of semantics by concentrating explicitly on form. Thus [LON80] states that program verification aims to demonstrate the consistency of a program and a specification but says nothing about the adequacy of that specification. Others [GOR79] [ROS84] use denotational semantics in an attempt to capture as much context information as possible in a format which can be formally manipulated.

Several workers argue that while formality may bring rigour, this is beneficial only if it leads to an better understanding of the problem domain and the program construction and operation. Thus [WO084] states that formality is not justified by mathematical elegance alone, but by the extent to which it increases confidence in the quality of the produce and reduces life-cycle costs overall. [MAR84] [GYR84] [BER84] argue that the complete set of non-trivial requirements are drawn up by a diverse group of people, only a few of whom may be expected to be familiar with formal methods, and that the formal approach is largely inaccessible at present, although the situation should be improved by the introduction of automated specification tools [DAV80] [GER80] [FIC84]. Such tools would make the formality more manageable by reducing the tedium of detailed work. They would automatically include minor conditions and lemmas which "humans understand to be present" [FIC84] without explicit statement, and would facilitate changes (e.g. refinement, elaboration, generalisation) while automatically checking that certain desirable properties (e.g. self consistency) are preserved.

2.2.4 - The credibility of rigorous reasoning

A deeper issue of product quality concerns the extent to which one is convinced by a formal and allegedly correct proof. Thus [LAM79]
constructs a formal specification whose "correctness is manifest to humans". Similarly [JONES80] points out that his formal technique may be used with a greater or lesser degree of formality, depending on how easily the user of the specification can be convinced of its correctness.

The point at issue - that a proof does not exist in its own right but must always be interpreted and accepted by humans - is discussed at length in [DEM79]. The essence of the argument is that a mathematical proof is a largely informal plausibility argument which rarely depends on rigorous appeal to first principles. A proof becomes accepted when many people succeed in relating the new result to an existing body of knowledge. In this sense, a mathematical proof does not exist in isolation from all other proofs. As a consequence, a single proof (no matter how rigorous) does not produce a general increase in the confidence of correctness and product quality. [GER75b] produces evidence that formal proofs of programs may contain serious errors which remain undetected for a significant time.

On a more encouraging note, [GUT81] describes the process by which an experienced formal specifier develops a repertoire of generalised scenarios which he then calls upon when presented with a new problem. This appears to be one mechanism by which the new formal proof can be related to an existing body of knowledge concerning a particular problem domain. It is also conceivable that the existing expertise can yield powerful insights which can then be used as informal but convincing lemmas in new proofs [PLA80].

2.3 - Completeness and ambiguity

The completeness of a specification is the degree to which it includes all conceivably relevant details and excludes all possible ambiguity. There are two extreme viewpoints on the question of how complete a specification can or should be: the formalists' view, and the pragmatists' view.
2.3.1 - The formalists' view

The first view - which is common among supporters of strictly formal methods - is that a specification is inadequate if it is incomplete in any detail.

Thus, [BAU82] proposes a wide-spectrum language (i.e. abstract data types, predicate calculus, etc.) to state requirements which are clear, complete, self-consistent and which can be transformed into an executable program with all of these properties being demonstrably preserved at each stage. [AND81] demands that his pre- and post-condition approach to specifying concurrent behaviour be capable of constructing invariant assertions which are "strong enough ... to allow whatever property is of interest to be proved".

One supporter of this viewpoint is [GYR84], who is unusual in expecting a largely informal notation to be capable of establishing that all redundancy and ambiguity has been removed, and that all implied requirements have been addressed.

2.3.2 - The pragmatists' view

The opposing viewpoint [ROM85] is that a specification is necessarily incomplete when design and implementation begin, and becomes more complete and clearer as these development stages proceed and even as the completed system is in use.

Workers from an AI background [GRE84] [BOR85] argue that the interpretation of requirements (even when formally stated) requires so much context information that one must be prepared to iterate judgements and revise earlier conclusions. [JACOB83] produces evidence that, in the area of user interfaces, the consequences of a precise specification are not always intuitively clear.
The pragmatists' viewpoint is stated more generally by [SWA82] who argues that a specification changes throughout the lifetime of a system (or equivalently, the interpretation of the specification changes) for two sorts of reasons: physical imperfections, and imperfect foresight.

Regarding physical imperfection, the designer or implementer is forced into certain interpretations of the specification by real-world constraints (e.g. economics, technology, etc.). This view is supported by the argument [PRE87] that functional and non-functional requirements may be intimately linked, and by the empirical evidence of [WAL80] who writes a second specification to take advantage of the improved insight gained from the attempt to implement the first specification.

Regarding imperfect foresight, the designer and implementer may subtly alter the interpretation of a specification as its less obvious consequences become apparent [WAL80]. These alterations may continue throughout the actual use of the system as users "learn to read" the specification (or the documentation which is derived from it), and also as users' behaviour changes in response to the new facilities offered by the new system. (This adaptive behaviour also influences the perceived reliability of the final system. See [VEE87] for a discussion of adaptive reliability.)

[WHI85] classifies the causes of changing specifications:

1 better understanding of the processes controlled by the system (i.e. improved insight [SWA82] [WAL80])
2 adjustments to non-functional requirements e.g. improved performance (i.e. "functional and non-functional are linked" [PRE87])
3 changes in the external environment. Such changes may be independent of the new system, or may have been induced by it.
4 technology changes, resulting in changes to the design and implementation constraints.
A further source of uncertainty in requirements concerns the handling of error conditions. From a specification viewpoint, the problem is that error behaviour is usually to do with unforeseen events: if an event can be foreseen, then clearly one can state in advance what action should be taken whenever that event occurs. But what provision should be made (or indeed, can be made) for unforeseen events if one is attempting to produce a complete specification?

First, distinguish between an error and an unexpected event. An error is the result of an incorrect (i.e. not in agreement with the intention stated in the specification) program operation. An unexpected event is one which has not been anticipated in any way. In a numerical application, the former would be exemplified by a roundoff error, and the latter by a out-of-range argument. Exception handling is concerned with foreseeable errors and not with unforeseen events. [BID85] defines operations on abstract data types such that the set of all acceptable results is augmented by the set of all (foreseen!) unacceptable results. Thus, an erroneous result is explicitly specified and the corresponding recovery action is prescribed.

In the context of demonstrably safe systems, a similar approach is taken by [ANDER78]. The original specification S is replaced by a weaker specification W which is "S or error". W is such that

1. any program which conforms to S also conforms to W, and
2. W describes the acceptable behaviour of the program.

A program is then developed to conform to S but is validated with respect to W.

But in all cases note that the erroneous condition must condition must be foreseen in the specification in order to be analysable. There is no way to make provision (formal or otherwise) for the unforeseen, apart from the trivial solution of a catch-all result: "if all else fails, do this ...".
2.4 - Functional and non-functional requirements

It is accepted practice to distinguish between functional and non-functional requirements [YEH82].

A functional requirement states some necessary interaction between the program and the environment in which it operates, while a non-functional requirement take the form of a constraint on program behaviour [ROM85].

Note that the term functional is often used in a very loose sense to mean something that the system does. For example, one talks of the system's functionality, meaning the complete collection of system behaviour and resource usage. A more precise definition is needed for the purposes of this investigation.

2.4.1 - Functional requirements and environment models

The statement of a functional requirement [ROM85] needs an abstract model to describe the behaviour of both the program and its environment. The power of this generalised functional requirement is limited by the power of the process/environment models. The full generality of environmental interaction is usually unmanageable, even for fairly simple applications, and it is common to adopt a simple model which leads to a narrower sort of functional requirement. This restricted functional approach abstractly models a program as a mathematical function and the operating environment as an unchanging input and output data domain.

Thus, the functional approach (in the common sense of the term) considers a program to be a black box [PRE87] whose internal detail is of no interest and whose observable behaviour is completely described by a mathematical mapping which relates input value to output value.

An example of this approach is [HOA72] where the specification of data operations takes the form of input/output assertions. [BAL79]
generalises this slightly by permitting the input/output relationship to be
an arbitrary predicate, but the essential restriction remains. [HOW87] gives a comprehensive account of (input/output) functional specifications.

2.4.2 - Functional specifications of embedded systems

A major limitation of the restricted functional approach is the assumption that the operating environment is unchanging. But it has been observed that some applications are apt to cause changes in their environment. Thus, [BAL79] states that a specification should be *process oriented* (i.e. should take account of a dynamic environment), while [WHI85] [FER88] describe the same problem in terms of the environmental feedback produced by an embedded application.

2.4.3 - Non-functional requirements

*Non-functional* requirements take the form of constraints on program behaviour [ROM85].

Where the constraints on functional behaviour can be stated in functional terms (e.g. range limits of input values, etc.), these constraints can be included in the functional specification. (But note that [MAJ77] points out some inherent limitations of a class of functional techniques and demonstrates the need to introduce auxiliary or *hidden* operations.)

The remaining non-functional constraints tend to be a catch-all for all behaviour which cannot be easily stated functionally. This includes:

1. failure response, which demands a significant amount of detailed design information not available during specification
2. human factors, which require much experiment
3. contentious matters such as reliability and maintainability, on whose meaning there is little general agreement

[WHI85] lists the following important non-functional requirements: performance, fault detection and recovery, safety, security, availability, reliability, ease of change.
[ROM85] surveys the diversity of non-functional requirements.

1. interface constraints require the program to accommodate particular classes of users (e.g. interactive, operating system, hardware)

2. performance constraints deal with issues of space/time bounds, reliability, security

3. operating constraints impose limits on how the program will be used (e.g. necessary skill levels, procedures for maintenance and upgrade)

4. life-cycle constraints address the design and production phases (e.g. portability, flexibility, re-use of components) and impose limits on resource usage, development methods, quality assurance methods, etc.

5. economic constraints relate to all costs during the entire life cycle: development, maintenance, production, marketing, etc.

6. political and legal constraints deal with policy and legal issues

2.4.4 - Coupling of functional and non-functional requirements

It is easy to see an overlap between the various non-functional requirements. Less obvious is the argument [PRE87] that functional and non-functional requirements may be intimately related. Thus, a particular performance goal (non-functional) might be impossible to achieve if a particular (input/output) functional specification is adopted.

Several attempts have been made to apply AI techniques to non-functional specification. [BOR85] [GREN84] point out that a considerable amount of context information is needed to interpret even a functional specification, and propose an object-oriented [REN82] approach using a knowledge database [STU84] and semantic network [BRA80] to describe the problem domain. A less ambitious approach [SCH84] makes use of a collection of promising specification and design methods (algebraic specification, pre/post conditions, abstract data types, explicit interfaces, concurrency primitives, etc.) and applies a Prolog verifier to check that the specification observes a particular set of guidelines.
Several workers have applied AI techniques to the inverse problem: how to infer the intended effect of a body of executable code. [FUK85] automatically annotates a executable program, while [JOH84] constructs an expert system to debug a Pascal program. However, some workers [SIE85] bluntly reject the usefulness of dealing with executable code and insist that an abstract specification is essential. This conclusion is supported by the difficulty [HOW87] of extracting even fairly straightforward input/output functional information from executable code.

2.5 - Classifications of specification techniques

Informative taxonomies of formal specification techniques are provided by [LIS80] and [ROM85].

2.5.1 - A general taxonomy

[LIS80] distinguishes between those techniques suitable for sequential programs and those for parallel programs. A sequential program has a single site of execution activity (a process), while a parallel one is executed by two or more co-operating processes. Specifications for parallel programs are mainly concerned with managing the co-operation of the co-operating processes of the computation: synchronisation and data sharing. The trivial case of processes which do not co-operate can be handled by sequential techniques.

Sequential specifications are either procedural abstractions or data abstractions.

Procedural abstractions are concerned with two aspects of computation: the observable interface, and the observable behaviour. Procedural abstractions subdivide into input/output specifications [NAU66] [HOA69] and operational specifications [ZAV82]. An input/output specification maps the input data domain to the output data domain but omits all details of internal program structure. An operational specification defines the required computation in terms of some idealised
computation whose level of detail stops well short of a fully executable program.

Data abstractions are concerned with abstract operations on data types [LIS74], and can be subdivided into special-purpose languages [WUL76] [AMB77], axiomatic operations [LIS75], and abstract models [WUL76] [HOA72]. The axiomatic operation approach treats each data item (i.e. an instance of a particular data type) as a well-defined object on which well-defined operations (and no other operations) can be performed. An abstract model specification conveys semantics by insisting on the use of widely accepted mathematical concepts (e.g. sets, sequences, etc.) whose semantics are widely known. A special purpose language enforces certain restrictions on data structure and usage, with the aim of ensuring that all specifications or programs thus produced are amenable to rigorous analysis.

Parallel specifications are concerned either with process communication mechanisms (e.g. message buffers [AMB77]), or with data sharing mechanisms (e.g. critical sections and monitors [HOWA76], or global state variables [BRAND83]), or with event sequences (e.g. Petri nets [PET77], path expressions [CAM74].

2.5.2 - A taxonomy of formal techniques

All of the techniques in the above general classification are formal (i.e. open to rigorous analysis) to some degree. However [ROM85] is particularly concerned to identify those techniques which are formal to a very high degree. He offers the following classification:

1. finite state [DAV79],
2. dataflow [TEI77],
3. stimulus/response [ALF79],
4. concurrent processes [HOA85] [ZAV81],
5. functional decomposition [HAM76],
6. data oriented semantics [PAG81] (which in turn subdivide into
   6.1 denotational [GOR79],
   6.2 axiomatic [HOA72] and operational [ZAV82]),
   6.3 predicate logic [CLO81] [GOOD83].
2.5.3 - Pertinent specification concepts

To some extent, every scheme to classify specification techniques is approximate, since most practical techniques exhibit features of more than one class. However, the above classifications highlight the specification concepts which are of particular interest in this thesis:

1. A distributed system has several components operating concurrently. Therefore, the specification should consider a process-based approach to take account of concurrent activity.

2. The interaction between a distributed system and its environment is likely to be of greater significance than for a centralised system. The specification should therefore address the problems of environment modeling.

3. The need to reason about the consequences of a specification demand the use of logic (in the most general sense of the term). It seems likely that the axiomatic approach (either to the specification of operations on data, or to the definition of data semantics, or both) will play a significant role.

4. A communication protocol is very conveniently modeled using a stimulus/response notation.

The representative specification techniques selected for detailed assessment in this thesis (see below) can be expected to display the all above characteristics to some degree.
3.1 - The application area: distributed applications

The investigation considers the specification of the major application area known as distributed applications. These display the following characteristics, which have traditionally proved very difficult to handle systematically:

1. the co-operation of two or more distinct components is needed to achieve the overall function of the application.
2. correct local operation of each component is needed to maintain detailed self-consistency of the overall application.
3. reliable communication is needed: correct co-operation is in general impossible if data transmission fails.

Communication is therefore a fundamental aspect of distributed systems. We consider communication behaviour in an abstract sense by examining the difficulties of specifying a particular class of distributed system - a communications protocol. Two protocols are considered in detail:

1. the 2-phase commitment protocol
2. a user/server protocol.

3.2 - Definition of terms: formal specification concepts

Our notions of formal specification are expressed by Jones’ description [JON88] of mathematical methods of development:

"typically, they use a mathematically based notation in which to record the specifications and, based on these, they have a notion of an implementation satisfying a specification which can be susceptible to proof."
For the purposes of this investigation, the defining characteristics of a formal specification are the following:

1. it consists of abstract mathematical objects (numbers, sets, predicates, etc.) which are precisely defined and whose interaction with all other objects in the specification is precisely known.

2. it can be built up systematically by assembling several smaller (i.e. less complex, or which describe subsystems of more limited functionality) specification components into a specification of the overall system of interest.

3. it is amenable to rigorous analysis which will expose specification faults (e.g. contradictions, omissions, consequences).

3.3 - Specific research objectives

This investigation considers which aspects of correct program behaviour can be rigorously specified, and the extent to which that rigour contributes to the validation.

3.3.1 - The formal specification of required behaviour

Developments of recent years have led to considerable speculation that a formal specification can establish program quality in a demonstrable fashion. This investigation considers three main strands of current knowledge in the area of formal specification: state-space, predicate calculus, and process algebra. These areas of major practical importance are represented by the following specification techniques:

1. finite-state models, which demonstrate the use of a well-developed and widely accessible body of results in state-space theory.

2. the Z technique, which demonstrates the use of set theory and predicate calculus.

3. CSP, which demonstrates the use of a process algebra.
These three techniques of formal specification are examined in detail to assess their applicability to the communication behaviour of distributed applications. Each technique is assessed on the following criteria:

1. to which aspects of the program construction and behaviour does the technique apply (e.g. functional, non-functional, etc.)
2. a characterisation of the specification faults detectable using this technique
3. ease of use in application area: has the technique all the necessary concepts to state the required communication behaviour of a distributed application?
4. effectiveness of validation. By the earlier definition of terms, a formal specification is amenable to rigorous analysis. This investigation considers the difficulties (both theoretical and practical) of carrying out an analysis which addresses the relevant modes of behaviour.

3.3.2 - Purity of specification

A commonly held view is that a system specification should ideally be independent of all implementation considerations. That is, a specification should consider what behaviour is to be achieved but not how it is to be achieved. This viewpoint is investigated. The issue is: to what extent can a useful specification be free from assumptions regarding implementation?

3.3.3 - The role of the environment

Much of the published literature on formal specification makes use of undemanding examples to illustrate techniques of specification and analysis. This simplicity is usually achieved by considering a problem which can be treated in severe isolation from the operating environment of the overall system (e.g. the push/pop operations on a stack). In contrast, practical applications of specification are often complicated by the need to take into account the behaviour of the operating environment, and maybe also the effect of the system’s operation on
that environment. This investigation considers the ability of formal specification techniques to take into account these environmental influences on the statement and analysis of required behaviour.

3.3.4 - Functional and non-functional behaviour

The term *functional* is often used very loosely to mean *what the system does*. A more careful consideration distinguishes between functional and non-functional requirements, the former being an obligation while the latter is a constraint on fulfilling an obligation. This investigation considers the ability of formal description techniques to state and analyse both functional and non-functional behaviour.

3.3.5 - An excluded question: system realisation

Validation with respect to a requirements specification is concerned largely with the self-consistency of the requirements specification. However, it is clearly important that one should be able either (1) to derive from the specification an implementation of demonstrable quality, or (2) otherwise show the equivalence of a specification and an implementation. The question of relating a specification to an implementation is considered, but as a relatively minor issue. This does not necessarily reflect the practical importance of the question, but merely the emphasis of this investigation.
Chapter 4 - Finite-state analysis of protocols

This chapter considers the use of finite-state techniques to specify and verify communication protocols. The finite-state specification approach is an appropriate study topic for several reasons. First, it is probably the most commonly used technique in this area, mainly because it is an easily visualised method of specifying sequences of events and it therefore has great intuitive appeal. Second, it is mathematically well founded and is therefore amenable to rigorous analysis.

The basic notions of state automata are summarised in an appendix, and a comprehensive account of state automata can be found in [MIN72]. This investigation adopts the following as a definition of a finite-state machine:

1. a finite (but arbitrarily large) set of input symbols \( I \)
2. a finite (but arbitrarily large) set of output symbols \( O \)
3. a finite (but arbitrarily large) set of internal states \( S \)
4. a response function \( R: I \times S \rightarrow O \)
5. a state transition function \( T: I \times S \rightarrow S \)
6. an initial state \( I_S \)

We first introduce (using an informal graphical finite-state technique) the two communication protocols which are studied in detail in this and in subsequent chapters of this thesis.

We then consider in detail the characteristics of the finite-state specification technique and assess its value as a validation approach.

4.1 - Definition of two communication protocols

Two protocols are considered throughout this thesis:

1. a master/slave protocol (the well-known 2-phase commitment protocol), from [SKE83]
2. a user/server protocol, from [BRAND83]
Both protocols illustrate the need for communication and co-ordinated action of two or more partners.

4.1.1 - master/slave protocol (2-phase commit)

The notion of a commitment protocol has been summarised in the introductory chapter. The 2-phase commit protocol is comprehensively described in [SKE83]. This 2-partner protocol (Fig.4-1) is centralised in the sense that one partner acts as a co-ordinating master while the other acts as a slave. Each partner has an IDLE state, a WAITING state which corresponds to the commitment point of the protocol, and two states COMMIT and ABORT which represent the protocol actions after reaching the commitment point.

![Diagram of 2-phase commit protocol](image)

**FIG.4-1. 2-PHASE COMMIT PROTOCOL**

4.1.2 - A user/server protocol

[BRAND83] describes a simple protocol to regulate the behaviour of two partners, one the provider of a service and the other a user of that service. The protocol (Fig.4-2) allows the provider to notify the user when a fault makes the service unavailable.
4.2 - Indeterminacy in the finite-state specification

The two automata which together specify the 2-phase commit protocol are indeterminate in the sense that their behaviour is not determined solely by the current state and input. This is more clearly seen if the protocol is defined algebraically than graphically, as in Fig.4-1 and 4-2.

4.2.1 - Algebraic definition of 2-phase commit

Algebraically, the master automaton is

\[ I = \{-, no, yes\} \]
\[ O = \{start, -, abort, commit\} \]
\[ S = \{IDLE, WAITING, ABORT, COMMIT\} \]

The master response function \( R \) is

<table>
<thead>
<tr>
<th>I</th>
<th>S</th>
<th>IDLE</th>
<th>WAITING</th>
<th>ABORT</th>
<th>COMMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>yes</td>
<td></td>
<td></td>
<td>abort</td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>start</td>
<td></td>
<td></td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

**FIG.4-2. USER/SERVER PROTOCOL**
The master transition function $T$ is

<table>
<thead>
<tr>
<th>$I$</th>
<th>IDLE</th>
<th>WAITING</th>
<th>ABORT</th>
<th>COMMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>WAITING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>ABORT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>ABORT COMMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The slave automaton is

$I = \{\text{start}, \text{commit}, \text{abort}, -\}$

$O = \{\text{yes}, \text{no}, -\}$

$S = \{\text{IDLE}, \text{WAITING}, \text{ABORT}, \text{COMMIT}\}$

The slave response function $R$ is

<table>
<thead>
<tr>
<th>$I$</th>
<th>S</th>
<th>IDLE</th>
<th>WAITING</th>
<th>ABORT</th>
<th>COMMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The slave transition function $T$ is

<table>
<thead>
<tr>
<th>$I$</th>
<th>S</th>
<th>IDLE</th>
<th>WAITING</th>
<th>ABORT</th>
<th>COMMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>WAITING ABORT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td>ABORT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.2.2 - Three sources of protocol indeterminacy

One can distinguish three sources of indeterminate behaviour in the algebraic definition of the 2-phase commit protocol. We classify these sources as types 1, 2 and 3 indeterminacy.

**Type 1:** the two transition functions of Fig.4-1 are multi-valued, causing some transitions to be ambiguously triggered. The master in WAITING state can enter either the ABORT or COMMIT state on receiving a "yes", while the slave in IDLE state can enter either the WAITING or ABORT state on receiving a "start". Both master and slave are clearly using some source of information other than the current state and input.

In formal language theory it is necessary to distinguish between an acceptor and a generator of a language. A generator of a language produces all sentences of the language (and no other sentences) and may be indeterminate in the above sense. But an acceptor recognises all sentences of the language (and no other sentences) by allocating a unique sequence of state transitions to each valid language sentence. The protocol partner of interest in this application is an acceptor of sentences (i.e. sequences of signals from the remote partner), and must be unambiguous.

**Type 2:** in some transitions of Fig.4-1 the input is unspecified. The input which drives the master from the IDLE to the WAITING state is not stated, so what controls the master’s behaviour?

In fact, there is a good use for such unspecified inputs. [DEN78] uses such a device (the lambda-transition) to eliminate certain difficulties in the composing of state automata, thus simplifying the proof of Kleene’s theorem. But this is never more than a constructional device and [DEN78] is careful to give a procedure for eliminating these transitions.

**Type 3:** for both master and slave in Fig.4-1, both the transition and response functions are incomplete: the behaviour of the
automata is not specified for all possible combinations of input and state. The protocol is specified for all correct signal sequences, but not for the incorrect sequences. These incorrect sequences are of great practical interest. They can occur because of a communication breakdown which causes the two automata to lose synchronisation, or because one of the automata has been incorrectly implemented.

Note that unspecified outputs do not cause indeterminacy. A transition of the master which produces no output merely risks being unobservable to the remote slave. It does not violate the principle that the master’s behaviour is determined solely by the current state (however arrived at) and input.

The types 1 and 2 indeterminacy are described by [AHO86]. The two automata of Fig.4-1 are nondeterministic, whereas a deterministic automaton fulfils two conditions:

1. for each state S and input symbol A, there is at most one edge labelled A leaving S
2. no state has a leaving edge which is labelled by the empty input symbol E

A recogniser can be either deterministic or nondeterministic, but there is a space/time trade-off. A deterministic automaton is fast in operation because the decisions are very simple at each transition, but the transition function may need to be very large to hold each decision explicitly. A nondeterministic automaton is economical in storing the transition function, but the indeterminacy resulting from the empty input symbol E significantly complicates the transition decisions. In practice, a deterministic recogniser is preferable to a nondeterministic one. [AHO86] describes the subset construction algorithm which generates a deterministic automaton from a nondeterministic one.

4.3 - Removing protocol indeterminacy

These sources of indeterminacy have also been investigated at length by [BRAND83], who has devised elaborate algebraic tests to detect these
structural faults in the response and transition functions. However it seems more constructive to remove the faults by recognising explicitly the non-protocol sources of information. The following section describes a procedure for removing some indeterminacy. It is less formally presented that [AHO86], but more complete in the sense that it acknowledges the semantics of the Fig.4-1 application by generating mnemonic signal identifiers.

4.3.1 - Removing indeterminacy - types 1 and 2

There is a simple procedure which will transform the protocol of Fig.4-1 so as to remove the indeterminacies of types 1 and 2 (above). It relies on the fact that the protocol automata do not operate in their own right, but are merely the front ends of two communicating application programs. Conditions arising in the application programs allow the automata to make those seemingly indeterminate decisions.

A convenient way to take account of this application-derived information is to consider that each automaton receives input signals from 2 sources:

1. from the other automaton
2. from the local application program on whose behalf it is communicating.

The protocol definition of Fig.4-1 includes only source (1). To take account of source (2) it is necessary to list all the application conditions which relate to communication and to name a distinct signal for each one. This device is described by several authors. For example, [SUN82] recommends that some events be interpreted as "external" or "callable by a user of the protocol". [LEM82] formally models events which are external (in the sense of [SUN82]) in order to formulate hierarchical protocols.

To remove Type 1 indeterminacies,
1. look for multi-valued entries in the state transition function of each protocol automaton.
where a multi-valued entry is found, introduce a unique new state DECIDING and invent $N$ unique application input signals to decide which of $N$ transitions is taken on exit from the new state.

Thus, the master of Fig.4-1 in WAITING receives a "yes" and enters a new state DECIDING, the exit from which is controlled by the signals AM2 and AM3. The slave’s IDLE state is similarly expanded.

To remove Type 2 indeterminacies, look for unspecified (i.e."-" ) inputs to the local transition function. Replace each "-" input with a unique application signal.

4.3.1.1 - 2-phase commit with no type 1 or 2 indeterminacy

The augmented set of input and output signals for the master is

$$I = \{\text{no}, \text{yes}, \text{AM1}, \text{AM2}, \text{AM3}\}$$
$$O = \{\text{start}, -, \text{abort}, \text{commit}\}$$

where

AM1 = "master application wants to communicate"
AM2 = "master application wants to continue"
AM3 = "master application wants to abort"

The augmented set of input and output signals for the slave is

$$I = \{\text{start}, \text{commit}, \text{abort}, \text{AS1}, \text{AS2}\}$$
$$O = \{\text{yes}, \text{no}, -\}$$

where

AS1 = "slave application agrees to communicate"
AS2 = "slave application refuses to communicate"
4.3.1.2 - user/server with no type 1 or 2 indeterminacy

The augmented set of input and output signals for the user is

\[ I = \{ \text{alarm, done, AU1, AU2} \} \]

\[ O = \{ \text{req, ack, -} \} \]

where

AU1 = "user wants service"
AU2 = "user acknowledges remote fault"

The augmented set of input and output signals for the server is

\[ I = \{ \text{req, ack, AS1, AS2} \} \]

\[ O = \{ \text{alarm, done, -} \} \]

where

AS1 = "provider completes the service request"
AS2 = "provider develops fault, cannot offer service"
4.3.2 - Removing indeterminacy - type 3

The transformed specifications of Fig.4-3 and Fig.4-4 still contain indeterminacies of type 3: the transition function contains blank entries, and the behaviour of the protocol is therefore not specified in all possible circumstances. The blank entries correspond to errors in transmission and are of considerable practical importance. It seems reasonable to require a specification of the action to be taken when errors occur.

Essentially, one needs to convert the partial transition function into a total function by adding error elements to the definition.

The simplest solution is to treat every error as fatal: every blank entry in the transition function brings the protocol exchange to a final stop with no provision for recovery and restart. A more ambitious approach is to distinguish between those conditions from which the protocol can recover without data corruption, and those conditions which must be abandoned.
However, a pessimistic conclusion is justified here. A formal description is simply an algebraic form: there can be no demonstrable proof that the form accurately represents any particular informal concept. The formality allows us to recognise that certain signal sequences (e.g. indefinite repetitions) can never be accepted by a protocol partner. Of the remaining valid sequences, we have no formal way of distinguishing between correct sequences, recoverable errors and fatal errors. The best hope is to recognise informally that certain protocol configurations are correct (or recoverable or fatal), and to look for a formal representation of them. This formal representation can then be added to a list of tests to be applied.

4.4 - Global state analysis

The technique of global state enumeration is adopted to analyse a finite-state specification of N co-operating automata. The usual analytic analysis techniques applied to a state automaton are intended to establish such properties as the observability and (of particular interest in this connection) the reachability of selected states. We are certainly interested in these properties, but we are also interested in the co-operative behaviour of N automata. To examine this, it is necessary to consider how the operation of automaton A affects the operation of automaton B, and vice versa. The concept of the global state permits a complete description of the instantaneous condition of N co-operating automata.

A global state crystallises a particular stage in the conduct of a protocol exchange. For any 2-partner protocol, the global state consists of

1. the instantaneous local state A of partner 1
2. the instantaneous local state B of partner 2
3. the instantaneous contents of the message channels linking the partners.

The global reachability relationship (also known as the dependency or precedence relationship) is represented as a tree. A single directed arc links (i.e. "makes adjacent") those global states SS and SSS where SSS can be derived from SS as the result of a single instantaneous protocol
event in a single partner. The set of all global states (together with their associated reachability tree) describes every possible combination of events which can occur during any protocol exchange.

4.4.1 - Enumerating global states

The detailed construction of global states and reachability trees is described in Appendix C. This treatment of global state analysis is inspired by [BRAND83], [SUN82], and [LEM82]. Some of the generality of [BRAND83] is omitted in this treatment, leading to a reduction in theoretical power and a large reduction in program complexity. In particular:

1. the number of protocol partners is fixed at 2
2. the channel bandwidth is limited to at most 1 unprocessed message
3. complicated tree-pruning heuristics are avoided by recognising certain conditions for halting the growth of a tree branch (e.g. bandwidth errors, deadlock, repetition of earlier global states)

The definition of the condition (the stable global state) which indicates a deadlock is due to [BRAND83]. The method for removing Type 2 indeterminacies is essentially that suggested by [SUN82] and [LEM82].

The global reachability trees of the protocols of Fig.4-3 and 4-4 are shown in appendices D and E respectively. The leaves of the tree are terminating global states. A terminating state indicates that one of the following protocol situations has arisen:

1. a bandwidth limit has been exceeded
2. a deadlock has arisen
3. an illegal (or unspecified, to use the notation of [BRAND83]) signal has been received
4. a state identical to a previously enumerated state has been discovered. There is no need to continue the enumeration of the new state since this will result in the same subtree as did the earlier state
4.5 - Observations and conclusions

4.5.1 - Memory limitations

Consider how to generalise the 2-phase commit protocol from 2 to N partners, where N is finite but is fixed by the operating conditions rather than by the protocol definition. The master must wait until each of N-1 slaves has offered to commit and must count each offer as it arrives. It is not possible to count the offers by looping around the single WAITING state because that state (like every other state) is an equivalence class of automaton histories. It is by definition impossible to distinguish between the first transition into WAITING and the (N-1)th transition. The master therefore needs N-1 distinct WAITING states to count the N-1 offers. These states form a sequence where the first state means "one slave has offered", the second means "two slaves have offered", etc.. (Although note that even this complication does not rule out the possibility that a rogue slave might offer twice, thus confusing the count.) It follows that a strictly finite-state automaton can specify this protocol only for specific values of N, but not for an arbitrary N.

More generally, a finite-state automaton is unable to count an arbitrarily large number of events. It can count at most up to the number of its states. It is restricted to problems which require the sorting of stimuli into no more than a specific and finite number of classes. It is pointed out in an appendix that the class of languages recognised by a finite-state automaton is precisely the set of strings generated by Kleene's regular expressions. We can formalise the counting limitation of a finite-state protocol by observing that an arbitrarily long sequence of counting states are characterised by the regular expression

\[ A; (B^*); C \]

There is no guarantee that the number of repetitions of B will always be less than the number of states of the automaton.

This limitation comes to light in the following practical circumstances:

1. In a particular fault-tolerant network, the master is required to decide to commit after a majority of the slaves have offered.
The remaining slaves might be incapable of responding, so there is no point in waiting until all offers have arrived.

2 if two protocol partners lose synchronisation (e.g. through a fault in the signal transmission) they might try to resynchronise by making a number $M$ of handshakes. Each partner must count up to $M$ before abandoning the attempt.

4.5.2 - State explosions

A finite-state specification is formally analysed by considering all possible state configurations. For an $N$-partner protocol, this is done by constructing the set of total or global states (e.g. [BRAND83]). A total state is an $N$-element vector where each element $i$ represents the current state of the $i$th partner. If there are $N$ partners each of $M$ distinct states, then the number of total states is $M^N$.

This illustrates a general feature of exhaustive finite-state analysis: the number of total states is exponential in the number of distinct states. There are broadly two ways of dealing with this exponential growth:

1 some workers deal not directly with the states but with classes of states which have been axiomatically defined. These approaches require the specification to be extended beyond the capabilities of strictly finite-state machines. Thus, [JOS86] constructs variables and logical assertions over the states, while [THOM81] manipulates states whose function is informally a stack or queue or some other memory device.

2 [BRAND83] sticks to the strict finite-state approach, and expands the specification to its complete set of total states. He then looks for certain structural properties similar to the three types of indeterminacy described earlier.

The first approach is not strictly finite-state, while the second is soon limited by the computational size of the problem.
4.5.3 - Correctness criteria are informal

The global reachability trees of the protocols of Fig.4-3 and 4-4 are shown in Appendices D and E respectively. The correctness of the protocol is decided by inspecting the tree leaves (i.e. the terminating global states).

The 2-phase commit protocol has three terminating states, all of which indicate that either both partners commit or both partners abort. This is indeed what the protocol is intended to achieve, but note carefully that the enumeration of Fig.4-3 contains no definition of correct termination. The test of correctness comes from the informal semantics of the protocol.

Of course, it is possible to state the 2-phase commit criterion neatly and formally in terms of states, but the user/server protocol of Fig.4-4 shows that even a modest increase in complexity makes this very difficult. Here, the finishing states are more numerous and it is far from obvious which indicate success and which indicate failure. In fact, the criterion for the successful user/server finishing states comes from the observation that the user and the server cycle endlessly, starting from the initial global state and eventually returning to that state: any protocol event which breaks this cycle must be wrong. The successful finishing states are either identical to the initial global state, or to global states which in turn develop into the initial state.

This observation that correctness is stated informally is consistent with the earlier pessimistic conclusion that Type 3 indeterminacies cannot be removed by any formal means. In both the above protocols it is necessary to consider the semantics of the protocol operation to arrive at a test of correctness.

4.5.4 - Protocol complexity is not obvious

The complexity of a protocol is indicated by the number of possible interactions between partners. A simple protocol has few possible outcomes, while a complex protocol has relatively many. This
complexity measure has a neat interpretation in global state analysis: a simple protocol has a reachability tree with few leaves. A casual comparison of Fig.4-3 and 4-4 might encourage the conclusion that the 2-phase commit protocol is more complex than the user/server protocol, but a comparison of the two trees shows that the reverse is true. There are two reasons why the user/server is the more complex. The first is concerned with unobservable states in Fig.4-3, and the second with the genuine complexity of events in Fig.4-4.

4.5.4.1 - Unobservable states in Fig.4-3

Recall that a protocol partner is interested only in the observable behaviour of the remote partner. Fig.4-3 has more states (and therefore potentially more interactions) than Fig.4-4, but some of the 4-3 states are unobservable to the remote partner. These are the DECIDING states introduced to remove Type 1 and 2 indeterminacies. They are effectively housekeeping states which give rise to no new protocol outcomes.

4.5.4.2 - Complexity of events in Fig.4-4

Compare the semantics of the two protocols. In the 2-phase commit protocol, partner B acts only in response to the actions of partner A, and vice versa. The actual response of B depends on signals received from the local B application (i.e. "agree to communicate", "refuse to communicate", etc.) but no state transition visible to A occurs except in response to a signal from A.

In contrast, each partner of the user/server protocol makes an externally visible transition in response to two independent sources of signals: from the remote partner, and from the local application. The relative timing of these sources is unpredictable and race hazards can arise. The reachability tree represents a race hazard as a node where a partner must choose which of N available input signals is to be processed first: each of the N signals gives rise to a distinct subtree.
The tree of Fig.4-4 contains several interesting examples. With the user in state READY and the server in state FAULT, the user has the choice of processing either an AU1 or an ALARM, or the local application might erroneously repeat its AU1 before the user has been able to process the earlier AU1. The five outcomes result directly from this race hazard and represent genuine protocol complexity.

Note that it is inadequate to argue that the partner should avoid race hazards by imposing a queue discipline on the inputs. This works only when all potential signals are available and the partner can make a choice based on full information. But the partner has no control over the arrival of signals and can never be sure that the best current decision will remain valid after the next signal has arrived.

4.6 - Conclusions on finite-state protocol specifications

The aim of this study is to investigate the extent to which the correct behaviour of a program can be formally stated and then analysed to expose consequences and contradictions. From the consideration of finite-state protocol specifications it becomes clear that although the manipulation is straightforward, the correctness criteria are largely informal and therefore outside the specification. The formal analysis will enumerate all possible sequences of events, but will not identify a particular sequence as correct or incorrect. This is consistent with the earlier conclusion that Type 3 indeterminacy cannot removed mechanically.

Certain structural faults can be formally detected, such as the exceeding of a bandwidth limit or the receiving of an unspecified stimulus. But the purpose of the protocol must be taken into account to decide whether or not the formally enumerated outcomes represent correct behaviour. The formal enumeration which is made possible by the finite-state specification does not itself answer the question "Is the protocol correct?". Rather, the formal enumeration lists the possible outcomes and draws attention to certain structural events.
In some cases it is possible to state correctness simply in such structural terms. For example, the 2-phase commit protocol is correct if all finishing global states leave both partners in the same local state. But it seems unlikely that such neat statements of correctness are common. Even the modest complexity of the user/server protocol supports this speculation.

However, a more optimistic attitude is justified if one sets aside the essentially philosophical problems of formalising correctness and meaning. The global state enumeration exposes all possible behaviour of the protocol: within the limitations of the tree-growing algorithm (e.g. bandwidth limitations, implied or explicit queue disciplines, etc.) a global state tree says all there is to say about the operation of the protocol. If the algorithm restrictions are realistic, and if an adequate technique is available for pruning the (in general) infinite tree, then the global state analysis describes all possible protocol behaviour and therefore makes possible a verification of the protocol specification. Appendix C defines such an algorithm, and Appendices D and E illustrate its application.
Chapter 5 - The Z specification of communications protocols

This chapter reconsiders the two communication protocols which have already been described in detail: the 2-phase commit protocol, and the user/server protocol. The two are specified in terms of the Z specification language, which is defined in [SUF83], [MOR84] and [ABR87]. The Z specification language illustrates the use of two major fields of mathematics i.e. set theory and predicate calculus.

This chapter is structured as follows:
1 a brief introduction to Z
2 indeterminacy, and how to express it in Z
3 comments on the fundamental role of state in a class of applications
4 a detailed consideration of the Z specification of the 2-phase commit protocol
5 an examination of the Z facilities for stating protocol properties.
6 a clean example illustrating issues from the previous sections - the Z specification of the user/server protocol.
7 conclusions on the use of Z to specify protocols

5.1 - Introduction to Z

The inspiration for the Z specification technique is the belief that informal techniques (and especially natural language) are incapable of communicating a sufficiently precise system specification. This imprecision has the effect that changes which are apparently minor and localised in fact cause significant disturbances in unpredictable areas of the system. It is asserted [ARB87] that a precise specification can reduce or eliminate such uncertainty in the system behaviour.

5.1.1 - Basic concepts of Z

The Z specification technique is set out succinctly in [ABR87] and more descriptively in [SPI89] [W0089].
A Z specification is based on a general system model known as an abstract machine. An abstract machine consists of a state and one or more actions which modify that state. The analysis of such a system consists in studying the statics and the dynamics. The statics define the state, and the dynamics define the possible changes of state.

It is a requirement of every Z specification that the statics and dynamics be independent of each other. This arises from the practical consideration that some actions might remain unknown until the development of the specification is well under way, and it would be undesirable for the addition of a new action to force a new specification of the state.

The state of a system consists of three parts:

1. one or more basic sets of objects to represent information. Any set may be used: finite or infinite, numbers, functions, etc.
2. one or more variables which determine the various components of the state
3. an invariant which makes clear the static laws of the system. The invariant is defined in terms of the variables and basic sets using any relevant facilities of predicate calculus and set theory.

The dynamics of the system specifies the actions which are permitted to alter the state, but always within the constraints imposed by the static invariant. The action is considered to take place instantaneously between two adjacent instants and is specified by the value of the relevant variables in the instant immediately before and immediately after the action. The value after the action is indicated by a prime. For example, the action INCREMENT is specified by the values of the variable SUM as follows:

\[ \text{Eq-5.1} \quad \text{INCREMENT} \rightarrow \quad \text{SUM}' := \text{SUM}+1 \]

An action is more generally described as a substitution. For example, given a formula \( P \), an expression \( E \) and a variable \( V \), the following
construct denotes \( P \) after all (free) occurrences of \( V \) in \( P \) have been replaced by \( E \).

**Eq-5.2** \([V:=E]P\)

A substitution may replace several (free) variables simultaneously

**Eq-5.3** \([V_1,V_2:=E_1,E_2]P\)

But note that multiple substitution and successive single substitutions may produce different effects: care must be taken to observe the rules of free and bound variables.

There are several conditional forms of substitution: the simple choice, the pre-conditioned choice, the guarded substitution, and the non-deterministic substitution. These constructs are considered in detail in a later section on indeterminacy.

From the viewpoint of rigorously establishing specification properties, the significance of a substitution is that it must preserve the static invariant. This can be neatly stated in terms of Hoare's notation [HOA69]. Given an invariant \( I \) and a substitution \( S \), then

**Eq-5.4** \( I(S)I \)

which reads "if \( I \) holds just before the activation of \( S \), and if \( S \) terminates, then \( S \) establishes \( I \)". In other words, \( S \) preserves the invariant \( I \). More detailed statements of termination conditions and proof rules for various circumstances are considered in [ABR87].

### 5.1.2 - Indeterminacy in Z

It is important to realise that in Z indeterminacy differs from ignorance. Indeterminacy arises where several acceptable solutions are known, but the final choice of a unique solution has not yet been made. In contrast, ignorance indicates a situation where there is insufficient information to arrive at any non-trivial solution.
Consistent with this distinction, [ABR87] points out that if a specification is indeterminate in the sense that it fails to distinguish between several possibilities, then:

1. all possible solutions must preserve the static invariant
2. the eventual implementation must fix the choice, as opposed to relying on some non-deterministic implementation technique

5.1.2.1 - Protocol indeterminacy

It seems reasonable to assume that the Types 1 and 2 indeterminacy of the 2-phase commit protocol of Fig 4.1 cannot be resolved at implementation time. Consider the indeterminate action of the master in state WAITING on receiving a yes signal from the slave. It is possible (but impractical) that a particular implementation could constrain the master always to reply commit. But a more reasonable interpretation is that the decision to commit or abort is intended to be based on the instantaneous condition of the application which is driving the master protocol partner. The Types 1 and 2 indeterminacy of Chapter 4 represent ignorance of the behaviour of the protocol automaton rather than implementation freedom. This ignorance can be removed only by making explicit those conditions of the communicating applications (see Chapter 4) which relate to communication.

5.1.2.2 - Z indeterminacy mechanisms

The Z indeterminacy mechanisms [ABR87] are:

1. simple choice \([S \lor T]Q = [S]Q \text{ and } [T]Q\)
2. pre-conditioned choice \([P \mid S]Q = P \text{ and } [S]Q\)
3. guarded substitution \([P \rightarrow S]Q = P \text{ implies } [S]Q\)
4. non-deterministic substitution \([\@x.S]Q = \forall x.[S]Q\)

In the simple choice, one is permitted to implement either the [S] or the [T] substitution, while the non-deterministic substitution allows the [S] substitution to be implemented with any specific value of the (bound) variable x. The simple choice and the non-deterministic substitution both
represent implementation freedom, and are therefore irrelevant in the context of Types 1 and 2 indeterminacy.

The guarded substitution is defined in terms of the \text{implies} operator and raises the well known ([QU180], [ABR87]) problem of how to interpret the conditional when the guarding predicate is false.

<table>
<thead>
<tr>
<th>P</th>
<th>[S]Q</th>
<th>\text{implies}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When P is false, the \text{implies} relation is true irrespective of the value of the substitution [S]Q. Z recognises this difficulty by noting that such a \textit{miraculous} [ABR87] substitution can achieve an arbitrary postcondition. Z counters this unreasonable possibility by introducing a special test of \textit{feasibility}: a substitution is not miraculous if there exists at least one unachievable postcondition. That is, a substitution (guarded or otherwise) [S]Q is feasible if (and only if) \textbf{not} [S]P is true for some postcondition P.

The obligation to prove feasibility (in the Z sense) is an added burden, so let us consider the remaining mechanism: the pre-conditioned choice. The construction [P|S]Q means that the postcondition Q is true after the substitution [S] in all cases where the pre-condition P is true, and that nothing is guaranteed about Q if [S] is carried out when P is false.

The semantics of the pre-conditioned choice corresponds nicely with the idea that a fatal error results from processing which takes place in unexpected circumstances. It seems to be adequate for specifying the correct protocol sequences of Fig. 4.1. For the remainder of this chapter, the pre-conditioned choice is the only Z indeterminacy mechanism which will be used.
5.1.3 - Appropriate levels of rigour and abstraction

The Z specification language is rigorously based on the formal disciplines of set theory and predicate calculus. However, Z has the concept of appropriate formality, and a complete (constructive) definition is not demanded for every object in the specification. This is in line with the approach typified by [JONES80], where the degree of formality is no greater than is needed to satisfy the user (whoever that may be) of the specification.

A Z specification may begin as a highly abstract description of behaviour and be successively refined [MOR85] [SPI89] to arrive at an executable program. However, this process of refinement is not of direct interest to this study of the specification facilities of Z. For the purposes of this study it is sufficient to note that an abstract Z specification is not usually executable. It may even be impossible to derive an equivalent executable program: see [ABR87] for examples of correct but unrealisable specifications.

This study considers the Z specification at an appropriate level of abstraction and neglects any subsequent refinement stages.

In arriving at an appropriate level of Z abstraction for the two protocols of interest here, one must anticipate the eventual implementation to some extent. At one (very abstract) extreme one might ignore all internal structure of the distributed application. But recall that one is concerned to specify and analyse the communication behaviour which co-ordinates the several distributed components of the application. One must therefore constrain the abstract specification to describe several distinct application components, in just the same way that the finite-state specification is structured into master and slave, user and server.

5.1.4 - Choice of notation

In addition to the mathematical elements of Z which are set out in [ABR87], an elaborate notation has developed. This notation is largely concerned with the statement of schemas and their relationships.
Although not fundamental to the primitive Z concepts, this schema notation has a significant impact on the practical use of Z. In this thesis the notation set out in [SPI89] [W0089] will be used.

5.2 - The fundamental role of "state" in certain applications

The two protocols of Chapter 4 are described in terms of state transition diagrams. It might be thought that the use of such diagrams is simply the result of choosing a particular specification technique. But in fact the concept of state plays a more fundamental role in a certain class of applications, including communication protocols. This class is considered below.

A repetitive process is characterised by the periodic nature of its behaviour. Periodic behaviour has the consequence that two identical stimuli (separated by a sufficiently long interval) produce an identical response.

In some cases the behaviour of a process is conveniently thought of as a sequence of checkpoints. A checkpoint has the property that, once achieved, it is never lost. This checkpoint approach is useful for two types of application:

1. those which model processes whose semantics indicate irreversible actions
2. those which model processes where a great amount of detail may need to be taken into account before a decision is made. In these cases it is practical to classify the major decisions as irreversible so that once the decision is made, the context of the decision (i.e. all the minor detail which contributes to the decision) can be ignored

Both repetition and checkpointing are exactly formalised by the concept of state [MIN72]. Each possible history is a sequence of events. A state is an equivalence class of histories. The nature of the equivalence relation guarantees that no two histories which achieve a particular state are distinguishable. This loss of detail matches well the semantics
of an irreversible decision or checkpoint, while repetition consists of visiting the same state several times.

Communication protocols exhibit both repetition and checkpointing, and the state description therefore plays a fundamental role. One might argue that a description based on some message-passing notation (e.g. CCS [MIL80] or CSP [HOA85]) is equally fundamental. However, the analysis of message sequences usually requires formal language theory, a significant amount of which is formulated in terms of state automata. The message-passing approach will examined in a later chapter.

This chapter will therefore concentrate on a state-based description, but the ability of the Z approach to avoid some of the fundamental finite-state restrictions will be examined.

5.3 - The 2-phase commit protocol in Z

In this section we use the Z concepts to specify the 2-phase commit protocol of Fig.4-1. Although several Z tutorials (e.g. [SPI89]) explicitly describe the static component of a Z schema as the set of states (and their invariant relationships) which a system can occupy, it appears that Z is not primarily concerned with finite-state concepts: a later section will demonstrate the use of Z variables in a non-state way to avoid some of the fundamental limitations of finite-state automata. However, an earlier section argues that these concepts are of fundamental importance in certain circumstances, and to this extent the Z specification must reflect the finite-state structure of Fig.4-1. The approach here is to use the Z statics to describe the structure of the protocol (i.e. communicating partners, legal signals, etc.), and to use Z substitutions involving primed variables to describe Z actions which mimic state transitions.

5.3.1 - Basic sets, variables, static invariant

The basic sets of Z objects are:

1 the legal states of the master
The 2-phase commit protocol is specified in terms of two communicating components - a *master* and a *slave*. There is exactly one master and exactly one slave. With each is associated a set of states, a set of input signals, and a set of output signals.

The static invariant for this protocol application simply requires that no inputs, outputs, or states other than those above may appear during the operation of the protocol. This amounts to the restatement of set membership.

The definition of the protocol actions requires three variables to denote the instantaneous value of the master's state input and output \((ms,mi,mo)\), and three similar variables for the slave \((ss,si,so)\).

<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ms) : state-m</td>
<td>(ss) : state-s</td>
</tr>
<tr>
<td>(mi) : input-m</td>
<td>(si) : input-s</td>
</tr>
<tr>
<td>(mo) : output-m</td>
<td>(so) : output-s</td>
</tr>
<tr>
<td>(ms \in {idle,waiting,abort,commit})</td>
<td>(ss \in {idle,waiting,abort,commit})</td>
</tr>
<tr>
<td>(mi \in {no,yes})</td>
<td>(si \in {start,abort,commit})</td>
</tr>
<tr>
<td>(mo \in {start,abort,commit})</td>
<td>(so \in {no,yes})</td>
</tr>
</tbody>
</table>

This description is profoundly static. It assumes that the roles of master and slave never change, and that the membership of all sets is fixed by initialisation.
The sets ms and ss are exactly the states defined in the finite-state specification of Chapter 4. The Z notion of state corresponds to the *global* state of Chapter 4: it is a vector of the instantaneous values of all quantities necessary to characterise the application completely. The sets so and mi are necessarily identical, because all slave outputs end up as master inputs. For the same reason, the sets mo and si are identical.

5.3.2 - Dynamics - substitutions as state changes

We now use the Z pre-conditioned substitution to state the actions which may take place during the protocol operation. The following example adopts the notation of [SPI89] where the static invariant is treated as an implied component of every pre-condition. Given the static invariant Inv, and taking the master as an example, if an input I in a state S produces the output O and causes a transition to the new state SS, then we can write this

```
Substitute in master

ΔMaster
i? : input-m

ms = S
i = I
ms’,mo! := SS,O
```

Note the interpretation of the pre-condition. If the substitution is applied in the wrong context (i.e. when the pre-condition is *false*) then a fatal error results.

In defining the protocol actions as substitutions, we are actually enumerating the members of the relationships which are (in finite state terms) the transition and response functions of the communicating components.

This notation using a multiple substitution makes it clear that the whole substitution is a single atomic action: there is no ambiguity concerning the relative timing of the state change and the generation of the output
signal. (The earlier remarks on the need to pay careful attention to free and bound variables do not apply here, since no quantifiers are used and all variables are therefore free.)

5.3.3 - Dynamics - removing decision ambiguities

But we must first deal with the decision ambiguities of Fig. 4.1 which have already been discussed at length in Chapter 4. These arise from the fact that a protocol partner is driven by an application, and Fig. 4.1 contains no explicit description of the various circumstances of the application. This ambiguity shows up in two ways: non-specific input, and ambiguous decisions.

A non-specific input is demonstrated by the master in state idle: the input signal is unspecified.

<table>
<thead>
<tr>
<th>Substitute in master</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔMaster</td>
</tr>
<tr>
<td>i? : ???????????</td>
</tr>
<tr>
<td>ms = idle</td>
</tr>
<tr>
<td>i = ???????????</td>
</tr>
<tr>
<td>ms',mo! := waiting, start</td>
</tr>
</tbody>
</table>

An ambiguous decision is demonstrated by the master receiving yes in state waiting, and then deciding either to commit or abort:

<table>
<thead>
<tr>
<th>Substitute in master</th>
<th>Substitute in master</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔMaster</td>
<td></td>
</tr>
<tr>
<td>i? : input-m</td>
<td></td>
</tr>
<tr>
<td>ms = waiting</td>
<td></td>
</tr>
<tr>
<td>i = yes</td>
<td></td>
</tr>
<tr>
<td>ms',mo! := commit, commit</td>
<td></td>
</tr>
</tbody>
</table>

| ΔMaster              |
| i? : input-m         |
| ms = waiting         |
| i = yes              |
| ms',mo! := abort, abort |

These ambiguities cannot be ignored. To remove them from the specification, we must take into account the circumstances of the communicating applications which enable the protocol partners to make
unambiguous decisions. This is a fundamental feature of any communication protocol and is independent of the particular specification technique adopted. The application circumstances have already been listed in Chapter 4. They are

<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms : state-m</td>
<td>ss : state-s</td>
</tr>
<tr>
<td>mi : input-m</td>
<td>si : input-s</td>
</tr>
<tr>
<td>mo : output-m</td>
<td>so : output-s</td>
</tr>
<tr>
<td>me : environment-m</td>
<td>se : environment-s</td>
</tr>
</tbody>
</table>

- **ms** ∈ \{idle, waiting, abort, commit\}
- **mi** ∈ \{no, yes\}
- **mo** ∈ \{start, abort, commit\}
- **me** ∈ \{AM1, AM2, AM3\}
- **ss** ∈ \{idle, waiting, abort, commit\}
- **si** ∈ \{start, abort, commit\}
- **so** ∈ \{no, yes\}
- **se** ∈ \{AS1, AS2\}

At this point one could repeat the procedures of chapter 4: one could treat these conditions as explicit signals and introduce auxiliary states (WAITING, DECIDING, etc.) to describe the protocol behaviour when they arise. However, this introduction of artificial states is unnecessary in a Z specification. A better solution is to stay closer to the spirit of the Fig.4-1 specification by encoding the above application conditions as the guarding predicates of pre-conditioned substitutions. This avoids introducing unnecessary structure into the specification, and makes full use of the appropriate Z facility with its well-considered theoretical foundation. We therefore introduce these environmental conditions as another basic set of the Z protocol specification.

But note finally that there is one variety of ambiguity which can be tolerated. This is the indeterminate output, as demonstrated by the master on receiving *no* in state *waiting*. This is of minor importance and
can be removed by setting the instantaneous output to the special value undefined.

Note that one should not model this situation as a persistent and unchanging output value, as in

To see why not, consider the semantics of protocol operation. Each output is a discrete signal which remains valid only long enough to allow the receiving partner to act on it. In physical terms, it must be considered a pulse rather than a persistent signal. By adopting the rule that an undefined output takes on the previous output value by default, one is in fact modeling a persistent signal. Chapter 4 on finite-state analysis points out that an undefined output does not lead to decision ambiguity, but merely raises the possibility of unobservable states.

5.3.4 - Dynamics - the 2-phase commit master and slave

The dynamic behaviour of the 2-phase commit protocol is expressed by relationships which are known (in finite state terms) as the transition and the response functions. Each member of these relationships is an action of the protocol, and is therefore appropriately specified as a Z schema. The initial state and the protocol actions of the master and slave are specified by the following schemas.
<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialise master</td>
<td>Initialise slave</td>
</tr>
<tr>
<td>Master</td>
<td>Slave</td>
</tr>
<tr>
<td>ms = idle</td>
<td>ss = idle</td>
</tr>
</tbody>
</table>

### Master transition 1

**Master**
- ms = idle
- me = AM1
- ms', mo! := waiting, start

**Slave**
- i? : input-s
- ss = idle
- se = AS1
- i = start
- ss', so! := waiting, yes

### Master transition 2

**Master**
- i? : input-m
- ms = waiting
- i = no
- ms', mo! := abort, undefined

### Master transition 3

**Master**
- i? : input-m
- ms = waiting
- me = AM2
- i = yes
- ms', mo! := commit, undefined

### Master transition 4

**Master**
- i? : input-m
- ms = waiting
- me = AM3
- i = yes
- ms', mo! := abort, abort

### Slave transition 1

**Slave**
- i? : input-s
- ss = idle
- se = AS1
- i = start
- ss', so! := waiting, yes

### Slave transition 2

**Slave**
- i? : input-s
- ss = idle
- se = AS2
- i = start
- ss', so! := abort, no

### Slave transition 3

**Slave**
- i? : input-s
- ss = waiting
- i = commit
- ss', so! := commit, undefined

### Slave transition 4

**Slave**
- i? : input-s
- ss = waiting
- i = abort
- ss', so! := abort, undefined
5.4 - Z facilities for stating protocol properties

The previous section is a complete specification of the 2-phase commit protocol. The basic sets of specification objects are stated; the state variables are given in terms of those basic sets; the dynamic operations which alter the value of those state variables are defined.

One can now make two major observations concerning the power of the Z facilities, one favourable and one unfavourable. An unfavourable observation concerns the surprising difficulty in specifying certain reasonable properties relating to the correct operation of a protocol. We consider the source of this difficulty, propose a major (but not fundamental) extension to Z, and illustrate the use of this extension.

The favourable observation relates to Z's ability to avoid the fundamental limitations of a strict finite-state automaton which are described in Chapter 4 and which lead to considerable inflexibility in specifying adaptive behaviour. We illustrate the use of Z in a practical approach to avoiding these fundamental limits.

5.4.1 - The enumerative statement of properties

The construction of the Z specification is transparent: one assembles a collection of dynamic operations which modify some global state, all operations subject to a static invariant. The modular construction is clearly an effective technique for separating the subgoals of the specification task: in this case, each subgoal corresponds to some protocol action. The rigorous basis of the Z notation facilitates the self-consistency of the overall specification i.e. the arguments and results of operations and the set membership are well defined at all times.

In a strict sense there is nothing more to say about the protocol. The Z specification has recorded in a self-consistent way all the available information. But in a practical sense one also wishes to state and demonstrate certain desirable properties of the protocol (e.g. absence of unspecified stimuli, liveness, etc.). That this proof facility is an important
motivation for, and an integral part of, the mathematical approach to specification is argued convincingly by [SPI89] [WO089].

The Chapter 4 analysis of protocols focussed on the following properties: absence of unspecified stimuli, liveness, absence of race hazards, and observation of bandwidth limit. Consider now how these may be expressed in Z terms.

The Z approach is fundamentally concerned with set memberships and functional relationships. The absence of unspecified stimuli is a static property which can be stated in terms of unchanging subsets of states and stimuli. This is essentially the approach of [BRAND83].

However [BRAND83] and the detailed analysis of Chapter 4 both illustrate the considerable practical difficulty of defining these subsets: one must simulate the operation of the protocol by enumerating all legal pairs (more generally N-tuples, where the protocol contains N partners) of states of the protocol.

1. one compares the output signals which partner 1 can legally produce in its current state with all input signals which partner 2 can legally accept in all states which are 1-reachable from the current state of partner 2

2. one then reverses the roles of partners 1 and 2 and repeats the above comparison

These two comparisons are recursively repeated for all state pairs where one state is 1-reachable from the current state of partner 1 and the other state is 1-reachable from the current state of partner 2. This recursive enumeration is essentially the procedure described in [BRAND83] and illustrated by the global state analysis of Chapter 4.

Similar arguments illustrate the need to enumerate sets in order to state the other protocol properties of interest:

1. liveness (i.e. the absence of deadlock) can in principle be stated in terms of the statically available information on unspecified inputs, but again the need arises to enumerate all
legal state N-tuples to examine the order of occurrence of stimuli

2 the concepts of a bandwidth limit and relative timing (i.e. to trigger race hazards) introduce a queue discipline into the master/slave communication. Both the bandwidth limit and the search for race hazards require the queued stimuli to be considered as an ordered set, all permutations of which must be generated. This is necessary to produce all possible signal juxtapositions in the message queue (of whatever discipline) and thus all possible relative timings of input and output signals.

This enumerative approach is significantly more complex than the basic specification of the protocol, but without it the proof techniques of [WOO89] concerning set membership cannot be applied to protocol properties.

One can state generally that the enumerative approach will be necessary in any specification whose proof depends on distinguishing set members, and unnecessary in any specification which can be proved in terms of indistinguishable set members. For example, the operation of a protocol depends crucially on which states are the legal successors of a given state i.e. the specific membership of the successor set of the given state. It is not sufficient to argue that the successor set is non-empty. But in contrast [WOO89] can prove the specification of his store manager application largely by considering the cardinality of sets of indistinguishable elements i.e. storage blocks.

5.4.1.1 - Separating statement from proof - temporal logic

The above section argues that standard [SPI89] Z treatment of protocol properties requires a property to be stated by enumerating all circumstances which exemplify that property. This approach confuses two issues: the statement of the property, and the demonstration that the property is satisfied. In the case of a protocol, we now argue that one can separate these issues by modifying the Z notation, but in a way which is mathematically equivalent to strict Z.
The standard Z notation makes use of primed variables to describe changing values: \( V \) is the value of a variable and \( V' \) is the value of the same variable one instant later. The difficulty of this scheme is that the significant dynamic properties of protocols involve events which are separated by indefinite time intervals (or more precisely, indefinitely long sequences of events). For example, the correctness of the 2-phase commit protocol requires that all partners arrive at the same conclusion eventually, but there is no obvious indication of how many signal exchanges are needed to achieve that condition. Similarly, the liveness requirement demands that certain events will never occur.

It is possible to make some progress in the standard Z primed notation by introducing explicit counters of events. However, for all but very simple dynamic conditions, the use of such counters rapidly becomes more complex than the application which is the main interest of the specification effort: the counter initialisation conditions can be troublesome, and it may become necessary to keep track of combinations of counters.

But further consideration reveals that it is not necessary to record the progression of time explicitly. Rather, it is required to state that events will occur "sometime in the future". That is, the events will be separated by some interval which is unknown but finite. These concepts are very well modelled in the notation known as temporal logic.

5.4.1.2 - A brief summary of temporal logic

Temporal logic discusses changes due to the passage of time by treating a process as a sequence of execution states and examining the transformation of one state into another.

Temporal logic is an example of modal logic, which is formally identical to first-order predicate calculus. [MAN81] contains a detailed tutorial and a comprehensive list of useful theorems.

Consider a classical scheme of constants, variables, quantifiers, propositions, and predicates. Consider also a formula \( F(Q,R) \) of at least
two arguments, where one of the arguments Q takes discrete values. One can then write the formula F with one explicit argument R and one implicit argument Q. The distinct values of Q are known as modes. In each such mode the rewritten formula F'(R) has an unambiguous first-order interpretation i.e. an assignment of values to its arguments such that the truth value of F'(R) can be evaluated. The interpretation of F'(R) may differ in different modes. In effect, each mode of the formula is a domain. All domains have identical structure but different contents.

A modal formula is constructed from the classical symbols and operators above, and from the modal operators invariance (□), eventuality (♦), precedence (U), and sequence (●). The universe of a modal formula F consists of a (finite or infinite) set of modes, an accessibility relation stating which mode can be reached from which other mode, and a distinguished (or initial) mode. The above modal operators effect the changes of mode, but always within the constraints of the accessibility relation.

In a temporal logic scheme the modes are informally interpreted as time instants. The accessibility relation must take the form of a linear monotonic-increasing sequence. This is a reasonable model of time for most purposes: time is single-valued, and it neither stands still nor reverses.

For the purposes of stating the dynamic properties of protocols, the temporal invariance operator □ conveniently states the safety properties ("the undesirable event X will never happen"), while the eventuality operator ♦ conveniently states liveness ("the desirable property X will eventually happen").

5.4.1.3 - Relation of modal logic to protocol specification

At first sight, modal logic, and temporal logic in particular, appears to be no more than an elaborate notation. But note that the modal operators can formulate statements about events which are separated by indefinite time intervals. In contrast, the standard Z primed notation is restricted to
statements about adjacent time instants. In formal terms, the Z primed notation is limited to the sequence operator. This ability of temporal logic to make statements about indefinitely separated events is exactly what is needed when stating protocol properties.

5.4.1.4 - A general procedure

This suggests a general procedure for incorporating protocol properties into the Z specification of the protocol. The property is formulated using whatever (dynamic) notation is convenient. Then a check is made that the property is invariant in the general sense that no protocol action may violate it. If this holds, then the property is conjoined with the static invariant to form one term in the overall protocol invariant.

5.4.1.5 - Correctness of the 2-phase commit protocol

The correctness of the protocol is stated in terms of the expected state transitions of the master and slave partners. The object here is to list the sequence of state pairs which indicate correct progression of the negotiation, and then to assert that this sequence occurs.

Correct operation of the Fig.4-1. protocol is expressed as a sequence of state pairs. The three correct modes of operation are the following three sequences of state-pairs. (The prefixes "m-" and "s-" indicate master and slave states respectively.)

<table>
<thead>
<tr>
<th></th>
<th>(m-idle,s-idle), (m-waiting,s-waiting), (m-commit,s-commit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(m-idle,s-idle), (m-waiting,s-waiting), (m-abort,s-abort)</td>
</tr>
<tr>
<td>2</td>
<td>(m-idle,s-idle), (m-waiting,s-abort), (m-abort,s-abort)</td>
</tr>
</tbody>
</table>

Observe that these sequences all result in the master and slave arriving at the same decision. This correctness criterion is therefore consistent
with that of Chapter 4. The correctness condition is formulated as the conjunction of the following temporal predicates

1. the master and slave can always be initialised to the *idle* state. (More formally, for all instants reachable from the current instant, the expression ... will eventually become true.)

\[ \Box \Diamond (m\text{-}idle \text{ and } s\text{-}idle) \]

2. after initialisation, one of the following state pairs always results. (More formally, from the initialisation condition either ... or ... eventually becomes true.)

\[ (m\text{-}idle \text{ and } s\text{-}idle) \Rightarrow \]
\[ \Diamond (m\text{-}waiting \text{ and } s\text{-}waiting) \text{ or } \]
\[ \Diamond (m\text{-}waiting \text{ and } s\text{-}abort) \]

3. from the above intermediate stages, one of 3 termination conditions always results. Be careful to attach the right termination to the right intermediate.

\[ (m\text{-}waiting \text{ and } s\text{-}waiting) \Rightarrow \]
\[ \Diamond (m\text{-}commit \text{ and } s\text{-}commit) \text{ or } \]
\[ \Diamond (m\text{-}abort \text{ and } s\text{-}abort) \]

\[ (m\text{-}waiting \text{ and } s\text{-}abort) \Rightarrow \Diamond (m\text{-}abort \text{ and } s\text{-}abort) \]

Reinitialisation is guaranteed by Eq-5.6, which states that the initialisation condition becomes true infinitely often (i.e. the protocol cycles infinitely).

At this point one observes that the above correctness statement says nothing about how the protocol behaves during state-pairs other than those mentioned above. In fact, one can rely on the memoryless nature of a protocol state. Following [MIN72], a state is an equivalence class of histories, none of which is separately distinguishable. It is sufficient to say that the above pairs are indeed achieved: the intermediate sequences by which they are achieved are of no interest.
One can reasonably argue that the above correctness criterion is properly part of a more complete static invariant, despite its formulation in terms of state changes. Although Z insists on separating the static and dynamic concerns, the correctness of operation is surely a static invariant: it would certainly be unacceptable to introduce an action which violates the above state-pair sequences. The criterion can therefore be conjoined to other invariant terms, as described in the above section setting out as general procedure.

5.4.1.6 - Liveness of the 2-phase commit protocol

It is quite straightforward to state in temporal terms the condition which guarantees freedom from deadlock. Observe that both the master and the slave must start in state idle and both must progress either to commit or to abort.

Eq-5.9 \( (m\text{-idle and } s\text{-idle}) \Rightarrow \sum ((m\text{-commit and } s\text{-commit}) \text{ or } (m\text{-abort and } s\text{-abort})) \)

By [MAN81] the eventuality operator can be distributed over or to give the intuitive result that eventually either (1) both master and slave commit, or (2) both master and slave abort.

Eq-5.10 \( (m\text{-idle and } s\text{-idle}) \Rightarrow \sum (m\text{-commit and } s\text{-commit}) \text{ or } \sum (m\text{-abort and } s\text{-abort}) \)

5.4.1.7 - Other properties of the 2-phase commit protocol

Protocol properties other than correctness and liveness were of interest during the finite-state analysis of Chapter 4; absence of unspecified stimuli, absence of race hazards, observation of bandwidth limit. It is now argued that these statements of behaviour are unnecessary in a Z protocol specification.

The strictly finite-state specifications of Chapter 4 are essentially operational in nature. (This usage is consistent with [ZAV82], who defines operational in terms of an idealised computation whose level of
detail stops well short of a fully executable program.) They state not what is required to be achieved, but instead they prescribe the process by which the desirable objective will be achieved. In this sense, the finite-state specifications state the protocol requirements implicitly rather than explicitly.

The above three properties are actually operational necessities to enable the operational specification to express the concepts of correctness and liveness. But if correctness and liveness can be expressed more directly, then the unspecified stimuli, the race hazards and the bandwidth limits are of no interest. This is precisely what is achieved by the Z specification of the dynamic properties using temporal logic.

However, although earlier sections adequately express the correctness and liveness of the terminating 2-partner protocol of Fig.4.1, it should not be thought that this is a general scheme which is adequate in all circumstances. [MAN81] discusses a framework of concurrent processes which is more general than the application class which is exemplified by the protocol of Fig.4.1. He finds useful definitions for concepts such as fairness (which relates to competing processes), intermittently true assertions, and several variants of liveness (responsiveness, partial correctness) which relate to terminating and cycling programs.

5.4.2 - Z and some fundamental finite-state problems

It is instructive to reconsider a fundamental limitation of the strictly finite-state specification which was pointed out in Chapter 4. A finite-state automaton such as that of Fig.4.1 cannot memorise an arbitrary number of events. Chapter 4 points out that this has practical consequences in a fault-tolerant network where a decision is made after a majority vote of the surviving protocol partners. Consider how a Z specification might approach this problem.

Observe that the memory limitation inherent in a finite-state automaton is completely absent from the predicate calculus which underlies Z.
Although this chapter has noted the fundamental role of the concept of state in many processes and has modeled state using substitutions, there is nothing to prevent the Z formulation of state from having extended memory (i.e. history). This is very similar to the approach of [SUN82] who uses state-like objects which are in fact stacks, queues, etc..

In terms of predicate calculus, a state is merely a variable which records some context. There is no formal reason why a Z specification cannot contain non-state (in the sense of [MIN72]) context variables and define functions over them.

The remainder of this section illustrates how the 2-phase commit protocol may be extended in Z to accommodate more than one slave, the decision to commit or abort depending on a majority vote. (But note that this illustration is incomplete. This extension forces other extensions on the protocol e.g. to allow a dissenting partner to somehow reverse its vote and adopt the majority action.)
5.4.2.1 - Z specification of 1-master-many-slaves

The master is defined as before, but the slave is now defined as one of a set of slaves, where there is at least one slave.

The master now counts the replies from the slaves and records the running total of yes and no replies using two predicates majorityyes and majorityno over 3 variables: the number of slaves (numberslaves), the current number of yes votes (voteyes), and the current number of no votes (voteno).

\[
\text{Eq-5.11} \quad \text{majorityyes} = (\text{voteyes} > \text{numberslaves}/2) \\
\text{Eq-5.12} \quad \text{majorityno} = (\text{voteno} > \text{numberslaves}/2)
\]

We assume here that the maximum number of slaves is a known constant. A more elaborate protocol could accommodate a varying
number of slaves (e.g. in a self-repairing network) but this major protocol complication would add little to this Z illustration.

The slave substitutions of section 5.3.4 are unchanged. The master transition 1 is unchanged. Master transitions 2, 3 and 4 are replaced with 2' and 3' (which record the incoming yes and no replies from the slaves) and with 4', 5' and 6' (which use the truth values of the majorityyes and majorityno predicates to trigger a commit or abort conclusion).

<table>
<thead>
<tr>
<th>Master transition 2'</th>
<th>Master transition 3'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔMaster</td>
<td>ΔMaster</td>
</tr>
<tr>
<td>i? : input-m</td>
<td>i? : input-m</td>
</tr>
<tr>
<td>ms = waiting</td>
<td>ms = waiting</td>
</tr>
<tr>
<td>i = no</td>
<td>i = yes</td>
</tr>
<tr>
<td>ms',voteno' := waiting,voteno+1</td>
<td>ms',voteyes' := waiting,voteyes+1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Master transition 4'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔMaster</td>
</tr>
<tr>
<td>ms = waiting</td>
</tr>
<tr>
<td>majorityno</td>
</tr>
<tr>
<td>ms',mo! := abort,undefined</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Master transition 5'</th>
<th>Master transition 6'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔMaster</td>
<td>ΔMaster</td>
</tr>
<tr>
<td>ms = waiting</td>
<td>ms = waiting</td>
</tr>
<tr>
<td>majorityyes</td>
<td>majorityyes</td>
</tr>
<tr>
<td>AM2</td>
<td>AM3</td>
</tr>
<tr>
<td>ms',mo! := commit,commit</td>
<td>ms',mo! := abort,abort</td>
</tr>
</tbody>
</table>

5.4.2.2 - Correctness and liveness

Using the results of the previous section concerning temporal logic, the correctness invariant is generalised to take account of more than one slave. For example, the initialisation requirement becomes
Eq-5.13  $\mathbb{P} \Box (m\text{-idle and } \Pi s\text{-idle and voteyes}=0 \text{ and voteno}=0)$

to conjoin the initialisation condition over all slaves. A similar conjoining is carried out everywhere to convert all single-slave states into a conjunction of single-slave states.

The liveness invariant is extended by a term to require that a majority of votes will eventually be received by the master

Eq-5.14 $m\text{-waiting }\supset$

$\Diamond (\text{voteyes} > \text{numberslaves}/2) \text{ or } (\text{voteyes} > \text{numberslaves}/2)$

which is more succinctly expressed as

Eq-5.15 $m\text{-waiting }\supset \Diamond (\text{majorityyes or majorityno})$

5.5 - A clean example: the user/server protocol in Z

Earlier sections have developed ideas on how the concepts of the Z notation and temporal logic can be applied to the specification of a class of communication protocols. These ideas are again illustrated by a complete example: the user/server protocol of Fig.4.4.

5.5.1 - Basic sets, variables, static invariant

The basic sets of Z objects are:

1. the legal states of the user
2. the legal states of the server
3. the legal input signals to the user
4. the legal input signals to the server
5. the legal output signals from the user
6. the legal output signals to the server
7. a set of environment conditions

The user/server protocol is specified in terms of two communicating components- a user and a server. There is exactly one user and one
server. With each is associated a set of states, a set of input signals, and a set of output signals. The static invariant for this protocol application (in part) requires that no illegal inputs, outputs, or states may appear during the operation of the protocol. This amounts to the restatement of set membership. The definition of the protocol actions requires three variables to denote the instantaneous value of the user's state input and output (us, ui,uo), and three similar variables for the server (ss, si, so).

The application circumstances which enable the protocol partners to make unambiguous decisions have already been listed in chapter 4. They are:

<table>
<thead>
<tr>
<th>AU1</th>
<th>user wants service</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU2</td>
<td>user acknowledges a remote fault</td>
</tr>
<tr>
<td>AS1</td>
<td>server completes the service request</td>
</tr>
<tr>
<td>AS2</td>
<td>server develops fault</td>
</tr>
</tbody>
</table>

These application conditions are introduced as another basic set and are used as the guarding predicates of pre-conditioned substitutions.

<table>
<thead>
<tr>
<th>User</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>us : state-u</td>
<td>ss : state-s</td>
</tr>
<tr>
<td>ui : input-u</td>
<td>si : input-s</td>
</tr>
<tr>
<td>uo : output-u</td>
<td>so : output-s</td>
</tr>
<tr>
<td>ue : environment-u</td>
<td>se : environment-s</td>
</tr>
</tbody>
</table>

us ∈ {wait,ready,register}  
ui ∈ {alarm,done}  
uo ∈ {req,ack}  
ue ∈ {AU1,AU2}  

| ss ∈ {service,idle,fault} | si ∈ {req,ack} |
| so ∈ {alarm,done} | se ∈ {AS1,AS2} |

5.5.2 - Dynamics - operation of user/server protocol

The correct dynamic behaviour of the user/server protocol is expressed by the following pre-conditioned substitutions. The Z basic sets, the variables and the static invariant are specified in earlier sections. The
initial state and dynamic actions of the user and server are specified by the following schemas.

<table>
<thead>
<tr>
<th>Initialise user</th>
<th>Initialise server</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Server</td>
</tr>
<tr>
<td>( us = \text{ready} )</td>
<td>( ss = \text{idle} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User transition 1</th>
<th>Server transition 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \text{User} )</td>
<td>( \Delta \text{Server} )</td>
</tr>
<tr>
<td>( us = \text{ready} )</td>
<td>( ss = \text{service} )</td>
</tr>
<tr>
<td>( ue = \text{AU1} )</td>
<td>( se = \text{AS1} )</td>
</tr>
<tr>
<td>( us',uo! := \text{wait,req} )</td>
<td>( ss',so! := \text{idle,done} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User transition 2</th>
<th>Server transition 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \text{User} )</td>
<td>( \Delta \text{Server} )</td>
</tr>
<tr>
<td>( i? : \text{input-u} )</td>
<td>( i? : \text{input-s} )</td>
</tr>
<tr>
<td>( us = \text{wait} )</td>
<td>( ss = \text{idle} )</td>
</tr>
<tr>
<td>( i = \text{alarm} )</td>
<td>( i = \text{req} )</td>
</tr>
<tr>
<td>( us',uo! := \text{wait,undefined} )</td>
<td>( ss',so! := \text{service,undefined} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User transition 3</th>
<th>Server transition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \text{User} )</td>
<td>( \Delta \text{Server} )</td>
</tr>
<tr>
<td>( i? : \text{input-u} )</td>
<td>( ss = \text{idle} )</td>
</tr>
<tr>
<td>( us = \text{wait} )</td>
<td>( se = \text{AS2} )</td>
</tr>
<tr>
<td>( i = \text{done} )</td>
<td>( ss',so! := \text{fault,alarm} )</td>
</tr>
<tr>
<td>( us',uo! := \text{ready,undefined} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User transition 4</th>
<th>Server transition 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \text{User} )</td>
<td>( \Delta \text{Server} )</td>
</tr>
<tr>
<td>( i? : \text{input-u} )</td>
<td>( i? : \text{input-s} )</td>
</tr>
<tr>
<td>( us = \text{ready} )</td>
<td>( ss = \text{fault} )</td>
</tr>
<tr>
<td>( i = \text{alarm} )</td>
<td>( i = \text{req} )</td>
</tr>
<tr>
<td>( us',uo! := \text{register,undefined} )</td>
<td>( ss',so! := \text{fault,undefined} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User transition 5</th>
<th>Server transition 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta \text{User} )</td>
<td>( \Delta \text{Server} )</td>
</tr>
<tr>
<td>( us = \text{register} )</td>
<td>( i? : \text{input-s} )</td>
</tr>
<tr>
<td>( ue = \text{AU2} )</td>
<td>( ss = \text{fault} )</td>
</tr>
<tr>
<td>( us',uo! := \text{ready,ack} )</td>
<td>( i = \text{ack} )</td>
</tr>
<tr>
<td></td>
<td>( ss',so! := \text{idle,undefined} )</td>
</tr>
</tbody>
</table>
5.5.3 - Correctness of the user/server protocol

Correct operation of the Fig. 4.4 protocol is expressed as a sequence of state pairs. The two correct modes of operation are the following sequences. (The prefixes "u-" and "s-" indicate user and server states respectively.)

<table>
<thead>
<tr>
<th></th>
<th>(u-ready, s-idle), (u-wait, s-service), (u-ready, s-idle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(u-ready, s-idle), (u-register, s-fault), (u-ready, s-idle)</td>
</tr>
</tbody>
</table>

The correctness condition is formulated as the conjunction of the following two temporal predicates

1. the master and slave can always achieve the initial state

   Eq-5.16 ■♦(u-ready and s-idle)

2. there are two circumstances of interest. The first is a correctly handled transaction request. The second is a correctly acknowledged fault report. One of these two must result from the initialisation condition.

   Eq-5.17 (u-ready and s-idle) ⊨
   ♦(u-wait and s-service) or ♦(u-register and s-fault)

From either of these two circumstances, the initialisation condition must be recovered. This requirement is already guaranteed by the first of the above predicates: the initial condition can always be achieved.
This is the complete statement of the correctness of the user/server protocol.

5.5.4 - Liveness of the user/server protocol

The freedom from deadlock (or more generally, the liveness of the protocol) is a statement to the effect that the user and server will succeed in cycling endlessly i.e. no delay will last forever.

The state achieved immediately after initialisation is denoted by
From this state we require the initial state to be recovered eventually:

\[
(u\text{-ready and } s\text{-service}) \Rightarrow \Diamond\Diamond(u\text{-ready and } s\text{-service})
\]
An important feature of the Z technique is the facility to construct a specification as a collection of distinct modules. Z facilitates an effective separation of distinct issues, and permits common concerns to be specified once only and used many times. This modularity is expressed in the schema calculus, a notational device which is not (strictly speaking) fundamental to the underlying set-theoretic concepts of Z, but which offers significant practical benefits.

In drawing conclusions concerning any protocol specification technique, it is important to be aware of the fundamental value of the concept of state in describing processes which are designed around stable stages or checkpoints. The finite-state specifications of Chapter 4 do not distinguish between the concepts of the specification technique and the concepts which define the protocol process. But a Z specification is not fundamentally concerned with states (in the sense of [MIN72]). This distinction between specification technique and protocol process is largely obscured by the relative simplicity of the protocols considered and a large part of the Z specification is devoted to modeling the protocol state transitions. However, the distinction becomes explicit when one considers some non-state extensions of the basic protocol.

Compared to the finite-state specifications of Chapter 4, the use of predicate calculus to express correctness and liveness is very powerful and flexible. The inherent memory limitations of a strict finite-state approach are avoided. Further, a predicate description is not limited to this operational approach, which requires much execution detail (e.g. the imposition of a bandwidth limit to permit a finite tree to be unfolded from a potentially infinite graph).

The use of Z gives access to a self-consistent body of theorems (in set theory and predicate calculus) by which the operation of a process can be stated with adequate formality. By adequate is meant that the user of the specification sees enough detail to convince him of its correctness. In the small examples considered here, the adequately rigorous specification does not contain that degree of detail which would cause a
reader to lose track and therefore demand a formal proof. But it seems likely that a large application consisting of many major Z-specified modules would require a fairly detailed proof of self-consistency. The protocol applications considered here do not shed much light on this aspect of Z.

However, the standard Z notation for describing time-dependent changes appears to be weak, and a full temporal logic scheme appears to be a practical alternative. Although temporal logic is formally identical to first-order predicate calculus, the notation for making statements about events which are separated by indefinitely long intervals is invaluable. The invariance and eventuality modal operators are ideally suited to expressing the correctness and liveness properties of protocols. In effect, the standard Z notation is confined to the sequence modal operator and can easily discuss only adjacent instants. This is inadequate for stating liveness properties ("something will eventually happen") and is impractically clumsy for stating invariance properties ("something will remain true forever").

This chapter has concentrated on stating the required properties of specific protocols but has made no attempt to consider the practicality of implementing those specifications. However, infeasibility in the Z sense (i.e. a statement of requirements which cannot possibly be implemented) can be avoided by prudent use of the Z indeterminacy mechanisms. The feasibility of the specification in a more general sense must be separately considered under the heading of the refinement [MOR85] of an otherwise acceptable Z specification. This consideration is explicitly excluded from this chapter on Z specifications.
This chapter reconsiders the two communication protocols which have already been described in detail: the 2-phase commit protocol, and the user-server protocol. These two are now specified using the facilities of CSP (Communicating Sequential Processes) which is defined in [HOA85].

This chapter is structured into the following principal sections:

1. a brief introduction to CSP: process concepts, interaction mechanisms, and sources of indeterminacy
2. a CSP specification of the 2-phase commit protocol, and a critical examination of the CSP facilities
3. an examination of the facilities for stating protocol properties
4. a clean example: a CSP specification of the user/server protocol illustrating issues from the previous sections
5. conclusions on the use of CSP to specify protocols

6.1 - Introduction to CSP

CSP is probably the best known example of the process algebra approach to specification. In this approach, a specification is built up from a number of abstractly defined interacting agents, or processes. The interaction of processes is governed by a set of rigorous laws - an algebra - which renders the specification amenable to rigorous mathematical analysis.

An abstract framework for a family of process algebras (including CSP) is provided by [MIL80] in CCS (Communicating Sequential Systems) which sets out a rigorous mathematical account of the general theory of concurrency and synchronisation. CCS is more general than CSP in the sense that CCS makes many distinctions between process behaviours which CSP treats as identical. According to [HOA85], the narrower approach of CSP allows it to arrive at a set of process laws which are
richer (i.e. their consequences can be investigated to a greater depth) and which permit more transformations and optimisations.

CSP is particularly significant among process algebras because it forms the basis of an internationally accepted standard specification language LOTOS [BRI87] which is especially applicable to telecommunications and distributed systems.

6.1.1 - Basic concepts of CSP

CSP is concerned with the rigorous description of a set of interacting agents - or processes - and the imposition of laws - an algebra - to regulate their interactions. The ensemble of CSP processes constitutes a specification of some application. This specification has mathematical structure which is defined by the CSP process laws [HOA85] and which is therefore open to formal analysis.

A CSP process $P$ is characterised by the set of events in which it may engage. This set is known as the alphabet of $P$ and is denoted

\[ \alpha P \]

For example, if $P$'s possible events are one, two and three, then

\[ \alpha P = \{one, two, three\} \]

It is by definition impossible for a process to engage in any event not in its alphabet.

A CSP event is instantaneous and atomic. Any occurrence of duration must be modelled as a start/end pair of events. Such extended events are permitted to overlap.

The concept of absolute time is absent from CSP. The ordering of events which are engaged in by two or more processes is always relative to some reference event, and the question of simultaneity is excluded by definition. When two or more processes are required to synchronise, they achieve this by engaging in a single event which is common to all.
Questions of causality are also excluded by refusing to distinguish between those events which are initiated by $P$, and those events which are initiated by another process and to which $P$ merely reacts. CSP events are always ordered, but the order never implies cause or effect. (It will be seen later that this lack of distinction is of great benefit in modelling the interactions between a system and its operating environment. Briefly, the ensemble of system and environment is simply another collection of interacting processes.)

The basic device for describing process behaviour is the prefix operator $\rightarrow$. Consider the process $P$ where

$$\text{Eq-6.3} \quad \alpha P = \{\text{one, two, three}\}$$
$$\text{Eq-6.4} \quad P = (\text{one} \rightarrow (\text{two} \rightarrow (\text{three} \rightarrow Q)))$$

This denotes a process which engages in a sequence of events one, two, three and then behaves exactly like a process $Q$. The prefix operation is technically known as guarding and it supplies certain conditions which guarantee the uniqueness of the equations which the process specifications represent.

Recursion is used to describe repetitive behaviour:

$$\text{Eq-6.5} \quad \alpha \text{CYCLE} = \{\text{do\_it\_again}\}$$
$$\text{Eq-6.6} \quad \text{CYCLE} = (\text{do\_it\_again} \rightarrow \text{CYCLE})$$

Two standard processes are available: STOP and SKIP. STOP characterises a process which cannot proceed because it refuses to engage in any event of its alphabet. STOP effectively halts any process in whose specification it appears. SKIP is another process which precludes further activity but unlike STOP, SKIP achieves a satisfactory process termination by engaging (once only) in a standard event known as success. STOP is useful in describing a deadlock of two or more concurrent processes, whereas SKIP indicates the successful termination of a sequential process.
A process achieves variation of behaviour by making use of the choice mechanisms of CSP. The basic choice mechanism is the $I$ operator, the deterministic choice.

$$\alpha = \alpha Q = \alpha PQ = \{\text{one two}\}$$

$$PQ = (\text{one} \rightarrow P | \text{two} \rightarrow Q)$$

This denotes a process $PQ$ whose behaviour depends on which event it engages in: if one, then the subsequent behaviour is that of $P$; if two, then $Q$. A more general notation permits an N-way choice of behaviour. Given an event $e$ which is a member of a set $E$, and a family of processes where each process $P_e$ is selected by some member of $E$, then the N-way choice is denoted by

$$e: E \rightarrow P_e$$

To say that a choice is deterministic does not mean that the result can always be predicted in advance. For example, in the process $PQ$ above, the behaviour is fixed by whichever of the events one or two occurs first.

Two more complicated choice operators are $I$ (general choice) and $\square$ (nondeterministic choice). General choice is related to both the nondeterministic and the deterministic choices as follows:

$$c \rightarrow P I d \rightarrow Q =$$

$$\begin{align*}
(c \rightarrow P | d \rightarrow Q) & \quad \text{if } c \neq d \\
(c \rightarrow P \square d \rightarrow Q) & \quad \text{if } c = d
\end{align*}$$

To state this distinction informally, a choice is deterministic if the initial event (i.e. $c$ or $d$) uniquely determines the subsequent behaviour (i.e. $P$ or $Q$), and nondeterministic otherwise. A formal and more precise
distinction requires the extra concepts of process *environment* and *deadlock*. These are introduced in a later section.

Note carefully that [HOA85] always uses the nondeterministic operator ■ to indicate implementation freedom rather than arbitrary process behaviour. In this respect, CSP takes the same view as Z: nondeterminism is not the same as ignorance. In a system containing the process P■Q, any system property of interest must be established for both P and Q independently. In particular, if either P or Q is liable to deadlock, then so also is P■Q. In this respect both Z and CSP take the consistent view that a fundamental property of a specification is that it should hold for every possible implementation.

Behaviour which is inherently nondeterministic (i.e. not merely unresolved implementation details) results from the *interleaving* || of two or more processes. Interleaving is related to *parallel composition* ||, described later. The interleave P|||Q of processes P and Q describes their concurrent but unsynchronised behaviour, and the action is that of exactly one of the processes, chosen indeterminately. Consider the following P and Q:

Eq-6.13 \[ \alpha P = \alpha Q = \{one,two,three\} \]
Eq-6.14 \[ P = (one \rightarrow P | three \rightarrow P) \]
Eq-6.15 \[ Q = (two \rightarrow Q | three \rightarrow Q) \]

If P|||Q engages in event *one* then the action can only be that of P, and of Q if the event is *two*. But where both processes are capable of engaging in event *three*, the action of P|||Q is that of either P or Q, chosen indeterminately.

A formal definition of deterministic behaviour needs some other concepts of process interaction. These concepts are set out in the following sections.
6.1.3 - Process traces

The trace of a process is a finite sequence of symbols which denote the events engaged in by the process up to some moment. A trace is necessarily sequential, and no two events are ever engaged in simultaneously by the one process. (In some applications it may be necessary to introduce a queueing discipline into the CSP specification to satisfy this condition.)

Given a process P

\[ \alpha P = \{ \text{one, two, three} \} \]
\[ P = (\text{one} \rightarrow (\text{two} \rightarrow (\text{three} \rightarrow P))) \]

then the following event sequences are valid traces of P:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;one,two&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;one,two,three&gt;</td>
</tr>
<tr>
<td>4</td>
<td>&lt;one,two,three,one&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;one,two,three,one,two...&gt;</td>
</tr>
</tbody>
</table>

Note that the empty trace <> is a valid trace of every process which has not yet engaged in an event, and is the only trace of the standard process STOP.

A useful concept is that of the set of all possible finite traces of a process. This set is denoted by \((\alpha P)^*\) and every valid trace of P is a member. Note that \((\alpha P)^*\) is not simply the set of all possible sequences which can be generated from \(\alpha P\), because not all such sequences are valid traces. For example, the above process P has no trace of the form <...one,three,...> and so no member of \((\alpha P)^*\) contains this subsequence.

A CSP trace is necessarily finite (although a recursively defined process may generate an arbitrarily long trace).
CSP processes interact in several well-defined ways: mutual recursion, parallel composition, interleaving, message passing and sequential composition. Interleaving has been described in an earlier section.

6.1.4.1 - Mutual recursion

The simplest form of interaction uses mutual recursion to couple the operation of two or more processes. This is illustrated by two processes ON, OFF, which model the operation of a toggle switch.

\[ \alpha \text{ON} = \alpha \text{OFF} = \{\text{up, down, switch-on, switch-off}\} \]
\[ \text{ON} = (\text{down} \rightarrow (\text{switch-on} \rightarrow \text{OFF})) \]
\[ \text{OFF} = (\text{up} \rightarrow (\text{switch-off} \rightarrow \text{ON})) \]

6.1.4.2 - Parallel composition

A more versatile form of process interaction is the operation of parallel composition \( \parallel \). This operation explicitly models the synchronisation of two or more concurrently executing processes. The parallel composition of two or more process results in a single process whose behaviour consists of the behaviour common to the component processes. The alphabet of the composite process is the union (N.B. not the intersection) of the alphabets of the components. Where the component processes have no common behaviour, the composite process deadlocks.

A deadlock occurs when the interacting processes cannot engage in any further events. Their traces therefore cannot be extended in any way, and no further execution is possible. Consider the processes P, Q:

\[ \alpha \text{P} = \alpha \text{Q} = \{\text{one, two, three}\} \]
\[ \text{P} = (\text{one} \rightarrow (\text{two} \rightarrow \text{P})) \]
\[ \text{Q} = (\text{one} \rightarrow (\text{three} \rightarrow \text{Q})) \]
When \( P \) and \( Q \) are parallel composed to run concurrently, they first synchronise by engaging in the common event \( one \), but they then fail to find any further behaviour in common. Therefore

\[
\text{Eq-6.24} \quad P \parallel Q = (one \rightarrow \text{STOP})
\]

Proof of absence of deadlock usually consists in showing that an arbitrary trace can be extended indefinitely i.e. the process can continue to engage in further events in all circumstances.

Where two processes \( P, Q \) have identical alphabets, their parallel composition denotes a process whose behaviour is identical to that of \( P \) and \( Q \) executing concurrently and synchronised in lock-step. Each event occurs in the independent behaviour of each component. An event in \( P \parallel Q \) requires the simultaneous participation of both \( P \) and \( Q \). The behaviour of \( P \) and is therefore synchronised. For example:

\[
\begin{align*}
\text{Eq-6.25} & \quad \alpha \text{BANDIT} = \alpha \text{WISE} = \alpha \text{FOOL} = \\
& \{\text{coin,win,lose}\} \\
\text{Eq-6.26} & \quad \text{BANDIT} = (\text{coin} \rightarrow (\text{win} \rightarrow \text{BANDIT} \\
& \quad \mid \text{lose} \rightarrow \text{BANDIT})) \\
\text{Eq-6.27} & \quad \text{WISE} = (\text{coin} \rightarrow (\text{win} \rightarrow \text{WISE} \\
& \quad \mid \text{lose} \rightarrow \text{SKIP})) \\
\text{Eq-6.28} & \quad \text{FOOL} = (\text{coin} \rightarrow (\text{win} \rightarrow \text{FOOL} \\
& \quad \mid \text{lose} \rightarrow \text{FOOL})) \\
\text{Eq-6.29} & \quad \text{BANDIT} \parallel \text{WISE} = (\text{coin} \rightarrow (\text{win} \rightarrow \text{BANDIT} \parallel \text{WISE} \\
& \quad \mid \text{lose} \rightarrow \text{STOP})) \\
\text{Eq-6.30} & \quad \text{BANDIT} \parallel \text{FOOL} = (\text{coin} \rightarrow (\text{win} \rightarrow \text{BANDIT} \parallel \text{FOOL} \\
& \quad \mid \text{lose} \rightarrow \text{BANDIT} \parallel \text{FOOL}))
\end{align*}
\]

\( \text{BANDIT} \) and \( \text{FOOL} \) will continue indefinitely in lockstep, but \( \text{BANDIT} \) and \( \text{WISE} \) will deadlock if the event \( \text{lose} \) ever occurs. But note that this does not guarantee that the deadlock will ever actually occur. (This point is the crux of an important and to some extent philosophical argument \[\text{DIJ88}\] \[\text{SCH88}\] concerning the limitations of a finite description of behaviour. See later.)
If the parallel-composed processes P and Q have disjoint alphabets then participation in common events is impossible by definition and there can therefore be no synchronisation. In this case the composition P||Q is characterised by a trace which is an arbitrary interleaving of the trace of P and Q.

6.1.4.3 - Message passing

Another type of process interaction is by means of a special event which represents the explicit passing of a message between the processes. Note that such a message event is fundamentally identical to any other CSP event, but a more convenient notation is introduced to more easily express the semantics of message-passing applications. As with all CSP events, the passing of a message event is instantaneous and atomic.

A message event is a structured value, written c.m, meaning that the message m is transferred on channel c. A channel is a 1-directional link between two processes. The transfer of a message m on channel c from process P to Q is denoted by

\[ (c!m \rightarrow P)|| (c?m \rightarrow Q) \]

The alphabet of the channel c (i.e. the set of messages communicable on it) must be identical in both the sending and receiving processes if the possibility of deadlock is to be excluded. That is, all messages from P are acceptable to Q

\[ \alpha c(P) = \alpha c(Q) \]

A process Q may be prepared initially to accept any one of the legal messages of a given channel, the particular message being chosen by the sending process:

\[ (c!x \rightarrow P)|| (c?x \rightarrow Q) \text{ and } x \in \alpha c(P) \]

A process may be prepared initially to communicate on any one of several distinct channels:
But note carefully that CSP does not require this choice to be resolved by choosing (for example) the first input message to become available. If some explicit queueing discipline is not imposed on the process at the sending end of the channels c1, c2, etc., then their traces are arbitrarily interleaved. The resolution of the above indeterminism is an implementation decision and is in no way constrained by the CSP specification.

6.1.4.4 - Sequential composition

Sequential composition is a simpler interaction than the concurrency of the parallel composition. The effect of two process FIRST and SECOND in strict sequence is described by their sequential composition:

Eq-6.35 \[ \text{ONETWO} = \text{FIRST;}\text{SECOND} \]

Sequential composition may be combined with recursion to yield some of the power of an infinite-state automaton [MIN72] to recognise indefinitely long sequences of events. For example, if the event B occurs in the context of one or more preceding A events, then the following process AB terminates with a C event. But if there are no A events, then AB terminates with B.

Eq-6.36 \[ \alpha \ AB = \{A,B,C\} \]
Eq-6.37 \[ \text{AB} = (B \rightarrow \text{SKIP} \mid A \rightarrow (\text{AB};C \rightarrow \text{SKIP})) \]

[HOA85] suggests that this provides a mechanism for context-dependent operation, in the sense of a recogniser of Chomsky-1 languages. However, later sections will demonstrate that the context-dependent decisions are significantly less wieldy than [HOA85] suggests, and that there is no easy way to compose two or more context-free modules into a single context-dependent process. Note also in passing that the AB process above achieves the effect of an infinite-state automaton only by treating each recursive call as a distinct state.
To do this, AB must be able to discover its depth of recursion and vary its processing accordingly.

6.1.5 - The operating environment of a process

CSP has a particularly satisfying and self-consistent way of characterising the environment of a process P. In general, the environment reacts to events initiated by P, or initiates events to which P reacts. This behaviour is readily expressed as another process E which interacts with P in any of the ways described in an earlier section. The parallel composition P||E may therefore be used to couple process and environment.

In short, the environment is simply another process whose behaviour may be modelled in however much detail is considered useful. The behaviour of P in a particular environment E is described precisely by the process P||E.

6.1.6 - A formal definition of determinacy

The above concepts of process behaviour now permit a formal definition of deterministic behaviour. Informally, given

1. a process P in an environment E, and
2. P may engage in any one of several events

then P is said to be deterministic if its choice of behaviour is always determined *externally* by E. (Note that E determined the choice either by actively influencing P, or in a weaker sense by observing the event the instant it occurs.) Otherwise P chooses its behaviour in some arbitrary or implementation-dependent way, and is then said to be nondeterministic.

The formal CSP definition of nondeterminism is stated in terms of process deadlocks and the events which lead to those deadlocks - *refusals*. 
Consider two processes $P$, $Q$ and the conditional behaviour which results from using the choice operators $\|\|$ (general choice) and $\|$ (nondeterministic choice).

$$
\text{Eq-6.38} \quad \alpha P = \alpha Q = \{x,y\} \\
\text{Eq-6.39} \quad P = (x \rightarrow P) \\
\text{Eq-6.40} \quad Q = (y \rightarrow Q)
$$

Now place $(P \| Q)$ in the environments $P$ and $Q$ in turn and observe the behaviour.

$$
\text{Eq-6.41} \quad (P \| Q)\|P = (x \rightarrow P) \\
\text{Eq-6.42} \quad (P \| Q)\|Q = (y \rightarrow Q)
$$

In both cases the environment unambiguously selects the appropriate common behaviour, and $\|$ is therefore equivalent to the deterministic $\|$ operator. No deadlock ever results.

Now place $(P \| Q)$ in either environment $P$ or $Q$, and observe that in either case a deadlock is now possible.

$$
\text{Eq-6.43} \quad (P \| Q)\|P = (P\|P) \| (Q\|P) = P \| \text{STOP} \\
\text{Eq-6.44} \quad (P \| Q)\|Q = (P\|Q) \| (Q\|Q) = \text{STOP} \| Q
$$

The event $x$ of environment $P$ which causes $(P \| Q)$ to deadlock is called a refusal of $P$. If the $(P \| Q)$ choice is ever made in favour of $Q$ (and recall that the $\|$ represents implementation freedom), then event $x$ will always cause $(P \| Q)\|P$ to deadlock. A similar argument applies to event $y$ and $(P \| Q)\|Q$.

The CSP definition of deterministic behaviour follows immediately: a process is deterministic if it can never refuse any event in which it may legally engage.

6.2 - The 2-phase commit protocol in CSP

The CSP specification of the protocol is derived from two sources:

1. an informal understanding of the protocol's operation
2 a few basic assumptions concerning the eventual operating environment and the likely implementation structure of the protocol

For reasons already discussed in the chapter on Z specification, the informal understanding of the protocol will depend heavily on the concept of \textit{state}. Similarly, the operating environment and implementation structure has also been discussed in the context of Z. The CSP process is a very convenient structuring primitive, and CSP's emphasis on events rather than on explicit messages excludes a considerable amount of tedious detail.

This section will adopt the following conventions for structuring the CSP specification:

1. the master/slave communication is modelled by explicit message-passing. This preserves a resemblance to the informal protocol description, but is formally equivalent to the use of shared events for synchronising processes

2. the interaction of the master and its environment (and similarly the slave and its environment) is modelled by shared events

3. the master, slave, master and slave environments are all specified as distinct processes. The main justification is that these components are functionally distinct and should therefore be treated as separate modules. In addition, this modularisation also expresses the likely implementation structure

The aim is to specify the complete protocol as the concurrent operation of these four components: master (M), slave (S), and master and slave environments (ME, SE).

\textbf{Eq-6.45} \quad 2PC = M \parallel S \parallel ME \parallel SE
The events of interest in the CSP specification are the following:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>master initialises</td>
</tr>
<tr>
<td>AM1</td>
<td>master initiates a commitment</td>
</tr>
<tr>
<td>AM2</td>
<td>master is ready to commit</td>
</tr>
<tr>
<td>AM3</td>
<td>master is ready to abort</td>
</tr>
<tr>
<td>ss</td>
<td>slave initialises</td>
</tr>
<tr>
<td>AS1</td>
<td>slave is ready to commit</td>
</tr>
<tr>
<td>AS2</td>
<td>slave is ready to abort</td>
</tr>
<tr>
<td>AMS4</td>
<td>master and slave commit</td>
</tr>
<tr>
<td>AMS5</td>
<td>master and slave abort</td>
</tr>
</tbody>
</table>

Note the asymmetry between master and slave: the master may initiate a commitment (AM1) but the slave may only react. Note also that the AMS4/5 events must be shared to guarantee that the master and slave synchronise in commitment.

6.2.1 - The master operating environment

The operating environment ME of the master component M represents the communicating application which is making use of the commitment protocol. The conditions which arise in ME supply the context which enables M to resolve decisions which would otherwise be ambiguous.

ME first initialises and eventually initiates a commitment (AM1), although this second event may be delayed indefinitely.

\[
\alpha \text{ ME } = \{ ms, AM1, AM2, AM3, AMS4, AMS5 \} \\
\text{Eq-6.46} \\
\text{ME } = (ms \rightarrow \text{ME1}) \\
\text{Eq-6.47}
\]

Before a commitment is initiated, various ME conditions may arise which determine whether or not the commitment will succeed. All such conditions are summarised abstractly by the two events AM2 (ready to commit) and AM3 (ready to abort). The AM2/3 conditions arise independently of the decision to initiate a commitment, and may arise repeatedly right up to the initiation. Alternatively, the decision to initiate
may precede all the $AM2/3$ events. To summarise, the $AMI$ event may occur in any one of 3 distinct contexts:

1. $AMI$ precedes both $AM2$ and $AM3$
2. $AM2$ immediately precedes $AMI$
3. $AM3$ immediately precedes $AMI$

Eq-6.48 $\text{ME1} = (AMI \rightarrow C1 \mid AM2 \rightarrow C2 \mid AM3 \rightarrow C3)$

It is necessary to decide in which context the $AMI$ event has occurred. If the $AMI$ precedes all $AM2/3$ events, then the next $AM2/3$ event will decide the subsequent behaviour. If $AMI$ immediately preceded by $AM2$, then a successful commitment is possible (but not guaranteed). If $AMI$ immediately preceded by $AM3$, then a successful commitment is impossible.

Each of the $C1/2/3$ processes above represents a distinct context. Switching between contexts is achieved by mutual recursion.

Eq-6.49 $C1 = (AM2 \rightarrow (AMS4 \rightarrow SKIP | AMS5 \rightarrow SKIP) \mid AM3 \rightarrow AMS5 \rightarrow SKIP )$

Eq-6.50 $C2 = (AMI \rightarrow (AMS4 \rightarrow SKIP | AMS5 \rightarrow SKIP) \mid AM2 \rightarrow C2 \mid AM3 \rightarrow C3 )$

Eq-6.51 $C3 = (AMI \rightarrow AMS5 \rightarrow SKIP \mid AM2 \rightarrow C2 \mid AM3 \rightarrow C3 )$

The complete behaviour of the master operating environment is specified by Eq.6-46 to 6-51.

Process ME1 and the mutual recursion of processes C1/2/3 together permit random interleaving of the $AM2/3$ events, but it is not clear from the informal description if this is acceptable. Several other interpretations are possible:

1. $AM2$ and $AM3$ should occur in strict alternation
2. $AMI$ should occur before any of $AM2/3$ (i.e. don’t worry about the decision before it becomes relevant)
These alternatives expose the importance of implementation considerations at the specification stage, and provide evidence for Swartout's argument [SWA82].

6.2.2 - Some other approaches to defining ME

Consider now some other process constructions which might be expected to specify the action of the master operating environment.

6.2.2.1 - Parallel composition

An attractively modular approach to defining ME is to set up a free-running process ME0 which alternates between events AM1/2/3, and to execute concurrently a number of distinct context-recognising processes CE1/2/3.

\[
\begin{align*}
\text{Eq-6.52} & \quad \text{ME0} & = & (\text{ms} \rightarrow \text{ME1}) \\
\text{Eq-6.53} & \quad \text{ME1} & = & (AM1 \rightarrow \text{SKIP} \\
& & & | AM2 \rightarrow \text{ME1} \\
& & & | AM3 \rightarrow \text{ME1}) \\
\text{Eq-6.54} & \quad \text{CE1} & = & (AM1 \rightarrow (AM2 \rightarrow (AMS4 \rightarrow \text{SKIP} \\
& & & | AMS5 \rightarrow \text{SKIP})) \\
& & & | AM3 \rightarrow AMS5 \rightarrow \text{SKIP})) \\
\text{Eq-6.55} & \quad \text{CE2} & = & (AM1 \rightarrow (AMS4 \rightarrow \text{SKIP} \\
& & & | AMS5 \rightarrow \text{SKIP}) \\
& & & | AM2 \rightarrow CE2 \\
& & & | AM3 \rightarrow CE3) \\
\text{Eq-6.56} & \quad \text{CE3} & = & (AM1 \rightarrow AMS5 \rightarrow \text{SKIP} \\
& & & | AM2 \rightarrow CE2 \\
& & & | AM3 \rightarrow CE3)
\end{align*}
\]

and finally

\[
\text{Eq-6.57} \quad \text{ME} & = \quad \text{ME0} \parallel \text{CE1} \parallel \text{CE2} \parallel \text{CE3}
\]

But this approach contains a serious fault. The \( \parallel \) operation allows only the common behaviour to take place.
Eq-6.58 \[ CE_2 \| CE_3 = (AM_1 \rightarrow AMS_5 \rightarrow \text{SKIP} \]
\[ | AM_2 \rightarrow CE_2 \| CE_3 \]
\[ | AM_3 \rightarrow CE_2 \| CE_3 ) \]

CE2||CE3 (and therefore ME) is incapable of engaging in the sequence <ms,AM2,AM1,AMS4...>. There is therefore at least one correct commitment trace which ME is incapable of generating.

6.2.2.2 - Interleaving

Another possibility is to interleave III rather than compose || the component processes.

Eq-6.59 \[ ME = ME_0 || CE_1 || CE_2 || CE_3 \]

The difficulty here is that the III indeterminism is not always acceptable. It is sufficient that whenever AM2 occurs, then exactly one of CE2 and CE3 will execute. But the indeterminate processing of AM1 by CE1|||CE2|||CE3 is not acceptable. One could remove this determinism by ensuring that the interleaved processes have disjoint sets of initial events, but this would remove the flexibility of III and defeat its purpose. It seems that III is not a useful operation for context recognition.

6.2.2.3 - Selective failure

Another possibility is to use || to execute the context recognisers concurrently, and to define the process actions so that only one can succeed while the others fail in deadlock. But the event which causes P to deadlock also causes P||Q to deadlock (by law L3A, [HOA85]), so this approach can never succeed.

6.2.2.4 - Angelic nondeterminism

The problem addressed by ME can be formulated as one of parsing sentences of a formal language. A process P is said to accept a sentence S if it terminates successfully after engaging in the sequence of events denoted by S. Formal languages are classified in complexity
according to Chomsky's scheme (see Appendix A): regular expressions (Chomsky-3), context-free (Chomsky-2), context dependent (Chomsky-1), and unrestricted (Chomsky-0).

CSP is immediately capable of recognising a regular expression and therefore has all the power of a (strict) finite-state automaton. A subclass of context-free languages is also within CSP's power: those LL(1) languages which require no backtracking. (To understand this restriction, consider the semantics of CSP events. An event cannot be undone once engaged in, so backtracking is impossible. The simple choice operator | requires the first event to be different for all alternatives, so a look-ahead of more than 1 is impossible.) Although [LEW76] points out that LL(k) languages for k>1 have negligible practical significance for compiler construction, it is far from clear that LL(1) is adequate for the much wider class of process interactions addressed by CSP.

But the problem posed by the ME definition is of Chomsky-1 complexity: AMI in the context of AM3 always leads to AM5, while in context AM2 it leads to either AM4 or AM5. The use of the || synchronisation operator allows a degree of context-dependence (see examples X6 and X7, section 5.1 of [HOA85]) but a more general mechanism is needed. The CSP approach is to execute the several alternative contexts concurrently and to delay the choice between them until the environment (i.e. some interacting process) supplies an event which resolves the issue. Then, the chosen process continues and all the alternative processes are stopped and their effects reversed.

This delay mechanism is known as *angelic nondeterminism*. When making an angelic choice between two processes P and Q, the essence is to delay the choice of P or Q until exactly one of them is capable of executing. A suitable definition of angelic choice along the lines of [HOA85] is:
Eq-6.60 \[
\text{angel } (P,Q) = \begin{cases} 
\text{if } P(x) \neq \text{BLEEP} & \text{then } P(x) \\
\text{else if } Q(x) \neq \text{BLEEP} & \text{then } Q(x) \\
\text{else angel } (P(x),Q(x))
\end{cases}
\]

This definition is easily extended to more than two processes.

The context-dependent behaviour of ME1 (Eq-6.48) can be stated as follows

Eq-6.61 \[
\text{ME1} = \text{angel } ((AM1\rightarrow C1), \\
(AM2\rightarrow C2), (AM3\rightarrow C3))
\]

Two comments can be made on this use of angelic choice. First, angelic choice always requires the ability to reverse the effects of the unsuccessful alternatives. The choice between N alternatives will be delayed as long as they continue to execute shared events.

Eq-6.62 \[
\text{ALT1} = (A \rightarrow B \rightarrow C \rightarrow X \rightarrow \text{SKIP})
\]

Eq-6.63 \[
\text{ALT2} = (A \rightarrow B \rightarrow C \rightarrow Y \rightarrow \text{SKIP})
\]

But the act of engaging in the shared events may well have some semantic significance for the processes, and it might not make sense to reverse (say) ALT1 and then expect ALT2 to continue successfully. [HOA85] would argue that such a consideration is beyond the scope of CSP, which holds that events are free of semantics and are significant only to the extent that they are observable and distinguishable. However, a trustworthy and practical mechanism appears to need some extra conditions to govern how the angelically-chosen processes share events. In short, the semantics of angelic choice is not fully defined for the practical purposes of protocol specification.

Second, angelic choice is unnecessary in this example. The C1/2/3 definitions (Eq-6.49/50/51) show that it is sufficient for the contexts C2 and C3 to toggle. However, this coupling of contexts is an accident of this example and cannot be expected to occur generally. A more
elaborate example is needed to explore CSP's ability to express context-dependent behaviour.

6.2.3 - The remaining 2-phase commit processes

The remaining components of the 2-phase commit protocol specification of Eq-6.43 are the master (M), the slave (S), and the slave environment (SE). Given the considerations of the previous section, these definitions are straightforward. In the following section, "ms" denotes the message channel from master to slave, and "sm" the reverse channel.

6.2.3.1 - The master process M

\begin{align}
O \cdot M &= \{ms, AM1, AM2, AM3, AMS4, AMS5, \\
& \quad ms!start, ms!abort, ms!commit, \\
& \quad sm?yes, sm?no\} \\
M &= (ms \rightarrow M1) \\
M1 &= (AM1 \rightarrow ms!start \rightarrow \\
& \quad (sm?yes \rightarrow MC1 \\
& \quad | sm?no \rightarrow AMS5 \rightarrow SKIP) \\
& \quad | AM2 \rightarrow MC2 \\
& \quad | AM3 \rightarrow MC3) \\
MC1 &= (AM2 \rightarrow ms!commit \rightarrow AMS4 \rightarrow SKIP \\
& \quad | AM3 \rightarrow ms!abort \rightarrow AMS5 \rightarrow SKIP) \\
MC2 &= (AM1 \rightarrow ms!start \rightarrow \\
& \quad (sm?yes \rightarrow ms!commit \rightarrow AMS4 \rightarrow SKIP \\
& \quad | sm?no \rightarrow AMS5 \rightarrow SKIP) \\
& \quad | AM2 \rightarrow MC2 \\
& \quad | AM3 \rightarrow MC3) \\
MC3 &= (AM1 \rightarrow AMS5 \rightarrow SKIP \\
& \quad | AM2 \rightarrow MC2 \\
& \quad | AM3 \rightarrow MC3)
\end{align}

Process MC3 illustrates the incompleteness of the informal specification and the influence of implementation decisions. After events...
the informal specification leaves unanswered the question whether or not the M process is reinitialised: \texttt{SKIP} expresses the assumption that M is not re-initialised. But if M is reinitialised, there is still the question whether or not the most recent AMS remains valid. If yes, then the MC3 definition is $AM1 \rightarrow AMS5 \rightarrow MC3$, and if not then $AM1 \rightarrow AMS5 \rightarrow M1$. This is likely to be an implementation decision. Similar remarks apply to the MC2 definition.

6.2.3.2 - The slave process S

Eq-6.70 $\Omega S = \{ss,AS1,AS2, ms?start,ms?commit,$
\hspace{1cm} \text{ms?abort,sm!yes,sm!no}\}$

Eq-6.71 $S = ss \rightarrow S1$

Eq-6.72 $S1 = (AS1 \rightarrow SC1$
\hspace{1cm} | AS2 \rightarrow SC2$
\hspace{1cm} | ms?start \rightarrow (AS1 \rightarrow sm!yes$\rightarrow$
\hspace{2cm} \text{(ms?commit$\rightarrow AMS4 \rightarrow \text{SKIP}$}$\hspace{1cm} \text{| ms?abort$\rightarrow AMS5 \rightarrow \text{SKIP}$})$
\hspace{1cm} | AS2 \rightarrow sm!no \rightarrow AMS5 \rightarrow \text{SKIP}))$

Eq-6.73 $SC1 = (AS1 \rightarrow SC1$
\hspace{1cm} | AS2 \rightarrow SC2$
\hspace{1cm} | ms?start \rightarrow sm!yes$\rightarrow$
\hspace{2cm} \text{(ms?commit$\rightarrow AMS4 \rightarrow \text{SKIP}$}$\hspace{1cm} \text{| ms?abort$\rightarrow AMS5 \rightarrow \text{SKIP}$})$

Eq-6.74 $SC2 = (AS1 \rightarrow SC1$
\hspace{1cm} | AS2 \rightarrow SC2$
\hspace{1cm} | ms?start \rightarrow sm!no \rightarrow AMS5 \rightarrow \text{SKIP})$

6.2.3.3 - The slave environment SE

Eq-6.75 $\Omega SE = \{ss,AS1,AS2, ms?start,ms?commit,$
\hspace{1cm} \text{ms?abort,sm!yes,sm!no}\}$

Eq-6.76 $SE = (ss \rightarrow SE1)$
Process \( SE \) provides the \( AS1/2 \) context for process \( S \). \( SE \) free-runs until one of the AMS4/5 events causes it to terminate.

The complete CSP definition of the 2-phase commit protocol is summarised by

Eq-6.80  \( 2PC = M \parallel S \parallel ME \parallel SE \)

6.3 - Stating protocol properties

The properties of interest in a protocol specification are: (1) legal interaction of components, (2) liveness, (3) deadlock, and (4) correctness. Experience of CSP is in agreement with that gained from the other formal methods studied here: all such properties can be stated only to the extent that an algebraic structure with suitable semantics can be constructed.

6.3.1 - Legal interaction, liveness, deadlock

Properties concerning the legality of component interaction are readily stated using the basic CSP facilities by which processes are constructed (prefix, deterministic choice, etc.) and how they interact (synchronisation on shared events, sequential composition). The shared event is an elegant and powerful way of specifying the simultaneous achievement of a goal by several protocol components.

However, it is not easy to demonstrate that a system of processes is free from deadlock. The essence is to demonstrate that no interaction leads to \( STOP \). In most systems this involves an examination of the results of applying (mostly) the parallel-composition operator. The
results of this highly recursive operation can be very difficult to enumerate exhaustively even for apparently simple systems. It is interesting to note that the proof [HOA85] of the well-known dining philosophers problem does not depend on enumeration (indeed, a simple calculation shows that the state space is far too large for such an attempt to succeed). Rather, the proof considers in an ad-hoc way the structure of the possible traces, and shows that none contains STOP. It is clearly intended that one should reason about a CSP specification. Such an approach clearly requires the ad-hoc construction of lemmas, and its success is strongly influenced by one's insight into the particular problem.

Correctness properties (in the sense that event $B$ is a necessary consequence of event $A$) are also readily stated as CSP events, and the use of shared events is a powerful and elegant way to specify the simultaneous achievement of a goal by several protocol components. For example, the correct operation of the 2-phase commit protocol is indicated by the occurrence of either event $AMS4$ ("both partners commit") or $AMS5$ ("both partners abort").

6.3.2 - Temporal properties

Fairness properties (i.e. that some desirable and live event must eventually occur) present a serious problem in CSP. [HOA85] emphasises that the trace of a CSP process must be finite, and then draws an interesting conclusion on temporal properties such as fairness and liveness. [HOA85] takes fairness to mean "any event which may happen infinitely often must happen eventually". He then argues that such an event may also be indefinitely delayed and therefore does not necessarily appear in any finite trace. The conclusion, which is supported by [DIJ88], is that any property which depends on the eventual achievement of a goal is beyond finite reasoning, and therefore cannot in general form part of a workable (i.e. demonstrable, enforceable) specification. This conclusion is resisted by [SCH88] who points out that if liveness is an unworkable concept, then so is termination.
One notes that several techniques (e.g. modal logics) have no difficulty in stating and proving fairness. But the CSP viewpoint is not wholly impractical: the finiteness of a trace corresponds to the practical notion that only a finite amount of information can be taken into account in making a decision. In the longer term one must acknowledge that CSP raises a serious question about the meaning of such a proof.

The practical value of fairness properties makes one reluctant to dismiss this issue as an arcane philosophical dispute. This matter is considered below. An alternative definition of fairness is offered which is compatible with accepted concepts of termination and finite behaviour, and which brings temporal properties into the scope of a CSP specification.

6.3.2.1 - Liveness, fairness

An important aspect of the correct operation of most systems is that

1. in all circumstances some action is possible, and
2. the system behaviour achieves one of several correct configurations (repeatedly so, in the case of non-terminating applications).

Requirement (1) is known as liveness, and is a necessary condition for the absence of deadlocks, while requirement (2) is a sufficient condition that the system will operate successfully in a certain sense.

A CSP specification is explicitly free of modality (although it is interesting to note that the very closely related CCS approach does contain modality). Liveness is expressed implicitly by the interaction of processes: one proves the assertion that the trace of the (composite) process which represents the whole system is capable of indefinite extension.

The concept of fairness is very closely related to liveness. Where several correct outcomes are possible, liveness (in the sense of requirement (2), above) can guarantee that all are capable of occurring, whereas fairness guarantees that the outcomes will occur in a particular discipline (e.g. in a particular sequence). Fairness is therefore a stronger
condition than liveness, and is of considerable interest when specifying schedulers and similar applications. [SCH88] points out that termination can be considered a special case of fairness.

The problem presented by fairness is significantly different from that of liveness. [HOA85] and [DIJ88] argue that the liveness of an event does not guarantee that this event will eventually take place. Because the event is live, it always remains possible that it will take place, but it may also be delayed indefinitely by other events. This is illustrated below in CSP terms. Event $B$ is clearly possible, but may be preceded by an indefinite number of $A$ events.

\[
\text{Eq-6.81} \quad \text{NotFair} = (B \rightarrow \text{SKIP} \mid A \rightarrow \text{NotFair})
\]

This viewpoint arouses passions (see [SCH88]) but it is eminently reasonable to point out that in general there is no rigorous justification for the commonsense notion that a process which executes for long enough must eventually display all possible behaviour.

A constructive approach to this problem is the following:

1. acknowledge that the [HOA85] definition of fairness (and note that [SCH88] does not dispute this definition) is indeed unjustifiable in rigorous terms
2. adopt a narrower and justifiable definition of fairness
3. distinguish between two sources of unfairness: perverse choice, and interrupted processing

Consider the following definition of fair behaviour:

*an event $A$ is fairly executed if, whenever particular conditions $C$ arise, event $A$ is the only possible behaviour if any further processing takes place.*

This reconciles the various interpretations of fairness by leaving open the possibility that processing may be halted ("the program crashes") after $C$ becomes valid and before $A$ takes place. This is consistent with the semantics of the Z preconditioned choice, and also with the widely accepted pre/postcondition scheme of [HOA69] which makes
successful termination a necessary condition of all conclusions. The meaning of a modal logic specification is unchanged: the modal expressions still describe reachability relationships, and a fair configuration can be achieved only if processing is not halted.

6.3.2.2 - Perverse choice

But the fairness of the NotFair process above still cannot be guaranteed even if processing continues forever. It appears that perverse choice can be eliminated only by careful attention to the indeterminism introduced into the specification. Any required fairness must be explicitly stated as process interactions. For example, the process TakeTurns below ensures that events A and B occur in equal numbers.

\[
\text{Eq-6.82 TakeTurns} = (A \rightarrow B \rightarrow \text{TakeTurns} \\
B \rightarrow A \rightarrow \text{TakeTurns})
\]

6.4 - The user/server protocol in CSP

Along the lines of the earlier definition of the 2-phase commit protocol, the aim is to specify the complete protocol as the concurrent operation of four components: user (U), server (SV), and user and server environments (UE, SVE).

The complete user/server protocol is defined by

\[
\text{Eq-6.83 USP} = U \parallel SE \parallel SVE \parallel UE
\]

The CSP events of interest are:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>us</td>
<td>user initialises</td>
</tr>
<tr>
<td>AU1</td>
<td>user wants service</td>
</tr>
<tr>
<td>ss</td>
<td>server initialises</td>
</tr>
<tr>
<td>ASV1</td>
<td>server ready to provide service</td>
</tr>
<tr>
<td>ASV2</td>
<td>server not ready to provide service</td>
</tr>
<tr>
<td>ASVU1</td>
<td>server successfully provides user</td>
</tr>
<tr>
<td>ASVU2</td>
<td>server and user correct fault</td>
</tr>
</tbody>
</table>
6.4.1 - The server environment SVE

\[ \alpha \text{SVE} = \{ss, ASV1, ASV2, ASVU1, ASVU2\} \]

\[ \text{SVE} = (ss \rightarrow \text{SVE1}) \]

\[ \text{SVE1} = (ASV1 \rightarrow (ASVU1 \rightarrow \text{SVE1}) \]

\[ \left| ASV2 \rightarrow ASVU2 \rightarrow \text{SVE1} \right) \]

The ASV1/2 contexts don’t free-run. An ASV2 fault must be corrected before ASV1 can occur again. The above interpretation is that ASV1 must be resupplied after ASV2 has been corrected. This is one of several possible interpretations of the informal description.

6.4.2 - The server SV

\[ \alpha \text{SV} = \{ss, ASV1, ASV2, ASVU1, ASVU2, sm!alarm, sm!done, us?req\} \]

\[ \text{SV} = (ss \rightarrow \text{SV1}) \]

\[ \text{SV1} = (ASV1 \rightarrow \text{SVC1} \]

\[ \left| ASV2 \rightarrow su!alarm \rightarrow \text{SVC2} \right) \]

\[ \left| us?req \rightarrow (ASV1 \rightarrow su!done \rightarrow \text{SV1} \right) \]

\[ \left| ASV2 \rightarrow su!alarm \rightarrow \text{SVC2} \right) \]

\[ \text{SVC1} = (us?req \rightarrow su!done \rightarrow ASVU1 \rightarrow \text{SV1} \]

\[ \left| ASV2 \rightarrow su!alarm \rightarrow \text{SVC2} \right) \]

\[ \text{SVC2} = (us?ack \rightarrow ASVU2 \rightarrow \text{SV1} \mid us?req \rightarrow \text{SVC2}) \]

Another implementation decision arises in the SVC1 definition. If the ASV1 event becomes invalid after the ASVU1, then the definition is ...\rightarrow ASVU1 \rightarrow \text{SV1}, and ...\rightarrow ASVU1 \rightarrow \text{SVC1} if the ASV1 remains valid.

6.4.3 - The user environment UE

\[ \alpha \text{UE} = \{us, AU1, ASVU1, ASVU2\} \]

\[ \text{UE} = (us \rightarrow \text{UE1}) \]
Eq-6.94 \[ UE1 = (AU1 \rightarrow (ASVU1 \rightarrow UE1 \quad | \quad ASVU2 \rightarrow ASVU1 \rightarrow UE1) \quad | \quad ASVU2 \rightarrow UE1) \]

6.4.4 - The user U

Eq-6.95 \[ \alpha(U) = \{us,AU1,ASVU1,ASVU2, \]

\[ su?alarm, su?done, us!req, us!ack\} \]

Eq-6.96 \[ U = (us \rightarrow U1) \]

Eq-6.97 \[ U1 = (AU1 \rightarrow us!req \rightarrow UC1 \mid su?alarm \rightarrow us!ack \rightarrow U1) \]

Eq-6.98 \[ UC1 = (su?done \rightarrow ASVU1 \rightarrow U1 \mid su?alarm \rightarrow UC1) \]

This definition of U allows a request to be interrupted by an alarm, and then be successfully completed without an ack from U to S. This is permitted by the informal description and by the earlier finite-state specification. But U now contains contradictory uses of ack, and the result is that U||S contains a deadlock. To see this, consider the following traces of U and S respectively:

1. \(<us,AU1,us!req, su?alarm, su?done, ASVU1>\) and

2. \(<ss, us?req, ASV2, su!alarm, us?ack or us?req>\)

U and S synchronise on the events us!req and su!alarm, but then fail to agree on the next synchronising event: U expects su?done, while S expects us?ack or us?req. Therefore S||U = STOP.

The use of CSP here exposes the peculiar semantics of the fault event ASV2. ASV2 arises in SE, but is corrected simply by S receiving an ack from U. In contrast, ASV1 is a context in the same sense as the AM2/3 events of the 2-phase commit protocol: entirely under the control of the environment.

6.5 - Conclusions on the use of CSP

One can now summarise the main conclusions on the use of CSP for the formal specification of a protocol.
6.5.1 - Expressive power

The expressive power of CSP can be related to the Chomsky classes of formal languages. CSP is immediately capable of recognising a regular expression and therefore has all the power of a (strict) finite-state automaton. A subclass of context-free languages is also within CSP's power: those LL(1) languages which require no backtracking. However, it is not clear that LL(1) is adequate for the whole class of process interactions addressed by CSP.

A degree of context-dependence can be achieved by the use of the I\& synchronisation, but a more general mechanism is needed. The angelic choice technique is suggested, but its semantic definition is incomplete in a practical sense.

6.5.2 - Modular development of a specification

Still on context-dependent processing, it appears that there is no easy way to compose a number of distinct contexts into a context-dependent process (i.e. to build up a complex recogniser from a number of context-free modules). This restriction arises in part from the particular semantics of the protocols considered, but the various CSP facilities for process interaction also appear to be fundamentally unhelpful.

6.5.3 - Temporal properties

CSP is not generally capable of facilitating proofs of temporal properties which depend on the guaranteed occurrence of a given event e.g. liveness and fairness properties. This contrasts dramatically with other formal techniques (e.g. modal logics). Progress can be made by taking termination as a primitive condition and adopting a modified definition of fairness. However, this approach is not part of CSP as defined by [HOA85].
CSP is a starkly abstract notation: events are explicitly free of semantics and are significant only in that they are observable and distinguishable. But however valuable this abstraction is for analysis purposes, a subtly different quality of abstraction appears to be needed to model real events, which of course have semantics.

Recall the discussion of section 6.2.1 which points out several interpretations of the informal requirements. It was found to be impossible to write a full CSP specification without first settling these questions of detail. [HOA85] would probably argue that such precision is a thoroughly desirable feature of the CSP approach, and indeed the discipline of considering (in fairly low-level detail) the event sequences can expose serious errors which other techniques reveal only after considerable analysis. For example, the deadlock caused in the user/server protocol by the inconsistent use of \textit{ack} is quickly revealed by the difficulty of writing the process descriptions. In contrast, the finite-state specification of an earlier section needs to be exhaustively enumerated before the deadlocking global state becomes apparent.

But recall the principle that a system specification should contain only those details which are \textit{necessarily} present in every possible implementation. From the discussion of section 6.2.1 it is clear that a detailed specification of the M and S protocol processes (and certainly the rigorous analysis thereof) depends on the details of the ME and SE specifications. To a large extent this problem arises due to the close coupling of a protocol component (i.e. master or slave) and its "driving" application, and is evident also in the Z and finite-state specifications. But CSP seems to be significantly more restrictive in its insistence on detail. In practice one would prefer to specify a protocol (i.e. the M and S processes) to accommodate all the environmental conditions of section 6.2.1, but this seems to be an impractical goal in CSP.
Chapter 7 - Summary and conclusions

7.1 - Objectives and limitations of the study

This investigation considers the extent to which a formal specification of requirements can contribute to the achievement of a distributed system of demonstrable correctness. More particularly, the investigation considers the extent to which several representative specification techniques are capable of expressing this required behaviour. Of special interest are formal techniques for specifying behaviour. Distributed applications are given special attention.

Several important questions are excluded from the investigation:

1 the substance of a specification will vary significantly according to the needs of the interested party. For example, an end-user might emphasise functional correctness, while a system development manager might demand maintainability. This generality of requirements introduces considerations of life-cycle stages and viewpoints. Such questions are ignored. Instead, the sole viewpoint considered is that of a system architect who is primarily interested in the correct co-operation of the distributed components.

2 a specification necessarily addresses requirements, but several techniques result in specifications which can be successively refined into implementations, or which can be animated as a prototype implementation. This merging of specification and implementation is not considered. To a first approximation, the approach of [BAL79] is adopted: a specification is concerned with what is required, and not with how it might be achieved.

3 Non-functional aspects (e.g. user-friendliness, reliability, maintainability, performance) usually take the form of constraints on how the program interacts with its operating
environment. Some researchers [LEM82] attempt to reduce environmental interactions to data mappings, while others [MEL86] argue that a model of the whole operating environment is required. This investigation is alert to the implications of a program's interaction with its environment, but proceeds on the practical assumption that a worthwhile specification does not demand a complete world model.

7.2 - Achievements of the study

This study has achieved a significant clarification of several issues (both practical and theoretical) in the use of formal specification techniques in obtaining a correct implementation.

1 despite the apparent differences in the Z, CSP and finite-state approaches, the concept of state is shown to be fundamental in practice. To a first approximation, the finite-state approach to protocol specification is as powerful as the use of predicate calculus and process algebra in processes which exhibit repetitive and checkpointing behaviour.

2 the application of three representative techniques (finite-state, Z and CSP) is considered in detail and their strengths and weaknesses compared, both in their description of protocol properties, and in their amenability to formal analysis. In particular,

2.1 a novel technique is developed which enumerates global-state trees more economically than [BRAND83]. In the particular context of a communication protocol, this novel technique achieves most of the effect of the [BRAND83] results but with much simpler criteria for limiting tree growth. This simplification is made possible by the definition of objective criteria for classifying the protocol behaviour modes which are characterised by the leaves (which represent global states).

2.2 the use of modal logic is shown to be necessary in a Z specification which addresses questions of liveness and
eventuality. This modification of Z readily permits the specification of protocol properties which are characterised by events separated by unstated time intervals.

2.3 CSP is shown to be strong in describing combinatorial behaviour, but this description is unexpectedly low-level and difficult to analyse systematically. The semantic completeness of some CSP constructions is questionable.

3 a contribution is made to the current debate on the meaning of fairness in finite descriptions of behaviour. This contribution unifies the conflicting interpretations of [DIJ88] and [SCH88] by imposing reasonable restrictions on the conclusions which can be drawn from a finite description of behaviour. The role of perverse choice in bringing about unfair behaviour is illustrated, and an example demonstrates how perverse choice can be avoided.

4 a theoretical argument questions the validity of the simple viewpoint that a system specification should not address matters of implementation. A consideration of impossible and implicit specifications concludes that a considerable degree of operational character is necessary in a workable specification.

5 it is concluded that the nature of formal specification techniques is essentially functional in the narrow mathematical sense. A treatment of the broader type of functional behaviour requires a model of the system’s operating environment. The three techniques are capable of this, at least for the particular semantics of a communications protocol, and there appears to be little difference in their analysis power.

6 some general observations are made concerning the overall value of a formal approach to specification and a significant limitation of the principal techniques in current use. The
character of non-functional behaviour is examined, and it is concluded that non-functional requirements are invalid elements of a demonstrable specification.

The above results and conclusions are summarised in the following sections.

7.3 - The fundamental role of "state" in certain applications

The concept of state appears to play a fundamental practical role in a certain class of applications, including communication protocols, although it is not immediately clear that the finite-state approach underlies other description techniques (i.e. that a Z or CSP specification of a communication protocol is at root a model of state-space semantics). However, one observes that in practice the Z and CSP approaches to protocol specification rely heavily on the state concept.

One can conclude that, to a first approximation, the finite-state approach to protocol specification is as powerful as the use of predicate calculus and process algebra in processes which exhibit repetitive and checkpointing behaviour.

A repetitive process is characterised by the periodic nature of its behaviour. Periodic behaviour has the consequence that two identical stimuli (separated by a sufficiently long interval) produce an identical response.

In some cases the behaviour of a process is conveniently thought of as a sequence of checkpoints. A checkpoint has the property that, once achieved, it is never lost. This checkpoint approach is useful for two types of application:

1 those which model processes whose semantics indicate irreversible actions
2 those which model processes where a great amount of detail may need to be taken into account before a decision can be made. In these cases it is practical to classify the major decisions as irreversible so that once the decision is made,
the context of the decision (i.e. all the minor detail which contributes to the decision) can be ignored

Both repetition and checkpointing are exactly formalised by the concept of state [MIN72]. Communication protocols exhibit both repetition and checkpointing, and the state description is therefore an adequate mechanism.

However, it is not immediately clear that the finite-state approach underlies other description techniques (i.e. that a Z or CSP specification of a communication protocol is at root a model of state-space semantics) despite (1) the prominent role of state in modern control theory [BAR75], or (2) the practice of many tutorial publications (e.g. [SPI89]) to explicitly describe the static component of a Z schema as the set of states (and their invariant relationships) which a system can occupy.

This investigation justifies the use of the state-space approach in a Z specification by reference to the particular semantics of a communications protocol. It appears that Z is not primarily concerned with finite-state concepts: Z variables may be used in a non-state way to avoid some of the fundamental limitations of finite-state automata.

One can point to a superficial resemblance between a [MIN72] state on the one hand, and on the other hand a CSP process. But it would be a major project to demonstrate the formal equivalence of these concepts. More pragmatically, one can point to

1. the ease of Z protocol specification when the substitutions are written in terms of the [MIN72] local states of the protocol components
2. the fact that CSP processes represent the stable configurations and event contexts of a system
3. the established body of control theory [BAR75] which attaches fundamental importance to state-space concepts

If one accepts the above informal conclusion, then it is plausible to argue that the Z and CSP specification techniques (or at least that subset of
their facilities which is used in this investigation) are restatements of state-space [BAR75] control concepts. This situation is not unprecedented. [MIL80] points out that a deep concept manifests itself differently in different formal settings, and admits that these various settings do not reinvent the concept in question, but merely "find it some new clothes".

7.4 - A comparison of finite-state, Z and CSP

The conclusions of this investigation regarding the application of the three representative techniques (finite-state, Z, CSP) to the specification of a distributed communications protocol are stated at length in Chapters 4, 5 and 6 respectively. They are summarised briefly here.

7.4.1 - Finite-state indeterminacy, bandwidth-limited state trees

The causes of indeterminacy in a finite-state protocol specification are classified into three types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>the stimulus of a transition is multi-valued, causing some transitions to be ambiguously triggered</td>
</tr>
<tr>
<td>2</td>
<td>the stimulus of a transition is unspecified</td>
</tr>
<tr>
<td>3</td>
<td>the transition and/or response function is incomplete: the behaviour of the automata is not specified for all possible combinations of input and state.</td>
</tr>
</tbody>
</table>

[LEM82] and [SUN82] have suggested a technique to remove types 1 and 2 indeterminacy. The conditions under which this technique can be validly applied to protocol automata are considered. The proof of a valid application is illustrated in Appendix B.
The relationship between Type 3 indeterminacy and a fundamental limitation of formal description techniques is discussed. Protocol indeterminacies of Type 3 cannot be removed by any purely mechanical means.

A novel global-state analysis algorithm is developed (Appendix C) to expose the structural faults of a protocol specification, and detailed analyses of the two study protocols are carried out (Appendices D and E). Similar work has been done by [BRAND83]. However, the innovation of this study is to introduce a bandwidth limit on the communication between the distributed components of the protocol, and thus produce a substantial simplification of the algebraic tests which characterise global states. These states correspond to significant protocol conditions.

The influence of the fundamental memory limitations of finite-state automata on protocol specification are explored. The counting ability of a finite-state automaton is fundamentally limited, and as a direct result, the finite-state specification is restricted to problems which require the sorting of stimuli into no more than a specific and finite number of classes. This limitation leads to inflexibility in situations where a degree of adaptive behaviour is desirable:

1 in a particular fault-tolerant network, the master is required to decide to commit after a majority of the slaves have offered.

2 if two protocol partners lose synchronisation they might try to resynchronise by making a number $M$ of handshakes. Each partner must count up to $M$ before abandoning the attempt.

7.4.2 - Indeterminacy, state, and modal logic in Z

7.4.2.1 - Z indeterminacy

The Z notation offers four indeterminacy mechanisms. Two of them (simple choice, non-deterministic substitution) are irrelevant to Types 1, 2 and 3 specification indeterminacy. A third (guarded substitution) can achieve an arbitrary post-condition when the guarding predicate is false,
and one therefore has the added task of carrying out a special test of \textit{feasibility}. The fourth (pre-conditioned choice) preserves the truth value of the substitution in all cases where the pre-condition is true, and guarantees nothing otherwise. The semantics of the pre-conditioned choice seems to be adequate for specifying a protocol sequences.

7.4.2.2 - States in Z

An earlier section argues that the concept of state is fundamental to descriptions of repetitive behaviour. A practical approach is therefore to use the Z features (i.e. static definitions and dynamic substitutions) to mimic the operation of a finite-state automaton. But a Z specification is not fundamentally concerned with states in the strict sense of [MIN72]. In terms of predicate calculus, a state is merely a variable which records some context. A Z specification can therefore contain non-state context variables and define functions over them. In this way, a Z specification can overcome the fundamental memory limitations of a finite-state specification. By defining variables with extended memory (e.g. stacks, queues, etc.) the Z specification can describe adaptive behaviour which eludes a finite-state specification.

7.4.2.3 - Modal logic

Although Z explicitly addresses the dynamic aspect of behaviour, the well-known commentators [ABR87] [SPI89] [WOO89] on Z neglect the need to specify change by means of events which are separated by indefinite time intervals. One might think that a structural property (e.g. the absence of unspecified stimuli, liveness, etc.) can be readily stated statically using a knowledge of (1) the inputs and outputs which are legal in each state, and (2) the state changes which result from the reception of each input in each state. However, to do so, one must enumerate all legal pairs (more generally N-tuples, where the protocol contains N partners) of states by simulating the operation of the protocol. Such a recursive enumeration is essentially the exhaustive global state analysis of Appendices C and D. Clearly, such a laborious approach loses much of the benefit of the succinct expressiveness of the Z predicate calculus.
The problem lies in the unadorned first-order predicate calculus of the standard Z approach. The primed notation \( V \) and \( V' \) describes adjacent instants in the history of the variable \( V \) (i.e. immediately before and immediately after a value substitution). The difficulty is that the significant dynamic properties of protocols involve events which are separated by indefinite time intervals. For example, the correctness of the 2-phase commit protocol requires that all partners arrive at the same conclusion eventually, but there is no obvious indication of how many intermediate signal exchanges will occur. Similarly, the liveness requirement demands that certain events will never occur.

It is possible to model indefinite intervals in the standard Z primed notation by introducing explicit counters of events and by specifying arithmetic relationships between them. However, the use of such counters rapidly becomes more complex than the application which is the main interest of the specification effort.

A solution emerges when one observes that it is unnecessary to record the progression of time explicitly. Rather, it is required to state that events will occur "sometime in the future". That is, the events will be separated by some interval which is unknown but finite. These concepts are very well modeled in the notation known as temporal logic, a variety of modal logic, which is set out in tutorial form in [MAN81]. Temporal logic expresses the time interval between two events as a reachability relation in state-space, rather than as an explicit event sequence. Temporal logic is formally equivalent to first-order predicate calculus, but the benefits (of notation, of expressive power, of analysis ability) over the unadorned calculus are immediately apparent when stating dynamic protocol properties.

7.4.3 - CSP expressiveness, analysis, protocol properties

Protocol properties concerning the legality of component interaction (i.e. liveness, deadlock, correctness) are readily stated using the basic CSP facilities by which processes are constructed (prefix, deterministic choice, etc.) and how they interact (synchronisation on shared events,
sequential composition). However, the semantics of the process interaction mechanisms are surprisingly limiting, a fact which emerges when one attempts to specify context-dependent processing of events.

1. Nondeterministic choice is concerned with implementation freedom, and is therefore of no interest in this connection
2. Interleaving is indeterminate
3. Selective process failure is ruled out by the CSP laws governing deadlock behaviour
4. Parallel composition selects only the behaviour which is common to the composed processes, as opposed to permitting a N-way choice of process behaviour
5. It is not clear that the semantics of angelic nondeterminism are acceptable to an irreversible (e.g. checkpointing) process.

Parallel composition is adequate for describing high-level parallelism (e.g. the concurrent operation of "fairly large" subsystems), but mutual recursion is necessary for closer control over process interaction at lower levels. There is no easy way to compose a number of distinct contexts into a context-dependent process (i.e. to build up a complex recogniser from a number of context-free modules). It is possible that this restriction arises in part from the particular semantics of the protocols considered, but the various CSP facilities for process interaction also appear to be fundamentally unhelpful.

In terms of formal language theory, CSP is capable of recognising a regular expression (Chomsky 3) and the non-backtracking LL(1) subclass of context-free languages (Chomksy 2). But this Chomsky 2 capability relies on the use of angelic nondeterminism [HOA85], and the checkpointing semantics of a protocol are incompatible with the unconstrained backtracking of angelic nondeterminism. Further, it is far from clear that non-backtracking LL(1) semantics is adequate for the very large class of process interactions addressed by CSP.

The discipline of considering (in fairly low-level detail) the event sequences which make up a CSP protocol specification can expose serious errors which other techniques reveal only after considerable
analysis. Chapter 6 contains several examples which demonstrate the impossibility of writing a CSP specification without first clarifying the informal requirements. But note that a CSP event is explicitly free of semantics and its only significant properties are that it is observable and distinguishable from all other events. A CSP analysis always excludes questions of causality, and the order of events never implies cause or effect. (One speculates that this outlook is the root cause of the difficulty described earlier, where the angelic nondeterminism technique is acceptable for analysis purposes, but unacceptable for expressing the protocol semantics.)

Regarding analysis techniques, it is clearly intended that one should analyse a CSP specification by reasoning about consequences, as opposed to enumerating those consequences. For example, in proving that a system of processes is free from deadlock, the essence is to demonstrate that no process interaction leads to STOP. The model proof [HOA85] of the well-known dining philosophers problem does not depend on enumeration, but rather considers in an ad-hoc way the structure of the possible traces, and shows that none contains STOP. Such an approach clearly requires the ad-hoc construction of lemmas, and its success is strongly influenced by one's insight into the particular problem.

Correctness properties (in the sense that event B is a necessary consequence of event A) are also readily stated as CSP events. The use of shared events is a powerful and elegant way to specify the simultaneous achievement of a goal by several protocol components.

In contrast, temporal properties such as liveness and fairness present a serious problem in CSP. [HOA85] argues convincingly that there is no rigorous justification for the commonsense notion that a process which executes for long enough must eventually display all possible behaviour. He concludes that any property (e.g. liveness, fairness) which depends on the eventual achievement of a goal is beyond finite reasoning, and therefore cannot in general form part of a workable (i.e. demonstrable, enforceable) specification. One result of this thesis is to offer an alternative definition of fairness which is compatible with accepted
concepts of termination and finite behaviour, and which brings temporal properties into the scope of a CSP specification.

7.4.3.1 - Liveness, fairness and perverse choice

A CSP specification expresses liveness implicitly by the interaction of processes: one proves the assertion that the trace of the (composite) process which represents the whole system is capable of indefinite extension.

The concept of fairness is very closely related to liveness. Where several correct outcomes are possible, liveness can guarantee that all are capable of occurring, whereas fairness guarantees that the outcomes will occur in a particular discipline. Fairness is of considerable interest when specifying schedulers and similar applications. Termination can be considered a special case of fairness [SCH88].

But fairness raises a serious problem: the liveness of an event does not guarantee that this event will eventually take place. It always remains possible that an event which is live will take place, but the possibility cannot be eliminated that the event may also be delayed indefinitely by other events.

A constructive approach to this problem is to adopt a narrower and justifiable definition of fairness, and to distinguish between perverse choice and interrupted processing. One can adopt a definition which makes successful termination a necessary condition of all conclusions.

*an event A is fairly executed if, whenever particular conditions C arise, event A is the only possible behaviour if any further processing takes place.*

Termination is therefore a necessary condition for fairness. One then attempts to remove perverse choice by paying careful attention to the indeterminism introduced into the specification. Any required fairness must be explicitly stated as process interactions.
7.5 - The need for operational specifications

If one regards a formal specification as the first step towards a correctly implemented system, then one must address the problem that it is possible for a formal specification to describe behaviour which cannot be produced by any known system.

In the strict sense of [BAL79], a pure specification is a description of the observable behaviour required from a system, with no regard to how that behaviour might be achieved in practice. This is the view taken by strict formalists. Thus, [BAU82] proposes a formal notation consisting of abstract data types, predicate calculus, etc. to state requirements which are clear, complete, self-consistent. It is implied in this approach that the above properties are sufficient, and that such a statement can always be transformed into an executable program with all of these properties being demonstrably preserved at each stage of transformation. This view is taken by [H0089] who asserts (without a general demonstration) that a Z specification "could easily be mapped into the MALPAS or SPADE languages" and thus implemented.

More pragmatically, but still very much in support of the formalist camp, [COH89] points out that "... [formal methods] are all eminently misusable. The most appalling rubbish can be ... constructed in every notation ... ".

But an even stronger statement than [COH89] appears to be necessary. If a formal specification is the first step towards a correctly implemented system, then one is forced to conclude that the specification should be operational (in the sense of [ZAV82]) to a large degree.
7.5.1 - Impossible specifications

As an example of impossible behaviour, consider a system whose behaviour results in the following relationship holding between the integers A, B, C and d.

\[ A^d = B^d + C^d \text{ and } d > 2 \]

Eq-7.1

This is the well known last theorem of Fermat, and no solution is known for \( d > 2 \). [MIN72] discusses the question of whether or not a solution can exist, but for our purposes it is sufficient to observe that (1) to date no solution has ever been produced, (2) none is imminent, and yet (3) the question cannot be dismissed as inherently insoluble. For all practical purposes, this specification cannot be satisfied by any known system. Is this a valid specification, in the sense that it contributes to the development of a demonstrably correct program?

Consider the theoretical position. If a specification is demonstrably unsatisfiable, then it can be dismissed for all practical purposes. This occurs when the required behaviour is to solve a problem which is theoretically unsolvable (e.g. to test an assertion which can be neither proved nor disproved), or which is incomputable (i.e. it can be shown that no effective procedure exists). Such a specification corresponds to no possible algorithmic behaviour: no technique is known (and there is good reason to believe that none exists) which will derive from it a correct program.

It is interesting to speculate that an impossible specification (i.e. one which is believed to be unsatisfiable) could serve as an oracle to test the results produced by an allegedly correct program, however produced. Such a procedure (if small enough to be enumerated) could be tested exhaustively, but if it proved to be correct then the assumption of unsatisfiability would be contradicted. It is of course possible that the current theory of unsatisfiability contains some major flaw which could be exposed by the accidental discovery of an existence proof, but this is a highly implausible way of establishing program correctness. In short, an impossible specification is of no practical interest.
Related to the notion of an impossible specification is that of an implicit specification. Whereas an impossible specification has no implementation, an implicit specification is presumed to have some implementation but the specification in no way indicates that implementation. For example, a differential equation precisely specifies a function which satisfies a particular differential equality, but does not indicate how to discover that function (i.e. the solution of the differential equation). In practice, a differential equation is solved ("the specification is implemented") not by any deep analysis of the structure of the equation, but by referring to a well established body of experience which describes the behaviour of various classes of function i.e. a list of known solutions.

In a strict sense, such an implicit specification satisfies the view of [BAL79] that a specification should relate to requirements and should give no consideration to implementation. But this investigation takes the view that the purpose of a formal specification is to facilitate a thorough analysis of possible behaviour and thus lead to a correctly implemented program. It has already been argued that this objective makes the [BAL79] approach inadequate when dealing with impossible specifications. There are similar (but less severe) difficulties in dealing with implicit specifications.

7.5.3 - Operational specifications: avoiding impossible behaviour

The above consideration of impossible and implicit behaviour leads to a significant departure from the naive view that a specification concerns only what is required and not how it can be achieved. One concludes that a valid specification cannot relate to impossible or implicit behaviour.

One observes that the more amenable a technique is to producing an operational specification, the less scope it offers for specifying impossible behaviour. An operational [ZAV82] specification tends to avoid impossible behaviour by obliging the author of the specification to
devise an effective procedure for achieving the required behaviour. It defines the required computation in terms of some idealised computation whose level of detail stops well short of a fully executable program. (For example, the specification of [GOL85] is a collection of high-level algorithms expressed in an implementable notation which permits some degree of abstract description. In this case, the programming language Ada is used.)

7.5.3.1 - Impossible finite-state behaviour

A finite-state specification has a significant operational character. Given a suitable programming style, a finite-state specification is readily converted into an implementable program. Even the extra features of an extended finite-state notation such as Estelle [ISO00] are amenable to automatic implementation. The behaviour described by a finite-state specification can be impossible only to the extent that the specification contains structural inconsistencies (e.g. unspecified receptions, deadlocks, etc.). But these specification faults can be detected using the analyses of Chapter 4, and the resulting corrected specification is clearly valid in the sense of this section.

7.5.3.2 - Impossible CSP behaviour

A CSP specification consists of process interactions which ultimately denote event sequences (i.e. process traces). At first sight this appears to have a significant operational character. Further, [HOA85] defines LISP implementations for the CSP primitives, and it is tempting to conclude that every CSP specification has a corresponding effective procedure. But the author of a CSP specification can choose the semantics of events without constraint, and indeed [HOA85] insists that the only significant properties of an event is that it is observable and distinguishable, and not that it has any particular semantics. One can therefore construct impossible specifications. For example, consider the process SolveFermat.

Eq-7.2 SolveFermat = (A→B→C→D→SKIP)
where $A, B, C$ are the events "reads the value of $A"", "reads the value of $B"", "reads the value of $C"", and $D$ is the event "writes the value of $D$ which solves Fermat...". By earlier arguments this specification cannot be satisfied in practice, and is therefore invalid in the sense of this section.

7.5.3.3 - Impossible Z behaviour

A Z specification may contain two sources of impossible behaviour: a miraculous substitution, and an unsatisfiable constraint. The miraculous substitution (by which a guarded substitution with a false pre-condition can achieve an arbitrary post-condition) can be eliminated

1. by avoiding the use of the guarded substitution construct, as described in chapter 5, or

2. by proving the feasibility (in the special Z sense of the term) of every guarded substitution

But it is still possible to state a static invariant which is incapable of being realised, since the invariant may be any predicate over the basic sets of objects in the Z specification. For example, the invariant may constrain the system to solve Fermat's equation. This investigation has adopted the particular solution of using the pre-conditioned choice construct to mimic the operational transitions of a finite-state specification, and has thus imposed an operational character on the Z specification. But a Z specification in general may relate to impossible behaviour and may therefore be invalid in the sense of this section.

7.5.4 - Implementing implicit behaviour

The specification techniques considered offer different degrees of assistance in finding an acceptable implementation. A finite-state specification is readily implemented. A Z specification can usually be refined [MOR85] (a procedure which is largely unsystematic) into an equivalent implementation. A CSP specification is claimed [HOA85] to be directly implementable in LISP, but practical difficulties [FID88] [LOG88] arise, particularly in connection with recursion depth and the
management of backtracking, and it is likely that a direct CSP implementation would be acceptable only as a disposable prototype.

7.6 - Modelling the environment: functional behaviour

Behaviour is conventionally [YEH82] divided into functional and non-functional. Functional behaviour states some interaction between a system and its operating environment [ROM85]. A specification can model the functional behaviour in its full generality only if it can describe flexibly the environment. Note also that in some common applications (e.g. embedded systems) it is unreasonable to expect the environment to remain constant [FER88]. That is, the operation of the system alters the environment. But in practice one observes that formal techniques adopt the narrower mathematical meaning of function:

*functional behaviour is that which can be described as a mapping from an unchanging input domain to an unchanging output domain.*

A finite-state specification is explicitly characterised by the transition and the response function. A Z specification is stated in terms of well established mathematical objects which invariably have the character of a mathematical function or set. A CSP process has a well-defined input and output domains (the alphabet of permissible events), while the operation of the process maps the alphabet onto an event sequence (the trace).

The question arises: to what extent does this mathematical definition of function restrict the ability of a specification technique to describe functional behaviour in the broader sense? In particular, how well do formal specification techniques model the operating environment of a communication protocol?

A finite-state specification is fairly inflexible in modelling the operating environment. In the particular context of a distributed communication protocol, the finite-state technique of [SUN82] and [LEM82] results in
the environment manifesting itself as a set of external stimuli in addition to the internal stimuli which arise from the components of the system proper. However, at this level of abstraction the state structure of the environment is unknown and there is no possibility of any feedback to describe how the system influences its environment. One should therefore attempt to model the operating environment (at some suitable level of abstraction) as another state machine. The complete specification would then consist of two components (i.e. the system proper, and its operating environment), while the interface between the two is defined by the external stimuli of [LEM82] [SUN82]. Thus, the use of N coupled state machines to model a physically distributed system can also be applied to a system and its operating environment.

A Z specification does not differ significantly from a state machine in the modelling of an operating environment. As described in Chapter 5, the protocol stimuli manifest themselves as choice pre-conditions. One merges the distinct Z descriptions of the system and its environment as follows:

1. one appends the two lists of pre-conditioned substitutions, and
2. one conjoins the two static invariants

A CSP specification readily models the operating environment as a distinct process which is then parallel-composed with the process specifying the system proper. This seamless approach is very satisfying to use, but it appears not to differ significantly in analysis power from the finite-state (extended as above) and Z facilities.

7.7 - Some observations on formal specification techniques

7.7.1 - The value of formality

It is pertinent to consider in fundamental terms the benefits one can expect from a formal specification.

The principal benefit of any formal approach is that the resulting specification is amenable to rigorous analysis. A
formal technique has the potential to establish demonstrably the validity of a conclusion.

In considering what conclusions one can draw from a rigorous analysis, one must distinguish between a valid conclusion and a sound conclusion. A conclusion is valid if it has been derived in accordance with the laws of the formal system in which the specification is written. A conclusion is sound if it is both valid and based on true premises. A consequence of this distinction is that it is possible to draw a conclusion which is valid but false. For a conclusion to be demonstrably sound, the premises must be absolutely beyond doubt. Such premises are rare indeed. The philosophical aspects of knowledge in this context of mathematical program validation are discussed by [FET88].

This argument is well known to careful authors (e.g. [REV85]) who point out that there can be no demonstration that a particular informal notion is adequately expressed by a particular formal description. Therefore, quite separate from the practical difficulties of carrying out the formal analysis, one can have only as much confidence in the results of a program validation or the formal analysis of a specification as one has in the premises.

A formal specification technique is a powerful tool in the construction of a valid argument, and the formal approach deserves to be favourably considered. But one should beware of an unjustified degree of confidence, which may come about in several ways:

1 the interesting issues in protocol specification can be very difficult to formalise. Section 7.6 argues that formal specification techniques are primarily concerned with functional mappings. However, other sections argue that some significant concepts are very difficult to state in such terms: non-structural properties (section 4.3.2), temporal and fairness properties (section 6.3.2), non-functional requirements (section 2.4.3).
the practical and theoretical limitations of the formal analysis technique might not be well understood. Earlier sections have illustrated the practical difficulties of analysis (e.g. systematic global state generation, the difficult formulation of temporal properties) and the opportunities for theoretical misinterpretation (e.g. the logical incompleteness of predicate calculi, the difficult semantics of some CSP concepts).

insufficient attention might be paid to the difficult (and essentially informal) task of considering the truth of the premises. The formal requirements define the purpose of the system only as well as that purpose can be articulated by all concerned with the specification. The study of CSP (and to lesser degrees, Z and finite-state) demonstrates that a formal statement may readily subsume several informal assumptions which determine the semantics of the statement. Typical cases include the effects of (re)initialisation, and the persistence of once-valid conditions. A fully analysed formal specification may be correct (i.e. logically valid) but need not deliver the desired behaviour (i.e. unsound).

7.7.2 - Instantaneous events, time resolution

The formal techniques of specification considered in this investigation assume that significant events are instantaneous and atomic. It is meaningless to contemplate a finite-state machine partway through a transition, while both a Z substitution and a CSP event are explicitly defined to be instantaneous.

But in general, the possibility cannot be excluded that two distinct events in two distinct subsystems will occur simultaneously. However, none of the considered specification techniques is capable of analysing this situation: all insist on ordering simultaneous events into some arbitrary sequence.
This is of importance where the specified system consists of several independent but co-operating component. Arbitrary behaviour arises in two circumstances:

1. where the independent subsystems execute at different speeds
2. where two or more simultaneous events occur in the overall system

The overall system behaviour depends on the relative execution speeds of the independent subsystems: when their combined behaviour is interleaved into a single sequence, that sequence is arbitrary and the resulting behaviour is indeterminate.

A radical departure from the above approaches is to deny the possibility of atomic and instantaneous events. Thus, the FOREST specification language [GOL87] holds that the time of occurrence of an event can at best be confined to a finite interval but never pinpointed to a precise instant. The occurrence interval of a FOREST event is effectively an instant if the interval is shorter than the system's shortest reaction time. The physical system being specified cannot resolve certain events, and the specification therefore loses no information by imposing arbitrary sequencing.

CSP acknowledges this reality by suggesting as a model an extended event consisting of a pair of instantaneous events. The non-instantaneous interval of a real event is discussed briefly by [MIL80], but even the very general CCS insists on ordering events into some sequence and excludes the possibility of a system supporting two simultaneous observations.

It is a common restriction of formal specification techniques to date that genuinely simultaneous behaviour is treated by the approximate method of imposing arbitrary sequencing. A general solution is not imminent.
7.7.3 - Non-functional behaviour

Perhaps the biggest limitation on the useful application of formal specification techniques arises from a common inability to deal with non-functional behaviour.

The functional aspects of a specification prescribe the behaviour which must take place, while the non-functional aspects impose constraints on that behaviour [ROM85]. For example, "the autolander must not cause the plane to bounce unduly on landing". These constraints can arise from any aspect of the operation of the system [ROM85] [WHI85]: the character of interfaces, design restrictions to be observed by the implementation, performance efficiency, failure responses.

It has been argued earlier that the specification techniques considered in this study are all predominantly functional: they relate input quantities to output quantities. A very few non-functional properties (e.g. fairness, liveness) can be stated in structural terms and analysed by functional techniques, but the full generality of non-functional behaviour appears to be unmanageable.

But before dismissing the formal functional specification approach as fundamentally inadequate, observe that a number of important non-functional requirements fall into one of two categories:

1. those with no generally accepted definition
2. those which might be based on measurable quantities, but whose assessment is ultimately subjective

An example of the first category is provided by reliability. Although reliability is in principle an objective measure, it is in practice not a straightforward application of probability. In fact, it depends crucially on the viewpoint of the assessor. See [VEE87] for a discussion of several workable definitions.

An example of the second category is provided by requiring an interface to be user-friendly, or a system to be maintainable. A common approach to this sort of problem is to distinguish between the objectively
measurable attributes and the subjective quality to which they give rise. The attributes are measured by objective metrics, most of which fall into one of two classes: the structural metrics which yield some measure of graphical complexity (e.g. the cyclomatic number, [MCC76]), and Halstead-type metrics [HAL77] which purport to measure software properties by counting the various syntactic components of the software. See [INC88] for a comprehensive review of this area.

While these metrics are undeniably valuable in quantifying some aspect of the software, there is no general theory for interpreting the resulting numbers and the assessment of the quality remains subjective and highly context dependent: a particular set of attribute values might be acceptable in one case and unacceptable in another.

It is fair to conclude that some of the more common non-functional requirements are imprecise catch-alls for behaviour which cannot (currently) be described adequately by any means. The test for a catch-all is to observe that, given an implementation, it cannot decisively settle a dispute as to whether or not the requirement of interest has been satisfied. Such statements have no place in any specification whose properties are expected to be demonstrable.
Chapter 8 - Further research

8.1 - Non-functional behaviour

A major limitation on the useful application of specification techniques (formal and otherwise) arises from a common inability to deal with non-functional behaviour. This investigation concludes that formal description techniques necessarily depend on (1) the use of well-defined and readily measured quantities, and (2) on the construction of input/output mappings (i.e. functions in the mathematical sense). Further work is needed to facilitate the application of formal specification techniques to non-functional behaviour. There are two principal problem areas: (1) the objective measurement of non-functional behaviour, and (2) the application of mathematical mappings in the description and analysis of behaviour.

Non-functional behaviour is not readily formalised by objectively measurable metrics. One conventionally characterises this distinction by saying that objectively measurable attributes contribute to the presence or absence of a quality, but the assessment of that quality is predominantly subjective. There is a well developed body of knowledge on software metrics i.e. the objective measurement of attributes. However, much of the work in software metrics relates to executable code rather than to more abstract problem representations which could serve as a specification. Further, there is no general theory which convincingly relates attributes to qualities. Without such a theory (at least, for a subset of the qualities of frequent interest) any formal description technique is fundamentally limited in its approach to non-functional behaviour.

It is argued in this thesis that current formal description techniques are at root functional in the sense that they depend on mathematical mappings. One speculates that there exists behaviour which is inherently non-functional in this sense. If such behaviour is indeed found, then how can one adapt the existing specification techniques to express these requirements?
8.2 - Stating and analysing criticality levels

A common argument is that the use of formal specification techniques is justified in the area of safety-critical systems: a formal description at the specification stage offers considerable potential for an accurate and comprehensive assessment of the safe circumstances of use. However, a realistic treatment of system safety must address two problem areas: (1) the modeling of the operating environment, and (2) non-functional behaviour. This investigation has shown that these problems are very difficult.

One can approach this problem in a piecemeal fashion by adopting the approach of many of the statutory licensing authorities (e.g. the Health and Safety Executive, the Civil Aviation Authority, the Nuclear Installations Inspectorate) which are required to monitor the conduct of safety-critical industrial activities. These authorities recognise serious difficulties in assessing safety hazards and in enforcing safe practices in very varied industrial environments. An important technique for taking account of wide variations in industrial practice is to base safety assessments on levels of criticality: one considers the potential damage a system might cause, and then allocates priority of verification effort accordingly. The higher the level of criticality, the more trustworthy must be the system.

However, this investigation has discovered no mechanism in current formal techniques to indicate the importance of a requirement i.e. to state levels of criticality. All terms of a formal specification appear to have equal priority, and there is no analysis technique which can be biased towards a subset of the specification terms.

The use of criticality levels is a widely accepted and well proven approach to system safety in realistic circumstances. Work is needed to investigate the use of a formal specification technique in this way.
Current formal description techniques assume that the behaviour of interest is essentially deterministic: there may well be considerable difficulty in calculating the outcome of a given set of circumstances, but it is assumed that a unique solution always exists. Even in their treatment of indeterminism, the formal techniques assume that the effect of all possible variations of behaviour is known, although it may be impractical to enumerate those variations.

However, a probabilistic description of behaviour is widely accepted in many practical situations. Probability may be used to describe inherently non-deterministic behaviour (e.g. quantum mechanical effects) or to quantify the uncertainty in macroscopic systems. The non-mechanistic view of system behaviour is well established on both philosophical and on practical grounds, but current formal techniques invariably adopt a mechanistic approach which makes no provision for uncertainty.

This investigation considers this issue of uncertainty in the context of temporal aspects of behaviour, and concludes that a substantial difference of approach is responsible for the uninformative debate on CSP fairness. One side [DIJ88] advances the essentially mechanistic argument that sound analysis can be carried out only on behaviour which is known to take place. The other side [SCH88], while still strongly in favour of a formal approach, is willing to draw conclusions based on behaviour which might take place.

The confidence of many formalists in the mechanistic approach is incomprehensible to practising statisticians (e.g. [VEE87]) who devote their efforts to calculating reliability and similar uncertain quantities. Some basic research is needed to investigate the application of formal description techniques to systems which have traditionally been amenable only to a statistical approach. However, this research should be directed at devising hybrid analysis techniques, and not at trying to reinstate mechanistic beliefs.
8.4 - Graph-theoretic analysis

An informal statement of state-space fairness is that some particular global states are achieved in one or more *fair* sequences. Each such sequence is a traversal of the global reachability tree from root to a unique leaf, and the set of all such tree paths is the combinatorial enumeration of all fair behaviour. The demonstration of state-space fairness therefore requires an exhaustive examination of behaviour.

This investigation has illustrated the practical difficulty of the global-state analysis of a finite-state specification, and has speculated that a less laborious analysis would be possible on a global reachability graph (as opposed to the unfolded tree), where liveness would be expressed by simple connectivity, while fairness might be formulated in terms of flows and cuts.

Some effort is needed to investigate how a global state-space analysis might be conducted in terms of compact graphs, as opposed to completely unfolded trees. The immediate difficulty is how to derive a graph which represents the merged behaviour of two or more distinct graphs.

8.5 - Development of automatic tools

It has become clear during this investigation that the use of formal techniques gives rise to much tedious and error prone manipulation. It is attractive to develop automatic tools to carry this burden.

The automation of state-space techniques has already been considered in detail and shown to be practical, although the reporting of detailed results (e.g. a large reachability tree) needs further work. The use of CSP raises the need for automatic tools to handle the calculation of a parallel composition and the enumeration of a process trace. It appears to need little more than a syntax-directed editor to ease the difficulty of constructing a Z specification, but the analysis of the specification requires sophisticated theorem-proving facilities.
The development of automatic tools is not strictly a research activity in the sense of extending the scope of formal techniques, but the heavy manipulation burden makes these techniques impractical without machine assistance.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s) &amp; Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABR87</td>
<td>J. Abrial, C. Morgan, &quot;Formal software specification and provable design&quot;, EWICS TC7 seminar, 4-7/May/1987</td>
</tr>
<tr>
<td>AMB77</td>
<td>A. Ambler et. al., &quot;GYPSY: a language for the specification and implementation of verifiable programs&quot;, SIGPLAN Notices, V12, N3, March 1977</td>
</tr>
<tr>
<td>ANDER78</td>
<td>T. Anderson, R. Witty, &quot;Safe programming&quot;, BIT, V18, pp.1-8</td>
</tr>
</tbody>
</table>
BAU82  F.Bauer, "From specification to machine code: program construction through formal reasoning", Proc. ICSE 6, Tokyo, Sept 1982

BER84  D.Berry et. al., "On the requirements for a program design language", Proc. Internat. workshop on models and languages for software specification and design, ed. R.Babb, IEEE Comp. Soc., 1984, Orlando


BOR85  A.Borgida, S.Greenspan, J.Mylopoulos, "Knowledge representation as the basis for requirements specification", Computer, V18, N4, April 1985


BRAND83  D.Brand, P.Zafiropulo, "On communicating finite-state machines", JACM, V30, N2, April 1983

BRI87  E.Bricksma, "A tutorial on LOTOS", in "Proc. IFIP workshop on protocol specification, testing and verification", ed. M.Diaz, pp.73-84, North-Holland, 1987

CAM74  R. Campbell, A.Habermann, "The specification of process synchronisation by path expressions", Lecture Notes on Computer Science, V16, Springer Verlag, 1974
CHAD85  M.Chandrasekharan, B.Dasarathy, Z.Kishimoto, "Requirements-based testing of real-time systems: modeling for testability", Computer, V18, N4, April 1985

CHA78  B.Chaudhary, H.Sahasrabuddhe, "Suggestions about a specification technique", SIGPLAN Notices, V13, N12, December 1978


CUL83  K.Culik, "On formal and informal proofs for program correctness", SIGPLAN Notices, V18, N1, Jan 1983


DAV80  A.Davis, "Automating the requirements phase: benefits to the later phases of the software development life cycle", Proc. COMPSAC, 1980

DEM79  R.DeMillo, "Social processes and proofs of theorems and programs", CACM, V22, N5, May 1979

DEN84  C.Denham, "A systematic approach to semantic specification and implementation of non-procedural languages", Proc. Internat. workshop on models and
languages for software specification and design, ed. R.Babb, IEEE Comp. Soc., 1984, Orlando


DIJ88 E.Dijkstra, "A position paper on fairness", ACM SE Notes, V13,N2, Apr 1988


FET88 J.Fetzer, "Program validation: the very idea", CACM, V31, N9, Sept 1988


GER75a S.Gerhart, "Knowledge about programs: a model and case study", Proc. internat. conf. on reliable software, April 1975, Los Angeles, pp.88-95

GER75b S.Gerhart, L.Yelowitz, "Observations of fallibility in applications of modern programming methodologies", IEEE trans. Comp., C-24, n5, May 1975


GOL87 S.Goldsack, "Specifying requirements: the FOREST approach", Dept. of Computing, Imperial College of Science and Technology, London, July 1987
<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOOD83</td>
<td>A.Goodall</td>
<td>&quot;Programming in logic&quot;</td>
<td>Computer Systems, July 1983, pp.61-63</td>
</tr>
<tr>
<td>GRE84</td>
<td>G.Greiter</td>
<td>&quot;acce-Specifications: a formal method to define data types&quot;</td>
<td>SIGPLAN Notices, V19, N10, October 1984</td>
</tr>
<tr>
<td>GYR84</td>
<td>W.Gyre</td>
<td>&quot;Expectations from a systems specification language&quot;</td>
<td>Proc. Internat. workshop on models and languages for software specification and design, ed. R.Babb, IEEE Comp. Soc., 1984, Orlando</td>
</tr>
</tbody>
</table>
HAL77  M.Halstead, "Elements of Software Science", Elsevier North-Holland, 1977


HEN83  P.Henderson, "Functional programming: applications and implementation", Prentice Hall series in computer science, editor C.Hoare, 1983


HOWA76  J.Howard, "Proving monitors", CACM, V1, N5, May 1976

HOW87  W.Howden, "Functional program testing and analysis", McGraw Hill, 1987, ISBN 0 07 030550 1
D.Ince, "MUSE metrics reference survey", working paper, MUSE Q/R project, Esprit project 1257

Information processing systems - open systems interconnection, "Estelle - a formal description technique based on an extended state transition model", ISO 9074


R.Jacob, "Using formal specifications in the design of a human-computer interface", CACM, V26, N4, April 1983


KAR84  R. Karp, "Proving failure-free properties of concurrent systems using temporal logic", ACM TRANSPLAN, V6, N2, April 1984


LAM79  L. Lamport, "On the formal proof of a calendar program", CACM, V22, N10, Oct 1979


LEW76  P. Lewis, D. Rosenkrantz, R. Stearns, "Compiler design theory", Addison-Wesley, 1976, ISBN 0 201 14455 7

LIS74  B. Liskov, S. Zilles, "Programming with abstract data types", SIGPLAN Notices, V9, N4, April 1974


LIS77  B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, "Abstraction mechanisms in Clu", CACM, V20, N8, August 1977


LOG88  I. Logrippo, A. Ohaid, J. P. Briand, M. C. Fehri, "An interpreter for LOTOS, a specification language for
distributed systems", Software Pract. & Exper., V18, N4, April 1988


MAJ77 M.Majster, "Limits of the algebraic specification of data types", SIGPLAN Notices, V12, N10, Oct 1977

MAN75 Z.Manna, R.Waldinger, "Reasoning and knowledge in program synthesis", Artificial Intelligence, 1975, V6, N2


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIL80</td>
<td>R. Milner</td>
<td>&quot;A calculus of communicating systems&quot;</td>
<td>Lecture Notes in Computer Science 92, Springer Verlag, 1980</td>
</tr>
<tr>
<td>NAU66</td>
<td>P. Naur</td>
<td>&quot;Proof of algorithms by general snapshots&quot;</td>
<td>BIT, V6, N4, 1966</td>
</tr>
<tr>
<td>NAU69</td>
<td>P. Naur</td>
<td>&quot;Programming by action clusters&quot;</td>
<td>BIT, V9, N3, 1969, pp.250-258</td>
</tr>
<tr>
<td>PAR77</td>
<td>D. Parnas</td>
<td>&quot;The use of precise specifications in the development of software&quot;</td>
<td>IFIP Congress, Toronto, 1977</td>
</tr>
<tr>
<td>Reference</td>
<td>Author(s)</td>
<td>Title</td>
<td>Publication Details</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>---------------------</td>
</tr>
<tr>
<td>PET77</td>
<td>J. Peterson</td>
<td>&quot;Petri nets&quot;</td>
<td>Computing Surveys, V9, N3, Sept 1977</td>
</tr>
<tr>
<td>POT84</td>
<td>C. Potts</td>
<td>&quot;Understanding specifications&quot;</td>
<td>Proc. Internat. workshop on models and languages for software specification and design, ed. R. Babb, IEEE Comp. Soc., 1984, Orlando</td>
</tr>
<tr>
<td>REN82</td>
<td>T. Rentsch</td>
<td>&quot;Object oriented programming&quot;</td>
<td>SIGPLAN Notices, V17, N9, Sept 1982</td>
</tr>
<tr>
<td>REV85</td>
<td>G. Revesz</td>
<td>&quot;Introduction to formal languages&quot;</td>
<td>McGraw Hill, 1985, ISBN 0 07 051916 1</td>
</tr>
<tr>
<td>ROB77</td>
<td>L. Robinson, K. Levitt</td>
<td>&quot;Proof techniques for hierarchically structured programs&quot;</td>
<td>CACM, V20, N4, April 1977</td>
</tr>
<tr>
<td>ROM85</td>
<td>G. Roman</td>
<td>&quot;A taxonomy of current issues in requirements engineering&quot;</td>
<td>Computer, V18, N4, April 1985</td>
</tr>
</tbody>
</table>

SCH88 F. Schneider, L. Lamport, "Another position paper on fairness", ACM SE Notes, V13, N3, Jul 1988

SCO76 D. Scott, "Data types as lattices", SIAM J. Comp., N5, 1976


W.Swartout, R.Balzer, "On the inevitable intertwining of specification and implementation", CACM, V25, N7, July 1982


J.Thompson, "The use and abuse of formal proofs", SIGPLAN Notices, V18, N7, July 1983


T.Winograd, "Beyond programming languages", CACM, V22, N7, July 1979

S.White, J.Lavi, "Embedded computer system requirements workshop", Computer, V18, N4, April 1985


Appendix A

Computability, languages and automata

A1 - Computability and the Church-Turing thesis

A definition of *computability* is the outcome of a theory of algorithms, i.e. the set of problems which can be solved within very broad but physically realisable constraints. Such problems are known as the *computable functions* or *effective procedures*.

First, some definition of terms is needed.

1. A *mathematical method* is a *procedure* if it consists only of previously defined operations which are performed in a well-defined order and which are adequate to deal with every situation which can possibly arise. In short, a procedure can be performed fully mechanically.

2. An *algorithm* is a procedure which terminates after a finite time.

3. A problem which is formulated so as to require only a *yes* or *no* answer is called a *decision problem*. E.g. "is X a member of the set Y?"

4. A decision problem is said to be "*[algorithmically]* decidable" if there exists a [algorithm] procedure which solves it.

There are several equivalent formulations of computability (due to Post, Kleene, Church, Turing) but all share the idea that an effective procedure must be finite.

1. A procedure has a finite description
2. A procedure consists of a sequence of discrete steps
3. A procedure terminates after a finite number of steps
Historically, the developments in this field came from several sources: automaton theory, formal languages, recursive functions. The results from these sources were eventually shown to be mutually equivalent.

The work of Turing [DEN78] resulted in a finite description of a very general-purpose problem solving automaton ("the Turing machine"). This formalised the intuitive notion of a procedure. For problems where it was demonstrated that no such procedure could exist, this undecidability demonstrated the existence of fundamental limits on the problem-solving ability of realisable solutions. For problems where it was demonstrated that no procedure could terminate, this demonstrated the non-existence of an algorithmic solution.

An equivalent line of argument comes from the field of formal languages. A formal language is the set of strings ("sentences") generated by a formal grammar. A formal grammar (sometimes known as "generative") consists of
1 a finite set of primitive symbols P (an "alphabet")
2 a finite set of non-primitive symbols N, including one "distinguished symbol" S
3 a finite set of "productions" or rewriting rules which are successively applied to S to generate symbol strings containing only P symbols

The complete set of strings of primitive symbols generated by a grammar G is the language L which corresponds to G. A language is said to be finite if and only if it contains a finite number of such strings (or "sentences").

Chomsky [REV85] classified grammars on the basis of their productions, and stated a grammatical hierarchy of decreasing generality:

type 0 completely unrestricted productions
type 1 context-sensitive productions
type 2 context-free productions
type 3 regular, or finite-state
Church [REV85] defined a class of "recursively enumerable" languages which was shown to be identical to Chomsky's Type 0 languages and which demonstrated that the membership problem (ie. "is sentence X contained in language Y?") for Type 1 (and lower) languages is algorithmically decidable, while for Type 0 it is algorithmically undecidable.

The equivalence of the above arguments in formal languages and automata is stated in the Church-Turing thesis: "the computing power of the Turing machine represents a fundamental limit on the capability of realisable computing machines".

The significance of the above results is that many problems can be reformulated either as membership problems of formal languages or as decision problems. One can then refer to the results from the formal language work (or equivalently, the capabilities of the appropriate automaton) to detect undecidability or the absence of algorithmic solutions.

A2 - Chomsky types and their corresponding automata

Computability is defined by describing in abstract and rigorous terms several classes of problems. All problems in a given class have the same degree of computational difficulty. A problem class is defined by the Chomsky classification of the statement of the problem: all problems in class i are expressible as sentences of a Chomsky Type i language.

Equivalently, the problem class can be defined by the generalised automaton needed to solve every problem in that class. Consider an automaton as a decision procedure which produces either a "yes" or "no" answer when presented with an arbitrary symbol string. The automaton is said to recognise the language which consists of the set of strings which result in the answer "yes". A generalised automaton is said to solve all problems in a particular class i if it can recognise all languages of Chomsky Type i.
[REV85] demonstrates that the generalised automata which correspond respectively to Chomsky types 0-3 are: the Turing machine, the linear-bounded 2-pushdown automaton, the pushdown automaton, and the finite-state automaton.

**A3 - Finite automata**

The significance of the finite automata results from the work of Kleene [DEN78] on "regular expressions". A regular expression is defined over a finite set of symbols S using the operations of iteration (*), concatenation (;) and set union (\(\sqcup\)). Brackets are used to indicate the binding of operators.

1. the empty string is a regular expression
2. every member of S is a regular expression
3. if A is a regular expression over S, then so is \((A)^*\)
4. if A and B are regular expressions over S, then so is \((A);(B)\)
5. if A and B are regular expressions over S, then so is \((A)\sqcup(B)\)

Distinguish between the string of symbols which make up the regular expression and the set of strings which the expression describes: the former is the regular expression, while the latter is the regular set. A regular set is a language and the regular expression is said to denote that language. Kleene demonstrated that every Type 3 language is denoted by a regular expression, and conversely every regular expression denotes a Type 3 language.

[REV85] demonstrates that every Type 3 language is recognised by some finite automaton. (The converse - that every finite automaton generates a Type 3 language - is also true, but it involves a minor complication concerning determinism, and is irrelevant to this investigation.)

The notion of "state" is very useful in capturing the quality of finiteness. A comprehensive account of state automata can be found in [MIN72].

Consider a machine M as a black box with an input channel I (from which M receives signals) and an output channel O (to which M outputs
The number of distinguishable signals on both \( I \) and \( O \) is finite although arbitrarily large.

\( M \) operates in discrete time. Its input and output signals \( I(i) \) and \( O(i) \) are defined at particular instances \( T(i) \) and at no other times. This corresponds to the notion that an effectively computable function proceeds in a finite number of discrete steps rather than in a continuum of infinitely small steps.

The output signal \( O(i+1) \) depends only on the input signal \( I(i) \) received immediately prior and on the history \( H(i) \) of \( M \). \( H(i) \) is the sequence of \( M \)'s input and output signals up to and including time \( T(i) \). The delay of 1 instant between receiving \( I(i) \) and producing \( O(i+1) \) corresponds to the constraint that no physical calculation can respond instantly.

\( M \) is now finite in the number of distinct input and output signals, but the history \( H(i) \) is still unbounded. A physically realisable calculation cannot be expected to take account of the arbitrarily remote past. To bound \( H(i) \) we define "equivalent histories". Consider two identical copies \( M_1 \) and \( M_2 \) of machine \( M \), with respective histories \( H_1(i) \) and \( H_2(i) \). These histories are equivalent if (and only if) for every subsequent input sequence \( I(i+1)...I(i+j) \) applied both to \( M_1 \) and \( M_2 \), the two copies produce the same output sequence \( O(i+2)...O(i+1+j) \). This procedure defines an equivalence relation which partitions the set of all possible histories into mutually disjoint classes. Each such class is known as an "internal state" (or simply "state") of \( M \). The member histories of a particular state cannot be distinguished by any possible input sequence. As each new input signal is received to increment the history, the state of \( M \) may or may not change. \( M \) no longer records its whole history, but merely those changes of state which change its output behaviour.

Finally, we bound the number of states of the automaton to arrive at the most restricted definition of computability in common use, the finite-state machine, defined by
1. a finite (but arbitrarily large) set of input symbols $I$
2. a finite (but arbitrarily large) set of output symbols $O$
3. a finite (but arbitrarily large) set of internal states $S$
4. a response function $R: I \times S \rightarrow O$
5. a state transition function $T: I \times S \rightarrow S$
Appendix B

Validity of transformation on finite-state automata

This section argues that the transformation carried out on the finite-state specification of Fig.4-1 of chapter 4 to produce that of Fig.4-3 preserves the formal structure of the original specification.

DEFINITION

only the behaviour which is observable to the remote partner is of interest. Fig.4-1 and 4-3 are equivalent if and only if each protocol partner continues to exhibit the same observable behaviour after the transformation of Fig.4-1 to Fig.4.3.

Observe that the transformation procedure has two steps:

1 in each component the "-" input is replaced by an explicit set of application-derived signals. These signals by definition come from the local application and not from the remote component

2 the expansion of a state containing an ambiguous exit transition is of the form illustrated below. X is a signal from the remote partner while Y and Z are application-derived signals. The new state does not introduce any new output signals, and it continues to generate all previous output signals in exactly the same circumstances as does the untransformed state. The set of output signals is unchanged

The transformation is illustrated below. Fig.B-1 becomes Fig.B-2.
A transformed component neither expects nor receives any new signals from the remote component. It follows that this transformation does not alter the externally observable behaviour of the transformed component. Fig. 4-1 and 4-3 are therefore equivalent.

Although not shown formally here, the changes introduced by the above transformation are unobservable in the strict sense of automata theory: there exists no sequence of stimuli from the remote component which can demonstrate the existence of the new state. The introduction of unobservable transformations is analogous to the use of slack variables in a linear programming problem: the essential problem does not change, but the analysis becomes tractable.
Global state analysis of a 2-partner protocol

C1 - Restrictions on method

1 this treatment considers only 2-partner protocols. The general case of N partners is ignored. The extension to N partners is theoretically no more difficult, but results in a considerable increase in program complexity.

2 this treatment considers only communications protocols. In this application the operation to remove Type 2 indeterminacies has an obvious physical interpretation.

3 the message channels linking the protocol partners have a bandwidth of 1 ie. there may be at most 1 unprocessed message at any time.

C2 - Definition of a local protocol partner

A protocol partner is defined by a finite-state automaton, which is:

1 a set \( S \) of internal states
2 a set \( I \) of input signals from the remote partner
3 a set \( AI \) of input signals from the local application
4 a set \( O \) of output signals to the remote partner
5 the initial state \( IS \)
6 the state transition function \( T: (I+AI) \times S \rightarrow S \)
7 the output function \( R: (I+AI) \times S \rightarrow O \)

The 2 partners are linked by 2 simplex message channels, one channel in each direction. The channels are represented by a 2x2 matrix which records the current (unprocessed) contents of each channel. Element \( i,j \) represents the message sent by partner \( i \) and not yet processed by partner \( j \). Element \( i,i \) represents an unprocessed message from the \( AI \) set into partner \( i \).
C3 - A global state

A global state crystallises a particular stage in the conduct of a protocol exchange. The global state consists of

1. the instantaneous local state A of partner 1
2. the instantaneous local state B of partner 2
3. the instantaneous contents of the 2x2 channel matrix

The set of all global states (together with their associated reachability tree) describes every possible combination of events which can occur during any protocol exchange.

The global reachability (or "dependency" or "precedence") tree simply links by a single directed arc (ie. "makes adjacent") those global states SS and SSS where SSS can be derived from SS as the result of a single instantaneous protocol event in a single partner. The local states of SS (say SS1 and SS2) can differ from those of SSS (say SSS1 and SSS2) in at most 1 value: either SS1=SSS1, or SS2=SSS2, or both.

A transition between two global states occurs in two circumstances:

1. one or other partner scans its unprocessed input channels (note that the input channels to partner i are represented by column i of the channel matrix), extracts exactly 1 signal, and undergoes a local transition as a result of receiving that signal. Note carefully that this procedure does not fix the order of scanning the available inputs: the fully enumerated reachability tree will contain all possible orders of scanning

2. one or other partners undergoes a "delta" transition. This event is described fully in the next section

C4 - A special case - the delta transition

An automaton may receive signals from the AI set. From the protocol's viewpoint these signals appear from nowhere since the behaviour of the local application is nowhere explicitly stated, and therefore it cannot be predicted when the AI signals will be generated.
To make explicit the arrival of these AI signals one adopts a device from [DEN78] - the "delta" transition.

The delta transition has no input stimulus but it always delivers an AI signal into the channel matrix. These signals by convention always appear in the diagonal elements of the matrix.

It must be emphasised that a delta is not a real local transition (ie. is not the response of the automaton to a real protocol event). Its sole purpose is to make explicit the insertion of AI signals into the channel matrix.

C5 - Global enumeration

The aim of this process is to produce the complete reachability tree of global states. This tree (and in particular the leaves) can then be examined to detect undesirable combinations of local events. The combinations of interest are those which indicate unforeseen cases of uncoordination between the protocol partners.

The procedure for enumerating the global reachability tree has three stages. The first two deal with the removal of indeterminacies of Types 1 and 2. The third stage calculates the reachability tree.

From the complete tree it is possible to detect certain rudimentary "structural" faults (eg. no provision for the arrival of a particular signal). It is also possible to inspect the non-faulty situations to ensure that they are semantically acceptable. This inspection phase (which, by an earlier argument, cannot in general be automated) addresses those indeterminacies which have earlier been characterised as Type 3.
C6 - Enumerating the global reachability tree

Step 1 - set up the initial global state

Construct the initial global state from the initial local states IS of each protocol partner. This analysis assumes that the the channel matrix is initially empty, although this is not necessarily the case.

A global state always has a defined status: NEW, OLD, STOPPED.
1 when a global state is first created (ie. by initialisation, or by derivation from an earlier global state) its status is NEW. In this status, further global states can be derived from it. When all (zero or non-zero) permissible global derivatives have been found its status becomes OLD
2 when an attempt fails (see later) to derive any new global states from a NEW state, its status becomes STOPPED and a reason for STOPping is also recorded

From the initial global state one proceeds to generate systematically further global states until all possible candidate states have the status STOPPED or OLD.

Starting with the initial global state, find a global state NS (which might also be the initial state) whose status is NEW. This requires a recursive search down to the leaves of the reachability tree. A NEW state is necessarily a leaf of the tree. If all the leaves have the status STOPPED then the enumeration of the reachability tree is complete.

1 Given a NEW global state NS, check whether or not NS is identical (in the 2 local states and in the contents of the channel matrix) to any other distinct (ie. other than NS itself) global state of the whole reachability tree. If yes, the status of NS becomes STOPPED with reason "identical"
Step 2: trigger all delta transitions

Given a NEW global state NS which is not the immediate result of a delta transition, generate all possible delta transitions as follows.

Consider each local protocol partner i (1...) in turn. Each contributes exactly one local-state component S to the global state NS. For the local-state component S contributed by partner i to NS, list the AI signals which the local transition function of partner i permits in local state S. For each AI signal generate a descendant global state NSS whose local-state components are identical to those of NS, but whose diagonal element i,i (where i is the index of the local automaton is question) of the channel matrix is equal to the AI signal. If the i,i element of the matrix is already occupied, then a bandwidth error has been detected: the application is misusing the protocol by sending AI signals faster than they can be processed. The status of the newly-derived global state NSS becomes STOPPED with reason "application bandwidth".

Otherwise, the status of the newly-derived global state NSS becomes NEW and a note is made that the state is the result of a delta transition.

Step 3: trigger all non-delta transitions

Given a NEW global state NS which has had all permitted delta transitions triggered as above, trigger all transitions which are indicated by the contents (both I and AI signals) of the channel matrix.

Consider each local protocol partner i (1...) in turn and list the input signals (both AI and I) which await processing by partner i. For partner i, an unprocessed AI signal appears in element i,i of the channel matrix, while the unprocessed I signals appear in the non-diagonal elements of column i of the matrix.

The local protocol partner i contributes exactly one local-state component S to the global state NS. For the local-state component S contributed by partner i to NS, consider in turn each unprocessed input
signal from column $i$ of the channel matrix. The order of selection of signals is immaterial since the complete reachability tree will eventually contain all possibilities. For each unprocessed input signal $I$ (or $AI$) generate a descendant global state $NSS$ where the local-state component contributed by partner $i$ is the local state $SS$ which results from partner $i$ receiving signal $I$ in local state $S$. That is, $SS = T(S, I)$ where $T$ is the local transition function of partner $i$.

If the local transition function of partner $i$ is undefined for the reception of signal $I$ in local state $S$, then the newly-derived global state $NSS$ has local-state component $I$ set to null "?" and takes the status $STOPPED$ with reason "not specified".

If the local transition of partner $i$ produces an output signal $O$ to another partner $j$, then that signal $O$ is inserted into element $ij$ of the channel matrix. If the $ij$ element of the matrix is already occupied, then a bandwidth error has been detected: the partners are sending signals faster than they can be processed. The status of the newly-derived global state $NSS$ becomes $STOPPED$ with reason "protocol bandwidth".

Otherwise, the status of the newly-derived global state $NSS$ becomes $NEW$.

**Step 4: end the recursive search**

If a $NEW$ global state $NSS$ has no possible descendants (ie. no delta transitions and the channel matrix is empty) then its status becomes $STOPPED$ with reason "stable".

When all possible descendant global states of a $NEW$ state $NS$ have been found in the above steps, then the status of $NS$ becomes $OLD$. 
Appendix D

Global state analysis of the 2-phase commit protocol

<table>
<thead>
<tr>
<th>state of master</th>
<th>state of slave</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel contents: application to master</td>
<td>channel contents: master to slave</td>
</tr>
<tr>
<td>channel contents: slave to master</td>
<td>channel contents: application to slave</td>
</tr>
</tbody>
</table>

KEY

<table>
<thead>
<tr>
<th>waiting</th>
<th>deciding</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>waiting</th>
<th>abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>abort</th>
<th>abort</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>commit</th>
<th>commit</th>
</tr>
</thead>
</table>

stable state
Appendix E

Global state analysis of the user/server protocol

**KEY**

<table>
<thead>
<tr>
<th>state of user</th>
<th>state of server</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel contents:</td>
<td>channel contents:</td>
</tr>
<tr>
<td>application to user</td>
<td>master to server</td>
</tr>
<tr>
<td>channel contents:</td>
<td>channel contents:</td>
</tr>
<tr>
<td>slave to user</td>
<td>application to server</td>
</tr>
</tbody>
</table>

δ↓ δ

<table>
<thead>
<tr>
<th>ready</th>
<th>idle</th>
<th>AU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait</th>
<th>idle</th>
<th>req</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait</th>
<th>idle</th>
<th>req</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait</th>
<th>fault</th>
<th>req</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait</th>
<th>fault</th>
<th>req</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stable state</td>
<td>stable state</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait</th>
<th>service</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>AS2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait</th>
<th>service</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>AS2,AS1</td>
</tr>
<tr>
<td>appln. bandwidth</td>
<td>not specified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>alarm</td>
<td></td>
</tr>
</tbody>
</table>

identical subtree

continue at A
continue at C

<table>
<thead>
<tr>
<th>ready</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU1,AU1</td>
<td>-</td>
</tr>
<tr>
<td>alarm</td>
<td>-</td>
</tr>
</tbody>
</table>

appln. bandwidth

<table>
<thead>
<tr>
<th>wait</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>req</td>
</tr>
<tr>
<td>alarm</td>
<td>-</td>
</tr>
</tbody>
</table>

identical subtree

<table>
<thead>
<tr>
<th>register</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU1</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>register</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

not specified

<table>
<thead>
<tr>
<th>register</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU1,AU1</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
continue at B

<table>
<thead>
<tr>
<th>ready</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>alarm</td>
<td>-</td>
</tr>
</tbody>
</table>

identical subtree

<table>
<thead>
<tr>
<th>register</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

\[ \delta \]

<table>
<thead>
<tr>
<th>register</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ready</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ack</td>
<td>-</td>
</tr>
</tbody>
</table>

\[ \delta \]

<table>
<thead>
<tr>
<th>ready</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

identical subtree. Success!

<table>
<thead>
<tr>
<th>ready</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU1</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

identical subtree. Successful error recovery!

<table>
<thead>
<tr>
<th>wait</th>
<th>fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>ack req</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

protocol bandwidth