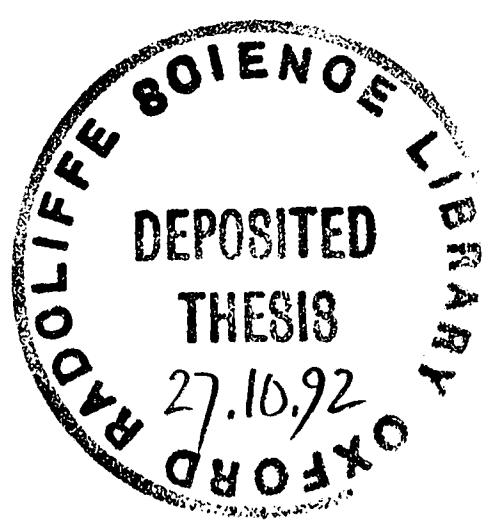
Towards a Unified Methodology for the Design and Development of Distributed Control System Software



Y. K. H. Lau
Robotics Research Group
Department of Engineering Science
University of Oxford

Trinity, 1991



This thesis is submitted to the Department of Engineering Science, University of Oxford, in partial fulfillment of the requirements for the degree of Doctor of Philosophy. This thesis has been composed by myself, and except where otherwise stated, describes my own work.

Towards a Unified Methodology for the Design and Development of Distributed Control System Software

Y. K. H. Lau

Christ Church, Oxford.

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy at the University of Oxford.
Trinity, 1991.

Abstract

A unified approach to the design and development of distributed control software is presented. This method is the result of a 'tight' integration between a formal method for concurrent systems (CSP) and a structured method for distributed control system (DARTS). The work presented in this thesis does not seek to extend the semantic model of CSP nor to design a specific control algorithm, rather, efforts are made to apply the existing specification and verification techniques to enhance the formality of the well established and case-proven structured counterparts that benefits are captured from both methods.

As a *methodology* is the central aim, the suggested approach is a first step towards a complete unified software development environment, which engineers can follow from organising design ideas to system implementation with proven correctness. The thesis develops a set of parameterised CSP predicates for expressing concurrency and communication together with a corresponding set of generic processes to reflect these specified behaviours. These generic processes are formal building blocks for generating system implementations at different levels of abstraction. Utilisation of DARTS criteria and the parameterised CSP objects frame the refinement strategies. Also, mappings of generic processes to pictorial representations are suggested which enable easy assimilation of the evolving designs.

Applicability of the approach is demonstrated through a high level software design of a high-performance robot control system where its suitability is shown via requirement specifications, properties verification and implementation of salient behaviours using generic building blocks. Although verification often means rigorous mathematical reasoning, the thesis presents a proof assistant — the Causality Diagram Evaluation Tool — to automate the manipulation of CSP processes according to the defined algebraic laws. It is shown to be of value in reasoning with designs and implementations of the robot system. It is found that the analysis facility and the graphical interpretation of communication provided by the tool allow effective analysis and manipulation of early designs.

The results derived from specifying essential design details, from transforming highly abstracted implementation models, and from investigation of system behaviours through formal reasoning and simulation conclude that formal methods, in particular CSP, has a niche value in enhancing software reliability at the sub-system level as well as providing a better underpinning to the structured method DARTS. The end product is a method to generate a correct and unambiguous document of the system concerned that is amenable to a direct implementation.

Acknowledgements

I would like to thank Ron Daniel for being my supervisor who has given me insight and encouragement over the past years, and the invaluable comments and constructive criticism he made on my thesis.

My special thanks to Wayne Luk for his continuous advice and enlightening suggestions made on my work, and his patience in reading the manuscript. I would also like to thank Steve Schneider and David Jackson for their discussions on the mathematics of CSP, Conrad Chan for his help in the UNIX and window based programming, Quentin Miller for his advice on the Orwell interpreter; also Jennet Batten for reading part of the manuscript from which I have very much benefited. And of course, all members of the 'real' robotics group, and in particular, Inaki Arocena, Paul Elosegui, Patrick Fischer and Terence Tsang for their expert advice on robot control, comments and friendship during the years.

I am also indebted to Croucher Foundation for their financial support to my research in Oxford.

My deepest thanks to my family for their support and understanding throughout the years and my wife Florence for her love and patience.

Contents

1 Introduction 1

1.1 Objectives . . . 1
1.2 Software Development for Control Systems . . . 2
1.3 Problems with a Single-tracked Approach . . . 3
1.4 A Unified Methodology . . . 4
1.5 Novelty of the Approach . . . 6
1.6 Outline of the Thesis . . . 7

2 Formal Methods 9

2.1 Formal Methods in Software Engineering . . . 9
2.1.1 Formal Systems . . . 9
2.1.2 Formal Specification . . . 10
2.1.3 Formal Verification . . . 10
2.1.4 Refinement . . . 11
2.2 Formal Specification Methods . . . 11
2.2.1 Algebraic Specification Methods . . . 14
2.2.2 Axiomatic Specification Methods . . . 15
2.2.3 State-Machine Specification Methods . . . 15
2.2.4 Conclusion . . . 17
2.3 Formal Verification Techniques . . . 18
2.3.1 The Operational Verification Technique . . . 19
2.3.2 The Denotational Verification Technique . . . 19
2.3.3 The Axiomatic Verification Technique . . . 19
2.3.4 Axiomatic Verification Methods on Concurrent Systems . . . 21
2.3.5 Limitations of Axiomatic Verification Technique . . . 22
2.3.6 Pragmatics . . . 23
2.4 Motivation for Formal Specification and Verification . . . 24
2.4.1 Advantages and Drawbacks . . . 24
2.4.2 Who will Benefit? . . . 26
2.5 Communicating Sequential Processes . . . 26
2.5.1 History and Motivation . . . 26
2.5.2 The Models . . . 27
2.5.3 The Notation . . . 29
2.5.4 CSP Processes . . . 29
2.5.5 Process Algebra . . . 29
2.5.6 Specification and Verification of Satisfaction . . . 36
2.5.7 Weaknesses . . . 40

3 Structured Methods for Control System Software 42
3.1 Structured Methods . . . 42
3.2 Existing Structured Methods . . . 43
3.3 Structured Software Development . . . 46
3.3.1 Essential Modelling . . . 47
3.3.2 Implementation Modelling . . . 49
3.3.3 The *Transformation* and *Data* Schemas . . . 50
3.3.4 Essential to Implementation Model Transformation . . . 53
3.4 DARTS — A Structured Method . . . 56
3.4.1 Why DARTS? . . . 56
3.4.2 The DARTS Way . . . 57
3.4.3 DARTS Criteria . . . 57
3.5 Formalisation of Structured Methods . . . 59
3.5.1 Motivation . . . 59
3.5.2 Bridging the 'Gap' — Refinement of the Yourdon Method . . . 60
3.5.3 Removing Ambiguity — the Formalisation of DARTS via CSP . . . 60
3.6 Conclusion . . . 61

4 CSP Abstraction Tool Set 63
4.1 Introduction . . . 63
4.2 Properties of Control Systems . . . 64
4.2.1 A Top Level View . . . 64
4.2.2 A Way Down the Hierarchy . . . 65
4.2.3 Generalised Properties . . . 71
4.3 The Abstraction Tool Set . . . 71
4.3.1 Assumptions on Real Systems . . . 71
4.3.2 CSP Models for Control Processes . . . 73
4.3.3 Objectives . . . 73
4.3.4 The Tool Set . . . 74
4.4 The Generic Processes . . . 79
4.4.1 FSM . . . 81
4.4.2 INIT and UPDATE . . . 91
4.4.3 BUFFER . . . 94
4.4.4 The PERM Family . . . 101
4.4.5 MUX . . . 104
4.5 Tool Set Utilisation: A Unification Issue . . . 107
4.5.1 A Unified Methodology . . . 108
4.5.2 Two General Proof Rules . . . 112
4.5.3 Remarks . . . 113
4.6 Completeness of Tool Set . . . 113

5 A Real-time Robot Control System: Specification 116
5.1 Introduction . . . 116
5.2 System Definition . . . 117
5.3 An Environmental Model Design Requirements . . . 118
5.3.1 The Requirements . . . 119
5.3.2 Formal Specifications . . . 121
5.4 Zeroth Level Model . . . 124

5.4.1 Tool Set Specification . . . 124
5.4.2 Verification Against Design Requirements . . . 126
5.4.3 Deduction . . . 132
5.5 First Level Model . . . 132
5.5.1 System Decomposition . . . 133
5.5.2 Tool Set Specification . . . 138
5.5.3 Verification . . . 149
5.5.4 Potential Problems and Re-design Strategies . . . 159
5.6 Higher Level Models . . . 166
5.6.1 Trajectory Generator: Refinement . . . 167
5.6.2 Controller: Refinement . . . 168
5.6.3 Remarks . . . 172
5.7 Properties with respect to Software Architecture . . . 172

6 Simulation and Implementation **176**
6.1 Introduction . . . 176
6.2 The Need for Machine Assisted Verification Tools . . . 177
6.3 The Design of the Causality Diagram Evaluation Tool . . . 178
6.3.1 The CSP Interpreter . . . 179
6.3.2 The User Interface . . . 183
6.4 Utilisation of CADET . . . 186
6.4.1 Application to a Simple Concurrent System . . . 187
6.4.2 A Communication Protocol . . . 189
6.4.3 Application to the Robot Control System . . . 191
6.4.4 The Value of CADET . . . 205
6.5 Implementation of Generic Processes . . . 206
6.5.1 A High Level Implementation . . . 206
6.5.2 Practical Implementations . . . 206
6.5.3 Mapping CSP into Occam . . . 207
6.5.4 Remarks on CSP to Occam Transformation . . . 209

7 Conclusion **211**
7.1 An Overview . . . 211
7.2 Formal Methods in Control Software Design: An Evaluation . . . 212
7.3 The Unified Approach: A Rational . . . 215
7.4 Suggestions for Future Work . . . 216

A CSP Notations **218**

B Proofs of the General Proof Rules for Parallel Processes **222**

C Proofs for the Properties of Generic Process *INIT* and *UPDATE* **226**
C.1 Properties 4.7 and 4.10 . . . 226
C.2 Properties 4.8 and 4.11 . . . 230
C.3 Properties 4.5 and 4.6 . . . 231

D Proof of Deadlock Freedom for the Process CONTROL$_2$ **232**

List of Figures

1.1 Three proposals for the combination of structured and formal methods for software development: (a), after-the-fact; (b), parallel; and (c), integrated approaches . . . 5

2.1 Abstraction and Refinement in Software Development 11
2.2 The metric space mapping of the different models of CSP 27
2.3 The CSP model of communication 29
2.4 A model of a simple communication protocol 34

3.1 A comparison between various structured system design methods 46
3.2 An environmental modelling — an "outside-in" approach 49
3.3 A Yourdon implementation model showing the different hierarchies of implementation details of a system 50
3.4 A transformation schema for discrete data flow 51
3.5 A collection of typical transformations used by structured method for modelling a system and representing design ideas 52
3.6 A "hand-in-hand" development of essential and implementation models 53
3.7 A separate development of models and a cascade transformation 54
3.8 A state transition diagram of process control system A 61
3.9 A state transition diagram of process control system B 61

4.1 A Computer Control System with Jacketing 64
4.2 Communication protocol between the calibration process and controlled plant . . . 65
4.3 A general closed loop control system 65
4.4 A Block Diagram of Feedforward Compensation Control Scheme 67
4.5 A compensated unity feedback control system 67
4.6 A typical Three Layer Hierarchy of a distributed control system 72
4.7 A transformation schema representation of the generic process *FSM* 87
4.8 A transformation mapping of the generic processes *INIT* and *UPDATE* 95
4.9 An event store formalised using *BUFFER* 99
4.10 A buffered synchronous communication 99
4.11 A structured model of a formalised PID controller 100
4.12 A transformation schema mapping of the generic process *PERM* 103
4.13 A physical process which is a formal mapping of the generic process *RPERM* . . . 103
4.14 Formalisation using *MUX* 107
4.15 A unified approach between formal and structured methods 111

5.1 The organisation of the Oxford 3-DOF Robot 117
5.2 An environment model of the robot control system 119
5.3 Decomposition of the robot system 133

5.4 A pictorial view of TRAJGEN 141
5.5 A pictorial view of DATALOG 143
5.6 A pictorial view of CTL 144
5.7 A pictorial representation of OPINT 146
5.8 A pictorial view of DATAPROC 148
5.9 A generic process representation of the communication topology of the overall CONTROL system 149
5.10 A refinement of the First Level sub-system TRAJGEN 168
5.11 An initial refinement of the First Level sub-system CTL 169
5.12 A refinement of the process CONTROLPROC 171
5.13 An "imploded" view of system refinement 173

6.1 The architecture of the causality evaluation tool 179
6.2 The main window of the Causality Diagram Evaluation Tool 180
6.3 The structure of the CSP interpreter 181
6.4 The structure of the User Interface 184
6.5 A typical Causality Diagram 185
6.6 The notion of deadlock, represented using a fictitious causality diagram 186
6.7 A simple three process system 187
6.8 An instance of the communication between the three process system 188
6.9 A trace from the original model of the communication protocol showing the uniqueness in the ordering of communicating events 189
6.10 A particular trace of the communication protocol with a modified receiving process $Receiver_2$ 190
6.11 Another trace of the communication protocol with a modified receiving process $Receiver_2$ 190
6.12 Only a single communicating event is displayed and there after the protocol fails to proceed, a deadlock is detected. 190
6.13 The initial actions offered to the environment by the three sub-systems OPINT, TRAJGEN and CTL running in parallel 192
6.14 A typical traces *after* an operator's input 192
6.15 A typical traces in the absence of operator's input 192
6.16 Behaviour of the three concurrent sub-systems under abnormal input conditions . 193
6.17 Behaviour of the three concurrent sub-systems under abnormal input conditions . 193
6.18 The sequence of actions taken by TRAJGEN and DATALOG which leads to the scenario of a deadlock 194
6.19 An alternative sequence of actions taken by TRAJGEN and DATALOG which leads to the scenario of a deadlock 194
6.20 A trace which leads to a subsequent deadlock 195
6.21 The sequence of actions taken by the parallel sub-systems TRAJGEN and DATALOG *after* an output from the sub-system OPINT 195
6.22 The sequence of actions taken by the parallel sub-systems TRAJGEN and DATALOG *after* an input from the sub-system DATALOG 195
6.23 Subsequent communication pattern of the three parallel sub-systems: DATAMASTER, TRAJGEN_2 and DATALOG_2 *after* communications across channel *Ddata* are signalled 196

6.24 Subsequent communication pattern of the three parallel sub-systems: DATAMASTER, TRAJGEN$_2$ and DATALOG$_2$ *after* communications across channel *Opout* are signalled 196
6.25 Communication pattern of the three parallel sub-systems: DATAMASTER, TRAJGEN$_2$ and DATALOG$_2$ 197
6.26 Initial actions offer by the overall system to its environment 198
6.27 The subsequent actions taken by the overall system CONTROL *after* an operator's input 199
6.28 The subsequent actions taken by the overall system CONTROL *after* receiving the set of sensor data from the robot 199
6.29 A typical traces of the system as specified by the First Level Model 199
6.30 Another typical trace of the system as specified by the First Level Model 199
6.31 The restrictive measures introduced by the sub-system DATAMASTER to internal transitions so as to avoid possible deadlocks. Guarding events *og* and *dg* are mutually exclusive events which are displayed on the same time step, appearing as alternative actions. 200
6.32 The initial communications, *Traj* and *Cdata* that are offered to the environment by the refined time critical sub-system CTL 202
6.33 Each event can be examined for details 202
6.34 A display of the inter-process communications of the refined sub-system CTL . . . 203
6.35 An infinite loop of communication across channel *Cdata* between process INVDYNGEN and its environment 203
6.36 A "Livelock" between communicating processes 203
6.37 The internal and external communications of the PID control process 204
6.38 An Occam implementation of the generic process *FSM* 208
6.39 An Occam implementation of the generic process *UPDATE* 210

List of Tables

2.1 A comparison of formal specification methods 13

5.1 List of abbreviations of sub-systems that are used in writing specifications 139
5.2 Formal parameters for specifying TRAJECTORY GENERATOR 139
5.3 Formal parameters for specifying DATA LOGGER 141
5.4 Formal parameters for specifying CONTROLLER 143
5.5 Formal parameters for specifying OPERATOR INTERFACE 145
5.6 Formal parameters for specifying DATA PROCESSOR 146
5.7 Justifications by DARTS criteria on the corresponding identified processes 169
5.8 A summary of the corresponding identified and formalised sub-process for the process CONTROLPROC 170

Chapter 1

Introduction

1.1 Objectives

This thesis is about the design and development of reliable software for distributed control systems using a *unified methodology*. Within a unified software development environment: the designer's perception can be built up structurally; the design can be specified precisely, independent of any implementation context; and it can be proved that the design meets its original requirements at various levels of abstraction and at different degrees of complexity while the design is refined step-by-step from a highly abstract description down to an implementation orientated form with the aid of clear guidelines and heuristics.

In order to provide a suitable environment, the thesis directly addresses the task of integrating the theory of formal mathematics for concurrency (which is less favourable to engineers) with the structured techniques developed for distributed control systems (that are more familiar to engineers) — the product is a methodology which benefits from both wings and is a step towards a complete unified methodology for the design and development of general distributed control system software.

Two main areas which are central to the idea of integration are explored in detail: the applicability of formal mathematics and its limitations in providing a platform for engineers to design and build control system software; and the exploration and evaluation of automated software tools in assisting the task of design and development. It is hoped that the answers to some of these questions will provide engineers with a systematic method for software production, allowing engineers to model a system formally, to analyse and predict a system's behaviour so that pre-established correctness constraints are met.

In the context of the thesis, robot control is the chosen area of the main case study, but the general principle of a unified methodology should be of use to practising engineers in the field of

software production for distributed control systems.

1.2 Software Development for Control Systems

It is generally known that producing a piece of 'correct' software is a non-trivial task and this is reflected in the fact that the variation in combination of software components is astronomically large even for a relatively small system. As shown in Gries [1981], a totally correct implementation of a system is hard to obtain because of the following statistics:

> Suppose a program consists of n small components, each with a probability p of being correct. Then the entire program has a probability of $P < p^n$ of being correct: this means that a program with 10 modules, each extensively tested and thought to have a 95% chance of being correct, has less than a 60% chance of being correct; for a 100-module program, the probability of correctness falls to less than 0.6%.

Since n is large compared to 100 in any good-sized program, in order to have any hope that a program is correct requires p to be extremely close to unity.

For the production of distributed control systems software, in addition to the assurance of functional correctness, severe constraints such as reliability, safety, responsiveness and timeliness have escalated the requirements of correctness to a stringent level. Therefore special disciplines have to be followed in order to comply with these requirements [Theaker, 1991]. Also, the distributed or sometimes the concurrent nature of the software must be represented in a design. Going back to the statistics for program correctness, the probability of being correct for a concurrent program not only depends on the correctness of its components, but it is also a function of *how* these components are combined. This further reduces the probability of obtaining a correct implementation. Hence, the following *ideas* proposed by Sufrin [1986] have emerged as the consensus for software production.

- Any agreements and contracts must be based on a *common understanding* between designers and clients.
- Designers and engineers should be able to formulate their objectives and express their decisions *unambiguously* and at a suitable level of *abstraction*.
- The evocation of *reasoning before building* a system after laying down the design requirements — this is the idea of correctness by construction.
- The *why* of a system design should be justified via simple models and the *what if* should be predictable. This leads to the idea of software prototyping.

The ultimate aim is, of course, to have a methodology which will produce quality software according to these consensus, which is *correct* and which exhibits a high degree of reliability, reusability, clarity and extendibility [Boriani, 1987].

1.3 Problems with a Single-tracked Approach

With the need to develop control software with such stringent requirements, various software design and development methods have been proposed. Historically, software development methods and techniques all fall into the class of techniques generally known as *Structured Methods*. These methods are built upon design experience in various commercial projects and case studies, and are generally well established and case-proven. These methods address the various stages in the software development life cycle and in particular, they start by helping designers to organize their thoughts, to represent their ideas through a mixture of graphical and textual entities, all the way down to code generation and program testing.

Recently, another class of software design and development technique has emerged called the *Formal Method*. This class of methods originated from a different root to its structured counterpart and the methods are based on the application of mathematical theories such as logic, sets, and predicate calculus in order to express requirements using a pre-defined set of notation. Relationships such as 'equality', 'satisfies', and 'implicates', are used in order to take advantage of the rigour of mathematical proof. In formal mathematics, system development is considered to be equivalent to the transformation of an abstract system specification to one which is implementation orientated in a step-by-step manner. The formality of these methods exist in their provision of proof-of-correctness, or verification of an implementation satisfying corresponding specified system properties. Following this idea, a correct implementation of the software of a system is guaranteed so long as each step of transformation is verified.

Apparently, the two classes of method are equally sound for producing quality software for a distributed control system and the choice of either of these methods seems arbitrary. However, detailed reviews on the two classes of methods in Chapters 2 and 3 reveal a number of major drawbacks in using exclusively methods from either class. Structured methods generally favour a top level design, which helps designers to organize design ideas and to present them in an understandable way, their inadequacy in reasoning with logical consistency and establishing *satisfies relations* of a design with respect to the desirable properties has always been a major limitation. Most often, reliability margins of structured methods are handicapped by approaching the assurance of 'cor-

rectness' via exhaustive case testing and syntax checking. Other drawbacks of applying structured methods to design problems are discussed in Chapter 3 which further limits their applicability.

On the other hand, even though the formality provided by formal methods allows the properties of a system to be reasoned rigorously with respect to its specification, they are not front-end design methods. They do not provide an environment for analysing a physical system, or structuring design ideas, and hence lack a good system concept for helping in the modification of the requirements in an iterative manner. For a simple system, it may be possible to write a specification directly from a physical model, however for a complex system, such as a multidegree-of-freedom robot control system, a direct specification will become as erroneous as producing an executable program.

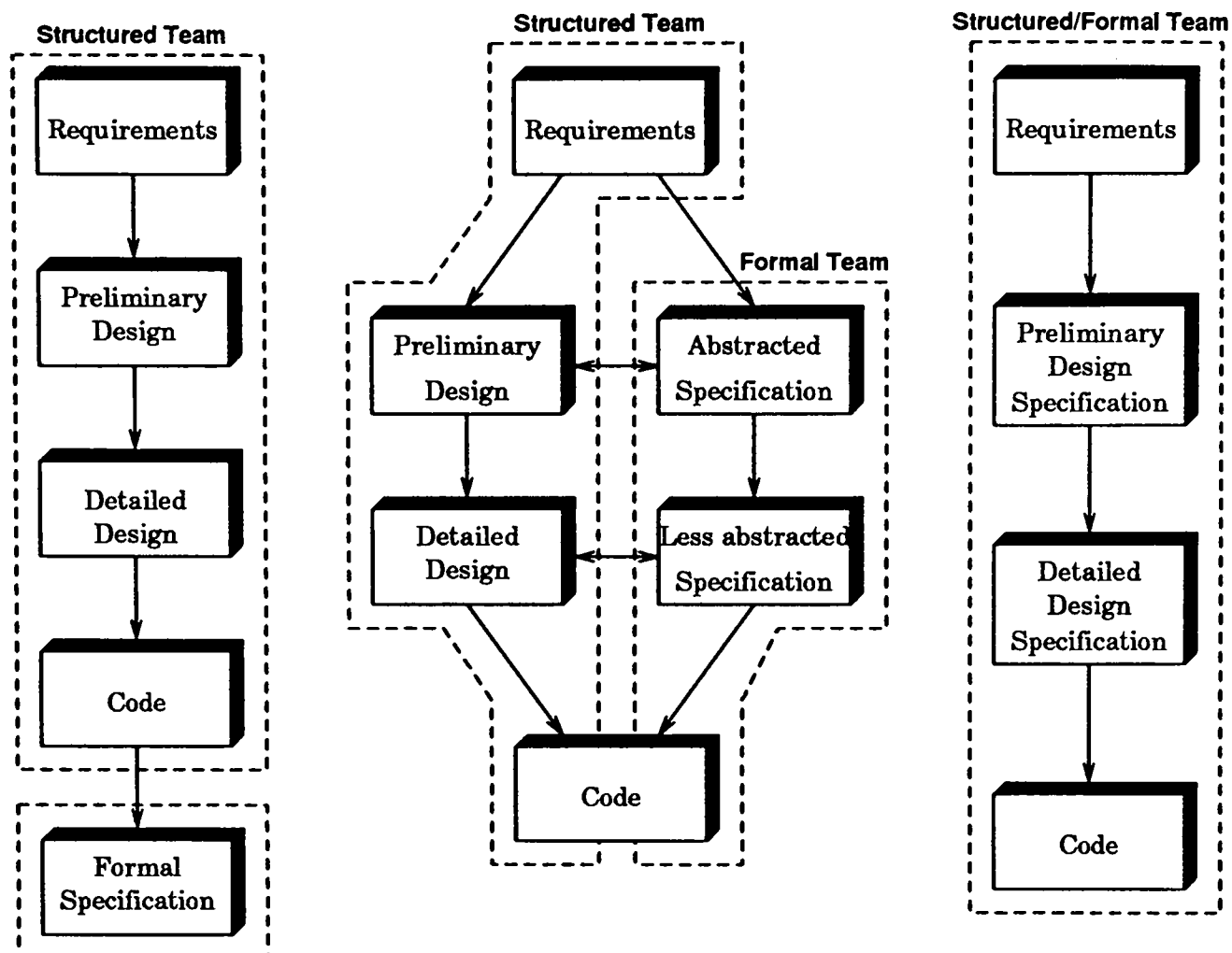
1.4 A Unified Methodology

From the foregoing discussion, it is clear that a single-tracked approach to control software design and development is undesirable for producing reliable software. Nevertheless, the use of a mixture of uncoordinated formal methods and structured techniques is equally unfavourable as the effort to relate the results or intermediate results produced has often overshadowed the benefit obtained in terms of costs and time. One of the solutions to this problem is to combine the two classes of method in a pre-planned manner so that cross-fertilization of ideas between the two classes of method results.

Various suggestions are put forward to this combination issue by Kemmerer [1990] and the three most popular approaches are: *after-the-fact verification*, *verification in parallel* and *integrated verification*. In Figure 1.1, the three approaches are presented with clear indications of the various development disciplines carried out by designated design teams with the relevant expertise.

Among the three different approaches, an after-the-fact approach is the one being used most and has the longest history of use. However, the cost effectiveness of this approach is poor because any errors detected by formal analysis after a completed development via a conventional structured method are costly to fix. On the other hand, verification in parallel suffers from the problem of the cost and time spent in relating the various sets of results produced from the different methods. Finally, a complete integration of the two classes of method has appeared to be the most desirable and is adopted in this thesis as the basis of the *unified methodology* discussed in Chapter 4.

With this truly integrated or unified approach to software development, a unified methodology is evolved which aims to provide an environment as described in the Objective and Section 1.2.



(a) **(b)** **(c)**

Figure 1.1: Three proposals for the combination of structured and formal methods for software development: (a), after-the-fact; (b), parallel; and (c), integrated approaches

The proposed method strongly adheres to the following properties for an ideal unified methodology.

- A 'tight' integration of conventional structured techniques and formal mathematics to an extent that the evolving design specifications become formal specifications expressed in a formal notation.

- Formal specification is introduced in the early stages of a design which follow directly after the creation of an essential system model. The 'specified' system can then be analysed using a formal requirement analysis. As a 'system' can be defined as a set of components working in concert, there are a number of constraints on the design concerning these components, e.g. system characteristics, properties and safety considerations. Formal requirement analysis allows:

  1. the identification and resolution of conflicts among interacting components to arrive at some tradeoff between functional goals and constraints; and

2. the assurance that system components, when put together, satisfy the system requirements.

- To associate a complex control system specification with the original requirements, a *hierarchical* specification approach is adopted. As such, the adopted formal method allows the control of abstraction spanning a range from a highly abstracted requirement specification to an implementation orientated specification.

- Both the specifications and their formal verification have to be designed in a hierarchical way. A unified method insists on the verification of system properties being satisfied at every stage of the transformation during system development so as to maintain overall consistency.

- A step-by-step refinement strategy is essential for transforming a highly abstracted specification to a lower level one, hence the provision of criteria and guidelines for this purpose is necessary.

- Reinstating the very first consensus made by Sufrin [1986], a common understanding between various parties involved in a system development program is essential. The unambiguous mapping of formally specified entities (specifications and process definitions) to a form which is familiar to engineers ("bubble-arrow" diagrams used in structured methods) is desirable.

1.5 Novelty of the Approach

The work in this thesis should be seen as an inter-disciplinary effort to integrate the two classes of software techniques: an evolving formal method (CSP) and the well established structured technique (DARTS) in the field of control. Through the act of integration and application, the applicability of formal methods to control software development is evaluated. It follows that much of the work which has been carried out has pioneered 'bridging the gap' and 'tying up' the two disciplines of software development. As such, the situation is quite different from, say, the application of a well understood control algorithm or optimization method to a particular control process, in which a whole body of theories and experimental results already exist, or a theoretical study on the specification and verification of hypothetical objects such as protocols and transaction systems in order to demonstrate the glory and elegance of the semantics model of the underlying formal system.

The following are the major contributions of the thesis to this inter-disciplinary field:

- The unification of CSP and DARTS through the introduction of a set of parameterized objects with clearly defined properties and laws that formalize the DARTS representation. The re-interpretation of DARTS task structuring criteria for system refinement through logical decomposition.
- The introduction and application of CSP modular formalism to the specification and verification of concurrent distributed control processes in association with the work on unification.
- The development of the Causality Diagram Evaluation Tool as a proof assistant for the prototyping of CSP processes that is used in the early stages of system design. The extension of the CSP interpreter enables process algebra to be automated and the graphical interface facilitates the analysis of process interactions.
- The application of formal specification and partial verification to the top level development of a robot control system which is a typical distributed real-time control system.

1.6 Outline of the Thesis

With an aim to integrate formal methods (CSP) with structured techniques (DARTS) and to investigate the applicability of CSP to the development of control system software, the thesis is structured as follows:

Chapter 2 reviews the various existing formal specification and verification techniques and their related applications to system development. The motivation for their use and the benefit from their use are analysed. The formal method CSP [Hoare, 1985b] is introduced which is chosen as the formal system for the unification issue discussed in Chapter 4. A subset of its notation is presented through illustrative examples of process definition and the two semantic models are discussed: *traces* and *refusals* for modelling and verifying properties. Being an evolving formal system, the weaknesses of the two semantic models are identified.

Chapter 3 reports and reviews the conventional structured techniques that are widely used by the engineering community. Although these are by far the most popular system development methods, the chapter identifies a number of their major shortcomings in producing reliable and 'correct' system software. Propositions are made with a view to eliminating these shortcomings and these propositions form the basis for the proposed unified methodology presented in Chapter 4. In particular, the structured technique DARTS [Gomaa, 1984] is chosen to be formalized via CSP. Apart from the discussion of the propositions, the DARTS

system structuring criteria are presented which play an important role in system refinement discussed in Chapter 5.

Chapter 4 identifies the salient properties of a typical control system and presents the specification, partial verification and full documentation of the formal building blocks for the full specification of such systems. A set of parameterized predicates and processes which form the main body of the unified methodology, are given for structuring designs at different levels of abstraction. A complete unified methodology is then proposed which is a step towards the integration of structured and formal approaches for the design and development of control software.

Chapter 5 describes the majority of the software development at a high level of a direct drive robot system. Emphasis is placed on the application of the formal building blocks for system specification and partial verification. Detailed top level development and refinement (Zeroth and First Level Models) are presented with partial verification of their properties. Potential problems within the design are predicted via verification and re-design strategies are discussed, based on both formal analysis and intuitive thoughts. The chapter concludes with a number of suggestions for a more refined system model which is obtained by applying the unified method iteratively to previously obtained system models. These suggestions will act as milestones which pave the way for a subsequent full implementation.

Chapter 6 is about implementation of formally specified systems such as the Zeroth and First Level Models of the robot control system developed in Chapter 5. In particular, the design and development of a proof assistant is discussed — the Causality Diagram Evaluation Tool, which is used for prototyping CSP processes. Its use forms part of the verification phase in the unified method and its applicability is assessed through a number of case studies on the sub-systems defined in Chapter 5 for the robot system. Also, the viability of a direct implementation of generic processes using a practical concurrent language, OCCAM, is demonstrated.

Chapter 7 summarizes the results and concludes on the advantages and limitations concerning the unification of the two methods, and through this, evaluates the applicability of formal methods to control software development. Comments on possible future research directions are also made.

Chapter 2

Formal Methods

2.1 Formal Methods in Software Engineering

The design and development of quality software involves learning a number of disciplines and following sets of criteria in order to obtain a sound end product which satisfies a designer's requirements. The primary objective of software engineering is to develop software which has the following features:

- A high degree of *correctness* with respect to its specifications.
- Software *reusability*.
- A piece of software should be clear to both designer and client.

It is found that properties such as reliability, modularity and extensibility which are equally crucial to quality software, are byproducts of the above features. To achieve such qualities, software design and development has to be based upon a thorough and formal approach, that is, one employing mathematical techniques for analysing all constituent elements within a system. Methodologies that allow system properties to be modelled unambiguously and to reason about these properties against the requirements mathematically are known as *formal methods* . The following sections define the terminologies and activities that are common to formal methods.

2.1.1 Formal Systems

A formal system is characterised by its *formal specification language* which provides a set of notations which form its syntactic domain, and a universe of objects which forms its semantic domain. And most importantly, there exists a set of precise rules defining which object satisfies which specification [Wing, 1990]. A formal model of a real system can be built from specifications with different *abstractions* through which strict attention can be paid to the specific aspects of a

phenomenon that is currently relevant and paying no attention to those that are not. A *formal specification* describing a real situation or phenomena is formed using elements from the syntactic domain which denotes a subset of its semantic domain. If this subset of the semantic domain *satisfies* the specification, then this *satisfies relation* provides the meaning or interpretation for the syntactic elements. Mathematical reasoning techniques are usually invoked to establish this *satisfies* relation.

Most often, different formal systems effect the way that *formal models* of systems are constructed. Two broad classes can be classified: the *model-orientated* and *property-orientated* formal methods. A formal system can be based on either of these methods or encompass both. Using a model-orientated method, a specifier defines the behaviour of a system directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets and sequences. Using a property orientated method, a specifier defines the system's behaviour indirectly be stating a set of properties, usually in the form of a set of axioms, that the system must satisfy. In the Sections 2.2 and 2.3, a detailed discussion of the two different classes of methods on the grounds of specification and verification is presented.

2.1.2 Formal Specification

A *specification* is a sentence written in a notation provided by the syntactic domain of a formal system. It is a *formal document* describing a system and serves as the central document for consultation by software designers during all stages of program development. It consists of a set of *predicates* defining the permitted observations for the system being specified [Hoare, 1982]. In most cases, they provide the possibility of proof-of-correctness so as to establish the truthfulness of a "specification satisfies another specification" of which the later specification is written at a higher level of abstraction or an "implementation meets its specification".

2.1.3 Formal Verification

As discussed in the previous section, the process of establishing a *satisfies* relation for a system realization with respect to its requirements with mathematical rigour is commonly known as *formal verification* [Blackhouse, 1986]. This process is also called the proof-of-correctness in some contexts.

Verification plays a very important part in program development using formal methods. It is often vital to prove that specifications constructed are indeed correct descriptions of the system of interest, or whether a system realization satisfies its formal specifications, so that these specifications can subsequently be relied upon.

2.1.4 Refinement

The process of *software refinement* [Woodcock, 1989] is defined as the transformation of a highly abstracted specification into a less abstracted one. By constructing successive specifications containing an increasing amount of implementation detail with the aid of formal reasoning, specifications which contain enough information for code generation are obtained. Usually, the process of software refinement is carried out in various stages and involving both data and procedural refinement as shown in Figure 2.1. However, it is vital to carry out this process formally in order to ensure all subsequent specifications correctly describe their predecessors. Therefore formal verification is used in each stage of refinement. Moreover, in some cases, it would be of great advantage in software development if a final specification produced is not only a precise description of a system in detail, but is also expressed in a form which is amenable for direct implementation.

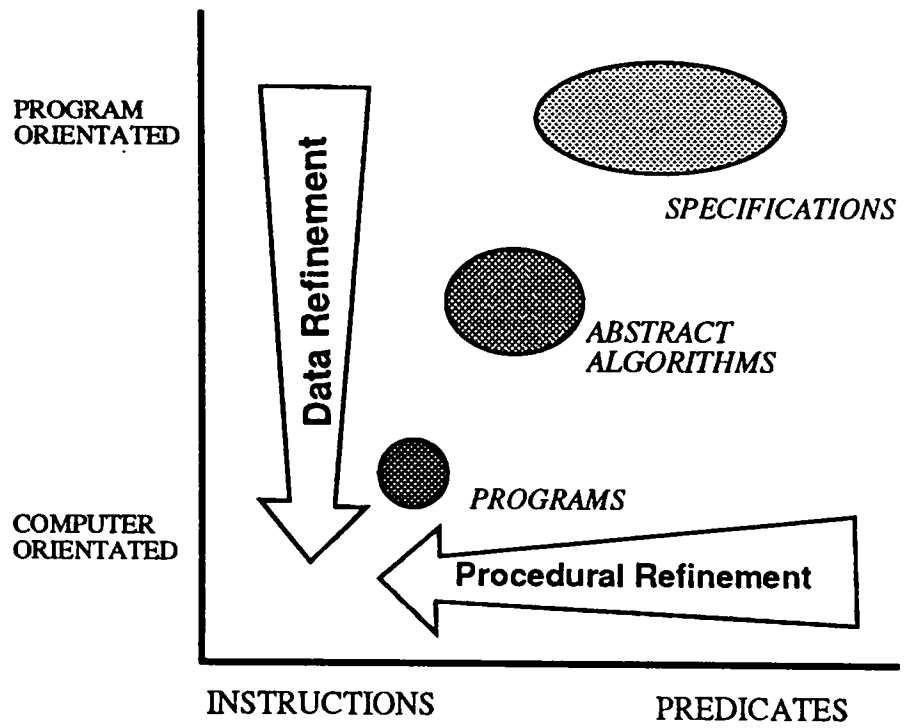


Figure 2.1: Abstraction and Refinement in Software Development

2.2 Formal Specification Methods

Among all the formal specification methods, there are three main families of specification approaches, they are the algebraic, the axiomatic (sometimes also known as an abstract model) and the state-machine approach. All the three families share common features such as they all define behaviours of an object expressed in units called *functions* and do so in a "result-orientated" or "non-procedural" way that suppresses most details of implementation [Berg *et al.*, 1982]. They define results in terms of an underlying abstraction and are usually associated with some set of

defined mathematical entities about which it is possible to reason with rigour.

Furthermore, according to the classification of formal specification methods described in Sub-section 2.1.1, the first two families, the abstract algebra method and the axiomatic method, are said to be *property-orientated* methods for which specifications are expressed in terms of axioms defining how system operations relate to one another. Whereas the State Machine method is *model-orientated*, where specifications constructed rely upon an explicit system model that is being built from a well defined set of primitives.

The following Sub-sections describe briefly the three families of approaches together with some examples of using these particular specification methods. Table 2.1 presents a comprehensive comparison of some of the existing specification methods which is based on Boriani [1990].

Methods	Approaches			Status	
	MODEL-ORIENTATED	PROPERTY-ORIENTATED			
	State-Machine	*Algebraic*	*Axiomatic*	*Recent*	*Established*
AFFIRM		√		√	
ANNA			√		
Act One		√			
BSI-VDM	√			√	
CCS	√				√
CLEAR		√			
CSP	√		√		√
ESM			√	√	
Gypsy		√			√
HOS	√				√
Ina Jo	√				
IOTA			√	√	
m-Verdi				√	
LARCH			√	√	
LOTOS		√		√	
OBJ		√	√		√
Petri Nets	√				√
PROLOG	√		√		
RTTL		√		√	
SIGNAL	√			√	
SPECIAL	√				√
VDM	√				√
Temporal Logic		√			√
TCSP	√		√	√	
Z	√	√			√

Table 2.1: A comparison of formal specification methods

2.2.1 Algebraic Specification Methods

An *algebraic specification method* specifies functions of objects using abstract algebra, functions are defined as in an algebraic technique by stating their interrelationships. Functions specified using an algebraic specification method must only map a value from their domain to a unique value in their range. Specified functions are used to describe particular operations of processes or systems and properties of a system are specified using a set of axioms in which their forms are restricted to algebraic equations. This approach is called "algebraic" because the values and functions of a specification can be viewed as forming an abstract algebra.

The initial theoretical work on algebraic specification was done by Guttag [1975], and some of the most popular specification methods that stemmed from this root are AFFIRM [Musser, 1979] and OBJ [Goguen and Tardo, 1979]; languages such as CLEAR [Burstall and Goguen, 1986], Act One (Algebraic Specification Techniques for Correct and Trusty Software Systems) and Gypsy [Ambler and et. la., 1976] for specifying the behaviour of sequential systems. Of these, OBJ is used to validate specifications by experimentation, by which functions are specified and then referenced in abstract programs. The OBJ system then computes the specified result using a rewrite technique developed by Goguen and Tardo [1979]. On the other hand, the AFFIRM verification system accepts quantifiers and logical connectives such as **or**, **not**, **and**, **imply**, etc. and is used to reason about the correctness of programs written in a Pascal-like language.

A number of algebraic specification methods for specifying concurrent and distributed systems have been developed, these include temporal logic [Pnueli, 1977], and later Real Time Temporal Logic (RTTL) [Ostroff, 1989] for real time systems, and Language of Temporal Ordering of Specifications (LOTOS) [Lotos, 1984b; Lotos, 1984a] which represents more recent work on the combination of Act One and CCS [Milner, 1980]. Also, the language Gypsy [Ambler and et. la., 1976] uses an algebraic approach with the intention of integrating the specification and program into a single text with a view to strike a middle position between extreme formality and "usability" within the domain of practical programming problems. Its approach has pioneered the integration of formal specification techniques with more practical software development methods.

Interesting applications of the algebraic specification approach may be found in Puente *et al.* [1986] where the language LOTOS is used to specify a real-time program for controlling a milling process in a cement plant and in Ditt [1986] where a mapping of an algebraic specification onto the programming language Modula-2 [Wirth, 1983] is presented.

2.2.2 Axiomatic Specification Methods

The *axiomatic method* is also known as the abstract model or the predicate transform method. Its names reflect the nature of the method which evolved from the work by Hoare [1969] on the proof-of-correctness for implementations of abstract data types, where predicate logic preconditions and postconditions are used for the specification of each operation. The syntax of an axiomatic specification language defines its functions in terms of an underlying abstraction, and such abstraction can be any of **sets**, **power-sets**, **sequences**, **arrays**, **vectors**, etc. about which it is possible to reason formally. Using this method, system behaviour are specified indirectly by stating a set of properties in the form of a set of *axioms* that this system must satisfy. Because of its nature, in Berg *et al.* [1982], the following conclusion has been made:

> *... the approach was developed by Hoare as part of a unified technique for the specification and verification of abstract data types. As a result, it is the most "verification-orientated" approach.*

Specification languages that support an axiomatic approach are those of ANNA [Luckham and von Henke, 1985], the Extended State Machine (ESM) [Ostroff, 1989], Communicating Sequential Processes (CSP) [Hoare, 1985b], OBJ [Goguen and Tardo, 1979], IOTA, LARCH [Guttag *et al.*, 1985] and the language Prolog. Of these languages, OBJ and Prolog are executable specification languages. As such, the logic programming language Prolog when used in an axiomatic approach, lets specifiers state logical relations on the objects being specified.

LARCH itself is a dual specification language that combines an axiomatic and algebraic specification into a two tiered specification. The axiomatic component specifies the state-dependent properties of a program whereas the algebraic component specifies the state-independent data accessed by the program. This characteristic is also shared by OBJ for which both axiomatic and algebraic specification approaches can be followed. A Case study on practical systems specification using ESM for specifying discrete event processes can be found in Ostroff [1989].

2.2.3 State-Machine Specification Methods

The *state-machine specification method* is a model-orientated approach by which a system's behaviour are defined using a model constructed from mathematical structures such as functions, relations, sets or sequences. This approach was first suggested informally by Parnas [1972b] and then formalised in the work done by Principato [1987]. The underlying abstractions of state machine specifications are boolean objects and integers. Specifications constructed in this approach define a set of functions that specify transformation on inputs. The set of functions may be viewed

as defining the nature of an abstract data type or describing the behaviour of an abstract machine. In general, since the technique arose from a more pragmatic root as compared to the algebraic technique, it is more widely used in practical system specification and there is a larger family of specification languages supporting this approach.

A number of the more well known specification languages supporting this approach includes the Vienna Development Method (VDM) [Jones, 1986] which is a relatively mature formal method which is based on set and predicate calculus. It has a long history of research and industrial applications and practical examples such as the partial software design for the protection system of a nuclear reactor which is illustrated in Bloomfield and Froome [1986], the specification of a computer graphics standards (GKS) described in Duce and Fielding [1987] is well documented. Other applications includes the specifications of compilers and database systems.

Similar to VDM, Z is a method based on set theory developed by Spivey [1989] from the Oxford Programming Research Group and is nowadays one of the most popular formal methods for both academic research and industrial application. The *schema* notation of the language provides a good platform for system structuring and scaling. There are numerous case studies being undertaken using Z and they fall in the line of software engineering and transaction processing such as the preservation of database integrity. Comprehensive summaries of case studies are found in Hayes [1987] and other works includes: Specification Directed Module Testing by Hayes [1985], a 6800 microprocessor instruction set specification by Bowen [1987a], and work on specification of network service by Bowen [1987b], the specification of Unix filing system by Morgan and Sufrin [1984] and the work supported by IBM U.K. on the partial specification (47K out of 1M of code) of a Customer Information Control System (CICS) by Hayes [1987] and the specification of an oscilloscope by Tektronix. Other well known model-orientated specification languages mainly for the description of sequential systems are that of SPECIAL (SPECIfication and Assertion Language) [Roubine and Robinson, 1977] which is supported by SRI International, Ina Jo [Scheid and Holtsberg, May 1989; Scheid, 1979], m-Verdi and SIGNAL [le Guernic, 1986] developed for specifying real-time synchronous, data flow programs which has inherited properties from both CSP and Ada.

Languages which are designed primary for describing the behaviour of concurrent and distributed systems includes Petri Nets [Peterson, 1977], Milner's Calculus of Communicating Systems (CCS) [Milner, 1980], Hoare's Communicating Sequential Processes (CSP) [Hoare, 1985b], TSL, I/O Automata and Timed CSP [Davies and Schneider, 1989b]. Applications of Petri Nets

are mainly found in the specification of complex distributed industrial control systems as in Banaszak [1986], the development of real-time systems using EDE (Embedded System Description Environment) by Goedicke [1986] of which EDE is a powerful version of Petri Nets and also work by Willson and Krogh [1990].

Furthermore, Milner's CCS is an algebraic model for defining the behaviour of asynchronous systems by means of recursive parameterised equations. An industrial case study on the specification of milling processes which uses CCS like expressions to specify the observable behaviour of the system was carried out by Puente *et al.* [1986]. CCS specifications have also been written for a lift control system.

2.2.4 Conclusion

In the previous sections, although the existing formal specification methods are classified into three different approaches under the two broad classes: property-orientated and model-orientated methods, a number of the specification methods are in fact inter-disciplinary. Several specification methods use or can be used in a mixture of approaches. The Z specification method can be used in either a model-oriented or a property-orientated way, whereas CSP uses a model-orientated method for specification and a property-orientated approach for stating properties about the model. Also, property-orientated methods such as OBJ can be used in an axiomatic or algebraic approach; or even a mixture of the two in system specification.

The recent trend of specification method development sees the amalgamation of a number of more established specification methods in search of greater expressiveness and versatility. Examples such as the EMS/RTTL system proposed by Ostroff [1989], the SIGNAL system by Guernic [1986] with properties that relate to the two languages ADA [US Dod, 1983] and OCCAM [INMOS, 1988a], BSI-VDM by Bloomfield *et al.* [1989], LOTOS which is a combination of Act One and CCS by LOTOS [1984a], the Raise project [Nielsen, et. la., 1989] describes more recent work on combining VDM and CSP, and Lamport's transition axiom method [Lamport, 1983] which combines an axiomatic method for describing the behaviour of individual operation with temporal logic assertions for specifying system properties.

Sometimes, even though effort has not been put in the development of a combined method, many system specification projects utilise a number of formal specification approaches such as the VIPER project [Cullyer, 1989] for the development of a correctly proven microprocessor which employs Gordon's Higher Order Logic (HOL) [Gordon, 1987] for a top level specification, VISTA for specifying operational software and ELLA for hardware specification; work by Goedicke [1986]

on real-time systems using an algebraic specification approach and Petri Nets; and also work by Puente *et al.* [1986] on a process control system using both Act One and CCS.

Apart from the major characteristics of formal specification methods (property and model-orientated method) being discussed, other characteristics which help to distinguish different formal methods include:

- *visual languages*: of which their syntactic domains involve graphical elements such as the Jackson System Development (JSD) [Jackson, 1983] which has also been classified as a structured method.
- *executable*: in other words, these specifications can be run directly on a computer, examples such as Prolog and OBJ.
- *Tool Support*: some specification languages are associated with syntactic checkers, model checkers or semantic analysis tools. Example such as the LARCH prover, B tool, the Edinburgh Concurrent Work Bench, etc. and in Chapter 6 further discussion on tool support is given.
- *Hardware Specification*: some specification languages are particularly developed for hardware specifications, an established example is the HOL system and its related work by Marshall [1986].

For general thoughts on specification methods for both software and hardware, the interested reader is referred to Parnas [1972b], Guttag [1975], Brown *et al.* [1988], Luk [1988], Parnas and Clements [1986], Bowen [1988], Djian and Probert [1990] and comprehensive overviews of formal specification methods is found in Berg *et al.* [1982], Wing [1990], and Blackhouse [1986].

2.3 Formal Verification Techniques

The notion of formal verification, in its broadest sense, is to show and reason with mathematical rigour, that a given implementation of a system *satisfies* its formal specifications, which reflect its original requirements. Hence, the objective of formal verification is to establish the truthfulness of statements such as "an implementation meets its specification" or "a specification satisfies another specification". As such if a "satisfies relation" holds for a particular implementation/specification pair or a related pair of specifications, it can be concluded that this is a *correct* implementation of its corresponding specification or this specification is a *correct* refinement of a more abstract specification and hence the notion of *correctness-proof*.

A number of techniques for formal verification have been developed and their fundamental differences arise primarily from the differences in approach to the specification methods used [Berg *et al.*, 1982]. Such differences result from the different semantic definitions of the syntactic units of the applied specification methods. The following three major verification techniques are distinguished and are discussed briefly in the following sub-sections:

- The operational approach.
- The denotational approach.
- The axiomatic approach.

2.3.1 The Operational Verification Technique

The operational verification technique verifies specifications written in a language with semantics defined in terms of state spaces and operational transforms by symbolic execution [Darringer and King, 1978]. Specifications are executed with symbolic inputs that system behaviour is observed through the symbolic values of the outputs which are independent of the particular initial and final states. Using an operational method for verification, system specifications can be tested without the need to generate lengthy numeric test cases. Since the correctness of a system is established by tracing through the whole system specification, in the occurrence of an alternative and/or iterative constructs in a specification, every alternative condition has to be traced and in the later case, an infinite trace can result which makes exhaustive tracing impossible.

2.3.2 The Denotational Verification Technique

A denotational approach to verification appears to be similar to the operational approach, as the specifications from which verification starts express the system software in the form of *functions*. Verification is then carried out by evaluating these functions to deduce the system behaviour. However, the method of evaluating these functions follows that of Blikle and Mazurkiewicz [1972], which in contrast to symbolic execution, the proof-of-correctness does not rely on traces. Rather, the input and output relationships that implicitly denote the symbolic values associated with the specifications are used to conclude system correctness. Other methods of evaluating a specification which follows a denotational approach include the use of λ-Calculus discussed in Church [1951].

2.3.3 The Axiomatic Verification Technique

The axiomatic verification technique is the most popular approach among the three techniques. The mathematical basis of the axiomatic technique is the *deductive system* which was proposed by

Hoare [1969], in which he suggested an alternative to the inductive assertion method of Floyd [1967]. But similarly, both methods define verification as some form of *predicate transformation.* Initially, the axiomatic verification method was applied to the verification of partial correctness of specifications. Later works by Gries [1976] and Dijkstra [1976] have developed new theorems for the verification of total correctness of specifications. Development of axiomatic methods then took a step towards the verification of parallel systems following the works by Owicki and Gries [1976] and Owicki [1975]. These works were then unified by Brookes and Hoare [1984] in the theory of Communicating Sequential Processes in which a method for specifying and verifying concurrent systems based on the axiomatic method was presented.

The core of an axiomatic method is its deductive system which is composed of a set of *axioms* and *rules of inference* by which *theorems* may be derived from these axioms and established theorems. Axioms can be thought of as "basic facts" about the arithmetic and comparison being taken place; whereas theorems are more complex facts.

The *rules of inference* are the means whereby new theorems are derived from established theorems, and in general, an inference rule is written as:

$$\frac{\begin{array}{l} T_1 \\ T_2 \end{array}}{\Rightarrow \quad T_3}$$

The theorems T_1 and T_2 which lie above the horizontal line are valid theorems and the consequent theorem T_3 is also a valid statement according to the rule. Translating the above inference rule into words is read as:

IF T_1 is true AND T_2 is true

THEN T_3 is also true

Hoare [1969] has proposed the following three basic inference rules for the deduction of new theorems and the derivation of more complex inference rules.

IR–I Rule of Consequence

IR–II Rule of Composition

IR–III Rule of Iteration

The *proof* of a theorem in a deductive system consists of a sequence of valid statements accompanied by justifications explaining why these statements are valid. A justification can either

be an axiom, or an inference rule by which a new theorem is shown to follow from another known theorem. And in many cases, the validity of the result of a system implementation will depend on the situation before an operation is executed and the situation when the operation terminates. Such a *before* situation is known as the *precondition* and the *after* situation is known as the *post-condition*. These conditions are specifications written as predicates which describe the behaviours of the implementation and they are said to be the *assertions* to the particular implementation. To state the required connection between a precondition P, an implementation I and a postcondition Q, the following statement is used:

$$\{P\}\ I\ \{Q\} \tag{2.1}$$

As interpreted by Hoare, Equation 2.1 says: "if the assertion P is true before the initiation of an implementation I, then the assertion Q will be true on completion." And if there exists no preconditions, Equation 2.1 is then written as:

$$\textbf{True}\ I\ \{Q\} \tag{2.2}$$

The three inference rules given above can be formally written in the pre/post-condition form and are given in Hoare [1969].

Application of the axiomatic verification method follows two basic approaches: the *forward* and the *backward* approaches. The forward approach is used when the precondition and the implementation are given such that:

$$\{P\}\ I\ \{\ \} \tag{2.3}$$

and the deductive system is used to find the corresponding postcondition Q. Whereas the backward approach starts with postcondition and the implementation:

$$\{\ \}\ I\ \{Q\} \tag{2.4}$$

and the precondition is deduced. For a conditional or iterative implementation, axiomatic verification can be applied *inductively* starting with an initial input assertion (the first precondition), successive output assertions (intermediate postconditions) can then be deduced until some termination assertions for the implementation are satisfied.

2.3.4 Axiomatic Verification Methods on Concurrent Systems

The introduction of non-deterministic behaviours in concurrent systems has made the task of verification more complicated than for sequential systems. This non-deterministic nature is often

due to the unpredictability in the speed of execution of each of the composite sub-system within a parallel system and the non-deterministic nature of the environment such as human beings. Hence, describing the evolution of the overall system is difficult. A number of attempts have been proposed to formalise and hence to verify the correctness of concurrent systems; approaches includes the use of *Monitor* by Howard [1976], the Communicating Shared Resources by Gerber and Lee [1989], the family of CSP formalisms originated from Brookes and Hoare [1984] and work by Owicki and Gries [1976] on the verification of partial correctness of parallel systems using a set of extended axioms.

Strategies for the verification of concurrent systems depend on the model of formalism being used. Two different families of formalism are proposed:

1. Non-interference Parallelism

2. Synchronised Parallelism

With the non-interference parallelism, processes of a system have no interactions with each other and therefore they exhibit "free" parallelism. To verify the correctness of such system implementations, component sequential processes are first proved to be correct by following the Hoare's axiomatic approach, then verification of no process interaction has to be undertaken to establish the condition of non-interference parallelism. This is carried out by showing the order of execution of component processes will not invalidate the postconditions of processes.

For synchronised parallelism, there exist interactions between concurrent processes and the existence of *shared resources* making it essential to ensure mutually exclusive accessing of all shared resources for system correctness. Further discussion on how to model and verify synchronised parallelism and the used of CSP as a formalism for concurrent system will be presented in the subsequent chapters.

2.3.5 Limitations of Axiomatic Verification Technique

Since an axiomatic approach is based on a set axioms and the inference rules, in order to believe that a proof in such a deductive system is correct, one must accept the validity of the axioms and the inference rules. In most cases, the length of proofs necessary to show the correctness of software are an order of magnitude longer than the implementation and as a consequence may be more error-prone than the actual implementation.

Another limitation arises from the need for an abstract view of the desired system implementation and this introduces a distance between a formally verified implementation and the correspond-

ing physical system. Moreover, although we may be able to verify a high level implementation, but before a correct system behaviour is to be guaranteed, the correctness of the compiler, the assembler or even the hardware have to be verified. Even though it may be possible to verify the correctness down to the 'chip level' with great effort, there exists a 'bottom line' in the verification hierarchy in which the functioning of hardware is governed by the laws of nature which are impossible to verify. It is therefore necessary not to over claim the validity of a proof, and accept at best that an implementation is correct with respect to its execution on some abstract machine, which is modelled with a deductive system.

Furthermore, the expressiveness of a specification language is limited and the completeness of the deductive system used is constrained. There are real life instances such as stochastic and non-linear behaviour which at present cannot be described formally and hence their semantics are not expressed completely as rules of inference in a deductive system. This in effect limits the scope of application of the axiomatic approach to only a subset of the real behaviour that can be formally specified. The theoretical treatment of the incompleteness of a deductive system can be found in Gödel [1931].

Finally, as the process of verification or proof-of-correctness is highly creative, some proofs do seem to be more credible than others. And in most cases, a complete step-by-step proof is not presented because the task of producing each step is so large so as to be prohibitive, and the arguments and justifications given are sometimes too informal. As a result, only a person who can internalise the steps of the proof will be convinced.

2.3.6 Pragmatics

Despite the primary objective of formal verification to establish the correctness of specifications with respect to a system design, there are a number of verifiable properties which prompt the various proof obligations. A subset of these verifiable properties that are related to control systems includes the logical consistency of system components with respect to the design, consistency of overall system topology, continuity of processes constituting the system as described by Hoare [1985b], Wojcik and Wojcik [1989] and Owicki and Gries [1976], termination of process described by Chen and Hoare [1982], *livelock* and *deadlock* freedom of interacting processes described by Roscoe and Dathi [1986], stability of real-time systems described in the preliminary work by Davies and Schneider [1989a] and various properties for the correctness of hardware systems described by Cohn [1989]. Among the set of verifiable or provable properties, they can be divided into two main categories: the *safety* and *liveless* properties.

- Safety properties are defined as those properties of a system which state that "bad things will not happen". All the above properties except the termination of processes are safety properties which are used to establish the *partial correctness* [Chen and Hoare, 1981a] of the system.

- Liveness properties are defined as those properties of a system which state that "good things will happen". Liveness properties include the safety properties plus the *termination* condition. To consider the liveness properties of a system is equivalent to considering its *total correctness* [Chen, 1982].

2.4 Motivation for Formal Specification and Verification

The drive for the application of formal methods to systems design and development which include system software and hardware resulted from the emergence of stringent *standards* which products must comply within *safety-critical* applications [Cullyer, 1989] such as public transport systems and nuclear installations. Failures of these systems will risk human lives and it is of vital importance to have a sound and proven methodology to design and develop correct and reliable systems software as suggested by Leveson [1991]. Other main driving forces behind the use of formal methods include the increase in *complexity* of problems that are otherwise unmanageable (such as designing a single VLSI chip with over 10^6 devices; and for software systems, the freedom in design and development is of orders of magnitude greater than that of hardware), the need to reduce *product design cycle time* so as to remain competitive, the desire to *reuse* parts or the whole of a previous design with minimum re-design and the fact that testing on a limited number of experimental cases can never guarantee system correctness and to predict system behaviour. Applications of formal methods to system development has indeed revealed a number of these appealing features such as the design and development time of the T800 Transputer has been shortened by 12 months [Jones, 1990] and the correctly verified VIPER microprocessor [Technologies, 1988; Wise, 1989]. These results have strengthened the argument for such applications.

2.4.1 Advantages and Drawbacks

It is apparent that the task of producing a *correct* formal specification is certainly as demanding as constructing a correct implementation, and the same degree of understanding of the real system behaviour is required. Nevertheless, the effort of specification construction is considerably less than that required for an implementation as a result of the abstraction employed. In the long term, going

through the process of formal specification and refinement rather than a direct implementation will save time and money.

As said, one of the strengths and properties of a specification is that it *should* be implementation independent. On the other hand, because of this very fact, looking at a specification alone cannot provide answers to those questions such as how difficult it is to implement a system and what is the subsequent performance.

Using formal methods for system specification provides all parties concerned with some procedures that have a theoretical foundation, which supports the validation of requirements, and is largely self documenting in a syntax of a particular formal language used. In addition to be a valuable thinking tool, formal methods shift the inevitable *iterative* aspects of the design process from the implementation phase to the specification phase, thereby treating the *cause* and not the *symptoms* of any potential problems. Formal methods address the *prevention* or *avoidance* of undesirable situations whereas an implementation orientated method works on the recovery of problems.

However, the applications of formal methods for practical software and hardware development are still in their "first generation", and quite often, the underlying theoretical foundations are still under active research and development. There are few case studies on real-time and distributed systems.

Problems in communication between the people involved using formal methods in system design may arise: the designer (who may be an engineer) may not be familiar with the concept of a formal method, whereas the specifier (usually a mathematician) may be overwhelmed by the concepts and terminologies used by the designer about the system being designed. It is therefore necessary for the former to be educated in mathematics to a level at which formal specifications are to be understood and the later be introduced to the required information on the real system before the two parties can exchange ideas constructively. Quite often, the process of learning new knowledge and concepts may not be a straight forward task.

Another problem is that of keeping to the right level of abstraction in specifications, in that the specification must convey the primary operation of a system being specified while not being biased towards a particular implementation. A possible solution to overcome this problem is to "specify in small" as in "programming in small" [Gries, 1981]. Specifications can be constructed and structured in a hierarchical manner, each specification specifying only a single operation in a system and different components of a system are specified using separate, modular specifications

with less complexity. Overall system specification is then obtained by combining these modular specifications.

2.4.2 Who will Benefit?

Formal methods for system development, both in hardware and software can be used by:

- *designers*: formal methods are used to formulate and experiment with their design ideas in a highly abstracted format. A handful of promising ideas can therefore selected for further development.
- *implementors*: detailed formal specifications with adequate implementation information can assist implementors to build a precise realization of the system designed.
- *documenters*: as formal specifications are an unambiguous and precise document for a system, specifications may be used as a starting point for compiling a user manual and a development document for subsequent system modification and enhancement.
- *quality control*: formal specifications enable suitable system testing strategies to be developed.

2.5 Communicating Sequential Processes

2.5.1 History and Motivation

The introduction and full mathematical treatment of the CSP model was first presented by Professor Hoare. The design of the language CSP by Hoare [1978], Brookes and Hoare [1984] attempts to provide a notation for *expressing* and *reasoning* about systems of concurrent processes. In CSP, *expressing* means designing, specifying and implementing whereas *reasoning* means verifying, developing and modifying. CSP offers a succinct mathematical notation for the description of processes and a way of controlling the level of abstraction of these descriptions. Processes can be structured using combinators for parallel composition, external choice, communication, sequential composition, etc. Further discussion of the semantics of the set of combinators is found in Hoare [1985b] and Sanders [1988]. The resulting expressions from these notations are powerful enough to describe a wide range of cases, yet small enough to support a body of fairly simple laws and semantic theory. Recent developments of CSP involve a number of extensions to the present model such as the Timed model of CSP which was proposed by Davies and Schneider [1989b;

1989a], also a hybrid model of ordinary CSP and TCSP is used by Liu and Shyamasundar [1990] and the one presented by Gerber and Lee [1989] for modelling priority of communicating processes.

CSP provides a powerful means of building real-time, parallel, complex programs that are ideally suited to today's hardware. Many engineering systems such as distributed control systems, are *naturally* conceived of and implemented in parallel as communicating sequential processes. A number of pilot case studies of using CSP as the formal system for system specification and verification have been carried out. These include studies of the specification of protocols by Goh [1984] and Reed [1984], on fault tolerant pipeline processors by Sheehan [1985] and specification of JSD by Sridhar and Hoare [1985]. Also more recent attempts on real-time distributed systems includes the specification on aircraft engines by Jackson [1989], on traffic control systems by Lau [1990], on robot controllers by Lau and Daniel [1989], on distributed sensors by [Djian and Probert, 1990] and on an autonomous guided vehicles by [Stamper, 1990] using TCSP.

2.5.2 The Models

To characterise the behaviour pattern of an object or in general, a process, three fundamental semantic models are given in Hoare [1985b] and Sanders [1988] which are the *traces*, *failures* and the *stability* models, these three models are closely related to the *traces*, *refusals* and *divergences* of CSP.

Among the three semantic models which are based upon metric spaces, projection mappings have been defined which allows the various aspects of a process's behaviour to be considered separately. Figure 2.2 represents these mapping by arrows connecting the various models.

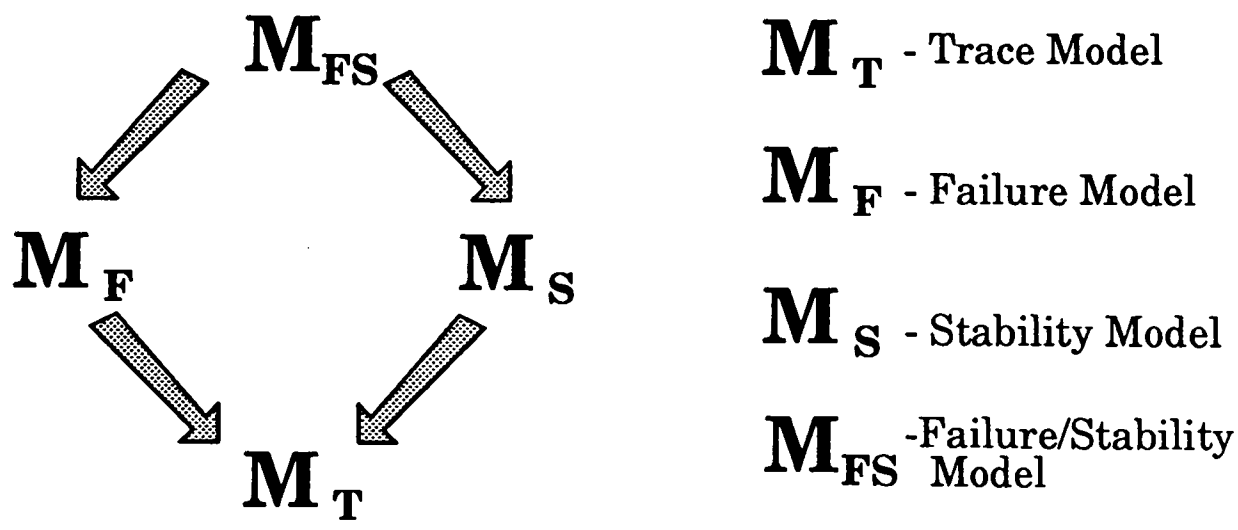


Figure 2.2: The metric space mapping of the different models of CSP

2.5.2.1 Traces

A process can be specified by its traces which is a record of a finite sequence of *events* in which the process has engaged up to a moment in time. In real terms, this represents a time history of the evolution of a process. An *event* is an action performed by a process, under the assumption that the actual occurrence of each event in the life of a process should be regarded as instantaneous or as an atomic action without duration. Also, a fundamental assumption is made on the exact timing of the occurrence of events: this timing parameter is abstracted from the model, which in effect generalises the model for all similar systems independent of speed and performance.

Equation 2.5 defines formally the *satisfies* relation of a process P that is exactly modelled by its specification in the traces model.

$$P \ \textbf{sat} \ S(tr) \tag{2.5}$$

Here, $S(tr)$ is a specification stating the behaviour of this process in terms of traces.

2.5.2.2 Failures

The introduction of the *failures* model [Brookes, 1983] distinguishes between the deterministic and non-deterministic behaviour of processes using the *refusal set*. Formally, a refusals set, *ref*, is written as in Equation 2.6.

$$ref \in refusals\,(P) \tag{2.6}$$

The refusal set or refusals for short, is a set of event sequences which is refused by the process P. In this equation, if P deadlocks on its first step when being placed in an environment, then *ref* is a refusals of P whereas *refusals* (P) is the set of all such refusals of P and is itself a super set. Furthermore, the set of refusals of a process is only indirectly observable.

For a non-deterministic process, both the traces and the refusals are used to specify its behaviour. In CSP, the formal definition for such a process is given by the following:

$$\begin{aligned} P \quad \textbf{sat} \quad & S(tr,\ ref) \\ & \Rightarrow \forall\,(tr,\ ref) \mid \left(\begin{array}{ll} & tr \in traces\,(P) \\ \wedge & ref \in refusals\,(P/tr) \end{array} \right) \end{aligned} \tag{2.7}$$

according to the semantics of the model, Equation 2.7 states that a process P is completely specified by its traces and refusals in the specification $S(tr, ref)$ in which the process's behaviour are described by its trace, tr, up to a point in time, and after that it refuses or deadlocks on the set of sequences given by its refusal set, *ref*. The definition is equivalent to the definition of the

failures model by which a process is defined as a relation such that:

$$failures\ (P) = \{(tr,\ ref) \mid tr \in \text{traces}(P) \wedge ref \in refusals\ (P/tr)\} \tag{2.8}$$

2.5.2.3 Stability

In addition to the traces and refusals of a process which record the reactions of a process in response to external stimuli, internal activities of a process can be modelled using the *stability* model of CSP [Davies and Schneider, 1989b]. This model is used to model internal transactions of a process and to investigate whether a particular process has stabilised at some internal event. Once a process is stabilised, there can be no further changes of state without an occurrence of external action. The concept of stability is dual to that of divergence which is discussed in Hoare [1985b].

2.5.3 The Notation

Although a full set of CSP notations is well defined in Hoare [1985b], a subset of these notations is given in Appendix A which is specific in reasoning about general distributed computer control systems. This particular set of notations given will also serve as a reference to the notations used in this thesis for the specification of control systems.

2.5.4 CSP Processes

The CSP models assume that processes communicate by sending messages across channels and a pair of communicating processes synchronise on a communicating event. The scenario is captured by Figure 2.3. Process A and B evolve concurrently in time and they both synchronise on a communication in which process A sends an output message **m** via channel c and process B inputs the message, with both processes participating simultaneously in this interaction.

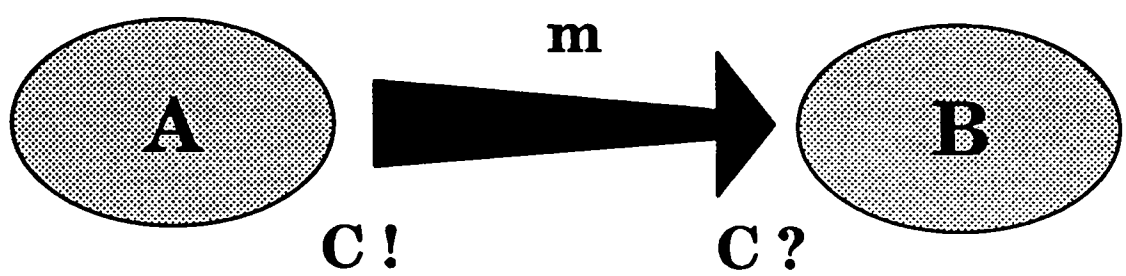


Figure 2.3: The CSP model of communication

2.5.5 Process Algebra

CSP provides a set of algebraic laws on processes — the *process algebra*. These laws are designed to transform and hence "to calculate" the set of traces and refusals of any processes defined using

CSP notations. In doing so, a complex CSP process can be transformed to a more transparent form in which its traces or refusals properties are well defined, and through such transformation, the behaviour of the original complex process is implied.

A subset of the full process algebra given in Hoare [1985b] is given by Equation 2.9 which are used in the later chapters of this thesis for specifying control system software.

$$\begin{aligned} P \quad ::= \quad & STOP \mid SKIP \mid a \rightarrow P \mid P/s \mid c\,!\,x \rightarrow P \mid c\,?\,x \rightarrow P(x) \mid \\ & P \,\Box\, Q \mid P \sqcap Q \mid P \parallel Q \mid P \,|||\, Q \mid P \,;\, Q \mid P \backslash A \mid f(P) \mid \mu X \bullet F(X) \end{aligned} \tag{2.9}$$

From a pragmatic point of view, a number of CSP operators are considered to be of engineering significance for expressing the behaviour and structuring control process definitions. These include the *hiding operator* for controlling the degree of abstraction, *process relabelling* for the control of modularity, and *input/output operators* for modelling synchronous communication. Also, in Hoare [1985b], the notion of alphabet consistency for the deterministic and non-deterministic choice operation has not been treated clearly. In this sub-section, a formal treatment of the alphabet sets of processes with the *choice* operators together with the physical interpretations of the above mentioned operators are presented. Operators given by Equation 2.9 that are not discussed in this sub-section and the corresponding laws on process algebra are referred to those given in Hoare [1985b].

2.5.5.1 Alphabet of Processes with *Choices*

It is *assumed* in Hoare [1985b] that the alphabet for a process with either deterministic or non-deterministic choices is defined according to Equations 2.10 and 2.11.

$$\alpha(P \,\Box\, Q) \;=\; \alpha P = \alpha Q \tag{2.10}$$

$$\alpha(P \sqcap Q) \;=\; \alpha P = \alpha Q \tag{2.11}$$

However in the literature, there is no clear indication of *why* this is the case or *what* is the interpretation of these equations, neither logically nor physically. A closer look into the various laws on processes with choices reveals that the consistency of alphabet laid down by Equations 2.10 and 2.11, that is:

$$\alpha P \;=\; \alpha Q \tag{2.12}$$

is a requirement or precondition for the choice operator to exist. If however, the choice operation is allowed to operate on processes with different alphabets, then the alphabet of the resultant

process will be dependent on which constituent process is being chosen at a particular instance, which violates the re-writing rules for process algebra — such as the laws on parallelism. Formally, two processes with different alphabet can be combined using the choice operators after the augmentation of their alphabets. Putting each constituent process in parallel with the null process *STOP* with a restricted alphabet set preserves the properties of the constituent process as well as augmenting its alphabet set. Similar augmentation to the other process satisfies the condition of alphabet consistency. To illustrate this point, suppose processes P and Q have different alphabets αP and αQ, and in order to combine the two processes using a deterministic choice, augmentation of their alphabet is carried out according to Equations 2.14 and 2.15 such that the condition set out by Equation 2.10 is satisfied as given by Equation 2.16 and their properties are preserved.

$$\text{If} \quad \alpha P \neq \alpha Q \tag{2.13}$$

$$P_{A'} \mathrel{\widehat{=}} (P \parallel STOP_{\alpha P - \alpha Q}) \tag{2.14}$$

$$Q_{A'} \mathrel{\widehat{=}} (Q \parallel STOP_{\alpha Q - \alpha P}) \tag{2.15}$$

$$\text{then} \quad \alpha(P_{A'} \,\Box\, Q_{A'}) = \alpha P_{A'} = \alpha Q_{A'} = \alpha P \cup \alpha Q \tag{2.16}$$

With this formal treatment, the requirement for alphabet consistency is clarified which also enables the combination of processes with different alphabets using the choice operators. However, in an implementation, such an augmentation is seldomly carried out and the above treatment is most significant in reasoning using process algebra. In the context of the thesis, processes with different alphabets are combined according to the above treatment, nevertheless, for conciseness, the operation given by Equations 2.14 and 2.15 are implicitly assumed in process definitions and proof construction.

2.5.5.2 Abstraction

The hiding operator $P \backslash A$ allows internal details of a process to be hidden from the environment. The hiding operation has the effect of forcing the hidden internal events to occur automatically and as soon as they are available, and these hidden events are no longer observable from the environment through a recorded trace. In this case, A is a set of events in which their occurrence in process P is hidden.

As an example, if the process P has channel b in its alphabet and suppose that b is a channel that connects two sub-processes of P together. Communication between subprocesses of P along channel b is regarded as an internal activity of P and occurs automatically and instantaneously as soon as the sub-processes connected by this channel are ready for it. To abstract this internal

event from the environment, the following is used:

$$P\backslash b \tag{2.17}$$

to denote a process which behaves like P, except that the channel b is concealed or localised. The notation can be extended to an alphabet set such that the alphabet of the resulting process is:

$$\alpha(P\backslash A) = (\alpha P) - A \tag{2.18}$$

2.5.5.3 Modularisation

The notion of process relabelling or event renaming enables the parameterisation of events in a process. This supports the design of generalised processes which adhere to the concept of modularisation. The process $f(P)$ is obtained by applying the alphabet transformation f to the events describing the of process P. And the second form of process relabelling is parallel to the reciprocal of a function that reverse the operation performed by the first transformation. $f^{-1}(P)$ allows all possible processes which behave in a fashion similar to P to be considered.

As an example, if P is a process with channel c in its alphabet, and let d be a new channel which is *not* in the alphabet of P. Then $P[d/c]$ is a process which behaves exactly like P, except that whenever P uses c for communication, $P[d/c]$ uses d instead. A new process is formed as a result. And,

$$P[d/c] \equiv f(P) \tag{2.19}$$

The alphabet of $P[d/c]$ is defined by Equation 2.20:

$$\alpha(P[d/c]) = (\alpha P - \{c\}) \cup \{d\} \tag{2.20}$$

This construct can be extended to multiple renaming where $P[d_1, d_2, \ldots, d_n/c_1, c_2, \ldots, c_n]$ denotes all channels named under c_i are renamed as d_i in the order described or trace relabelling.

2.5.5.4 Communication

The process $(c\,!\,e \rightarrow P)$ is used to model an output process where e is an expression or a message and the process algebra defined such a process as one which first outputs e on the channel c and then behaves like P afterwards.

Whereas $(c\,?\,x \rightarrow P(x))$ is used to model an input process. $P(x)$ is a process whose behaviour depends on the value of the free variable x. The process inputs a message which contains the free

variable x from channel c, there after, the process behaves as $P(x)$. The variable x is regarded as a bound or local variable, such that:

$$c\,?\,x \rightarrow P(x) \equiv c\,?\,y \rightarrow P(y) \tag{2.21}$$

In modelling a synchronous communication between two CSP processes using the output and input operators, the value of x in the inputing process is usually determined by the corresponding outputting process ($c\,!\,e \rightarrow P$). And if the communication is between two concurrent processes such as:

$$\begin{aligned}&(c\,!\,e \rightarrow P) \,||\, (c\,?\,x \rightarrow Q(x))\\ &\quad \widehat{=} \quad c\,!\,e \rightarrow (P \,||\, Q(e))\end{aligned} \tag{2.22}$$

The communicating process as given by Equation 2.22 can be interpreted as the scenario for which the outputting action is an *observable action* of the system. This corresponds to a real physical system where internal communication or transaction can be logged, such as in physical terms, by "tapping" off signals from the "wires" connecting the components using appropriate instrumentation. Whereas the inputing action is an *assignment* in which the free variable x in this case has been assigned the value of e after the communication has taken place.

$$x := e \tag{2.23}$$

As communication is a special form of prefix operation, the alphabet of the input and output processes are identical to the definition of a general prefixing.

2.5.5.5 Modelling Processes Using CSP

To illustrate the use of CSP notations for process modelling, a simple example is studied to reveal their usages and expressiveness in a distributed environment. Two properties that are central to this environment: the *sequential ordering* and *parallel execution* of events within parallel processes are illustrated together in this case study.

The system is a model of a communication protocol. It consists of two processes: a *transmitter* and a *receiver* together with four communication channels. Altogether, they model a flow of discrete messages from the *left* channel to the *right* channel via a non-dispersive media as shown by Figure 2.4. In order to ensure a smooth flow of information, a control signal is sent from the receiver to acknowledge the receive of messages before new messages are transmitted.

Specifications describing formally the behaviours of the two communicating processes are written down using the traces semantics according to the design of the model. The following predicates

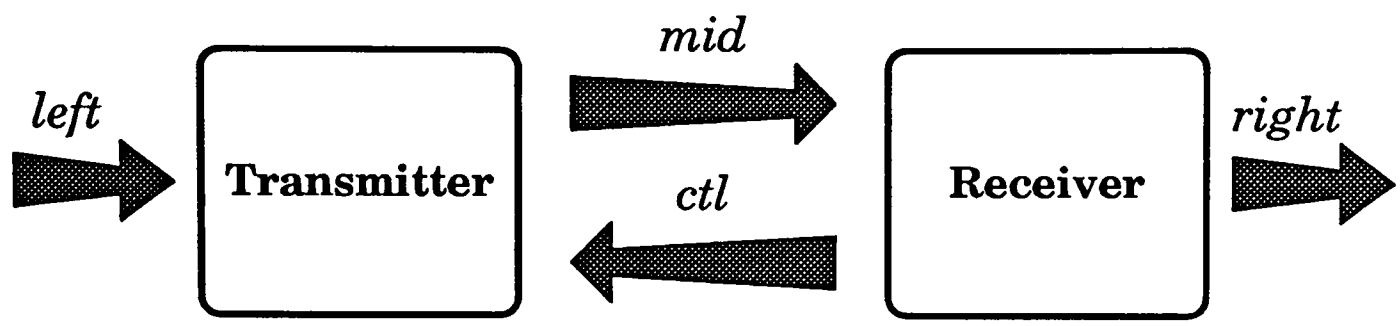


Figure 2.4: A model of a simple communication protocol

specify the actions of the two processes in terms of their input/output characteristics.

$$Transmitter \quad : \quad (tr \downarrow ctl \overset{1}{\leq} tr \downarrow mid) \wedge (tr \downarrow mid \overset{1}{\leq} tr \downarrow left) \tag{2.24}$$

$$Receiver \quad : \quad (tr \downarrow ctl \overset{1}{\leq} tr \downarrow right) \wedge (tr \downarrow right \overset{1}{\leq} tr \downarrow mid) \tag{2.25}$$

With these predicates, corresponding process definitions which reflect the operations of the two processes are constructed according to Equations 2.26 and 2.27.

$$Transmitter \quad \widehat{=} \quad \mu X \bullet (left \; ? \; x \rightarrow mid \; ! \; x \rightarrow ctl \; ? \; z \rightarrow X) \tag{2.26}$$

$$Receiver \quad \widehat{=} \quad \mu X \bullet (mid \; ? \; y \rightarrow right \; ! \; y \rightarrow ctl \; ! \; z \rightarrow X) \tag{2.27}$$

In order to predict and to understand the behaviour pattern of this communication protocol given the above two process definitions, process algebra is used. The overall system P which models the behaviours of this protocol, is given by Equation 2.28 with the processes *Transmitter* and *Receiver* executing concurrently.

$$P \quad \widehat{=} \quad (Transmitter \parallel Receiver) \tag{2.28}$$

Expansion of process P using process algebra to eliminate the parallel operators show that the communicating events of the two processes follow a unique sequence. Also, the resultant process definition of P is in the form of a recursive process implying that this communication protocol under the described realization is free from deadlocks . The relevant process algebra is given as follows:

$$P \quad \widehat{=} \quad (Transmitter \parallel Receiver)$$

$$\equiv \quad (\mu X \bullet (left \; ? \; x \rightarrow mid \; ! \; x \rightarrow ctl \; ? \; z \rightarrow X))$$
$$\parallel (\mu Y \bullet (mid \; ? \; y \rightarrow right \; ! \; y \rightarrow ctl \; ! \; z \rightarrow Y))$$

$$\equiv \quad \mu X \bullet (left \; ? \; x \rightarrow mid \; ! \; x \rightarrow right \; ! \; x \rightarrow ctl \; ! \; z \rightarrow X)$$

As one of the prime concern in this study is the ordering of communicating events, investigations of the effects of varying the ordering of communications within an individual process are carried out. Firstly, the effect on the overall system behaviour under a different realization of the receiving process is investigated. By reversing the order of the communications via channel *ctl* and *right*, a new receiving process definition, $Receiver_2$ is obtained. Equation 2.29 gives the new definition of this process.

$$Receiver_2 \;\widehat{=}\; \mu X \bullet (mid\ ?\ y \rightarrow ctl\ !\ z \rightarrow right\ !\ y \rightarrow X) \tag{2.29}$$

For the properties of the new model, the following process algebra provides the relevant information.

$$P_2 \;\widehat{=}\; (Transmitter \,||\, Receiver_2) \tag{2.30}$$

$$\begin{aligned} \equiv \quad & (\mu X \bullet (left\ ?\ x \rightarrow mid\ !\ x \rightarrow ctl\ ?\ z \rightarrow X)) \\ & ||\, (\mu Y \bullet (mid\ ?\ y \rightarrow ctl\ !\ z \rightarrow right\ !\ y \rightarrow Y)) \end{aligned} \tag{2.31}$$

$$\begin{aligned} \equiv \quad & left\ ?\ x \rightarrow \mu X \bullet (\ mid\ !\ x \rightarrow ctl\ !\ z \rightarrow right\ !\ x \rightarrow left\ ?\ x \rightarrow X \\ & \quad |\ mid\ !\ x \rightarrow ctl\ !\ z \rightarrow left\ ?\ x \rightarrow right\ !\ x \rightarrow X) \end{aligned} \tag{2.32}$$

From the definition of the expanded process P_2 given by Equation 2.32, it is seen that its evolution is not a unique recurrence of communicating events as in the first case. The existence of a choice between alternative evolutions has resulted in this non-uniqueness behaviour of this realization. In this case, the order of communication is determined by the readiness of the environment to offer communications through either channel *left* or *right*.

Finally, if the specifications of the process $Receiver_3$ is constructed such that its realization follows that of Equation 2.33, with its first and last communications across channels *mid* and *ctl* being reversed.

$$Receiver_3 \;\widehat{=}\; \mu X \bullet (ctl\ ?\ z \rightarrow mid\ !\ y \rightarrow right\ !\ y \rightarrow X) \tag{2.33}$$

The resulting model which is made up of processes *Transmitter* and $Receiver_3$ will have different properties given by the following process algebra.

$$P_3 \;\widehat{=}\; (Transmitter \,||\, Receiver_3) \tag{2.34}$$

$$\equiv \quad (\mu X \bullet (\mathit{left} \,?\, x \rightarrow \mathit{mid} \,!\, x \rightarrow \mathit{ctl} \,?\, z \rightarrow X)) \tag{2.35}$$

$$\| \, (\mu Y \bullet (\mathit{ctl} \,!\, z \rightarrow \mathit{mid} \,?\, y \rightarrow \mathit{right} \,!\, y \rightarrow Y)) \tag{2.36}$$

$$\equiv \quad \mathit{left} \,?\, x \rightarrow (\quad (\mathit{mid} \,!\, x \rightarrow \mathit{ctl} \,?\, z \rightarrow \mathit{Transmitter}) \tag{2.37}$$

$$\| \, (\mathit{ctl} \,!\, z \rightarrow \mathit{mid} \,?\, y \rightarrow \mathit{right} \,!\, x \rightarrow \mathit{Receiver}_3)) \tag{2.38}$$

$\Rightarrow$ **deadlock**

It is seen that the communication between the two recursive parallel processes ceases to proceed after an input from channel *left*, the scenario is generally known as a *deadlock*.

It is apparent that only a slight alteration in the ordering of communicating events in the process *Receiver* leads to significant changes in the properties of the overall system. The first parallel realization of the protocol given by Equation 2.28 produces a system with a well defined and unique ordering in the occurrence of communication which is *independent* to the readiness of the environment. Whereas in the second case, the ordering of communications has become unpredictable and the particular evolution depends on the environment. More significantly, the change in event ordering in the last case has resulted in the catastrophic deadlock. Also, in this example, the use of CSP formalism for system modelling allows the exploration of "true" parallelism of concurrently specified distributed processes.

Although this example has deliberately been made simple, it has illustrated some of the techniques of modelling concurrent systems using the notations of CSP and at the same time, demonstrated the use of process algebra as a powerful means of conducting simple proofs on the correctness of system realizations in an early stage of system design.

2.5.6 Specification and Verification of Satisfaction

CSP uses a model orientated method for specifying concurrent processes and a property-orientated method for stating and proving properties about the model. In the following sections, the idea of process specification and verification of *satisfies* relations are discussed.

2.5.6.1 Process Specification

A CSP specification is a predicate containing free variables such as *traces* and/or *refusals* which describes some of the observable aspects of the behaviours of a process. In the context of a process,

its traces are direct and obvious observable behaviour by which its operation is characterised up to a given moment in time. Its refusals is a more implicit and indirect description of its behaviour after successfully performing a sequence of events. In Sub-section 2.5.5.5, the use of traces for process specification has already been discussed briefly.

Here, as a more realistic example is an *unbounded buffer* modelled using mutual recursion in CSP. Its specifications are given by Equations 2.39, 2.40 and 2.41.

$$\mathbf{BUFFER} = P_{\langle\rangle} \tag{2.39}$$

$$\text{where} \quad P_{\langle\rangle} \mathrel{\widehat{=}} \mathit{left}\,?\,m \rightarrow P_{\langle m\rangle} \tag{2.40}$$

$$\text{and} \quad P_{\langle m\rangle^\frown s} \mathrel{\widehat{=}} (\mathit{left}\,?\,n \rightarrow P_{\langle m\rangle^\frown s^\frown\langle n\rangle} \mid \mathit{right}\,!\,m \rightarrow P_s) \tag{2.41}$$

Equation 2.39 states that BUFFER is specified by the process P. Equation 2.40 is a recursive definition of the process P when the buffer is empty or when it is being initialised. It states that when the buffer is empty, in the event that there is a message m appearing on the left channel, it will input it. Equation 2.41 states that when the buffer is non-empty and is in operation, then the buffer will either input another message n from its left channel, appending it to the end of the buffer, *or* output the first message in the buffer to the right channel. However, the preference of the choice between input and output communications in this case is totally determined by the environment.

The corresponding specifications for BUFFER are given by the following predicates:

$$\mathbf{BUFFER} \quad \mathbf{sat} \quad \left(\begin{array}{ll} & \mathit{tr} \downarrow \mathit{right} \leq \mathit{tr} \downarrow \mathit{left} \\ \wedge & s = \langle\rangle \Rightarrow \{\mathit{left}\} \not\subseteq \mathit{ref} \\ \wedge & s \neq \langle\rangle \Rightarrow \{\mathit{left}\} \not\subseteq \mathit{ref} \vee \{\mathit{right}\} \not\subseteq \mathit{ref} \end{array}\right) \tag{2.42}$$

These predicates describe the desired behaviours of the unbounded buffer. The first conjunct in Equation 2.42 defines the ordering of data flow: the sequence of messages output to the right channel is a *prefix* of the messages input from the left channel. The prefix property of traces guarantees that only messages input from the left will be delivered to the right, only one at a time and in the same order as input. The second conjunct states the initial requirement when the buffer is empty. In this case, the buffer will receive any messages input from the left channel. The last conjunct states the requirement when the buffer is non-empty. In this situation, the buffer cannot refuse to communicate on either of its left or right channels, implying that a message will eventually be delivered. Using traces and refusals, safety and liveness properties of a system are specified. In this example, the first conjunct defines the system safety property whereas the second and third conjuncts define its liveness properties.

2.5.6.2 Verification of Satisfiability

As CSP is built on a formal system, proof-of-correctness is supported and it is essential to check that a CSP specification satisfies its corresponding requirements. The proof-of-correctness is performed using Hoare's axiomatic verification approach. The primary proof obligation is to verify the validity of the relation:

$$P \text{ sat } S \tag{2.43}$$

Application of mathematic induction together with rules of inference enables the relation given by Equation 2.43 to be established such that the implementation P meets its specification S. CSP provides a set of laws giving properties of this *satisfies* relationship in general and those which apply exclusively to processes to assist the process of verification. Inference laws such as the two given are found in Hoare [1985b].

Law 2.1

If P **sat** S
and $S \Rightarrow T$
then P **sat** T

Law 2.2

If P **sat** S
and P **sat** T
then P **sat** $(T \wedge S)$

Considering the unbounded buffer described in Sub-section 2.5.6.1, the process BUFFER is proved to satisfied its assertions given in Equation 2.42. The following establishes the prefix relationship on traces of the process BUFFER.

Proof 2.1 *The correctness of the first assertion given by equation 2.42*

Case: The initial case when $tr = \langle\rangle$

$$\begin{aligned} \langle\rangle &\leq left \\ tr \downarrow right &\leq tr \downarrow left \end{aligned}$$

Case: $tr^+ \neq \langle\rangle$, that is the buffer is partially filled.

In this case, there are two sub-cases, either the left channel is ready to input or the right channel is ready to output.

Subcase: Right channel to output

$$\begin{array}{lrcll}
 & last\ (tr \downarrow right) & = & last\ (tr^+ \downarrow left) & \text{def of BUFFER} \quad [1] \\
 & tr^+ \downarrow left & = & tr \downarrow left & \text{Right to output} \quad [2] \\
\text{assume} & tr^+ \downarrow right & \leq & tr^+ \downarrow left & \\
\text{then} & \left(\begin{array}{l}(tr^+ \downarrow right)^\frown \\ \langle last\ (tr \downarrow right)\rangle\end{array}\right) & \leq & \left(\begin{array}{l}(tr^+ \downarrow left)^\frown \\ \langle last\ (tr \downarrow right)\rangle\end{array}\right) & \\
 & tr \downarrow right & \leq & \left(\begin{array}{l}(tr^+ \downarrow left)^\frown \\ \langle last\ (tr^+ \downarrow left)\rangle\end{array}\right) & \text{by [1]} \\
 & tr \downarrow right & \leq & \left(\begin{array}{l}(tr^+ \downarrow left)^\frown \\ \langle last\ (tr \downarrow left)\rangle\end{array}\right) & \text{by [2]} \\
 & tr \downarrow right & \leq & tr \downarrow left &
\end{array}$$

Subcase: Left channel input

$$\begin{array}{lrcll}
 & last\ (tr \downarrow right) & = & last\ (tr^+ \downarrow left) & \text{def of BUFFER} \quad [1] \\
 & tr^+ \downarrow right & = & tr \downarrow right & \text{Left to input} \quad [2] \\
\text{assume} & tr^+ \downarrow right & \leq & tr^+ \downarrow left & \\
\text{then} & \left(\begin{array}{l}(tr^+ \downarrow right)^\frown \\ \langle last\ (tr^+ \downarrow left)\rangle\end{array}\right) & \leq & \left(\begin{array}{l}(tr^+ \downarrow left)^\frown \\ \langle last\ (tr \downarrow left)\rangle\end{array}\right) & \\
 & \left(\begin{array}{l}(tr^+ \downarrow right)^\frown \\ \langle last\ (tr \downarrow right)\rangle\end{array}\right) & \leq & tr \downarrow left & \text{by [1]} \\
 & tr \downarrow right & \leq & tr \downarrow left &
\end{array}$$

Hence, the safety property is proved. □

The other two assertions given by the later two conjuncts of Equations 2.42 can be proved using the laws of refusals. The following proof establishes the second conjunct.

Proof 2.2 *Correctness of the second assertion in Equation 2.42*

We want to prove that when $\#tr = 0$, process BUFFER must not refuse to input from left. This is the case when BUFFER is initiated. Hence the proof follows:

To Prove: $\#tr = 0 \Rightarrow \{left\} \notin refusals\,(P)$

$$\begin{aligned} refusals\,(P) &= \{X \mid X \subseteq (\alpha P - \{left\})\} && \text{by refusals L2 from Hoare [1985b]} \\ &\Rightarrow \{left\} \notin refusals\,(P) && \square \end{aligned}$$

The last conjunct in Equation 2.42 can be verified in similar manner and the result follows from that of the second conjunct. From this example, the usefulness of CSP laws on the grounds of verification is revealed.

In addition to the set of laws which are used in the proof of *satisfies* relations, the process algebra discussed in Sub-section 2.5.5 is also used as a powerful means to infer the correctness of a system or process definition. Transformation using process algebra enables a complex definition to be put in a form which is more recognisable with well defined properties; and if these properties satisfy the requirements of the original complex definition, then it implies that the original definition is also correct. The use of process algebra for the purpose of system verification has already been demonstrated by the examples described in Sub-section 2.5.5.5.

In the development of verification techniques for CSP, a number of more refined techniques have been proposed such as the verification of partial correctness of processes by Chen and Hoare [1981a], Chen and Hoare [1981b], the verification of total correctness by Hoare [1981], Chen and Hoare [1982] and the proof of deadlock freedom for parallel processes by Roscoe and Dathi [1986]. Interested readers are referred to the works by Hoare [1985a], Davies [1987], Olderog and Hoare [1984] and Jeffrey [1988a] for these refined proof strategies.

2.5.7 Weaknesses

CSP is primarily designed for modelling process communication and parallel interactions among sequential processes in term of sequences of events such that properties of the systems are inferred and predicted. Nevertheless, part of the language that characterises the formal system is still under active development. In order to facilitate a more complete modelling of real systems, further refinements of the semantics and proof systems is necessary. At the time of writing this thesis, areas of the model which are still under active research are:

- Timed CSP which aims at specifying real-time systems and the associated laws to allow reasoning in the time domain.
- Probabilistic CSP for modelling signal processing systems. This model will be particularly useful for specifying multi-sensor devices such as those found in mobile robots.

- Prioritised CSP would be a useful model for analysing some lower level aspects of real-time systems such as interrupts, emergency handlers and event schedulers.

Furthermore, the verification technique used by CSP is based on an axiomatic approach and to construct full mathematical proof requires an in depth understanding of the different inference systems and requires substantial learning on the part of an unfamiliar engineer.

Chapter 3

Structured Methods for Control System Software

3.1 Structured Methods

With a different origin to formal techniques, structured design methods address the problems of organising vague design ideas and assisting the definition of what a designer actually **wants** a system to be. Although most of the developed structured methods provide a framework for system development starting from design ideas through to low level implementation, the strength of these methods is their provision of an established methodology for *requirement analysis* from a system point of view. The stage of requirement analysis concentrates on the *problem space* rather than the *solution space* which often involves:

- Idea generation
- Problem identification
- Problem analysis
- System requirement generation

These are carried out prior to system implementation. During the system requirement generation, three pieces of information are generated which are used to specify a system, these are the implementation requirements, hardware requirements and software requirements. The activity of generating these requirements is generally known as *system design* which is an optimisation of available resources and a trade off between constraints so as to arrive at a design which best fits the original needs.

For this, structured methods provide a methodology which may consist of design concepts, sets of heuristics and criteria to assist a designer to tackle the task of system analysis, design and final

implementation in a systematic manner. With an identical goal as formal methods, structured methods are designed for the production of a sound system implementation. In general, structured methods are characterised by the following historical system development methods:

1. *Structured Analysis*

   Provides a method for building a requirement definition model [Ross and Schoman, 1977].

2. *Structured Design*

   Provides a set of quality assurance criteria for defining the relationships among independently addressable groups of instructions as discussed in Parnas [1972b]. This is an implementation discipline method which assumes that computer instructions are organised to satisfy a set of well defined requirements.

3. *Structured Programming*

   This method consists of a set of guidelines for coding designs in order to maximise the clarity and maintainability of a group of computer instructions. The method is problem orientated which carries similar assumptions as the structured design method.

With this historical background, structured methods today are reasonably established and have inherited some if not all of these properties. Generally they are familiar to engineers and are being used in the design and development of a large number of projects such as those carried out by Simpson [1990b], Ward and Campbell [1983], Foulkes and Deitch [1982], Gomaa [1986] and Looney [1990]. In the next section, a comprehensive survey of a number of existing structured methods which are often used in the production of control system software are presented.

3.2 Existing Structured Methods

Among the existing structured methods, a subset of these are developed for the design and development of distributed, real-time systems such as control systems. One of these is the Yourdon Structured Method (YSM) by Mellor and Ward [1986], Yourdon and Constantine [1978]. The method analyses a system primarily in terms of flow, transformations and data storage in response to external events. More specific development methods which follow this particular style are Ward-Mellor and Hatley. The Ward-Mellor methodology is an extension of the Yourdon/DeMarco technique which allows not only data flow and functional decomposition of a system to be analysed, it also enables users to specify control flow via state transition diagrams.

The basis of the Yourdon technique for system development is functional decomposition, where a model is progressively broken down until it is adequately refined in order to define a system at its implementation level. An application of this extended method is carried out by Ward and Campbell [1983] for a process control system. The other Yourdon-style method was developed by Derek Hatley and is called after its developer, being an extension for specifying control and timing at the requirement analysis phase. The main difference between Hatley and Ward-Mellor is their graphical representations of data flow [McNulty, 1990]. Also, the Hatley method provides techniques for architectural modelling by which a designer can specify the distribution software across a target system.

The Jackson System Design (JSD) by Jackson [1983] takes a data driven approach to design. Being a method designed especially for the development of systems whose subject matters have a strong time dimension: such as control systems, data processing systems and switching systems, guidelines are given for practical usage. It analyses a proposed system in terms of events which are encapsulated into models. As a data driven design methodology, the JSD models are incorporated into a data flow network which expresses the logical structure of how information enters and leaves the system. The final stage of JSD consists of massaging the data flow network into an implementable form. The methodology thus follows the Yourdon approach of attempting to go straight from the analysis to an implementation.

One structured method popular in the defence industry in the UK is Mascot (Modular Approach to Software Construction, Operation and Test) that is discussed in Bennett [1990]. The original version of Mascot was developed during the period 1971 – 1975 in RSRE by Simpson and Jackson [1979]. The official definition of Mascot 1 was published in 1978 with the two other more recent major definitions — Mascot 2 (1983) and Mascot 3 [JIMCOM, 1987] (1987). Mascot provides a means of design representation, deriving the design, ways of constructing software so that it is consistent with the design, and facilities for the execution and testing of the design so that design structures are studied. Systems modelled by Mascot consist of a combination of a set of concurrent functions and the flow of data between such functions.

Having modelled a system with this representation, both Mascot 2 and Mascot 3 then map these functions into one or more active components each of which is a single sequential program. Active components can be run on several processors or to be scheduled by a run time kernel called Mascot Kernel, so as to run on a processor shared with other activities. This eases the design of a concurrent system since the kernel provides all of the relevant constructs. Between the two recent

versions of Mascot, Mascot 3 is a more graphical orientated method which is suited to the design of more complex systems.

However, neither Mascot 2 nor Mascot 3 handle requirement analysis. Both methods start from sets of requirements which are obtained from other structured methods. For Mascot 3, the major phase of development is in the network level from which Network Level Diagrams are produced. These diagrams contain three main constructs: sub-systems, intercommunication data areas (IDAs) and servers. IDAs are data stores which provide communication between concurrent processes. Whereas servers are used to interface the system to its environment. Applications of Mascot to the specification and development of asynchronous communication system were carried out by Simpson [1990b], real-time parallel systems by [Simpson, 1990a] and an implementation of Mascot using the language Ada by Looney [1990].

A structured method widely used for *requirement expression* is CORE (Controlled Requirement Expression) which is described in Mullery [1979]. Central to CORE is the concept of "viewpoints". Each viewpoint describes the nature of the problem from a particular person or object's point of view, the analyst looks at the problem space in term of information acquired, the way that information is processed and the generation of output results. Graphical means are used to describe these viewpoints. This method concludes by analysing the data flow models produced from two points of view: overall end-to-end transactions and constraints. The transactions are chosen to highlight how a system will react to input events where its behaviours must be constrained. CORE does not address the problem of design; despite this it is possible to use it to analyse at a level near to physical reality by choosing viewpoints which span different aspects of the system. Nevertheless, this is just a more detailed analysis which is not an architectural design.

Another method which is built on the basis of an object orientated design is HOOD [Group, 1989]. The HOOD methodology was originally developed for the European Space Agency and is intended to provide object orientated design facilities for Ada programs. On one hand, as this tool is heavily dependent on the language Ada, it is well suited for the development of real-time distributed systems. On the other hand, because of the language dependence, it is not easily applicable to more general object orientated languages such as C++ and Smalltalk. Also, as in the case of Mascot 3, this method does not handle requirement analysis.

The Design Approach for Real-time System (DARTS) [Gomaa, 1984], which is discussed in greater details for its criteria for system decomposition in the later sections, amalgamates the two structured techniques: system design and analysis, which results in a more general design method.

This method is similar in approach to the Yourdon-style methods. Other structured design methods include the RSL/REVS for specification and simulation of systems by Alford [1977], PAISLey for the specification and development which relies on some formal techniques [Zave, 1982] and SADT — Structured Analysis Design Technique which is similar to VDM, is a well defined method for a top-down decomposition of a problem. A summary of the described structured system design methods is given in Figure 3.1.

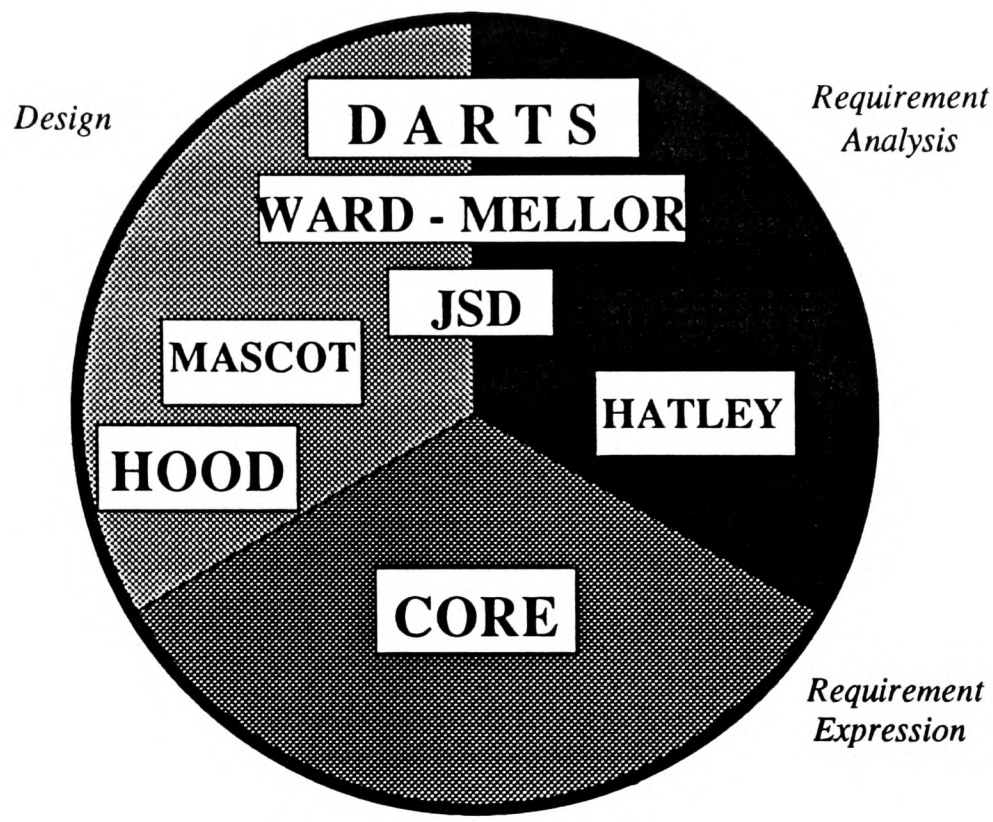


Figure 3.1: A comparison between various structured system design methods

Interested readers are referred to Jackson [1990] and McNulty [1990] for a more detailed discussion and comparison on the various structured methods used for real-time system development.

3.3 Structured Software Development

It is found that a general picture in system design and development follows the capturing of design ideas and the analysis of the interactions between a system and its environment. For this, a conceptual model of the system is required before any further development proceeds. The Yourdon-style modelling technique is one of the methods that enables such system definitions to be generated and this modelling concept has been chosen to be unified with the rigorous formal techniques discussed in Chapter 2. Before the discussion of the unification issue, this modelling techniques is discussed in the following sub-sections.

3.3.1 Essential Modelling

By following a Yourdon-style system development, the product of structured analysis is an *essential model*. This model is a description of the system which clearly defines WHAT the system's requirements are. The model separates the system's external interfaces from its internal behaviours and in its construction, implementation details are abstracted. A number of techniques have been adopted by various structured methods which attempt to obtain the essential model.

1. *Functional decomposition* is a specific application of the idea of a top-down development [DeMarco, 1978] by which statements of system requirements are obtained by combining the behaviours of decomposed sub-systems and processes.

2. *Environmental modelling* as an alternative technique to functional decomposition. By *incorporating* a model of the environment into the requirements such that a system essential model captures not only the functionalities of the 'system' under design, but also the interactions between such a system with its 'external world' — its environment. It specifies the events that the subject system must respond to and the specifications obtained form the basis of the behavioural specification.

3. *Subject-matter-centred modelling* consists of a set of common sense modelling heuristics which direct a designer to concentrate on the salient 'perception/action' of the design when capturing requirements. These heuristics assist the abstraction of un-related information from the subject matter in a design.

4. *Modelling with minimum complexity* states that the simplest representation that adequately models a situation should always be used. This point is exemplified by the following statement made by Hoare, a distinguished computer scientist:

   > *"There are two ways of constructing a software design: One way is to make is so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies ..... and the first is always preferred."*

Among the four techniques described, using a functional decomposition to obtain an essential model is the least preferred. Since at the stage of system analysis where design ideas are organised and captured, modelling through functional decomposition fixes the boundaries of sub-systems at

an early stage which will result in an unsatisfactory partitioning of parallel processes and non-optimal interfaces. Also, there is a tendency to produce an implementation dependent model when using decomposition. Furthermore, as it is a specific top-down approach, its application is limited. As such, the method addresses poorly the problem of requirement analysis and provides only vague guidelines for system design. At this stage of system development, the other three modelling techniques are all better for generating an essential model from a system analysis point of view. With these, a complete essential model is made up from an *environmental model* and a *behavioural mode* that are obtained by applying the last three modelling techniques.

3.3.1.1 An Environmental Model

An environmental model puts a system into its real world context. It defines the organisation, people, other systems and events which are classified as the 'environment' to which the subject system must respond. It is a description of the environment in which the system operates which consists of two main parts: the definition of a system boundary together with a description of the events that occur in the environment that the system must act on.

An environmental model can be obtained by mapping the relevant elements of the environment to the corresponding system responses. This is an "outside-in" development strategy as pictured by Figure 3.2, the result may be a functional description of the system which is not obtained via decomposition. Various structured methods provide both textual and graphical tools to facilitates the construction of an environmental model, these includes: Statement of Purpose, Context Diagram, Event List, Entity Relationship Diagram and Data Dictionary which are discussed in detail in Mellor and Ward [1986].

3.3.1.2 A Behavioural Model

While an environmental model defines what events a system must respond to, a behavioural model defines the actual responses the system must act in response to its environmental stimulus. It describes the required behaviour of a system. The model consists of two distinct parts: the transformation schema and the data schema which can be obtained from the environment model with the aid of tools such as: Levelled Sets of Data and Control Flow Diagram, Process Specification, Expanded Entity Relationship Diagram, State Transition Diagram and Data Dictionary that are discussed in Mellor and Ward [1986].

Together, the environmental and behavioural models form the essential system model — the WHAT of a system.

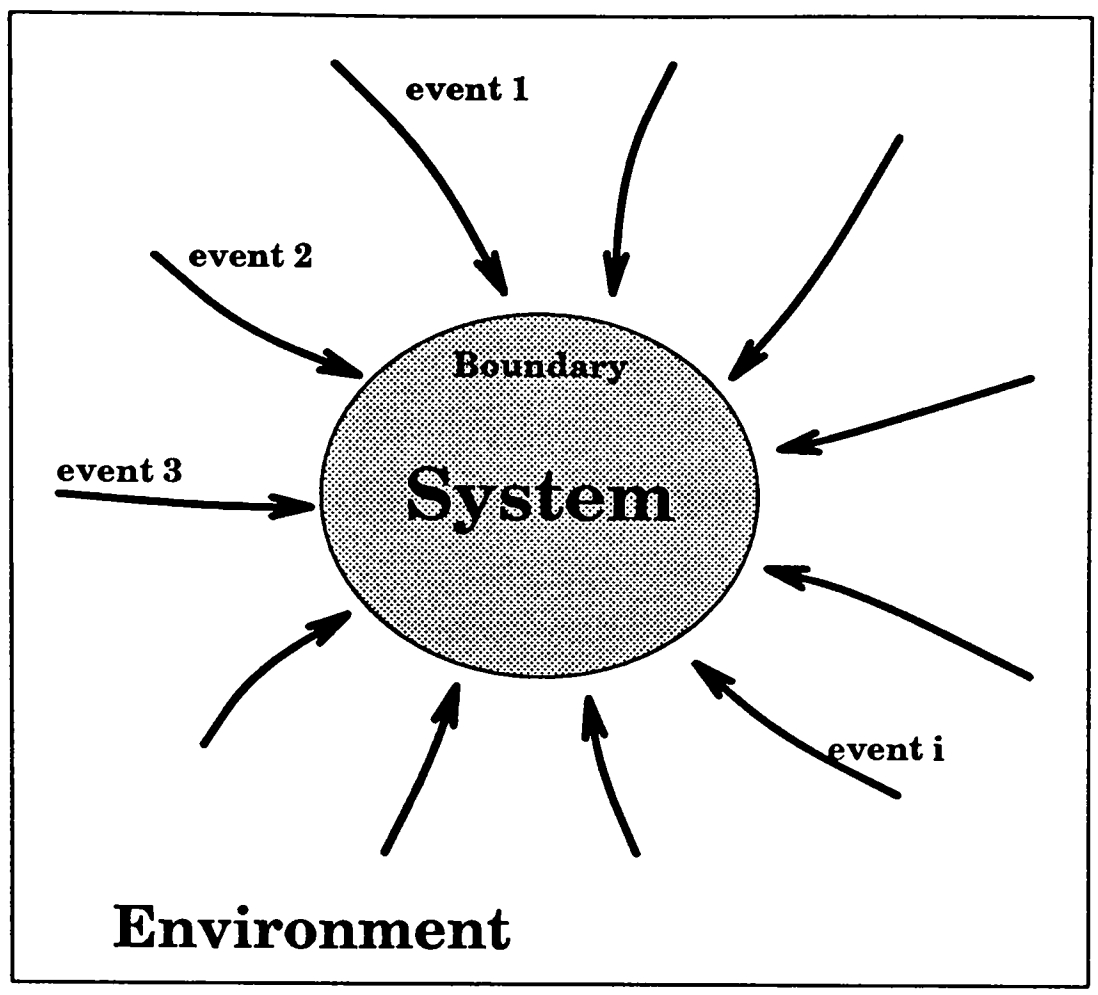


Figure 3.2: An environmental modelling — an "outside-in" approach

3.3.2 Implementation Modelling

The implementation model of a system is derived from the essential model. Its creation is fundamentally an activity of derivation and mapping the content of an essential model into an implementation environment, such as a given hardware platform. Requirement specifications from an essential model are mapped onto various implementation aspects. As constraints are inevitable in an implementation environment, the task of implementation modelling is to *optimise* the desired performance under the specified requirements among a set of constraints and conflicting objectives such as:

- correctness
- safety
- minimum distortion
- use of resources
- process partition
- efficiency

According to the Yourdon Methodology [Mellor and Ward, 1986], the task of generating an implementation model is carried out in a number of steps. The first level is the introduction of processors for carrying out the activities and storing of those data or data structures that are being declared in an essential model. The next step is to define more specifically the activities — *tasks* within a processor. Finally, the level of using a programming language is considered by which each operating module is identified and laid down according to the defined tasks. In Yourdon, the above three steps correspond to the creation of three models: *processor*, *task* and *module* models, altogether they make up the implementation model. An organisation of a typical implementation model is illustrated by Figure 3.3.

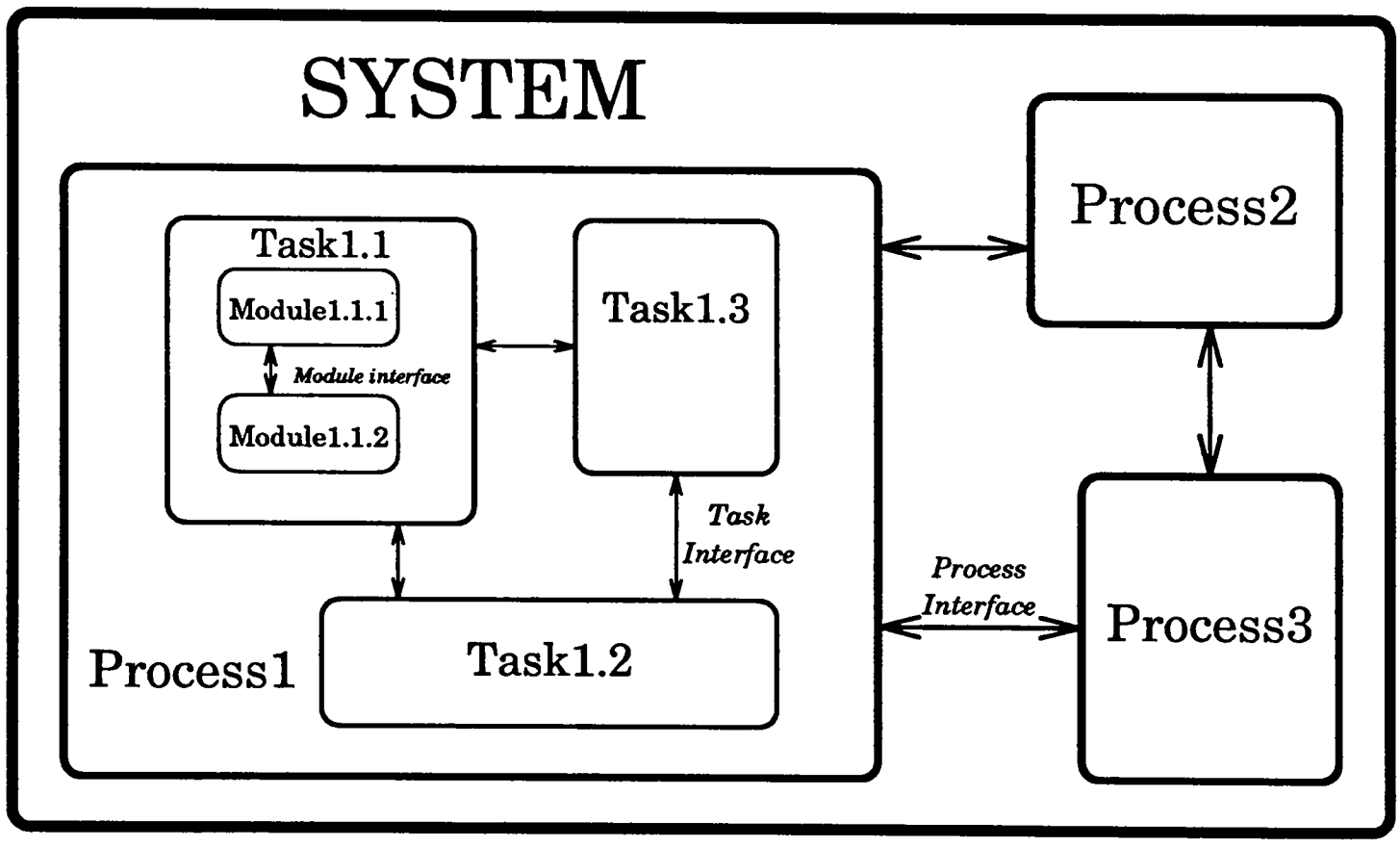


Figure 3.3: A Yourdon implementation model showing the different hierarchies of implementation details of a system

3.3.3 The *Transformation* and *Data* Schemas

It is a well known scene in a structured description document that the topology and operations of a system is often represented by a series of diagrams; each consists of number of "bubbles" and "arrows". These "bubble-arrow" diagrams are graphical tools that are used by structured methods to represent system activities and are called *schema*. There are various forms of schema that are used for representing control signal, data flow, data storage and state transition. Figure 3.4 illustrates the use of a simple *transformation schema* to represent a function that operates on a stream of discrete data — hence the transformation of data flow.

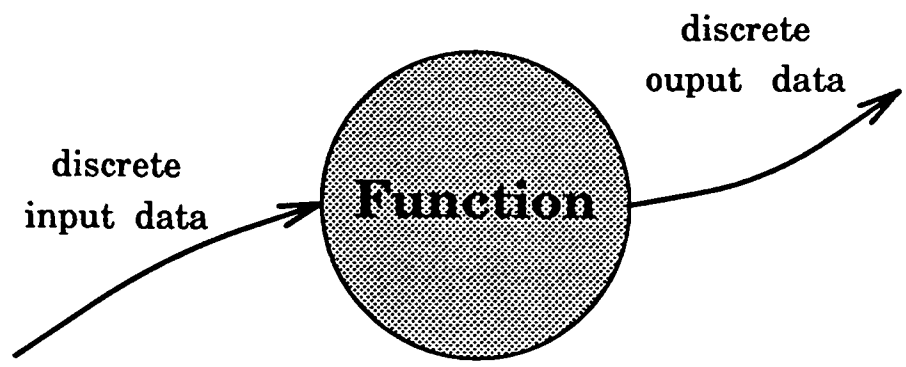


Figure 3.4: A transformation schema for discrete data flow

Apart from expressing activities using transformation schema, information stores are also vital to the functioning of a system. The *data schema* denote graphically the layout of information that must be remembered by a system. A number of the commonly used transformation and data schema are given in Figure 3.5 and detailed descriptions and their usages are found in Mellor and Ward [1986].

It is found that the application of these graphical transformations and data schemas are complimentary to the textual requirements for specifying a system as a partitioned set of interconnected components so as to allow easy assimilation and quality review. A transformation schema denotes graphically the layouts of the transformations that operate on flow that crosses the system boundary, and it represents the active portion of the system that responds to the events that have occurred in the environment. Whereas the data schemas denote the relevant data stores in a graphical way. In Yourdon, an essential model of a system is sometimes expressed as a combination of transformation and data schema; for an implementation model, the use of transformation schemas for defining processes, tasks and modules is frequently found.

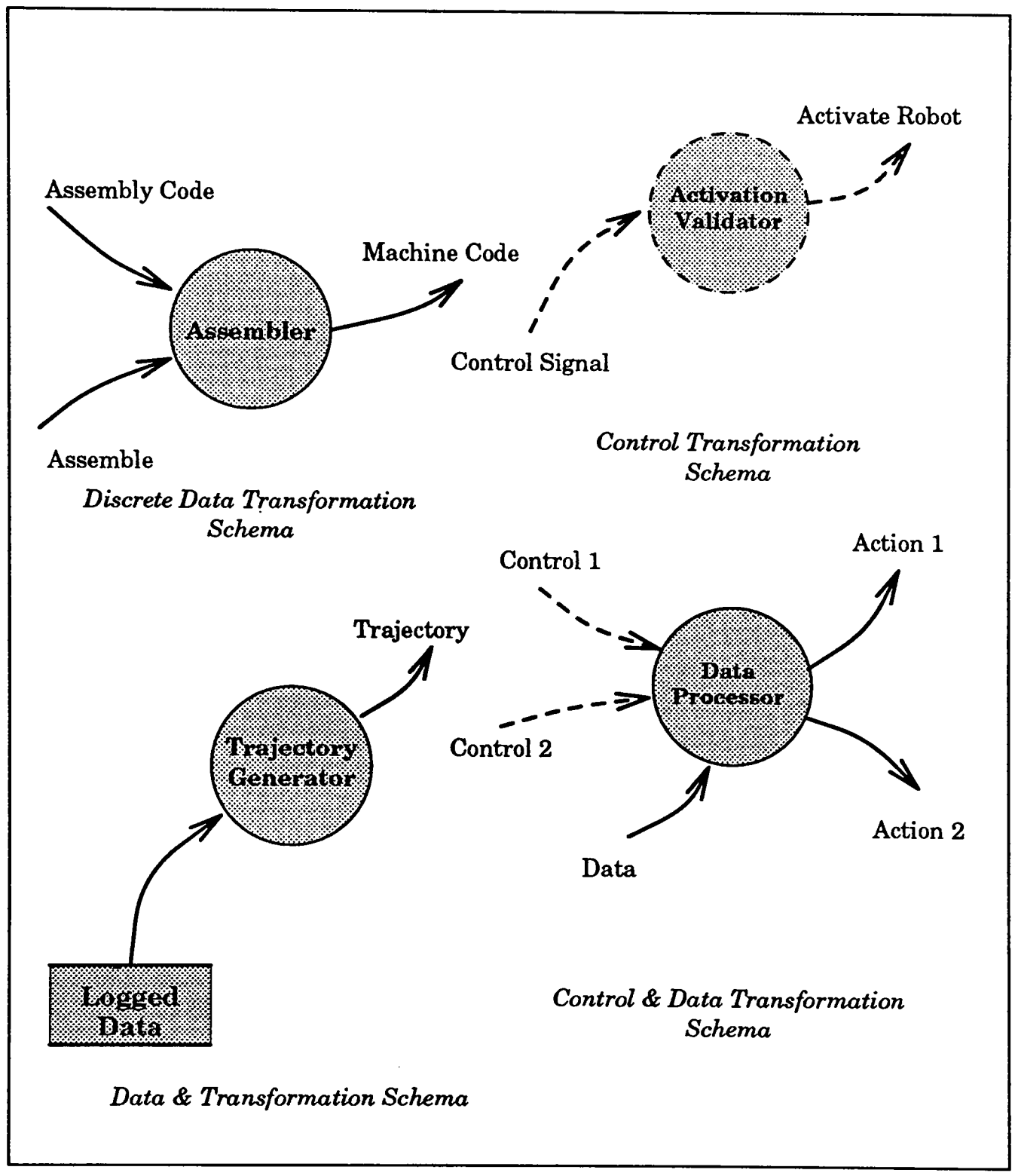


Figure 3.5: A collection of typical transformations used by structured method for modelling a system and representing design ideas

3.3.4 Essential to Implementation Model Transformation

There exists two distinct approaches of transforming from an essential model, by which a design is captured, to an implementation model that contains information of how the system can be realized. One approach is to develop both models concurrently with increasing detail. Figure 3.6 illustrates this concurrent transformation.

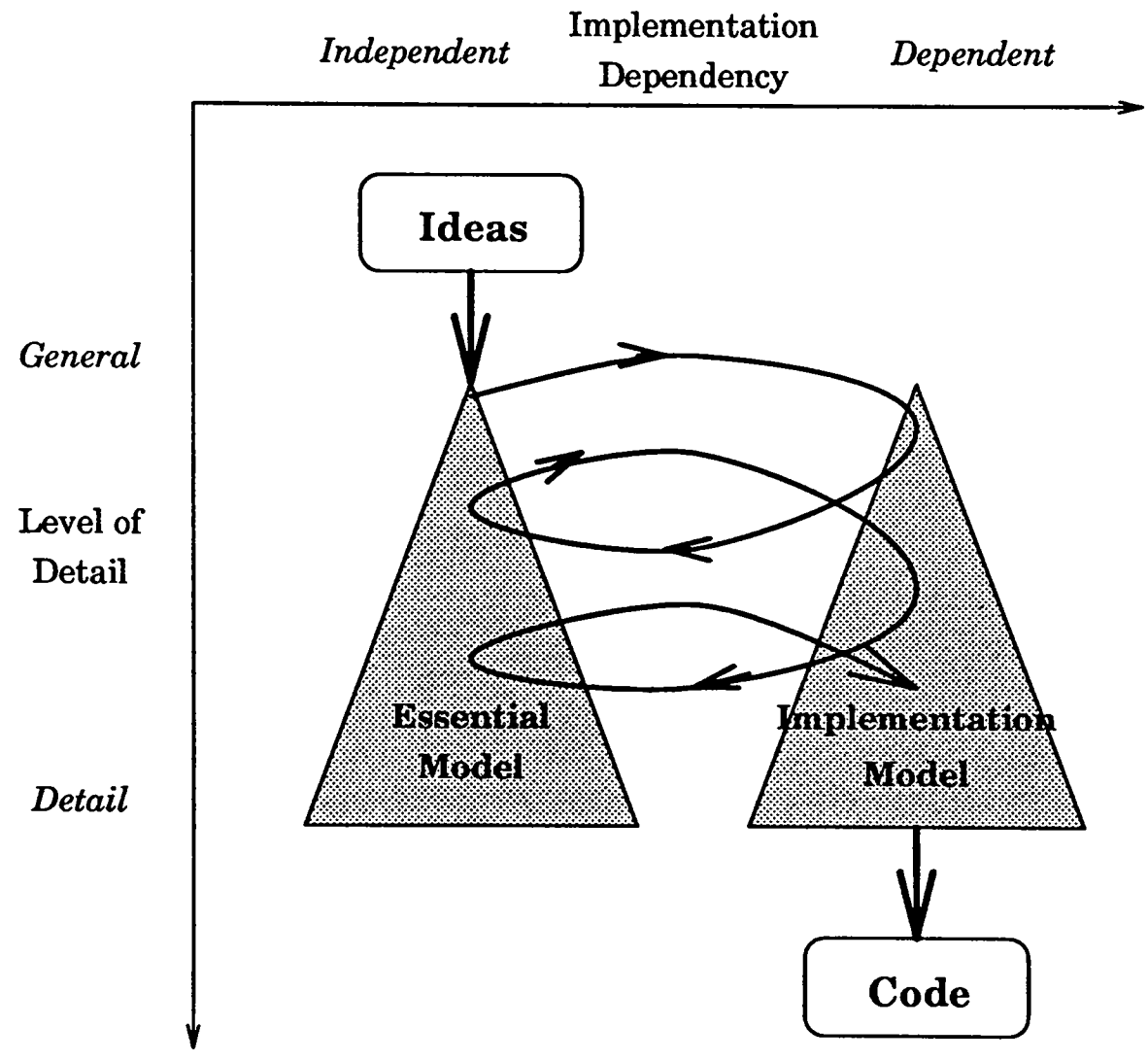


Figure 3.6: A "hand-in-hand" simultaneous development of essential and implementation models

In Figure 3.6, the starting point of the development is a set of high-level essential details. The corresponding implementation details are filled in, then the next lower-level essential details are created which is followed by the introduction of possible implementations, and this process goes on.

Another method of transformation is to separate the development of the two models, with the creation of an essential model followed by an implementation model. This development strategy is pictured in Figure 3.7.

As the aims of the two approaches are identical and the sequence of construction should be independent to the model contents, the decision to follow either of these models is up to the indi-

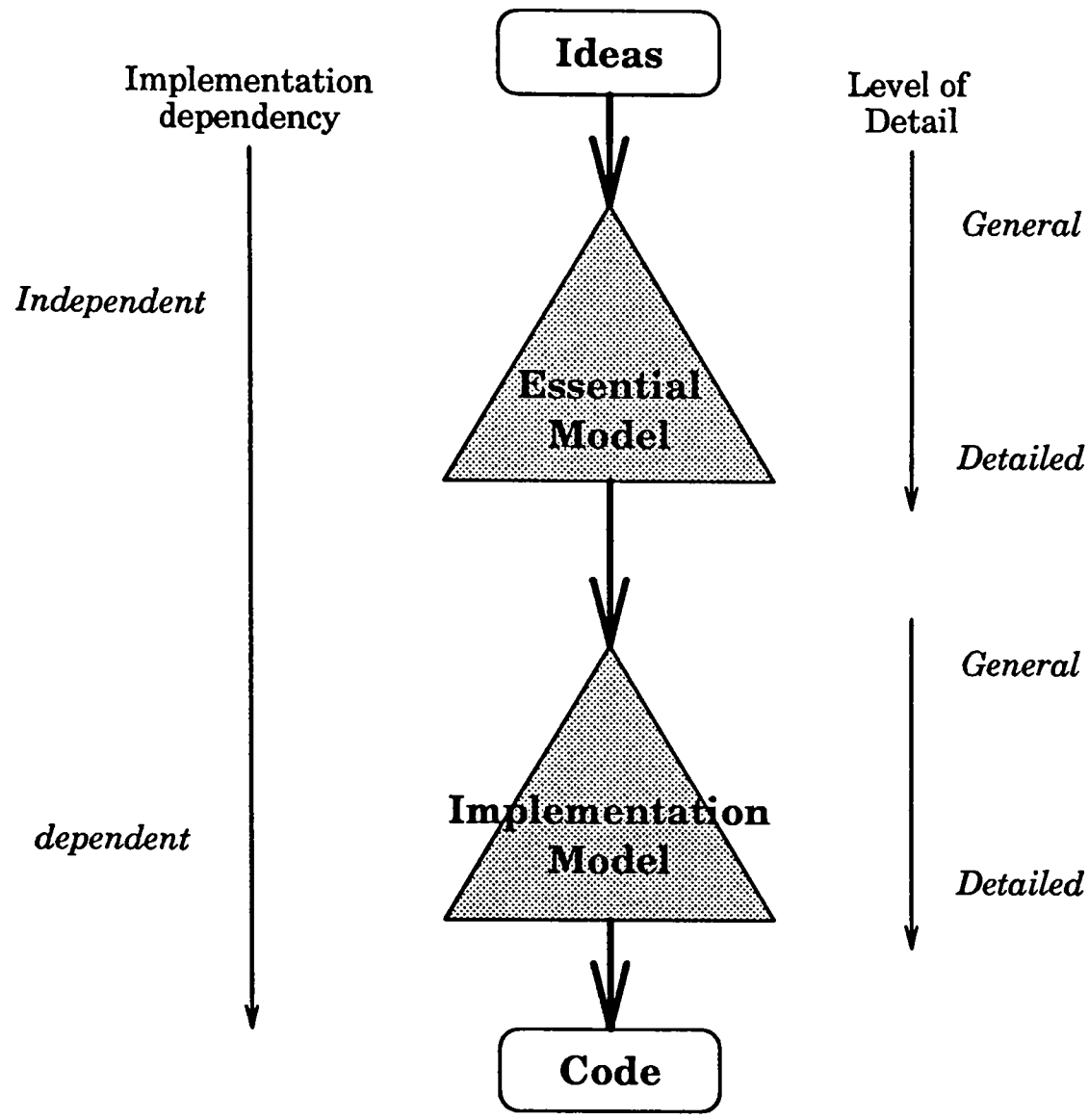


Figure 3.7: A separate development of models and a cascade transformation

vidual designer. However, for most structured methods, there seems to be no definite guidelines or methodologies to assist a stage-by-stage transformation. Nevertheless, the two different approaches to such transformation have their corresponding advantages and disadvantages. The later approach has a number of appealing properties such as enabling the separation of information on WHAT and How in the two models so as to minimise the interference between implementation and essential details and to allow the comparison between their consistency when both models are generated. Nonetheless, the former "hand-in-hand" approach enables an incremental generation of the two models which in effect has emphasized the assurance of consistency right into every incremental development step, rather than leaving this until the two models are fully developed. With this, not only an implementation model can be constructed which satisfies its requirements, essential requirements are modified to comply with the design. Moreover, to develop a practical system description such as an essential model of a control system, a minimum knowledge of the realities such as *"what is the design for?"* and *"in what environment is this system going to reside?"* are

often useful.

Hence, the former incremental philosophy of model transformation is adopted which forms the basis for the unified methodology discussed in Section 4.5.1.

3.4 DARTS — A Structured Method

The Design Approach for Real-time Systems (DARTS) is a structured method which is designed for the development of real-time systems with concurrent processes. The method incorporates the ideas of structured design and structured analysis. It features the Yourdon-style data flow analysis and functional decomposition, together with the MASCOT task structuring techniques. The method consists of two main components as discussed in Gomaa [1984]:

1. A dual set of criteria for task cohesion and task coupling which can be used for the evaluation of the quality of a design.

2. A design method which is fundamentally a top-down decomposition method, transforming a system into functional modules.

The main objective of this design methodology is to decompose a system into manageable, well defined modules with each target module having a high degree of cohesion and low coupling. The result is that the generated modules are suitable for parallel realization. By considering the decomposition and structuring criteria given in Sub-section 3.4.3, its design objective will become more apparent.

3.4.1 Why DARTS?

DARTS is a general design method with an approach to system design which follows closely the well established Yourdon Structured Development method. It starts with an essential model with the system requirements and aims at producing an implementation of a system with self-contained modules which facilitates modification and maintenance. The method enables the translation of a "flat" system model to a hierarchy of control structures with each of these hierarchies well defined by a related group of modules.

Through these, the method has attempted to fill the gap between the transformation from an essential to an implementation model. Although Yourdon has suggested a novel requirement capturing method for the creation of an essential model, there seems to be no clear guidelines of how the transformation to an implementation should be undertaken. The system decomposition and structuring criteria suggested by DARTS which produces highly cohesive functional modules with minimum interfaces that are suitable for parallel implementation provides insights into the problem of transformation. Its applicability to real-time distributed control software design was demonstrated by Gomaa [1986].

With these fundamental properties, the resultant system model complies with the formal computation model of CSP to some degree especially in the philosophy of developing a system with concurrent modules that communicate with one another. Moreover, being a smaller and "cleaner" method, it is easier to work with. For these reasons, the DARTS method was chosen to be integrated with the formal method CSP for a unified control system software design and development methodology.

3.4.2 The DARTS Way

The DARTS design method starts with the requirements of a system — the essential model. As a given specification can be designed and implemented in many different ways, the development of a *state transition diagram* is the first phase of the design process. This diagram is an interpretation of the essential model by which the external requirements asserted by the environment and the corresponding sequence of internal actions to be performed are clearly defined. The diagram also sets down the allowable states of a system. Following the creation of the state transition diagram, the development of a *data flow diagram* is then followed. At this stage, the system is decomposed into sub-systems; sub-systems into processes and each sub-system interface is identified. Moreover, DARTS provides a set of criteria for structuring processes logically into sub-systems or tasks and a mechanism for defining interfaces between tasks. The various stages that DARTS supports are summarised as follows:

- Development of the State Transition Diagram — a description of a chosen set of the allowable states and transitions of a system.
- Development of the Data Flow Diagram — identify the essential processes that constitute the system and their interactions.
- Task Identification — based on the data flow diagram, processes are structured according to defined criteria into logical processes called tasks. These tasks are self-contained modules that execute concurrently within the system.
- Development of a Task Structure Chart — this is the end product of the last stage which is used for subsequent implementation.

3.4.3 DARTS Criteria

DARTS adopts the criteria from the Yourdon Method for system decomposition via schema transformation which is discussed in Mellor and Ward [1986], and in addition, it introduces a unique

set of criteria for structuring a decomposed system into tasks. A subset of these relevant process structuring criteria are outlined and will be applied in the case study of a robot control system as part of the unified method discussed in Chapter 5. For the specific application of designing with communicating processes, an additional criterion DC–5 is proposed for task decomposition and structuring.

DC–1 *Dependency on I/O*:

The speed of interaction between processes and its environment is often determined by the rate of occurrence of external events. Functions having direct interactions with the environment across a system boundary are therefore desirable to be structured as separate concurrent processes.

DC–2 *Time-critical function*:

A time-critical function needs to run at a high priority and is preferred to be a separate process.

DC–3 *Computational requirement*:

A computational intensive operation consumes a large amount of CPU resources and in order for it to have the least interference with other processes, it is desirable to structure it separately.

DC–4 *Functional cohesion*:

Related functions are grouped together as a single process as the data traffic between these functions are usually high.

DC–5 *Communication requirement*:

Processes that are communication intensive should be grouped as a single process. As the communication overhead between these functions is high, separating them will increase the system overhead.

It is seen that these criteria all require some knowledge of the physical system in which the developed software is going to reside. In the process of task structuring using these criteria, an abstracted system model is gradually mapped onto a physical platform with an increasing amount of implementation detail. By recursively applying this process, a realization of the system at a level that is amenable for implementation is obtained.

3.5 Formalisation of Structured Methods

3.5.1 Motivation

From the survey of structured methods, it is found that the strength of these methods are the provision of design guidelines for capturing and organising vague ideas into structured documents. Most of these criteria and guidelines result from a large number of past design experiences on numerous case studies and projects so their applicability at the relevant levels of design is well proven. In the case of generating an essential model of a design, structured methods such as Yourdon, DARTS and JSD provide sets of heuristics for a designer to follow. Also a number of these methods have suggested criteria such as those for decomposition by Structured Design of Parnas [1972a] and Yourdon; and also criteria for structuring functional modules by Mascot and DARTS, which provide insight for system development.

Nevertheless, the transformation from an essential model to an implementation model proposed by these methods is relatively ad hoc. In particular, when the requirements of an implementation are stringent, as in the case when the software is used in a safety-critical system, these development methodologies will become unreliable and un-traceable. Although some methods such as Yourdon set out design laws such as the followings:

1. Internal consistency of a schema model
    - all input event flows must be used as conditions
    - all conditions must respond to input events
    - all output events must result from some actions
2. Inter-stage consistency among schema in different levels of abstraction
    - each flow in a higher level must match with a lower level flow
    - each store must be matched
    - external flow must be matched by internal flow

for ensuring the "correctness" of a design so as to increase the confidence of the end product; in effect, only consistency and hence syntactical errors are avoided.

Moreover, a majority of the documentation from structured design such as the essential models, implementation models together with their intermediate refinements are presented in a pictorial form. The use of transformation schema and data schema in the representation of a system

provides a strong visual perception of the design, but they are usually incomplete and ambiguous. Furthermore, in cases when system decomposition is being applied at a premature stage, the resultant system model will be embedded with a mixture of essential and implementation details which might lead to a sub-optimal and undesirable system realization.

In the following sub-sections, the two major weaknesses of structured methods: the discontinuous transition from an essential to an implementation model and the ambiguity of a pictorial representation are analysed together with propositions to overcome these weaknesses.

3.5.2 Bridging the 'Gap' — Refinement of the Yourdon Method

The problem of lack of continuity of the transformation from an essential model to an implementation model in system development according to Yourdon has already been pointed out in Sub-section 3.3.4. In order to bridge the gap between system analysis and implementation design, the use of a combination of structured methods is proposed. Although the Yourdon method does not provide guidelines for this transition, other structured methods such as Mascot and DARTS provide some of the relevant criteria for this intermediate stage. In this study, a refinement of Yourdon method — DARTS is seen as a promising candidate which provides a set of criteria to guide the stage of transformation

3.5.3 Removing Ambiguity — the Formalisation of DARTS via CSP

A major drawback of using a graphical representation of a system is the existence of ambiguities in its semantics. Figures 3.8 and 3.9 show two state transitions diagrams constructed by two different engineers representing two simple process control systems. It is seen that the complexity of system B given by Figure 3.9 is higher than that of system A which is given by Figure 3.8 with a greater number of states.

A closer look at the two representations concludes that even though the two state transition diagrams appear to be different, the two systems have the same behaviour patterns which can be realized with an identical implementation. This observation might be apparent for these two simple systems, but for a more complex system, such observations will be difficult to make. Nevertheless, the case has illustrated the non-uniqueness of a pictorial representation in general. In some cases, there exists a number of alternative interpretations of a single picture such as a "bubble-arrow" transformation schema in a data flow diagram, making subsequent system development both confusing and arbitrary. The existence of ambiguities in a graphically based modelling technique as adopted by most structured methods is that they are usually "in-complete" in term of specifying quantitative details such as those found in an implementation model.

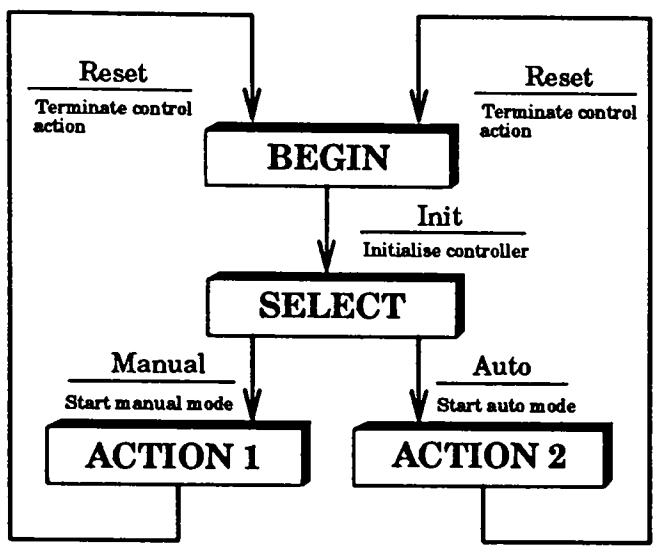


Figure 3.8: A state transition diagram of process control system A

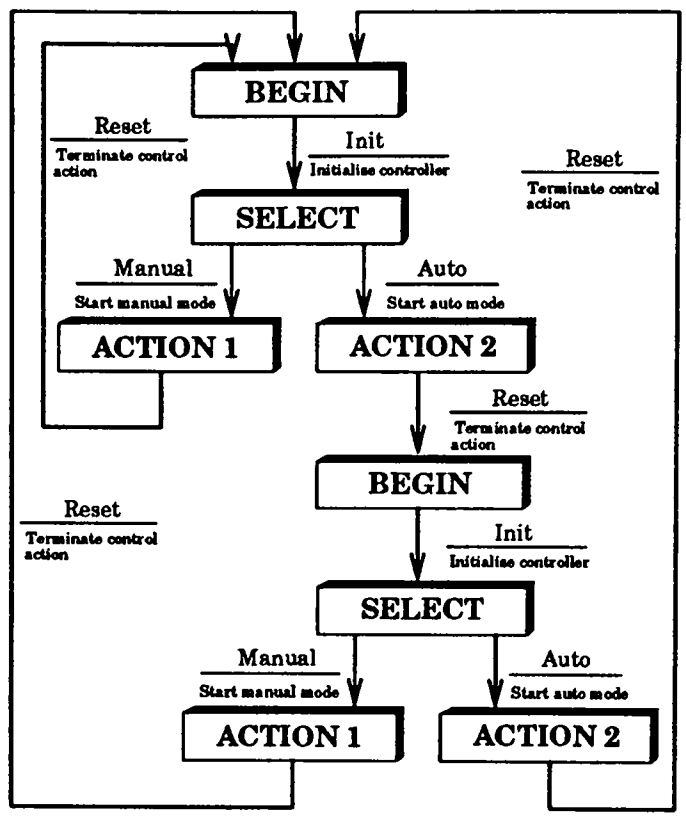


Figure 3.9: A state transition diagram of process control system B

quantitative details such as those found in an implementation model.

The use of formal descriptions — specifications to express the behaviours of a system — provides a resolution to the problem of eliminating ambiguities. For this particular process control system, a realization using CSP process algebra given by Equation 3.1 provides a precise and unambiguous description of the behaviour of this system. Moreover, such system definition is unique.

$$\text{PROCESS CONTROL} \;\widehat{=}\; \mu X \bullet (init \rightarrow \; (manual \rightarrow reset \rightarrow X \;\; | \;\; auto \rightarrow reset \rightarrow X)) \tag{3.1}$$

Hence by obtaining a formal specification prior to the generation of a graphical description, the formalised pictorial representation will benefit from both an unambiguous definition and a strong visual conception of the system.

3.6 Conclusion

From the discussion presented in this chapter, it is seen that structured methods are conventional and well established system development methods which are widely used and familiar to the engineering community. The large volume of projects and case studies conducted using these methods has shown their popularity of use and their applicability in tackling the design of distributed con-

trol systems. Nevertheless, there exists a number of major drawbacks in these methods such as the informality in the transformation between essential and implementation model and ambiguous graphical representation as discussed in Sub-sections 3.5.1.

Two propositions are made with a view to resolve these weaknesses: the utilisation of a number of different structured methods in order to 'bridge the gap', and the use of formal specification to strengthen the representation forms the basis of a unified methodology presented in Chapter 4. To aim at a structured and yet formal development, the use of CSP specifications and process definitions strengthens the pictorial representation. Furthermore, the inability of structured methods to reason with the compliancy of an implementation with respect to its specification may lead to a large deviation in the end product, that is the executable program, from the original design. At present, the "formality" that is advocated by structured methods is only capable of ensuring internal and inter-stage consistencies so as to minimise possible syntactical errors. Despite the use of several structured methods in tackling a design problem, this fundamental short-coming has never been solved. This very reason has prompted the necessity of the integration of formal methods within the structured development process. The mutual benefit to both structured and formal methods of a unified methodology will become apparent in the proposed integration discussed in Sub-section 4.5.1.

Chapter 4

CSP Abstraction Tool Set

4.1 Introduction

The central theme of this chapter is the definition of the *Abstraction Tool Set* which is used to capture a sub-set of the typical behaviours of a distributed control system in a formal and modular way under a number of assumptions given in Section 4.3.1. This abstraction tool set forms the core of the unified methodology for process modelling. In achieving this, general properties of a distributed control system are identified through the study of a typical control system and by examining the various control hierarchies. In particular, the structure and algorithm of a PID controller is discussed in some detail so as to facilitate subsequent formal analysis. As CSP is the chosen formal system, a subset of its notations and process algebra considered suitable for describing and reasoning with control systems has already been discussed in Chapter 2. Having defined a formal basis, the abstraction tool set is presented which is made up of a set of specification predicates and five *classes* of generic CSP processes such as *FSM*, *PERM* and *MUX*.

In this chapter, each generic process class is discussed in detail with its satisfies relation and corresponding specification, the laws and theorems concerned and most importantly, their usage for specifying real systems and processes are also discussed. Having described and defined the tool set, Section 4.5 integrates their use with the chosen structured technique — DARTS — to present a unified methodology. A number of guidelines and proof rules are described that engineers can follow for the design and development of a distributed control system. Finally, in order to present a fair view of the proposed methodology, the completeness issue of the tool set is discussed in Section 4.6, so as to enable users to apply the method in a sensible and judicious manner.

4.2 Properties of Control Systems

4.2.1 A Top Level View

To set the scene for identifying the various properties of control systems so that greater formality can be introduced during their design and development, a typical robot control system is considered. The task is to develop a control system for an industrial PUMA 560 robot which has been modified for fine force control [Dickinson, 1988b; Dickinson, 1988a]. Some typical requirements are that an operator will be able to move the robot to desired locations, monitor its movements and to log any salient parameters for further analysis.

The system given in Figure 4.1 consists of a tight-loop controlled plant, on top of it an interactive monitor, a low level datalogger and a demand generator, etc.. All these "devices" are running in parallel with the controller. We will call such a system with layers of processes interacting with the controlled plant a *jacketed* controller.

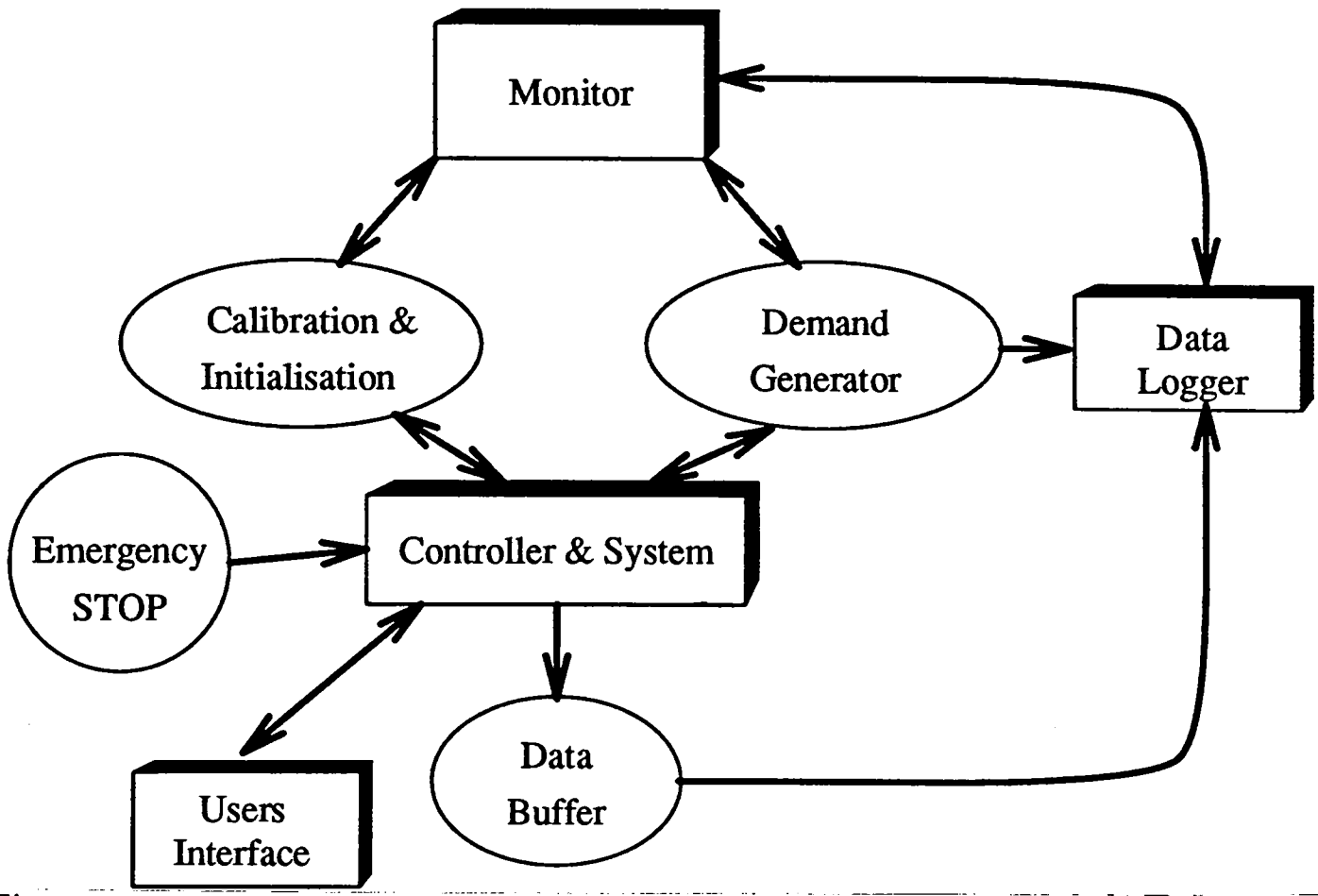


Figure 4.1: A Computer Control System with Jacketing. The boxes and ellipses indicate "desirable functional blocks" which are free from any implementation implications.

In this figure, all the conceptual processes run concurrently after the calibration of the system by pre-determined inputs from the environment. Each of these processes can be seen as an individual process performing a number of complex actions, and the links joining these boxes can be thought of as communication linkages by which messages are passed through using some protocol. An

example of such a protocol is shown in Figure 4.2, which is a communication protocol between the calibration process and the controlled plant.

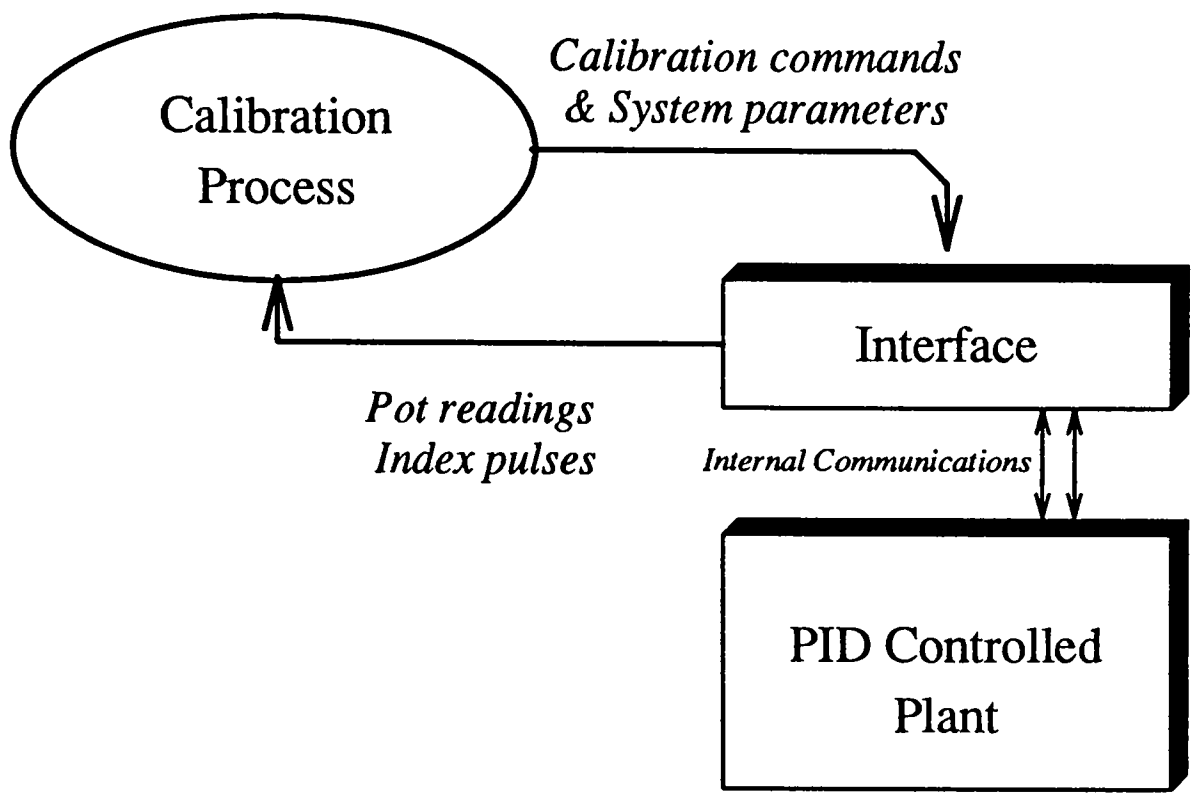


Figure 4.2: The communication protocol between the calibration process and a controlled plant

4.2.2 A Way Down the Hierarchy

However, the above model is only at a very high level, in real life, each *box* can be decomposed into more refined processes describing in greater detail the operation of the system. Taking the controller and system down into greater detail, a more explicit control architecture is revealed which is shown by Figure 4.3.

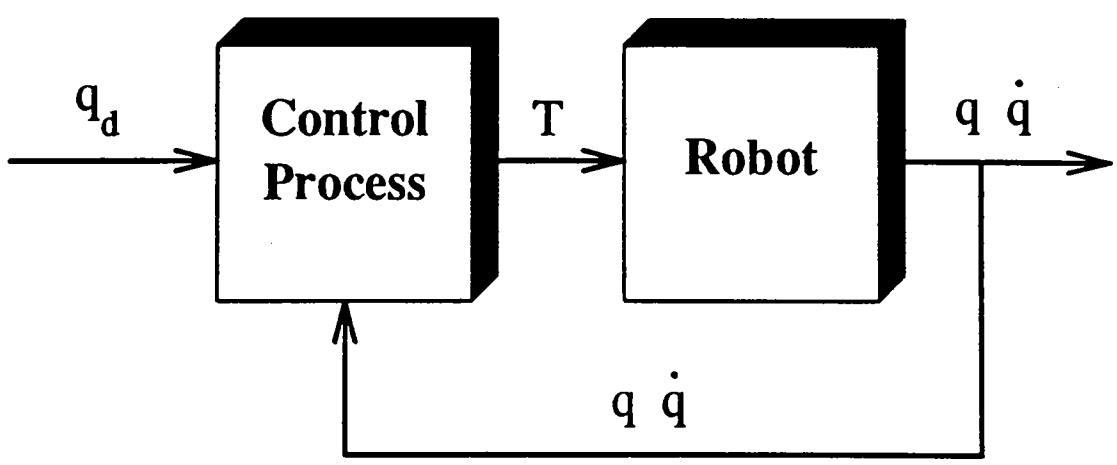


Figure 4.3: A general closed loop control system

4.2.2.1 Feedforward Compensation Control Scheme

A number of refined control schemes can be used to realize this controlled plant. One such analysis follows that of Craig [1986]. In Figure 4.3, the control parameters is the the state vector $(q, \dot{q})$ for describing a trajectory of the manipulator. Here q is an n-dimensional vector represented by Equation 4.1 with n equal to the degrees of freedom of this manipulator.

$$q = [\theta_1, \theta_2, \ldots]^T \tag{4.1}$$

A generally accepted model for robot arm dynamics takes the form as presented in Equation 4.2 which can be formulated from two distinct approaches, namely the *Lagrangian* and the *Newton-Euler*. While the Lagrangian formulation leads to physical insight, the Newton-Euler approach is computationally more efficient and more suited for real-time control applications. It can be shown that the difference is one of representation of notation.

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + k(q) = \tau \tag{4.2}$$

Equation 4.2 is a vector equation for the mechanical manipulator where q is a vector of joint measurements, τ is a vector of control torques or forces exerted on each joint by the actuators, M arises from the "inertial" properties of the links, C, from the coriolis and centripetal forces of motion, and k represent the gravitational forces. Even for kinematically simple robots, the entries of the matrices M, C, and the vector k, are extremely complex high order polynomials in transcendental functions of the joint variables, q. They are difficult to obtain for real manipulators. An important early study by Bejczy [1974] shows that these non-linear terms are not simply analytical artifacts, but represents the cross-coupling forces which may vary by as many as three orders of magnitude over the robot work space.

One of the most generally discussed control algorithms to achieve "robot tracking" around some *a priori* specified reference trajectory, $q_d(t)$, is the so-called "computed torque" or "inverse dynamics" algorithm [Craig, 1986] given by Equation 4.3:

$$\tau = k(q) + C(q, \dot{q})\dot{q} + M(q)[\ddot{q}_d + K_v(\dot{q}_d - \dot{q}) + K_p(q_d - q)] \tag{4.3}$$

This corresponds to the general linear strategy of pole-placement via state feedback, preceded by a forward-loop compensator intended to invert the non-linear dynamics of the feedback compensated plant. This is illustrated in Figure 4.4.

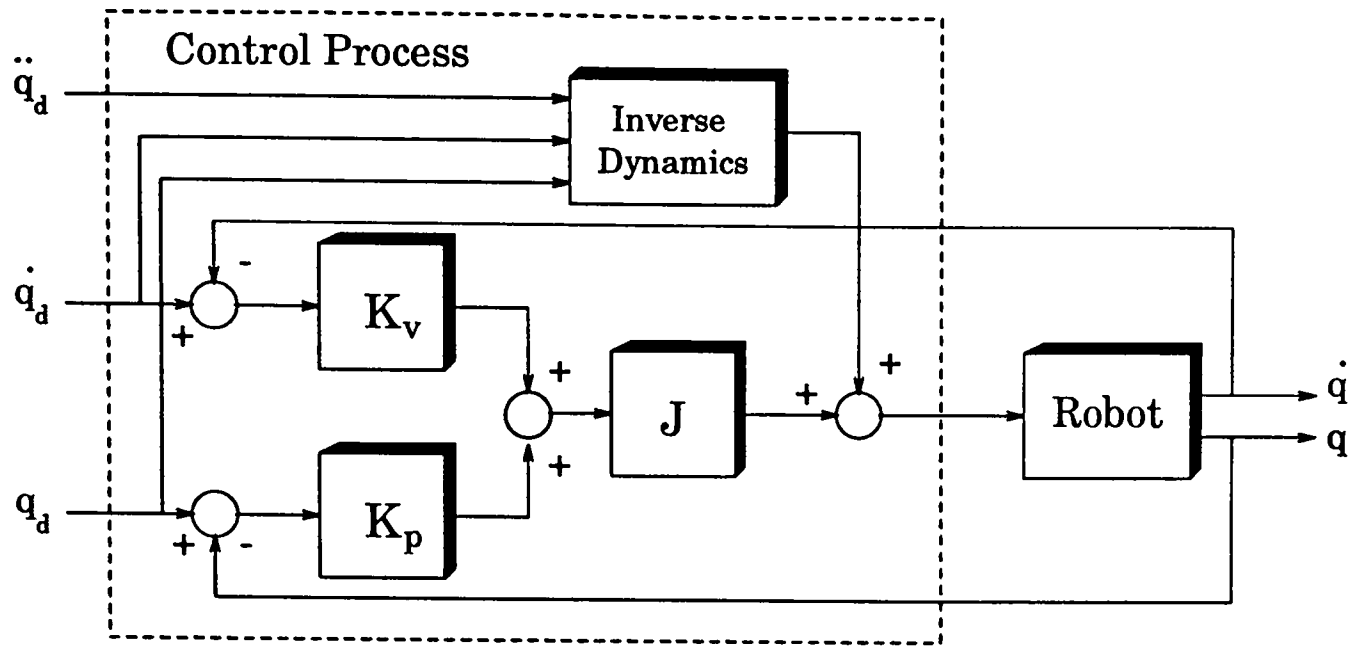


Figure 4.4: A Block Diagram of Feedforward Compensation Control Scheme

4.2.2.2 PID Controlled Plant

As another simple illustrating example, a PID controlled scheme shown in Fig. 4.5 is considered. However, it is emphasised that the described PID controller is not presented as the solution to the PUMA force controller described earlier. The primary objectives of presenting the PID controller are:

1. Being the most popular and widely used controller, its structure and functionalities is revealed so as to enable its properties to be identified.

2. To present the degree of complexity for an idealised control system so as to demonstrate the level which formal analysis has to cope with ever in the idealised case.

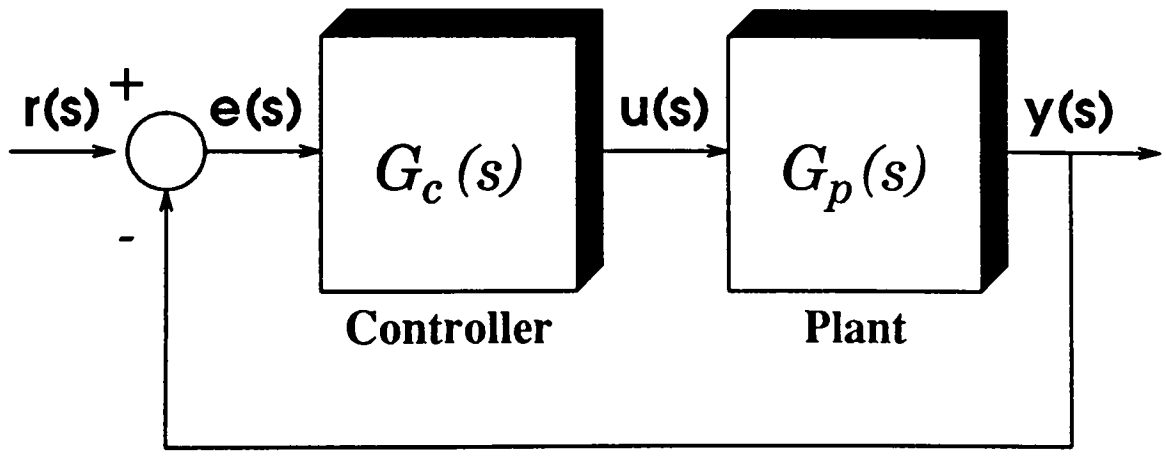


Figure 4.5: A compensated unity feedback control system

From the figure, the basic control sub-system consists of a model of the real plant, which in this case will be the robot itself defined by its dynamics and the controller which is an implementation of its designed *transfer function.*

General Description Basically, a PID controller is a *state machine* which is made up of two fundamental parts: a *recursive* part that continuously takes input from the present and past states periodically and after operating on these states, outputs a new state. The other part is a *memory* . It takes the present state and stores it up for the next cycle of operation.

In software engineering terms, the recursive part is made up of an *Invariant* which is the closed loop transfer function; a *guard* which is the error signal and the *body* which is the operation to calculate the control signal. These three define a loop operation. The memory part is some form of buffer that holds the calculated error signal for the next cycle of recursion.

Background Theory Consider a *single-input-single-output* unity feedback frequency model as shown in Figure 4.5, The *closed loop transfer function* is given by

$$\frac{y(s)}{r(s)} = \frac{G_p(s)G_c(s)}{1 + G_p(s)G_c(s)} \tag{4.4}$$

For a simple proportional controller, $G_c = K_p$, to have zero *steady state error*, y must be equal to r at d.c., i.e. $y(0) = r(0)$. From Equation 4.4,

$$y(0) = \frac{G_p(0)K_p}{1 + G_p(0)K_p}\, r(0) \tag{4.5}$$

where $G_p(0)$ is the d.c. gain of the plant G_p. Hence with a proportional controller, the steady state offset can only be reduced by making $G_p(0)K_p$ very large. Since $G_p(0)$ is a fixed parameter of the plant, this can be achieved by increasing the gain K_p. Since it affects the dynamics of the system, the range of K_p may need to be limited, in particular at higher levels, by considerations of stability.

Referring to Equation 4.4, the steady state error can-be eliminated if $G_c(0) = \infty$. This can be achieved if the controller contains an integrator as well as proportional control. The controller's transfer function then becomes:

$$G_c(s) = K_p + \frac{K_i}{s} \tag{4.6}$$

The effect of the integral action term is to give a controller output u(t) which changes at a rate proportional to the error $e(t)$. Thus,

$$\frac{d\,(u(t))}{dt} = K_i e(t) \tag{4.7}$$

$$u(t) = K_i \int_0^t e(t)dt \tag{4.8}$$

For a step error signal e, the integral component of the controller starts from zero and continues to increase as a result of the integral action. But the component due to proportional action remains constant. At some time the two terms are equal so that:

$$K_p e = K_i \int_0^t e(t)dt = K_i e(t) \tag{4.9}$$

This time is known as the *integral action time*, T_i, which is given by

$$T_i = \frac{K_p}{K_i} \tag{4.10}$$

Integral action is also known as 'reset' because of its ability to eliminate offset at the output. Combining Equations 4.6 and 4.10, the transfer function of the proportional plus integral, **PI** controller becomes:

$$G_c(s) = K_p(1 + \frac{1}{sT_i}) \tag{4.11}$$

Unfortunately, the integral action will increase the order of the characteristic equation, $1 + G_p(s)G_c(s) = 0$, and hence increase the possibility of instability. This increase in the phase lag due to the integral action has to be compensated by a reduction of proportional gain K_p.

Derivative action can be used where there is an excessive phase lag in the control process and to enhance stability. A controller with proportional plus derivative action, **PD**, has a transfer function of the form:

$$G_c(s) = K_p + K_d s \tag{4.12}$$

As in the case for integral action, a *derivative time*, T_d, is defined as the time required, in the presence of an error signal changing at a constant rate, for the proportional action component to become equal to the derivative component. Hence:

$$T_d = \frac{K_d}{K_p} \tag{4.13}$$

Combining Equations 4.11 and 4.12, the transfer function of the PD controller becomes:

$$G_c(s) = K_p(1 + T_d s) \tag{4.14}$$

This derivative action is 'anticipatory' and can be used to reduce the overshoot of a system. With all three terms included, the proportional plus integral plus derivative, PID, controller can be written in Laplace transform as:

$$G_c(s) = K_p(1 + \frac{1}{sT_i} + T_d s) = K_p + \frac{K_i}{s} + K_d s \tag{4.15}$$

or representing its input/output characteristics in the integral and differential form as:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{d\ e(t)}{dt} \tag{4.16}$$

This provides a controller pole at the origin and two zeros given by the root of the following equation.

$$s^2 + \frac{s}{T_d} + \frac{1}{T_i T_d} \tag{4.17}$$

There are various design methods by which values of K_p, K_i and K_d are to be chosen. One such method which was first inspired by Ziegler and Nichols [1942; 1964] by which the three parameters of the PID compensator are tuned according to different responses from experimental results. On the other hand, a digital computer controller [Houpis and Lamont, 1985] can be implemented via a discrete time, Z transformation of Equation 4.17. A general form of a discrete sampled time description of Equation 4.17 can be written as Equation 4.18.

The various discrete time controller design methods such as the *pole placement* method is documented in Aström and B. Wittenmark [1984], Aström and T. Haggland [1984] and are not discussed in this context.

$$G_c(z) = \frac{U(z)}{E(z)} = \frac{1 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}} \tag{4.18}$$

This can be rewritten in the following form:

$$U(z) = E(z) + a_1 z^{-1} E(z) + a_2 z^{-2} E(z) - b_1 z^{-1} U(z) - b_2 z^{-2} U(z) \tag{4.19}$$

Treating z^{-1} as a shift operator in the time domain as described in the last section, equation 4.19 is transformed into:

$$u(k) = e(k) + a_1 e(k-1) + a_2 e(k-2) - b_1 u(k-1) - b_2 u(k-2) \tag{4.20}$$

which is a recursive input/output relationship. Using Equation 4.20, a new control signal can be calculated from the previous sampled signals.

In this section, effort have been put into explaining and deriving the particular structure of a simple control system. This is justified by the need for a detailed description of a low level controller which forms part of the example in Section 4.4.3 and the case study discuss in Chapter 5 for which behaviours are to be specified.

4.2.3 Generalised Properties

Having considered the scenario of a general distributed control system in detail, in general terms, it is seen that such control systems consist of a number if real-time processes which may be distributed in space. The execution of these processes are strictly governed by hard real-time boundaries. A great proportion of these systems exhibit a significant degree of concurrency as pointed out by Cox *et la.* [1988] and in more concrete terms, control systems are characterised by the following properties:

- They have to be ready for and interact with external events from the *environment* and be able to handle interrupts. They are transaction orientated in which the sequence of actions carried out depends on the nature of the inputs by the operator and the sensory data.
- As in the case of a robot control system, data is entering the system and actions are being performed *continuously* from and to the environment resulting in a non-discrete time system.
- They exhibit some degree of *concurrency* in their architecture, such as a sensor-based robot system, processes controlling the data acquisition from multiple sensors are executed in parallel in order to avoid any skewed aliasing.
- Systems must be able to *synchronise* with external events; and processes within a system synchronise their actions with one another.
- A control system is seldom organised in a flat structure. Different levels of control can usually be identified and for a real system, processes constituting the system will be structured in a *hierarchical* manner. In the robotics example, a typical control hierarchy is given in Figure 4.6.
- One of the characteristics of distributed system is that constituent processes *communicate* with one another and/or with the environment. Information is being passed from one process to another either within the same or across hierarchies. Lower level control processes pass data among themselves and selected data may be passed to upper levels.

4.3 The Abstraction Tool Set

4.3.1 Assumptions on Real Systems

Real control systems are often arbitrarily defined in relation to their environment. An aircraft together with its well trained pilot can be visualised as a single control system, on the other

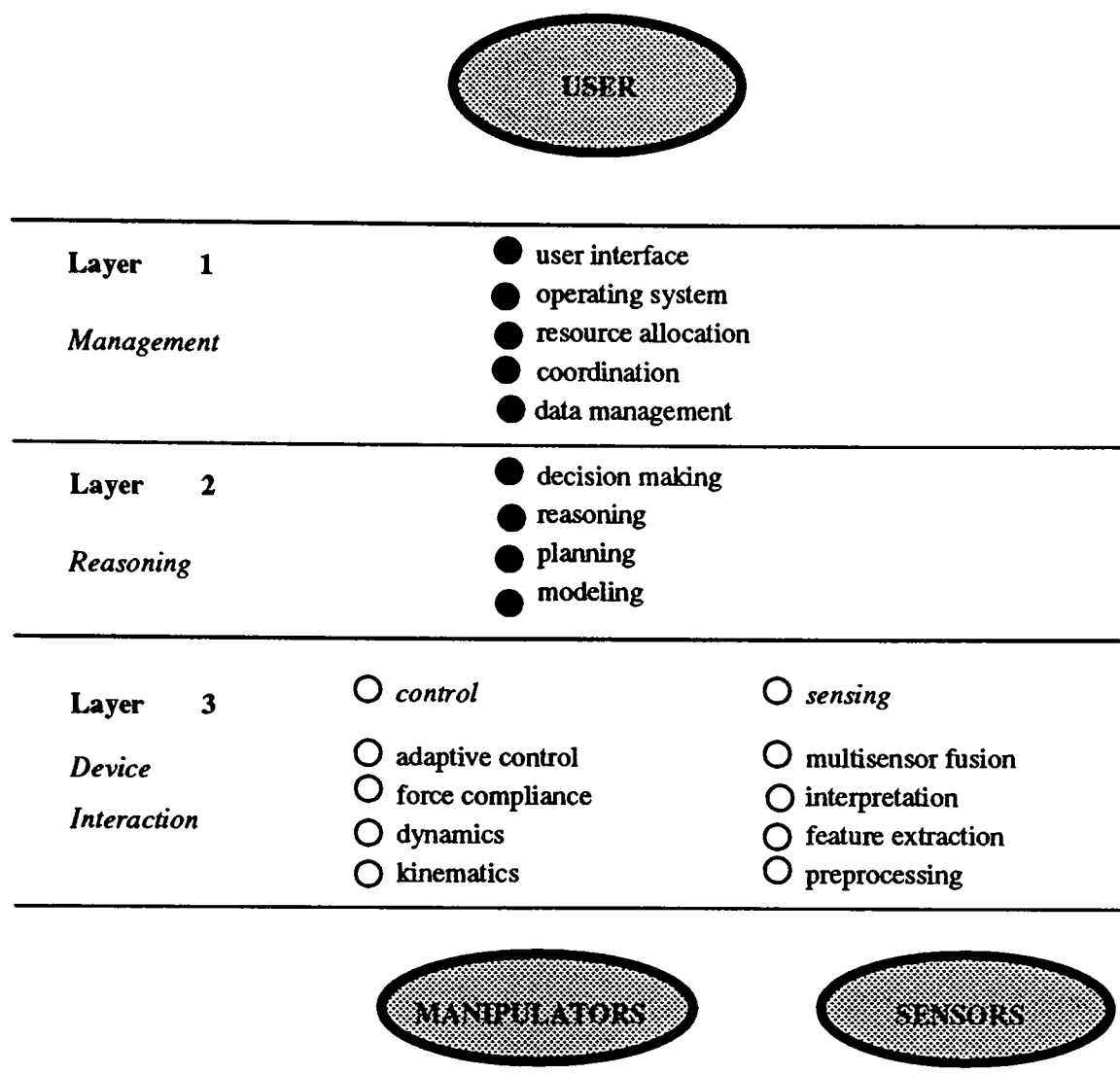


Figure 4.6: A typical Three Layer Hierarchy of a distributed control system

hand, the aircraft alone can be considered as a single control system, or in odd cases, one might even consider the airspace in which the aircraft is flying, plus the aircraft and the pilot as one single entity because the air flow in the atmosphere can affect the behaviour of the aircraft. Such arguments make it difficult to define a control system in concrete terms, and questions such as "what are the *external* stimuli?" or "what is *inside* a system?" must be clarified and made definite before any system can be designed and built successfully.

The assumption of an imaginary *boundary* must be made into which the system is being placed "inside"; it and the remainder of the universe been placed "outside" as the *environment*. This imaginary boundary has to be well defined so that a system can be established. This assumption has to be made before any substantial design work is undertaken and it is an essential and crucial decision to be made by a designer.

Another difficulty in understanding and analysing control systems is their continuous time nature. Actions happening within the system and with the environment happen continuously. In order to model such systems and to understand their behaviour and to implement them using digital controllers, approximations and assumptions have to be made. For most cases, it is justifiable to assume that a control system is a *discrete time* system in which its operations are characterised by

discrete events. A discrete event has no duration and it can happen at any moment in time. The use of discrete event controllers for practical applications has been demonstrated by work from Jacquot [1981] and Willson and Krogh [1990].

4.3.2 CSP Models for Control Processes

CSP has a number of semantic models which have been discussed in Section 2.5.2. Of the three models, two in particular are chosen to produce formal descriptions of control process behaviour and the abstraction tool set is designed and based on the *traces* and *failures* models.

As a trace is a record of the time history of a process, system safety properties, such as an operation performed by a process, can be captured. And for stronger arguments such as "some events will definitely occur", the refusals will be used to capture these liveness properties. The notations employed for traces and refusals are given in Appendix A.

Among the full set of process algebra given in Hoare [1985b], a subset of these discussed in Subsection 2.5.5 are adopted for the building of the abstraction tool set; and hence, they are used in the design and development of the control systems under the assumptions set out in Section 4.3.1.

4.3.3 Objectives

The central idea of the introduction and design of the CSP abstraction tool set is to provide a modular formalism to facilitate the development of correct control software. One of the main intentions is to formalise the transformation from an essential model of a control system to an implementation model in order to address the prime issue of integrating the formal and structured approaches in the unified methodology. This is a step towards the practical application of formal techniques for building an implementation in a step-wise manner. And the particular domain of formalism used is aimed to address, and hence expressed and reasoned with rigour about: process interaction through synchronous communication and parallelism among sequential processes. Moreover, the design of the tool set must satisfy a number of requirements. They have to fit in with the structured development technique of which vague design ideas are organised prior to implementation, the aim is to obtain a formal structured development method. The tool set should be able to express a set of salient behaviours of control systems as discussed in Section 4.2.3 through formal specifications, either directly or under acceptable assumptions and to reason about their properties with rigour.

In these respects, specification using CSP processes is chosen as one of the formalisms which encompasses the requirements stated and CSP is therefore the formal system being applied in this thesis. As has already been discussed in Section 2.5, CSP is designed to be a formal system for

concurrent systems with communicating processes. Maximum parallelism is implicitly defined and process interactions are well modelled by synchronous communication.

Following the first assumptions made in Section 4.3.1, an imaginary system boundary is defined prior to the design, notions of CSP process are applied to the definition of a system in which interaction with the environment is assumed through channel communication. The second assumption makes CSP the appropriate formalism for reasoning about discrete event systems. Since a CSP event is instantaneous and discrete, each of these can be mapped onto an exact transaction. Although the task of describing system states may not be apparent, they can be implicitly defined using two CSP events, one indicating the initiation of an action and the other signifies its termination.

4.3.4 The Tool Set

Considering the behaviours of control systems, a majority of these can be classified into a few major forms. A number of *specification predicates* are therefore introduced to encompass these general classes of behaviours based on the semantics of traces and refusals. However, the ultimate task of engineering design is to realize a system in real terms, that is to produce a working system implementation, and this prompts the design of a number of *generic processes* whose definitions correspond to the set of specification predicates.

This set of specification predicates together with their corresponding implementation as reflected by the generic processes make up the abstraction tool set for control system specification and development. Most importantly, since the tool set is built on the sound mathematical basis of CSP, the relation of a particular generic process satisfying its specification predicates can be verified rigourously. Hence, the transformation from system description (via specification predicates) to realization (with generic processes) is formalised. As such, it is intended that the application of the tool set formalises the process of system development. For the purposes of organisation of the tool set, the set of specification predicates is described which is followed by the discussion on generic processes with their corresponding sets of specification predicates which specify the behaviours of these processes.

4.3.4.1 Specification Predicates

It is noticeable from the detailed analysis of control systems that the *ordering* of control actions is often the most obvious observable phenomenon that captures its behaviour. Requirements such as *"sensor readings must be obtained before a joint torque is calculated"* and *"all actions must stop*

after the emergency stop has been activated" are common in control. Using the traces semantics, these ordering criteria are specified precisely as predicates such as:

$$\forall tr \mid ((tr \in \text{traces}(P)) \wedge (0 \leq tr \downarrow a - tr \downarrow b \leq n)) \tag{4.21}$$

Translated into English, Equation 4.21 states that for a process P with a trace tr, the number of occurrence of a must be greater than the number of occurrence of b with no more than n counts. In the case where P is a communicating process, the same predicate is interpreted differently since the sequence of events happened across channel b (that is the trace of P *restricting* the occurrence of events across channel b) is a prefix of the sequence of events across channel a with at most n events ahead. For this, the predicate has a stronger implication which specifies the occurrence of communication across channel a must *precede* that of b. And this is often written as:

$$tr \downarrow b \overset{n}{\leq} tr \downarrow a \tag{4.22}$$

with identical semantics.

Moreover, in describing the properties and behaviours of control systems in general, it is helpful to incorporate the notion of *state* information in a specification. For example, a common form of requirement will be:

"ACTION x WILL ONLY OCCUR WHEN THE SYSTEM IS IN A STATE Y" (4.23)

Although a CSP event is discrete and instantaneous, two distinctive events are utilised to capture the notion of state; with one event marking the initiation and the other marking the end of a state. To specify statement 4.23 formally in CSP, it is assumed that an action a defines the start of state Y and action b defines its end, then translating this conditional statement to CSP becomes:

$$(last\ tr = x) \Rightarrow (last\ (tr \restriction \{a, b\}) = a) \tag{4.24}$$

which is used to specify *state* information for a system.

A third type of specification predicate is used to define what is possible and what is impossible for a specific system. A specification of a system which is written in the refusals model given by Equation 4.25, in which x is an event of interest:

$$\text{CONDITION} \Rightarrow (x \notin ref) \tag{4.25}$$

states that the system must be ready to perform the action x under the given CONDITION such as after performing a specific sequence of actions.

As an example: "given the condition that c is the last action performed by the system, the system must be able to perform the sequence of actions s", is expressed formally in CSP as:

$$(last\ tr = c) \Rightarrow (s \cap ref) = \varnothing \tag{4.26}$$

Being common forms of system specification, it is therefore convenient to *abbreviate* the predicates describe above in more concise formats. This enables complex system behaviours to be written in clear and manageable forms. The following set of specification predicates are introduced in this thesis.

ORDER

$$ORDER\,(a, b, n) \;\widehat{=}\; 0 \leq tr \downarrow a - tr \downarrow b \leq n \tag{4.27}$$

The *ORDER* predicate is an abbreviation for describing the ordering property of events. The parameters a and b can be simple events in which the operator $\leq$ is a strict forward inequality, whereas in the case when both a and b are channel names, the operator $\leq$ becomes a prefix denotation between sequences of messages across channels a and b. However, it is essential that a and b must be compatible events in that both must have the same type.

START & END

$$tr\ START\,(a, b) \;\widehat{=}\; \left(\begin{array}{ll} & tr \restriction \{a, b\} = \langle\rangle \\ \vee & (tr \restriction \{a, b\} \neq \langle\rangle \wedge\ last\,(tr \restriction \{a, b\}) = a) \end{array} \right) \tag{4.28}$$

$$tr\ END\,(a, b) \;\widehat{=}\; \left(\begin{array}{ll} & tr \restriction \{a, b\} \neq \langle\rangle \\ \vee & last\,(tr \restriction \{a, b\}) = b \end{array} \right) \tag{4.29}$$

START and *END* [Jackson, 1989] are abbreviating predicates for defining system *states*. A particular state is set by an event which appears at the end of a trace. Equation 4.28 denotes the initial state of a process when there is only an empty trace or such a process is in a state that is set by an event which appears as the *first* parameter of the predicate. Equation 4.29 defines the state of a process which is set by the event that appears as its *second* parameter.

IFSTART & IFEND

$$c\ IFSTART\,(a, b) \;\widehat{=}\; (last\ tr = c) \Rightarrow (tr\ START\,(a, b)) \tag{4.30}$$

$$c\ IFEND\,(a, b) \;\widehat{=}\; (last\ tr = c) \Rightarrow (tr\ END\,(a, b)) \tag{4.31}$$

Equations 4.30 and 4.31 abbreviate the consequences after the execution of particular events. Equation 4.30 states that "c will only occur when event a occurs". This also implies that "c will only occur when the system is in a state initiated by the last occurrence of event a". Similarly, Equation 4.31 states the consequence of event b. Both of the equations stem from an earlier description given by Equation 4.24.

NOTREF & NOTFAIL

$$a\ NOTREF\ (s) \ \widehat{=}\ (a \notin ref) \wedge (ref \in \text{refusals}(P/s)) \tag{4.32}$$

$$s\ NOTFAIL\ (a,b) \ \widehat{=}\ \left(\begin{array}{ll} & (last\ tr \restriction \{a,b\} = a) \Rightarrow (b \notin ref) \\ \wedge & (ref \in \text{refusals}(P/s)) \end{array} \right) \tag{4.33}$$

Finally, Equations 4.32 and 4.33 are predicate abbreviations in relation to the failures semantics. They stem from Equations 4.25 and 4.26 for specifying system failures. They are generalised predicates with hybrid parameters as compared to the traces predicates. Parameters include traces and events in relation to the system specified and not only initial failures are described, failures of a process after performing any permissible trace are also catered for.

4.3.4.2 Generic Processes

The original idea of using simple and yet general processes for system definitions was due to Jackson [1989] in aiming at producing a system implementation by combining a number of these processes in parallel. This idea is extended with the introduction of a set of generic processes by which the repetitive structural behaviour of control systems is captured and implemented using parameterised CSP processes. In addition, provision has been made to formalise the DARTS representations through generic processes so as to generate a complimentary formalised pictorial representation of a design for the communication of ideas.

This group of CSP processes can be used to express real world objects with different degrees of abstraction and modularisation, hence their usage is not limited to a single level of a control hierarchy. The abstraction is controlled by using the *hiding* operation whereas modularisation is taken care of by *process relabelling*.

This set of generic processes are grouped into a number of classes which are discussed in detail in the following sections. The provision for each class of generic process to reflect the actions of control sub-systems in different hierarchies with different levels of abstraction has inherited the useful property of an object orientated description technique such that each "child" generic process

that stems from a particular "class" inherits all the properties of its parent together with its own more refined and specific properties.

To illustrate how a single class of generic process can be used to model systems and sub-systems within a control system hierarchy with different abstractions, the recursive CSP process F, given by Equation 4.34 models a system at its highest level of abstraction. Its interactions with its environment are undergone solely via events a and b. However, this top level process definition may be refined into two interacting sub-processes P and Q, each having its own atomic functions and are described by Equations 4.35 and 4.36 respectively.

$$F \;\widehat{=}\; a \rightarrow b \rightarrow F \tag{4.34}$$

$$P \;\widehat{=}\; a \rightarrow c \rightarrow P \tag{4.35}$$

$$Q \;\widehat{=}\; c \rightarrow b \rightarrow Q \tag{4.36}$$

$$F \;\Rrightarrow\; (P \parallel Q) \tag{4.37}$$

where $\Rrightarrow$ represents a refinement

Suppose that the refinement of F is given by Equation 4.37. It can be seen that the structures of the three processes F, P and Q belong to the same class of process, which are CSP processes with simple prefixes, yet they represent different abstractions of a system describing its behaviours in different hierarchies. Hence, a general process with a same structure as defined by F can be classified as a possible generic process. To verify the equivalence of the the refinement given by Equation 4.37, one can proceed by reversing the process of refinement and abstracting any internal transitions using the hiding operator on event c and the proof follows:

$$\begin{aligned}
(P \parallel Q)\backslash\{c\} &\equiv ((a \rightarrow c \rightarrow P) \parallel (c \rightarrow b \rightarrow Q)) \setminus \{c\} \\
&\equiv ((a \rightarrow c \rightarrow b) \rightarrow (P \parallel Q)) \setminus \{c\} \\
&\equiv ((a \rightarrow b) \rightarrow (P \parallel Q)) \\
&\equiv F \qquad \square
\end{aligned} \tag{4.38}$$

Since only events a and b are interactive with the environment, process F and process $(P \parallel Q)$ are equivalent from the point of view of the environment.

The idea of a modularisation in system construction is to introduce a set of well defined process definitions with provable properties so that complex systems can be built from smaller and more

manageable modules. Generic processes support this idea by manifesting themselves as formal building blocks for modelling a system. The use of process relabelling enables a single generic process to take different images in different applications. Using the above example, process P can be derived from process F using the operation of process relabelling, it is possible to write:

$$P = f(F) \tag{4.39}$$

and formally, this is written explicitly as:

$$\begin{aligned} P &\equiv F\,[b/c] \\ &\equiv \mu X \bullet (a \rightarrow c \rightarrow X) \end{aligned} \tag{4.40}$$

Process Q under the same argument can also be derived from F with a different relabelling, and it is seen that the overall system can be built up from a single form of generic process F.

After all, generic processes are more complex CSP processes, it follows that a number of algebraic laws that preserve the semantics of a process which are given in Hoare [1985b] and in Section 2.5.5 apply to generic processes as well. These laws allow the re-writing of a system definition using the tool set and to facilitate the reasoning with process definition and to establish properties of the corresponding processes; and if necessary, abstraction and parallel operators can be eliminated in the course of verification and refinement through process algebra. In the next section, each generic process is presented with its corresponding specification predicates in detail.

4.4 The Generic Processes

In order to reflect the behaviour of the specification predicates, a set of generic processes are designed so as to *implement* their action precisely. The set of processes are given by Processes 4.1 to 4.7. To differentiate the different generic processes, an in depth discussion of their properties, their corresponding specification predicates together with the satisfies relations, alphabet sets, the formalising of the representation used by their structured counterparts and possible applications is followed for each of them.

Process 4.1 *FSM* (a, b)

$$FSM\ (a, b) \ \widehat{=}\ \mu X \bullet (a \rightarrow b \rightarrow X) \tag{4.41}$$

$$\alpha(FSM(a, b)) = \{a, b\} \tag{4.42}$$

Process 4.2 *PERM* (a, b)

$$PERM\ (c, P, d, Q) \ \widehat{=}\ (\ c \rightarrow P \quad \Box\, d \rightarrow Q) \tag{4.43}$$

where $c \neq d$

$$\alpha(PERM(c, P, d, Q)) \ \widehat{=}\ \{c, d\} \cup \alpha P \tag{4.44}$$

and $\alpha P = \alpha Q$ as suggested by the explanation given in Sub-section 2.5.5.1

Process 4.3 *TPERM* (a, b)

$$TPERM\ (a, b) \ \widehat{=}\ (a \rightarrow b \rightarrow SKIP \quad |\ b \rightarrow a \rightarrow SKIP) \tag{4.45}$$

$$\alpha(TPERM(a, b)) \ \widehat{=}\ \{a, b\} \tag{4.46}$$

Process 4.4 *RPERM* (a, b)

$$RPERM\ (a, b) \ \widehat{=}\ \mu X \bullet \ (a \rightarrow b \rightarrow X \quad |\ b \rightarrow a \rightarrow X) \tag{4.47}$$

$$\alpha(RPERM(a, b)) \ \widehat{=}\ \{a, b\} \tag{4.48}$$

Process 4.5 *BUFFER* $(left, right, n)$

$$BUFFER_{\langle\rangle} \ \widehat{=}\ left\ ?\ x \rightarrow P_{\langle x \rangle} \tag{4.49}$$

$$BUFFER_{\langle x \rangle^\frown s} \ \widehat{=}\ (left\ ?\ y \rightarrow P_{\langle x \rangle^\frown s^\frown \langle y \rangle}\ |\ right\ !\ x \rightarrow P_s) \tag{4.50}$$

$$\text{if} \quad 0 \leq \#s + 1 < n \tag{4.51}$$

$$\widehat{=}\ right\ !\ x \rightarrow P_s \tag{4.52}$$

$$\text{if} \quad \#s + 1 = n$$

$$\alpha(BUFFER) \ =\ \{left.z,\ right.z\} \tag{4.53}$$

where z is any arbitrary messages (4.54)

Process 4.6 $INIT\ (c, a, b)$ & $UPDATE\ (c, a, b)$ [Jackson, 1989]

$$\begin{aligned} INIT(c,a,b) \;\widehat{=}\;\; & a \rightarrow UPDATE\ (c,a,b) \\ & \Box\ b \rightarrow INIT\ (c,a,b) \end{aligned} \tag{4.55}$$

$$\begin{aligned} UPDATE(c,a,b) \;\widehat{=}\;\; & c \rightarrow UPDATE\ (c,a,b) \\ & \Box\ a \rightarrow UPDATE\ (c,a,b) \\ & \Box\ b \rightarrow INIT\ (c,a,b) \end{aligned} \tag{4.56}$$

$$\alpha(INIT(c,a,b)) = \{a,b,c\} \tag{4.57}$$

$$\alpha(UPDATE(c,a,b)) = \{a,b,c\} \tag{4.58}$$

Process 4.7 $MUX(x, y, m)$

$$\begin{aligned} MUX(x,y,m) \;\widehat{=}\; \mu X \bullet\ & (x\,?\,a \rightarrow y\,?\,b \rightarrow m\,!\,f(a,b) \rightarrow X \\ & \mid y\,?\,b \rightarrow x\,?\,a \rightarrow m\,!\,f(a,b) \rightarrow X) \end{aligned} \tag{4.59}$$

$$\alpha(MUX(x,y,m)) = \{x.a,\ y.b,\ m.f(a,b)\} \tag{4.60}$$

As the primary motives behind the design of the set of generic processes is to obtain a formal realization of the behaviours defined by the specification predicates, *all* the generic processes implement the *ORDER* predicate given by Equation 4.27. In particular, Process 4.6 is designed to realize the behaviours of *IFSTART* and *IFEND* that are given by Equations 4.30 and 4.31. And all the processes implicitly reflect the failures conditions given by Equations 4.32 and 4.33. In the following sections, each generic process is discussed accordingly.

4.4.1 FSM

The *FSM* represents the largest class of control processes, such as all the transaction processes and finite state machines, from which its name originates. Any process with toggling actions between two distinct events is described by this process. The toggling actions will repeat indefinitely unless halted by external synchronising events. The order of the occurrence of the two toggling events must not be altered.

4.4.1.1 Specifications

With these requirements, *FSM* as defined by Process 4.1 satisfies the following specifications:

$$FSM(a,b) \quad \widehat{=} \quad \mu X \bullet (a \rightarrow b \rightarrow X)$$

$$\begin{aligned}
\textbf{sat} \quad & (traces\,(FSM) \subseteq \alpha\,(FSM(a,b))^*) && (4.61)\\
\wedge\ & (0 \leq tr \downarrow a - tr \downarrow b \leq n) && (4.62)\\
\wedge\ & (last\ tr \upharpoonright \{a,b\} = a \Rightarrow b \notin ref) && (4.63)\\
\wedge\ & (last\ tr \upharpoonright \{a,b\} = b \Rightarrow a \notin ref) && (4.64)
\end{aligned}$$

Assertion 4.61 specifies the closure property whereas assertion 4.62 specifies the strict ordering property. Assertions 4.63 and 4.64 define the liveness properties of the toggling events. Applying the abbreviations given by the specification predicates, the above specification can be rewritten in a clearer and more concise format as:

$$FSM(a,b) \quad \widehat{=} \quad \mu X \bullet (a \rightarrow b \rightarrow X)$$

$$\begin{aligned}
\textbf{sat} \quad & traces\,(FSM) \subseteq \alpha\,(FSM(a,b))^* && (4.65)\\
\wedge\ & ORDER\,(a,b,1) && (4.66)\\
\wedge\ & tr\ NOTFAIL\,(a,b) && (4.67)\\
\wedge\ & tr\ NOTFAIL\,(b,a) && (4.68)
\end{aligned}$$

4.4.1.2 Satisfaction of Specification

To establish *FSM* is a correct realization of the specifications and hence establishes its properties, it is necessary to prove that *FSM* satisfies assertions 4.65 to 4.68. The technique of recursive induction is applied to establish the truthfulness of the general case for this particular recursive process using the result of the inductive proofs. Assertions 4.65 and 4.66 are proved using the traces semantics, and the later two assertions require the assistance of the failures proof rules. A full proof strategy is presented in this section so as to give a general picture of constructing CSP proofs.

Proof 4.1 *Assertion given by Equation 4.65*

The specification describes a well behaved process in which every event must lie within the alphabet set of the process. The proof obligation is therefore:

To Prove:

$$FSM(a,b) \textbf{ sat } tr \in traces\,(FSM(a,b)) \subseteq \alpha\,(FSM(a,b))^*$$

Then using a case analysis:

Case: $\#\ tr = 0$

$$\begin{array}{rrcll} & \langle\rangle & \in & traces\,(FSM) & \\ & traces\,(FSM) & \subseteq & \alpha\,(FSM)^* & \text{traces of a process} \\ \Rightarrow & \langle\rangle & \in & \alpha\,(FSM)^* & \end{array}$$

Case: $\#\ tr \neq 0$

Assume: $tr^+ \in \alpha\,(FSM)^*$

$$\begin{array}{rrcll} \text{if} & tr^+ & = & \langle\rangle & \\ \Rightarrow & last\ tr & \in & \alpha(FSM \upharpoonright \langle a\rangle)^* & \text{def of } FSM \end{array}$$

$$\begin{array}{rrcll} \text{if} & (tr^+ \neq \langle\rangle) \wedge & & (last\ tr^+ \in \alpha\,(FSM \upharpoonright \langle a\rangle)^*) & \\ \Rightarrow & last\ tr & \in & \alpha\,(FSM \upharpoonright \langle b\rangle)^* & \text{def of } FSM \end{array}$$

$$\begin{array}{rrcll} \text{if} & (tr^+ \neq \langle\rangle) \wedge & & (last\ tr^+ \in \alpha\,(FSM \upharpoonright \langle b\rangle)^*) & \\ \Rightarrow & last\ tr & \in & \alpha\,(FSM \upharpoonright \langle a\rangle)^* & \text{def of } FSM \\ \Rightarrow & tr & \in & \alpha\,(FSM)^* & \end{array}$$

Hence the following property of FSM is established. □

Property 4.1

$$FSM(a,b) \textbf{ sat } tr \in \alpha(FSM)^*$$

Proof 4.2 *Assertion given by equation 4.66 to 4.68*

The proof is constructed using recursive induction. The indirectly observable behaviours specified using the refusals are combined with the ordering property as they are related to one another. A proof is first carried out to verify the ordering property of FSM's events and is then followed by the stronger arguments about its refusals.

ORDER $(a, b, 1)$ is a satisfiable specification, which is satisfied by at least the process *STOP* and this parameterised predicate is also continuous. The recursive definition of *FSM* is a sequential composition of two prefixes, and so it is a constructive function with a unique solution [Hoare, 1985b].

Assume that X **sat** *ORDER* $(a, b, 1)$, i.e.;

$$a \rightarrow b \rightarrow X \quad \textbf{sat} \quad \left(\begin{array}{ll} & ORDER\,(a, b, 1) \\ \wedge & tr\ NOTFAIL\,(a, b) \\ \wedge & tr\ NOTFAIL\,(b, a) \end{array} \right)$$

To facilitate the use of case analysis in recursive induction, the above assertions are translated into the following three cases:

$$a \rightarrow b \rightarrow X \quad \textbf{sat} \quad \left(\begin{array}{ll} & (tr = \langle\rangle \wedge a \notin ref) \\ \vee & (tr_0 = a \wedge tr' = \langle\rangle \wedge b \notin ref) \\ \vee & (tr_0 = a \wedge (tr')_0 = b \wedge ORDER\,(a, b, 1)[tr''/tr]) \end{array} \right)$$

Case: $tr = \langle\rangle$

$$\begin{array}{ll} & tr \downarrow a = tr \downarrow b = 0 \\ \Rightarrow & ORDER\,(a, b, 1) \end{array}$$

Case: $tr = \langle a \rangle$

$$\begin{array}{ll} & tr \downarrow a = 1 \wedge tr \downarrow b = 0 \\ \Rightarrow & tr \downarrow a - tr \downarrow b = 1 \\ \Rightarrow & 0 \leq tr \downarrow a - tr \downarrow b \leq 1 \\ \Rightarrow & ORDER\,(a, b, 1) \end{array}$$

Case: $tr = \langle a, b \rangle \frown tr''$

$$\begin{array}{ll} & ORDER\,(a, b, 1)[tr''/tr] \\ \Rightarrow & 0 \leq tr'' \downarrow a - tr'' \downarrow b \leq 1 \\ \Rightarrow & 0 \leq (tr \downarrow a - 1) - (tr \downarrow b - 1) \leq 1 \\ \Rightarrow & 0 \leq tr \downarrow a - tr \downarrow b \leq 1 \\ \Rightarrow & ORDER\,(a, b, 1) \end{array}$$

For the refusals properties, since *FSM* is constructive, Hoare's laws for process refusals can be invoked. Again, a case analysis is performed:

Case: $tr = \langle\rangle$

$$tr \downarrow a = tr \downarrow b = 0$$

Let $F(X) = a \rightarrow (b \rightarrow X)$

then $ref \in refusals\,(F(X)) = refusals\,(a \rightarrow (b \rightarrow X))$

let $P(X) \mathrel{\widehat{=}} b \rightarrow X$

then $refusals\,(F(X)) = refusals\,(a \rightarrow P(X))$

$\Rightarrow$ $ref \subseteq (\alpha P(X) - \{a\})$

$\Rightarrow$ $a \notin ref$

$\Rightarrow$ $tr\ NOTFAIL\,(b, a)$

Case: $tr \neq \langle\rangle$

Subcase: $tr \downarrow a = tr \downarrow b$

given $traces\,(FSM) = traces\,(F^n(FSM)) = \bigcup_{n \geq 0} \{tr \mid tr \leq \langle a, b\rangle^n\}$

$\Rightarrow$ $tr \in traces\,(a \rightarrow b \rightarrow F^{n-1}(FSM))$

if $last\,(tr \restriction \{a, b\}) = b$

$\Rightarrow$ $a \notin ref$

$\Rightarrow$ $tr\ NOTFAIL\,(b, a)$

Subcase: $tr \downarrow a \neq tr \downarrow b$

$\Rightarrow$ $tr \leq \langle a, b\rangle^n \frown \langle a\rangle$

$\Rightarrow$ $last\,(tr \restriction \{a, b\}) = a$

$\Rightarrow$ $tr \in traces\,(a \rightarrow b \rightarrow a \rightarrow F^{n-1}(FSM))$

$\Rightarrow$ $b \notin ref$

$\Rightarrow$ $tr\ NOTFAIL\,(a, b)$ □

These results have established the following three properties of FSM:

Property 4.2

$$FSM(a,b) \textbf{ sat } ORDER\,(a,b,1)$$

Property 4.3

$$FSM(a,b) \textbf{ sat } tr\ NOTFAIL\,(a,b)$$

Property 4.4

$$FSM(a,b) \textbf{ sat } tr\ NOTFAIL\,(b,a)$$

4.4.1.3 Laws

Since *FSM* is a process of double prefixing, and is continuous and constructive, which mean that all the laws that are valid to a CSP prefix process are also valid to *FSM*. It is associative and communitative; and operators such as general choice and parallel constructor as applied to *FSM* observe properties such as identity, symmeticity and associativity. In particular, it exhibits similar properties to a guarded process in the general choice operation.

If the operation *first* is defined as:

$$first\,(P) \;\widehat{=}\; \text{the } \textbf{first} \text{ event that } P \text{ is capable of performing}$$

$$\text{then} \quad first\,(FSM) = a$$

Law 4.1

$$FSM_1 \,\square\, FSM_2 = FSM_1 \mid FSM_2 \quad \text{if} \quad first\,(FSM_1) \neq first\,(FSM_2)$$

$$= FSM_1 \sqcap FSM_2 \quad \text{if} \quad first\,(FSM_1) = first\,(FSM_2)$$

4.4.1.4 Formalisation of DARTS Representation and Utilisation

According to its specifications, *FSM* is used to model any processes with Properties 4.1 to 4.4. It represents the behaviour of any sequential finite state machine with at least two distinct interactions with its environment. With this, the generic process *FSM* as given by Equation 4.41 can be formally mapped onto a structured transformation given in Figure 4.7 if a graphical representation is ever required. And hence, the formalisation of DARTS representations. Moreover, because of its generic nature, its arguments may not only take the form of simple events, but also communicating events or even other generic processes. This provides further flexibilities for *FSM* to formalise not only control transformations which involve synchronising signals, but also data transformations.

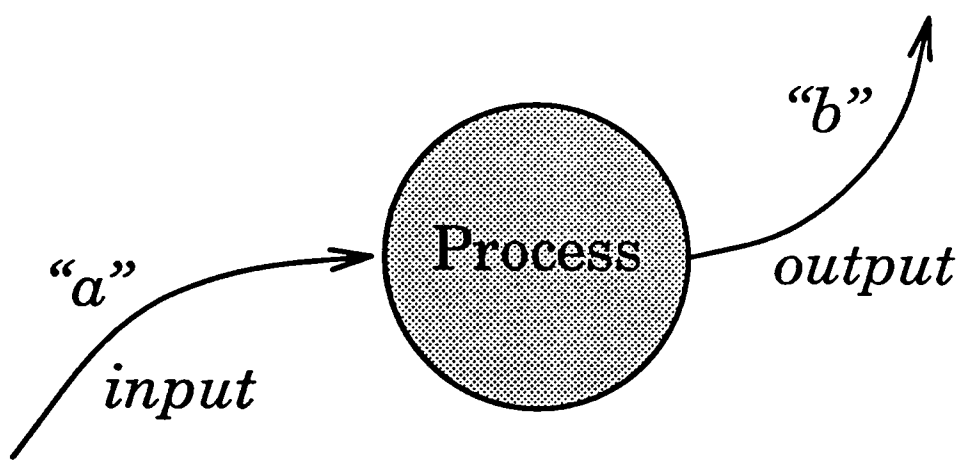


Figure 4.7: A transformation schema representation of the generic process *FSM* $(a,\ b)$

A very useful form of *FSM* can be derived from the basic form described by the Process 4.1. Using communicating events as its parameters such as: $(a\ !\ x)$, the extended generic process *FSM* is written as:

Process 4.8 *FSM (a.x, b.y)*

$$FSM(a.x,\ b.y) \equiv \mu X \bullet (a\ ?\ x \rightarrow b\ !\ y \rightarrow X) \tag{4.69}$$

$$\alpha(FSM(a.x,\ b.y)) = \{a.x,\ b.y\} \tag{4.70}$$

The alphabet of this form of *FSM* can be written in a more specific form in which the alphabet of each communicating channel is identified.

$$\alpha\ a(FSM(a.x,\ b.y)) = \{x\}$$

$$\alpha\ b(FSM(a.x,\ b.y)) = \{y\}$$

This extended form of *FSM* inherits all the properties of the original form of *FSM* either directly or with modified interpretations such as in the case of *ORDER*. For a communicating *FSM* given by Equation 4.69, its *ORDER* property as given below is interpreted as the traces property of the two traces recorded from its input and output channel communications. The following equation defines the traces from its output to be a *prefix* trace of its input, and hence gives the ordering property.

$$ORDER\,(a,\ b,\ 1) \ \widehat{=}\ tr \downarrow b \overset{1}{\leq} tr \downarrow a$$

Similar to ordinary *FSM*, many of the laws governing the process algebra of the basic form are equally valid on the extended form such as described by laws 4.2 and 4.3. Also, the extended *FSM* are idempotent, associative and symmetric over the general choice and parallel operators.

Law 4.2

$$FSM \parallel STOP = STOP$$

Law 4.3

$$FSM \ \Box\ STOP = FSM$$

However, since the extended *FSM* models channel communication, this has made the laws for parallel composition and choice between *FSM*s different to those for simple *FSM*. These new laws enable the extended version of *FSM* to be re-written during process algebra reasoning. The two laws concerning parallel construction and choice are given as follows:

Law 4.4

$$FSM_1 \parallel FSM_2 = FSM_1 \parallel\!\parallel\!\parallel FSM_2 \quad (4.71)$$

if $chan\,(FSM_1) \cap chan\,(FSM_2) = \varnothing$

where $chan\,(FSM_1) \ \widehat{=}$ the set of channels from FSM_1

$$= (FSM_1 \parallel FSM_2) \setminus (chan\,(FSM_1) \cap chan\,(FSM_2))$$

$$= FSM_3 \quad (4.72)$$

otherwise

Law 4.5

$$FSM_1 \,\square\, FSM_2 = FSM_1 \quad \text{if} \quad first\,(FSM_1) \in (\alpha\,(FSM_1) - \alpha\,(FSM_2)) \tag{4.73}$$

$$= FSM_2 \quad \text{if} \quad first\,(FSM_2) \in (\alpha\,(FSM_2) - \alpha\,(FSM_1)) \tag{4.74}$$

$$= FSM_1 \sqcap FSM_2 \quad \text{otherwise} \tag{4.75}$$

It can be seen that Law 4.5 is a direct deduction from the law on general choice and its validity is based on the corresponding CSP laws. Nevertheless, law 4.4 can be established from the following proof.

Proof 4.3 *Proof of ordering property for Law 4.4*

The constructive property of the law can be proved using the traces model, then the law itself can be established using process algebra as given by Proof 4.4. Starting with the following condition:

$$(chan\,(FSM_1) \cap chan\,(FSM_2) \neq \varnothing) \Rightarrow \begin{pmatrix} chan\ o(FSM_1) \cap chan\ i(FSM_2) \neq \varnothing \\ \vee \quad chan\ i(FSM_1) \cap chan\ o(FSM_2) \neq \varnothing \end{pmatrix}$$

where $chan\ o(FSM_1) \mathrel{\widehat{=}}$ set of output channel from the process FSM_1

and $chan\ i(FSM_1) \mathrel{\widehat{=}}$ set of input channel from the process FSM_1

Then follow by case analysis on the two disjuncts.

Case: $chan\ o(FSM_1) \cap chan\ i(FSM_2) \neq \varnothing$

$$FSM_1 \quad \textbf{sat} \quad ORDER\,(a_1,\ b_1,\ 1) \qquad \text{Property 4.2}$$
$$FSM_2 \quad \textbf{sat} \quad ORDER\,(a_2,\ b_2,\ 1) \qquad \text{Property 4.2}$$

$$(FSM_1 \parallel FSM_2) \quad \textbf{sat} \quad \begin{pmatrix} & ORDER\,(a_1,\ b_1,\ 1) \\ \wedge & ORDER\,(a_2,\ b_2,\ 1) \end{pmatrix} \qquad \text{traces of } \parallel \text{ processes}$$

$$\Rightarrow \quad (FSM_1 \parallel FSM_2) \quad \textbf{sat} \quad \begin{pmatrix} & (tr \downarrow b_1 \overset{1}{\leq} tr \downarrow a_1) \\ \wedge & (tr \downarrow b_2 \overset{1}{\leq} tr \downarrow a_2) \end{pmatrix}$$

But
$$\begin{pmatrix} & chan\ o(FSM_1) \\ \cap & chan\ i(FSM_2) \neq \varnothing \end{pmatrix}$$
$$\Rightarrow \quad b_1 = a_2 = c$$

$$\Rightarrow \quad (FSM_1 \parallel FSM_2) \quad \textbf{sat} \quad \begin{pmatrix} & (tr \downarrow c \overset{1}{\leq} tr \downarrow a_1) \\ \wedge & (tr \downarrow b_2 \overset{1}{\leq} tr \downarrow c) \end{pmatrix}$$

$$\Rightarrow \quad (tr \downarrow b_2 \overset{2}{\leq} tr \downarrow a_1)$$
$$\Rightarrow \quad ORDER\,(a_1,\ b_2,\ 2) \qquad \text{def of ORDER}$$

Case: $chan\ i(FSM_1) \cap chan\ o(FSM_2) \neq \varnothing$

The proof follows that of the previous case, and is not repeated.

Hence:

$$(FSM_1 \parallel FSM_2) \quad \textbf{sat} \quad ORDER\,(a,\ b,\ 2) \qquad \square$$

Proof 4.4 *Proof for Law 4.4*

Given:

$$FSM_1 \ \widehat{=}\ FSM(a_1.x_1,\ b_1.y_1)$$
$$FSM_2 \ \widehat{=}\ FSM(a_2.x_2,\ b_2.y_2)$$
$$chan\,(FSM_1) \cap chan\,(FSM_2) = b_1$$
$$\Rightarrow \quad a_2 = b_1$$

To Prove:

$$(FSM_1 \parallel FSM_2) \setminus (chan\,(FSM_1) \cap chan\,(FSM_2)) = FSM_3$$

Using process algebra, the proof is presented as follows:

$$
\begin{aligned}
&(FSM_1 \parallel FSM_2) \setminus (chan\,(FSM_1) \cap chan\,(FSM_2)) \\
&\quad = \quad (\ (\mu X \bullet (a_1\,?\,x_1 \rightarrow b_1\,!\,y_1 \rightarrow X) \\
&\qquad \parallel (\mu Y \bullet (a_2\,?\,x_1 \rightarrow b_2\,!\,y_2 \rightarrow Y))) \setminus (chan\,(FSM_1) \cap chan\,(FSM_2)) \\
&\quad = \quad (\ (\mu X \bullet (a_1\,?\,x_1 \rightarrow b_1\,!\,y_1 \rightarrow X) \\
&\qquad \parallel (\mu Y \bullet (b_1\,?\,x_2 \rightarrow b_2\,!\,y_2 \rightarrow Y))) \setminus (chan\,(FSM_1) \cap chan\,(FSM_2)) \\
&\quad = \quad (\mu X \bullet (a_1\,?\,x_1 \rightarrow b_1\,!\,y_1 \rightarrow b_2\,!\,y_2 \rightarrow X)) \setminus (b_1) \\
&\quad = \quad \mu X \bullet (a_1\,?\,x_1 \rightarrow b_2\,!\,y_2 \rightarrow X) \\
&\quad = \quad FSM(a_1.x_1, b_2.y_2) \\
&\quad = \quad FSM_3
\end{aligned}
$$

The result establishes the Law. □

4.4.2 INIT and UPDATE

In order to model conditional system behaviour which is determined by different embedded states, a pair of complementary processes, namely *INIT* and *UPDATE* are designed according to the specifications described in the following section. Considering the functionality of Process 4.6, *INIT* prevents the event c from occurring until an a event has occurred more recently than an event b, while *UPDATE* differs from *INIT* by allowing event c to occur first even though there is no occurrence of either a or b.

4.4.2.1 Specifications

INIT and *UPDATE* are designed to satisfy the following specifications:

$$
INIT\,(c, a, b) \quad \widehat{=} \quad \left(\begin{array}{ll} & a \rightarrow UPDATE\,(c, a, b) \\ \Box & b \rightarrow INIT\,(c, a, b) \end{array} \right)
$$

$$
\textbf{sat} \qquad (traces\,(INIT) \subseteq (\alpha\,(INIT) \cup \alpha\,(UPDATE))^*) \tag{4.76}
$$

$$\wedge\,(c\ IFEND\ (b,a)) \tag{4.77}$$

$$\wedge\,(\{a,b\} \not\subseteq ref \wedge tr = \langle\rangle) \tag{4.78}$$

$$\wedge\,(c\ IFEND\ (b,a) \Rightarrow (c \notin ref)) \tag{4.79}$$

and

$$UPDATE\,(c,a,b) \quad \widehat{=} \quad \left(\begin{array}{ll} & c \rightarrow UPDATE\,(c,a,b) \\ \Box & a \rightarrow UPDATE\,(c,a,b) \\ \Box & b \rightarrow INIT\,(c,a,b) \end{array}\right)$$

$$\textbf{sat} \quad (traces\,(UPDATE) \subseteq (\alpha(INIT) \cup \alpha(UPDATE))^*) \tag{4.80}$$

$$\wedge\,(c\ IFSTART\ (a,b)) \tag{4.81}$$

$$\wedge\,(\{a,b\} \not\subseteq ref \wedge tr = \langle\rangle) \tag{4.82}$$

$$\wedge\,(c\ IFSTART\ (a,b) \Rightarrow (c \notin ref)) \tag{4.83}$$

4.4.2.2 Satisfaction of Specification

Since both *INIT* and *UPDATE* are continuous mutually recursive functions, their properties can be proved simultaneously by considering a two dimensional product space. The properties of a closed function as given by Equations 4.80 and 4.76 is described as:

$$INIT \quad \textbf{sat} \quad \left(\begin{array}{ll} & traces\,(INIT) \subseteq (\alpha\,(INIT) \cup \alpha\,(UPDATE))^* \\ \wedge & traces\,(UPDATE) \subseteq (\alpha\,(INIT) \cup \alpha\,(UPDATE))^* \end{array}\right)$$

The proof proceeds using process algebra to expand the mutually recursive functions into a *tail-recursive* form and then the required property is deduced from the laws of general choice and recursion. The proof of the first conjunction is given as follows:

$$INIT(c,a,b)$$

$$= \quad b \rightarrow INIT(c,a,b) \;\Box\; a \rightarrow UPDATE(c,a,b)$$

$$= \quad (\mu\,X \bullet (b \rightarrow X \;\Box\; a \rightarrow SKIP)\,;\;\; UPDATE(c,a,b))$$

$$= \quad (\mu\,X \bullet (b \rightarrow X \;\Box\; a \rightarrow SKIP))\,;$$
$$(c \rightarrow UPDATE(c,a,b) \;\Box\; a \rightarrow UPDATE(c,a,b) \;\Box\; b \rightarrow INIT(a,b,c))$$

$$= \quad (\mu\,X \bullet (b \rightarrow X \;\Box\; a \rightarrow SKIP))\,;$$

$$(\mu\, Y \bullet (c \rightarrow Y \;\Box\; a \rightarrow Y \;\Box\; b \rightarrow SKIP))\;;\; INIT(a, b, c))$$

$$= \quad \mu\, Z \bullet (\mu\, X \bullet (b \rightarrow X \;\Box\; a \rightarrow SKIP))\;;$$
$$(\mu\, Y \bullet (c \rightarrow Y \;\Box\; a \rightarrow Y \;\Box\; b \rightarrow SKIP))\;;$$
$$Z$$

From the tail-recursive form of *INIT*, the property described by the first conjunct is proved. The second conjunct can be verified in a similar way and is not repeated. Hence the following properties of *INIT* and *UPDATE* are established:

Property 4.5

$$INIT\ (c, a, b)\ \textbf{sat}\ traces(INIT) \subseteq (\alpha\ (INIT) \cup \alpha\ (UPDATE))^*$$

Property 4.6

$$UPDATE\ (c, a, b)\ \textbf{sat}\ traces(UPDATE) \subseteq (\alpha\ (INIT) \cup \alpha\ (UPDATE))^*$$

The following properties correspond to assertions 4.77, 4.78, 4.81 and 4.82 are given below. The proofs for each of these properties are given in Appendix C.

Property 4.7

$$INIT\ (c, a, b)\ \textbf{sat}\ (c\ IFEND\ (b, a))$$

Property 4.8

$$INIT\ (c, a, b)\ \textbf{sat}\ \{b, a\} \not\subseteq ref$$

Property 4.9

$$INIT\ (c, a, b)\ \textbf{sat}\ c\ IFEND\ (b, a) \Rightarrow (c \notin ref)$$

Property 4.10

$$UPDATE\ (c, a, b)\ \textbf{sat}\ (c\ IFSTART\ (a, b))$$

Property 4.11

$$UPDATE\ (c, a, b)\ \textbf{sat}\ \{a, b\} \not\subseteq ref$$

Property 4.12

$$UPDATE\ (c, a, b)\ \textbf{sat}\ c\ IFSTART\ (a, b) \Rightarrow (c \notin ref)$$

4.4.2.3 Formalisation of DARTS Representation and Utilisation

The generic processes *INIT* and *UPDATE* are designed to formalise state dependent transformations which are often used to describe conditional operations in control systems.

As the properties of the two complementary processes suggest, they are best used to reflect the behaviours of a system whose actions are initiated by some known states. Figure 4.8 shows a formal mapping of the pair of generic processes given by Process 4.6 into a DARTS transformation. This reflects the action of a hypothetical system which only starts when it is enabled by a user's signal, "*a*". Once this system has been started, only a change in the system status parameter resulting from an external implied emergency signal such as "*b*" will reset the system. A re-initialisation is then necessary to restart the system. Otherwise, the system will undergo normal transitions with the execution of a normal action such as "*c*".

4.4.3 BUFFER

The generic process *BUFFER* is a super set of buffering processes which excludes the infinite buffer that is described in Sub-section 2.5.6.1. The process *BUFFER* has three parameters, two of these define its input/output channels and the third one, n, which defines the maximum size of the buffer.

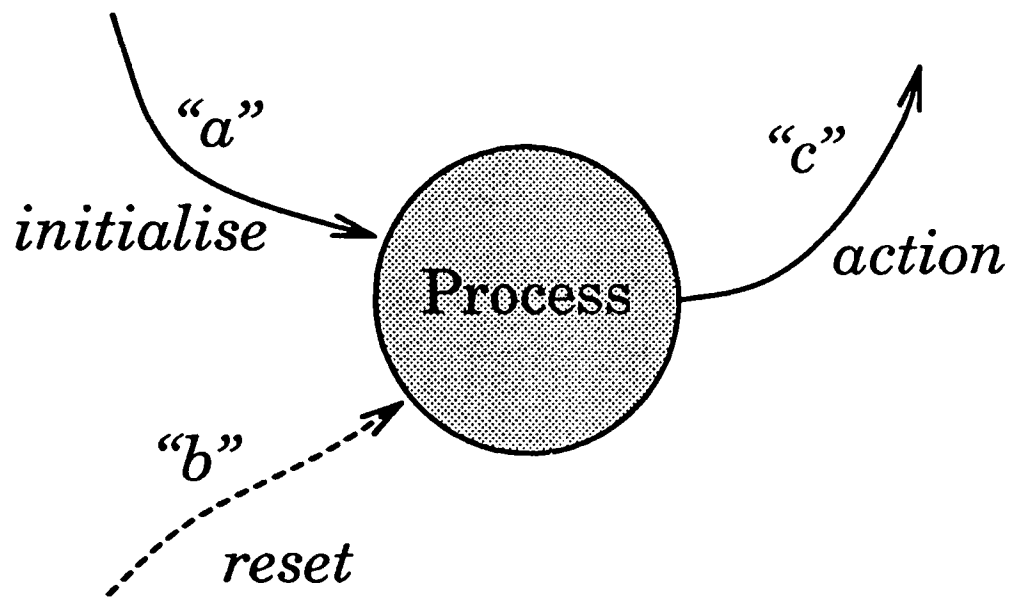


Figure 4.8: A transformation mapping of the generic processes *INIT* (c, a, b) and *UPDATE* (c, a, b)

4.4.3.1 Specification

As required, a buffer is capable of storing messages. Messages can be added from an input channel and be removed from another provided the size of the store is not exceeded. As the input and output actions both affect the *number* of items within the buffer, they have to proceed in a synchronous manner. Synchronisation is necessary as both actions share a common state variable — the level of the buffer.

Process 4.5 is therefore designed to satisfy the following specifications which define its property:

$$BUFFER\ (left,\ right,\ n)$$

$$\textbf{sat} \quad (ORDER\ (left, right, n)) \tag{4.84}$$

$$\wedge\ (tr\ START\ (right,\ left) \Rightarrow \{left\} \not\subseteq ref) \tag{4.85}$$

$$\wedge\ (tr\ END\ (right,\ left) \wedge (0 \leq \#s + 1 < n) \Rightarrow \{left, right\} \not\subseteq ref) \tag{4.86}$$

$$\wedge\ (tr\ END\ (right,\ left) \wedge (\#s + 1 = n) \Rightarrow \{right\} \not\subseteq ref) \tag{4.87}$$

$$\text{where} \quad \#s \mathrel{\widehat{=}} (tr \downarrow left - tr \downarrow right)$$

The specifications given resemble those given by Equation 2.42 in the example discussed in Subsection 2.5.6.1. The assertion given by Equation 4.84 defines an extra constraint for the *BUFFER* process in which the message must enters the buffer store from the left and leaves from the right and never in the opposite direction. Moreover, since the size of the buffer is parameterised by the

value of n, a *fullness* condition has been imposed onto the specifications given by Equations 4.86 and 4.87.

4.4.3.2 Satisfaction of Specification

The refusal property as described below is proved using the result from Proof 2.2 following a case analysis.

Property 4.13

$$BUFFER \textbf{ sat } tr\ START\ (right, left) \Rightarrow \{left\} \not\subseteq ref$$

Proof 4.5 *Proof for Property 4.13*

There are three cases to consider: when the buffer is empty, the buffer is partially filled and the buffer is full.

Case: *An empty buffer*

$$\begin{aligned} & \#tr = 0 \\ \Rightarrow\ & tr = \langle\rangle \\ \Rightarrow\ & \{left\} \not\subseteq ref \end{aligned}$$

Case: *A partially filled buffer*

$$\begin{aligned} \text{Given}\quad & (tr \neq \langle\rangle) \wedge (0 \leq \#s + 1 < n) \\ \Rightarrow\ & last\ (tr \downarrow \{left, right\}) = right && \text{definition of } START \\ \Rightarrow\ & \{left\} \not\subseteq ref && \text{result from Proof 2.2} \end{aligned}$$

Case: *An "over-filled" buffer*

This case is impossible according to the definition of *BUFFER* and the argument can be proved by contradiction.

Assume	$\#s + 1 = n$	Buffer is already full
Given	$\begin{pmatrix} (tr \neq \langle\rangle) \\ \wedge \quad (last\ (tr \downarrow \{left, right\}) = right) \end{pmatrix}$	definition of *START*
$\Rightarrow$	$(\#s = n) \wedge (tr = tr^+)$	definition of *BUFFER*
$\Rightarrow$	$(\#s + 1 > n) \wedge (tr = tr^+)$	contradicts assumption

This contradicts the definition of *BUFFER* and thus implies that the condition is impossible. And this completes the proof for the property. □

The following two Properties 4.14 and 4.15 define the extended requirements for any bounded buffer. Their validity can be established using a similar case analysis follows by the results from Proof 2.2 and definition of *BUFFER*.

Property 4.14

$$BUFFER \quad \textbf{sat} \quad (tr\ END\ (right,\ left) \wedge (0 \leq \#s + 1 < n)) \Rightarrow \{left, right\} \not\subseteq ref$$

Property 4.15

$$BUFFER \quad \textbf{sat} \quad (tr\ END\ (right,\ left) \wedge (\#s + 1 = n)) \Rightarrow \{right\} \not\subseteq ref$$

Proof of Property 4.14 follows directly from Proof 2.2 under the condition $0 \leq \#s + 1 < n$. *BUFFER* under such conditions behaves as a process composed of two prefix processes using the general choice operator. Whereas the proof of Property 4.15 follows that of the recursive definition of *BUFFER* when the condition $\#s + 1 = n$ holds, which is identical to a simple process with a single prefix communication along the *right* channel. □

To complete the proof of all the basic properties for process *BUFFER*, it is necessary to prove the remaining one property, that is the following Property 4.16.

Property 4.16

$$BUFFER \quad \textbf{sat} \quad ORDER\ (left,\ right,\ n)$$

The form of this proof is determined by the fact that *BUFFER* is defined as a mutual recursive process. Recursive induction is used to show that the specification $S_i(tr)$ where i corresponds to $\#s$, is satisfied by the three definitions of *BUFFER* given by Process 4.5 for $0 \leq i \leq n-1$.

$$BUFFER_i \quad \textbf{sat} \quad S_i(tr)$$
$$\text{where } S_i(tr) \mathrel{\widehat{=}} 0 \leq (tr \downarrow left - tr \downarrow right) + i \leq n$$

The following three results established the property:

1. $left\ ?\ x \rightarrow BUFFER_1$ **sat** $S_0(tr)$

   given $BUFFER_1$ **sat** $S_1(tr)$

2. $(left\ ?\ x \rightarrow BUFFER_{k+1} \ \Box\ right\ !\ y \rightarrow BUFFER_{k-1})$ **sat** $S_k(tr)$

   given $BUFFER_{k+1}$ **sat** $S_{k+1}(tr) \wedge BUFFER_{k-1}$ **sat** $S_{k-1}(tr)$

   where $1 \leq k < n-1$

3. $(right\ !\ x \rightarrow BUFFER_{k-1})$ **sat** $S_k(tr)$

   given $BUFFER_{k-1}$ **sat** $S_{k-1}(tr)$, where $k = n-1$

This completes the set of proofs for *BUFFER*. □

4.4.3.3 Laws

Since *BUFFER* is defined as a mutually recursive communicating process, laws on deterministic choice, recursion and prefixing apply equally well to it.

A special case is the one place buffer in which information being fed from the left channel is directly output from the right channel with a delay of one single step. When the buffer is initialised, its behaviour is equivalent to the generic process *FSM* as described by the following process.

Process 4.9 *BUFFER* (*left*, *right*, 2)

$$BUFFER_1 \mathrel{\widehat{=}} BUFFER_{\langle s \rangle} \tag{4.88}$$
$$BUFFER_{\langle\rangle} \mathrel{\widehat{=}} left\ ?\ x \rightarrow BUFFER_{\langle x \rangle} \tag{4.89}$$
$$BUFFER_{\langle x \rangle} \mathrel{\widehat{=}} right\ !\ x \rightarrow BUFFER_{\langle\rangle} \tag{4.90}$$

combining Equations 4.89 and 4.90:

$$BUFFER_1 = left\ ?\ x \rightarrow right\ !\ x \rightarrow BUFFER_1 \tag{4.91}$$
$$\equiv FSM\ (left.x,\ right.x) \tag{4.92}$$

The equivalence between the one place buffer and *FSM* implies all laws which apply to *FSM* will equally apply to the one place buffer.

4.4.3.4 Formalisation of DARTS Representation and Utilisation

The primary aim of introducing *BUFFER* is to formalise the representation of a bounded data store in structured methods. Process 4.5 can be formally mapped onto the structured representation given in Figure 4.9.



Figure 4.9: An event store formalised using *BUFFER*

Although the *BUFFERing* process is strictly synchronous, an asynchronous transaction between communicating sub-systems can be express by a *buffered* synchronous communications as described by Figure 4.10.

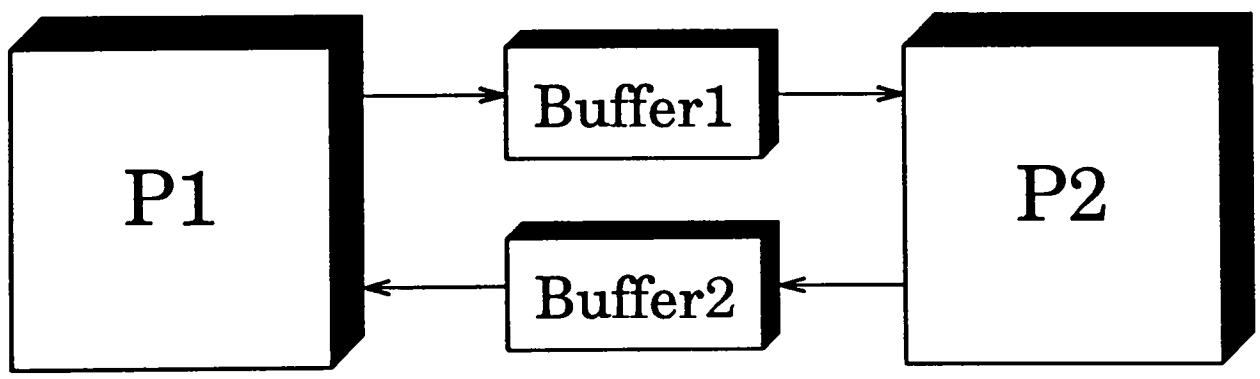


Figure 4.10: A buffered synchronous communication

In this respect, the buffering process is a useful element for *interfacing* between sub-systems where strict synchronisation is not a design constraint. Another possible usage of *BUFFER* is to introduce a fixed delay in an ordered sequence of events. An immediate example is the one place buffer of Process 4.9 where the sequence of events entering the left channel is delayed by exactly one step before they leave the right channel.

Another example of its use is to model the action of a PID controller described in Subsection 4.2.2.2. At the highest level of abstraction, the controller can be realized using the generic

process *FSM* as:

$$PID \;\widehat{=}\; FSM\ (x.e,\ y.u) \tag{4.93}$$

$$\equiv\; \mu X \bullet (x\ ?\ e \rightarrow y\ !\ u \rightarrow X) \tag{4.94}$$

Refinement of Equation 4.93 via Equation 4.20 describing the action of a general PID controller can be realized using the generic processes *FSM* together with a set of one place and two place *BUFFER*s running in parallel. A structured representation of such formalisation is given in Figure 4.11 with the corresponding process algebra given by Equations 4.95 to 4.99. It is seen that sub-process P is modelled using a form of the generic process *FSM* and sub-processes $Q1$ and $Q2$ are a one and two place *BUFFER* with the values of the parameter n equal to 2 and 3 respectively. The proof of correctness of the described realization with respect to its specifications has been carried out by Lau and Daniel [1989].

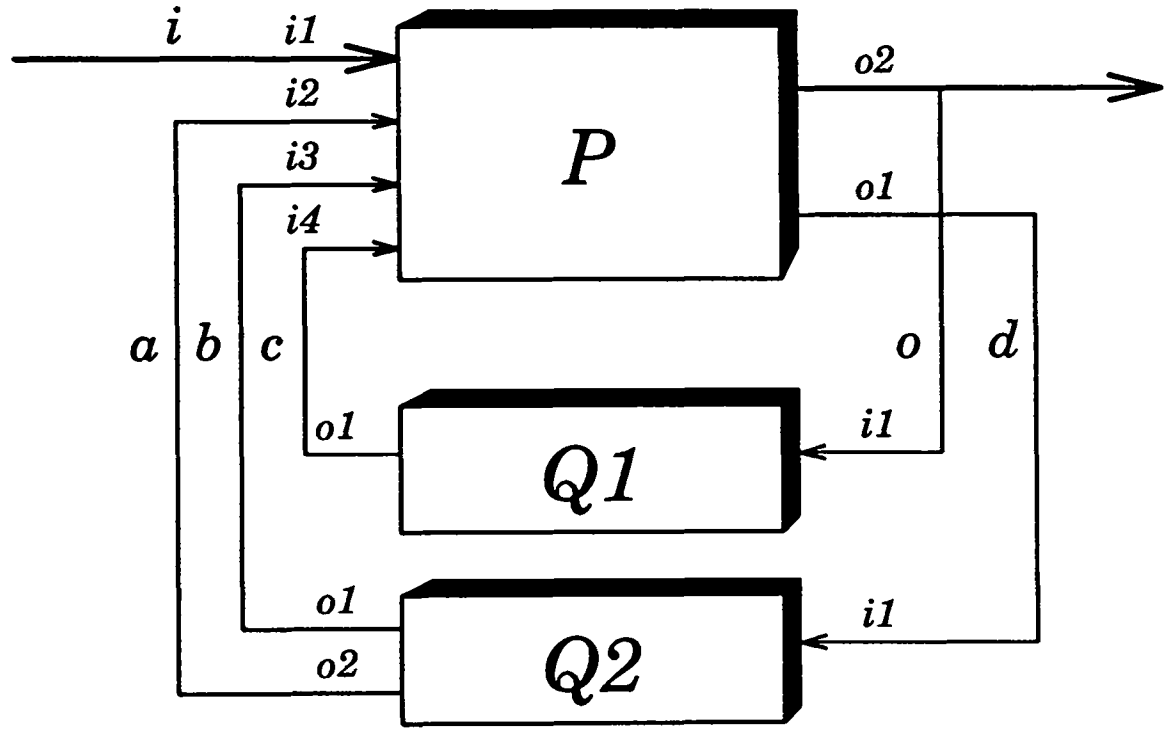


Figure 4.11: A structured model of a formalised PID controller

$$P \;\widehat{=}\; i\ ?\ e_0 \rightarrow a\ ?\ e_{-1} \rightarrow b\ ?\ e_{-2} \rightarrow c\ ?\ u_{-1}$$
$$\rightarrow o\ !\ (e_0 + a_1 e_{-1} + a_2 e_{-2} - b_1 u_{-1}) \rightarrow d\ !\ e_0 \rightarrow P \tag{4.95}$$

$$Q1 \;\widehat{=}\; Q1_{(x_0)} \tag{4.96}$$

$$Q1_{(x)} \;\widehat{=}\; c\ !\ x_0 \rightarrow \mu X \bullet (o\ ?\ y \rightarrow c\ !\ x \rightarrow Q1_{(y)}) \tag{4.97}$$

$$Q2 \;\widehat{=}\; Q2_{(x_0,m_0)} \tag{4.98}$$

$$Q2_{(x,m)} \;\widehat{=}\; a\,!\,m_0 \rightarrow b\,!\,x_0 \rightarrow \mu X \bullet (d\,?\,y \rightarrow a\,!\,m \rightarrow b\,!\,x \rightarrow Q2_{(m,y)}) \tag{4.99}$$

4.4.4 The PERM Family

There are three members of the *PERM* class of generic processes which implement the different forms of alteration. The most basic form is the *PERM* process whose subsequent actions are determined by a choice made by the environment between two distinctive events. According to Process 4.2, the parameters c and d are distinctive events which are designed to synchronise with the environment so that subsequently, either processes P or Q is chosen. By taking different forms of processes P and Q, various generic forms of *PERM* are derived. In this thesis, two possible substitutions are discussed which are most useful in reflecting the behaviour of control systems.

By substituting processes P and Q with terminating processes and specially chosen prefixes as described by Equations 4.100 and 4.101 the process *TPERM* is generated.

$$P \;\widehat{=}\; (d \rightarrow SKIP) \tag{4.100}$$

$$Q \;\widehat{=}\; (c \rightarrow SKIP) \tag{4.101}$$

Similarly, by substituting $(c \rightarrow P)$ and $(d \rightarrow Q)$ with recursive definitions of P' and Q' given by Equations 4.102 and 4.103 generates a recursive *PERM* process and is called *RPERM*.

$$P' \;\widehat{=}\; \mu X \bullet (c \rightarrow d \rightarrow X) \tag{4.102}$$

$$Q' \;\widehat{=}\; \mu X \bullet (d \rightarrow c \rightarrow X) \tag{4.103}$$

4.4.4.1 Specification

The corresponding specifications for the three different forms of *PERM* is derived generically from the basic properties of prefixing and general choice.

For a general guarded process or one with prefixes, its traces and refusals properties are given by Law 4.6 as:

Law 4.6

$$P(x) \quad \textbf{sat} \quad S(tr, ref)$$

$$x : B \rightarrow P(x) \quad \textbf{sat} \quad \left(\begin{array}{ll} & (tr = \langle\rangle \wedge (B \cap ref) = \varnothing) \\ \vee & (tr_0 \in B \wedge S(tr', ref)) \end{array} \right)$$

Combining two guarded processes using the general choice, the overall process will have traces and refusals properties as defined by Law 4.7.

Law 4.7

$$(P \;\Box\; Q) \quad \textbf{sat} \quad S(tr, ref)$$

$$\text{where} \quad tr \in (traces\,(P) \cup traces\,(Q))$$

$$\text{where} \quad ref \in (refusals\,(P) \cap refusals\,(Q))$$

Using the two laws as given, the properties of the three *PERM* processes are composed.

Property 4.17

$$PERM\,(c, P, d, Q) \quad \textbf{sat} \quad \left(\begin{array}{ll} & (tr = \langle\rangle \wedge \{c, d\} \not\subseteq ref) \\ \vee & (tr_0 = \langle c \rangle \wedge tr' \in traces\,(P) \wedge ref \in refusals\,(P)) \\ \vee & (tr_0 = \langle d \rangle \wedge tr' \in traces\,(Q) \wedge ref \in refusals\,(Q)) \end{array} \right)$$

Property 4.18

$$TPERM\,(c, d) \quad \textbf{sat} \quad \left(\begin{array}{ll} & (tr = \langle\rangle \wedge \{c, d\} \not\subseteq ref) \\ \vee & (tr_0 = \langle c \rangle \wedge tr' = \langle d \rangle \wedge c \notin ref) \\ \vee & (tr_0 = \langle d \rangle \wedge tr' = \langle c \rangle \wedge d \notin ref) \end{array} \right)$$

Property 4.19

$$RPERM\,(c, d) \quad \textbf{sat} \quad \left(\begin{array}{ll} & (tr = \langle\rangle \wedge \{c, d\} \not\subseteq ref) \\ \vee & (tr \in (\bigcup_{n \geq 0} \{tr \mid tr \leq \langle c, d \rangle^n\}) \wedge ref \subseteq \{c, d\}) \\ \vee & (tr \in (\bigcup_{n \geq 0} \{tr \mid tr \leq \langle d, c \rangle^n\}) \wedge ref \subseteq \{c, d\}) \end{array} \right)$$

As the *PERM* processes are customised CSP processes with prefixing combined using a general choice, their corresponding specifications are all satisfiable. Detailed proof of satisfaction for Properties 4.17, 4.18 and 4.19 will not be discussed in the context of the thesis and are assumed to be correct, interested readers are referred to Hoare [1985b] Sections 1.8, 1.10, 3.3 and 3.4 for detailed proofs and laws on these processes.

4.4.4.2 Formalisation of DARTS Representation and Utilisation

All the specifications corresponding to the *PERM* processes describe choices between alternative actions which are determined by the environment in which a system resides. The three sets of specifications associated with the three processes describe the different natures of reactions taken after an initial choice is made by the environment. And the associated generic processes are designed to reflect these actions.

Since the abstractions for these processes are controlled by their parameters, in real terms, they could be used to structure a wide range of systems and sub-systems with an *alternative* action which appears in different levels of the control hierarchy. Mappings of this class of generic process is given by Figure 4.12 for a process that is modelled by: *PERM* (*a*, *action*1, *b*, *action*2), which describes a process with two distinct operating modes; and Figure 4.13 for the process: *RPERM* (*A*, *B*) which describes the actions of a bang-bang controller.

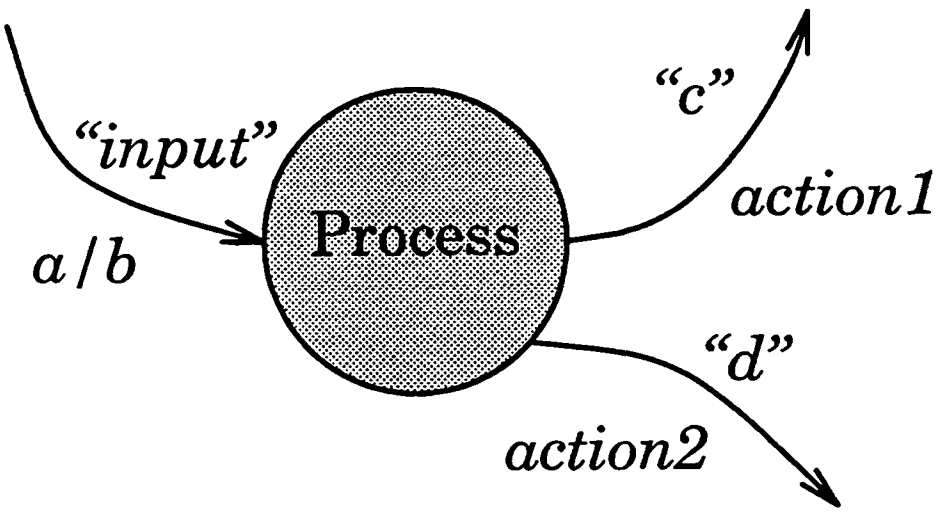


Figure 4.12: A transformation schema mapping of the generic process *PERM* (*a*, *action*1, *b*, *action*2)

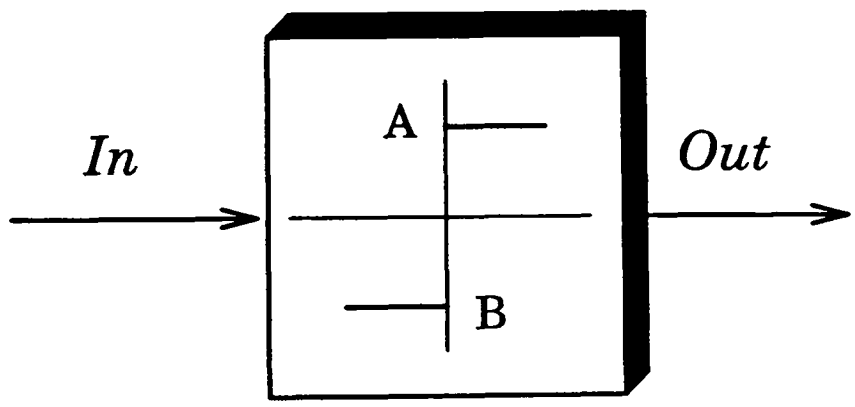


Figure 4.13: A physical process which is a formal mapping of the generic process *RPERM* (*A*, *B*)

4.4.5 MUX

The essential behaviour that the *MUX* process or *multiplexor* model is to perform the reverse of *PERM*, that is to *merge* the behaviour patterns produced by two separate sources into a single one. Two distinct traces of actions are combined in a manner such that the inputs are *toggled* between the two input channels. When messages are propagating along a *MUX*, it behaves as a one-place buffer for successive toggle events.

4.4.5.1 Specification

Apart form the general descriptions, the *MUX* must be able to take input from either of its inputting channel whenever its environment permits. After alternative events have been gathered, they are operated on and output from another channel. *MUX* should also be recursive and the propagation of a message is controlled by synchronous communications. In effect, the general behaviours as given by its specifications suggest that it is a hybrid of the processes *FSM* and *PERM*. Its behaviour is inherited from both of the two processes. A generic realization of this process satisfying the corresponding set of specifications is summarised by Property 4.20.

Property 4.20

$$MUX\ (x.a,\ y.b,\ m.f) \mathrel{\widehat{=}} \tag{4.104}$$

$$\mu X \bullet\ (x\ ?\ a \rightarrow y\ ?\ b \rightarrow m\ !\ f(a,b) \rightarrow X$$

$$\mid y\ ?\ b \rightarrow x\ ?\ a \rightarrow m\ !\ f(a,b) \rightarrow X)$$

$$\textbf{sat} \quad \left(\begin{array}{ll} & ORDER\ (x.a,\ y.b,\ 1) \\ \vee & ORDER\ (y.b,\ x.a,\ 1) \end{array} \right) \tag{4.105}$$

$$\wedge \left(\begin{array}{ll} & ORDER\ (x.a,\ m.f,\ 2) \\ \vee & ORDER\ (y.b,\ m.f,\ 2) \end{array} \right) \tag{4.106}$$

$$\wedge \left(\begin{array}{ll} & ORDER\ (y.b,\ m.f,\ 1) \\ \vee & ORDER\ (x.a,\ m.f,\ 1) \end{array} \right) \tag{4.107}$$

$$\vee\ (tr = \langle\rangle \wedge \{x.a,\ y.b,\} \not\subseteq ref) \tag{4.108}$$

$$\vee \left(\begin{array}{ll} & tr \in (\bigcup_{n \geq 0} \{tr \mid tr \leq \langle x.a,\ y.b,\ m.f \rangle^n\}) \\ \wedge & ref \subseteq \{x.a,\ y.b,\ m.f\} \end{array} \right) \tag{4.109}$$

$$\vee \left(\begin{array}{ll} & tr \in (\bigcup_{n \geq 0} \{tr \mid tr \leq \langle y.b,\ x.a,\ m.f \rangle^n\}) \\ \wedge & ref \subseteq \{y.b,\ x.a,\ m.f\} \end{array} \right) \tag{4.110}$$

$$\wedge \left(\begin{array}{ll} & (m.f)\ IFEND\ (x.a,\ y.b) \\ \vee & (m.f)\ IFEND\ (y.b,\ x.a) \end{array} \right) \tag{4.111}$$

4.4.5.2 Satisfaction of Specification

Assertions 4.105 to 4.107 define the traces properties of MUX and their proofs follow identically to proof 4.2. As MUX is made up of two recursive prefixed processes combined using the general choice, specifications describing the behaviours of the two similar processes are therefore conjointed and are proved separately. The refusals and their associated traces are given by assertions 4.108 to 4.110. They are set out as two different cases. Assertion 4.108 describes the initial situation whereas the rest of the two assertions describe the general case for each of the alternative processes. The correctness of the three assertions depends on the followings:

1. Traces of a triple prefixed recursive processes.
2. The refusals for general choice with prefixed processes given by Law 4.6.
3. Law 4.7 for combining specifications.

Taking one of the recursive processes from MUX, the first result can be proved as follows:

Proof 4.6 *Proof for result 1*

To Prove:

$$\begin{aligned} & traces\ (\mu X \bullet (x\ ?\ a \rightarrow y\ ?\ b \rightarrow m\ !\ f(a,b) \rightarrow X)) \\ = \ & traces\ (MUX_1) \\ = \ & \bigcup_{n \geq 0} \{tr \mid tr \leq \langle x.a,\ y.b,\ m.f \rangle^n\} \end{aligned}$$

Proof:

Define $F(X) = (x\ ?\ a \rightarrow y\ ?\ b \rightarrow m\ !\ f(a,b) \rightarrow X)$

Assume $traces\ (MUX_1) = \bigcup_{n \geq 0} \{tr \mid tr \leq \langle x.a,\ y.b,\ m.f \rangle^n\}$ inductive hyp.

Case: *Initially,* $n = 0$

$$\begin{aligned} traces\,(STOP) &= \{\langle\rangle\} \\ &= \{tr \mid tr \leq \langle x.a,\ y.b,\ m.f\rangle^0\} && t^0 = \langle\rangle \end{aligned}$$

Case: *for general* n

$$traces\,(x\ ?\ a \rightarrow y\ ?\ b \rightarrow m\ !\ f(a,b) \rightarrow F^n(STOP))$$

$$\begin{aligned} &= \{\langle\rangle,\ \langle x.a\rangle\} \cup \{\langle\rangle,\ \langle x.a\rangle,\ \langle x.a,\ y.b\rangle\} \\ &\quad \cup\{\langle x.a,\ y.b,\ m.f\rangle \frown s \mid s \in traces\,(F^n(STOP))\} && \text{prefix traces} \end{aligned}$$

$$\begin{aligned} &= \{\langle\rangle,\ \langle x.a\rangle\} \cup \{\langle\rangle,\ \langle x.a\rangle,\ \langle x.a,\ y.b\rangle\} \\ &\quad \cup\{\langle x.a,\ y.b,\ m.f\rangle \frown s \mid s \leq \langle x.a,\ y.b,\ m.f\rangle^n\} && \text{inductive hypothesis} \end{aligned}$$

$$\begin{aligned} &= \{s \mid s = \langle\rangle \vee s = \langle x.a\rangle \vee s = \langle x.a,\ y.b\rangle \\ &\quad \vee \exists s \bullet tr = \langle x.a,\ y.b,\ m.f\rangle \frown s \wedge tr \leq \langle x.a,\ y.b,\ m.f\rangle^n\} \end{aligned}$$

$$= \{tr \mid tr \leq \langle x.a,\ y.b,\ m.f\rangle^{n+1}\}$$

$$\text{If} \quad traces\,(\mu X \bullet F(X)) = \bigcup_{n\geq 0} traces\,(F^n(STOP)) \qquad \text{by definition}$$

$$\Rightarrow \quad traces\,(MUX_1) = \bigcup_{n\geq 0} \{tr \mid tr \leq \langle x.a,\ y.b,\ m.f\rangle^n\}$$

Using the above result, the traces properties of the other alternative recursive processes composing *MUX* follows directly. With the aid of the two CSP Laws 4.6 and 4.7, *MUX* is proved to satisfy the later three assertions which completes the proof for Property 4.20. □

4.4.5.3 Formalisation of DARTS Representation and Utilisation

According to its specifications, *MUX* combines two system outputs into a single one. It is a formalisation of the transformation as shown in Figure 4.14 with the corresponding generic process: *MUX* $(a,\ b,\ c)$.

The channels corresponding to the different data flow into and out of the transformation with some implicitly defined function such as $f(a, b)$. Due to the orderly nature of its toggling inputs, it

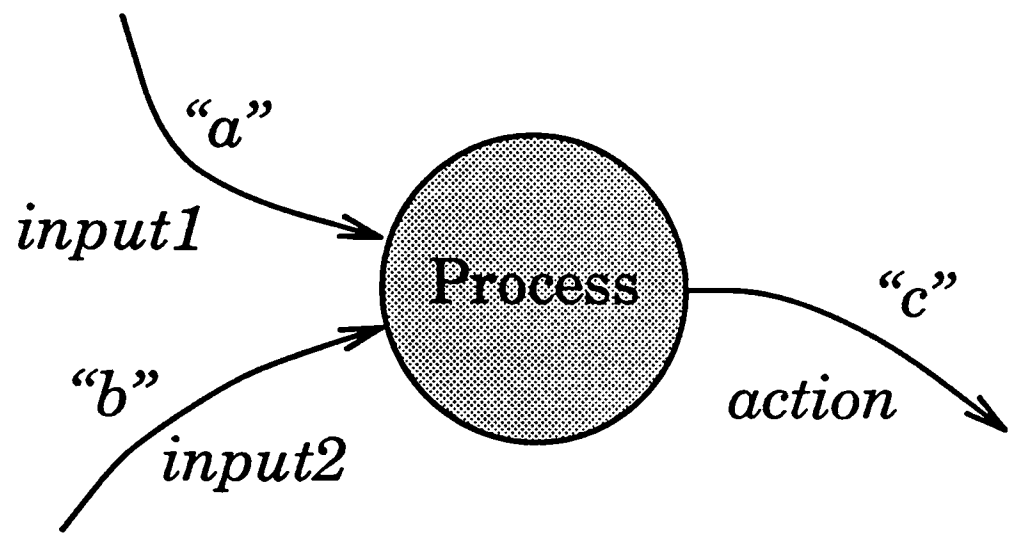


Figure 4.14: Formalisation using *MUX*

is used to synchronise freely interleaving processes into an ordered sequential process. The process has properties of preventing the occurrence of *infinite overtaking* suggested by Hoare [1985b], by a single parallel process by forcing the two processes to toggle. The second property enables shared resources to be controlled and accessed in a mutually exclusive manner so that the situation of deadly embrace as suggested by Dijkstra [1976] can be avoided.

4.5 Tool Set Utilisation: A Unification Issue

Having a primary objective of formalising the development of control systems software along the lines of structured methods and hence the integration of the methods, the set of specification predicates and generic processes is designed to formalise a number of steps in the transformation from an essential model to a more implementation orientated model. According to the structured design and development philosophy discussed in Chapter 3, the transformation from an essential model to the corresponding implementation model involves the structuring and a top-down decomposition of the following classes of element:

- different levels of transformations, pictured as *bubbles*
- data stores pictured as *boxes*
- dynamic data flow and synchronisation pictured as different forms of *arrow*

As such, generic processes can be formally mapped onto all the three classes of element such that not only a formal system description is obtained, an unambiguous and visually assimilable design document can also be generated. To summarise the intended formalisation suggested in the last section with each of the generic processes being mapped onto a structured object of DARTS,

the generic process class: *FSM*, *PERM*, *MUX*, *INIT* and *UPDATE* are mapped onto transformation schema by which the main activities of systems such as operations and functions are described. Whereas the *BUFFER* generic process class is directly mapped onto data stores. Aspects such as dynamic data flow, process synchronisation and interfacing are implicitly taken care of by the mathematical model of CSP on which all the generic processes are based. Synchronous communication between processes are explicitly modelled and the task of interfacing asynchronously between processes is taken care of by the *BUFFER* process as discussed in Sub-section 4.4.3.4.

4.5.1 A Unified Methodology

The proposed methodology is to integrate the application of formal specification and reasoning techniques with structured methods, and hence a unified approach. The following guidelines are laid down by which control systems software can be developed from their requirements, and as a consequence, benefits from both structured and formal approaches.

1. A high level essential model of the system concerned is constructed according to an environmental approach to system definition as described in Sub-section 3.3.1.1. This essential model is a document containing the design requirements about the system in question laying down preliminary ideas of what a designer wants the system to do.

2. By mapping the requirements onto the defined set of specifications predicates given in Section 4.3.4.1, corresponding generic processes in their highly abstracted forms are used to reflect the system behaviours. The initial product is called the *Zeroth Level Model* of a system realization in which highly abstracted generic processes are often used to represent clearly a transformation between inputs and outputs with the environment. In effect, the obtained Zeroth Level Model is already a high level system implementation model which contains information of how abstract actions are to take place.

3. Refinement of the highly abstracted generic processes from the Zeroth Level Model is carried out by applying the method's heuristics — the system structuring criteria suggest by the structured method DARTS. Although as discussed in Chapter 3, there exists two sets of criteria for system refinement: the decomposition and structuring criteria, only one set is necessary. It is seen that with the DARTS system structuring criteria, corresponding system decomposition strategies are used, the resulting refined system having the properties as described in Section 3.4.1. Hence, the unified methodology has adopted the structuring criteria for system refinement. Repeating this approach, higher levels, such as the *First,*

Second, ..., etc. Level Models of system realizations, each encompassing greater detail of the final implementable system, is obtained. Generic processes are used as formal building blocks at different levels of abstraction to implement the defined system requirements, translating a relatively "flat" architecture into a hierarchically organised form; with the level of abstraction controlled via the use of parameterising.

Through this approach of incremental refinement of system definitions, not only an implementation model is gradually being built with increasing complexity, the essential document specifying the system requirements grows which is reflected in the emergence of stronger predicates for system specification. Hence, the incremental philosophy of model transformation is upheld.

4. During each stage of refinement, such as the transformation from the Zeroth to the First Level Model of system realization, a more abstracted system model will be decomposed into a more refined and detailed description which is reflected by the use of less abstracted generic processes. Consistency among the generic processes and the satisfiability of the original essential requirements must be maintained. This is ensured by mathematical reasoning such as formal verification using the established laws on generic processes with CSP operators. A three-stage verification strategy is adopted which aims to build up a proof for the overall system from atomic proof results. Since maximum parallelism is a desirable property for high performance control systems, parallel combination of generic processes is considered when combining a number of processes that are refined from a higher abstracted processes for analysis.

   Theorems 4.1 and 4.2 described in Sub-section 4.5.2 are introduced which are particularly useful for proving properties of systems made up with concurrent processes. At this stage, although maximum parallelism is assumed in the analysis, a system final architecture with a combination of parallel and sequential processes is taken care of by the CSP model which depends on the synchronisation of communication defined in the specification of constituent generic processes during the stage of refinement. Moreover, if a pictorial representation about the evolving system structure and topology is required for the exchange of ideas during the stage of development, formal mappings of generic processes onto structured entities is facilitated via the suggested formal representations of DARTS given in Section 4.4.

5. Should an initial prediction and observation of the behaviours of the highly abstracted system be required, the system which is structured using the generic processes can be interpreted

automatically by the *Causality Diagram Evaluation Tool* described in Chapter 6. The tool is particularly designed to interpret high level CSP specifications and to investigate the traces and refusals properties of communicating processes. Through the use of this tool, design faults and syntactic errors are detected at an early stage of the development so that possible redesigns can be made. The more detailed design and usage of this tool is discussed further in Chapter 6. The above three steps, that are steps 3,4 and 5 concerning system refinement are to be repeated until a systems that is structured and represented in a satisfactory implementation model is achieved.

6. Having obtained an implementation model which describes the organisation embodying the required behaviour as defined by the specifications, a real system is then built. Structured techniques have provided a number of the criteria by which suitable hardware and software can be mapped from the implementation model in the form of tasks, processes and modules. These criteria are formed using the factors described in Section 3.3.2. Although the applicability of formal method does not necessarily stop at this point, formalisation at this stage of the development lies outside the context of this thesis, and will not be further discussed. As a matter of fact, much work on applying formal methods at the hardware level of specification and verification has been carried out by Luk [1988], Cohn [1989], Cullyer [1989] and interested readers are referred to these.

Figure 4.15 illustrates how the two approaches are integrated to a unified methodology.

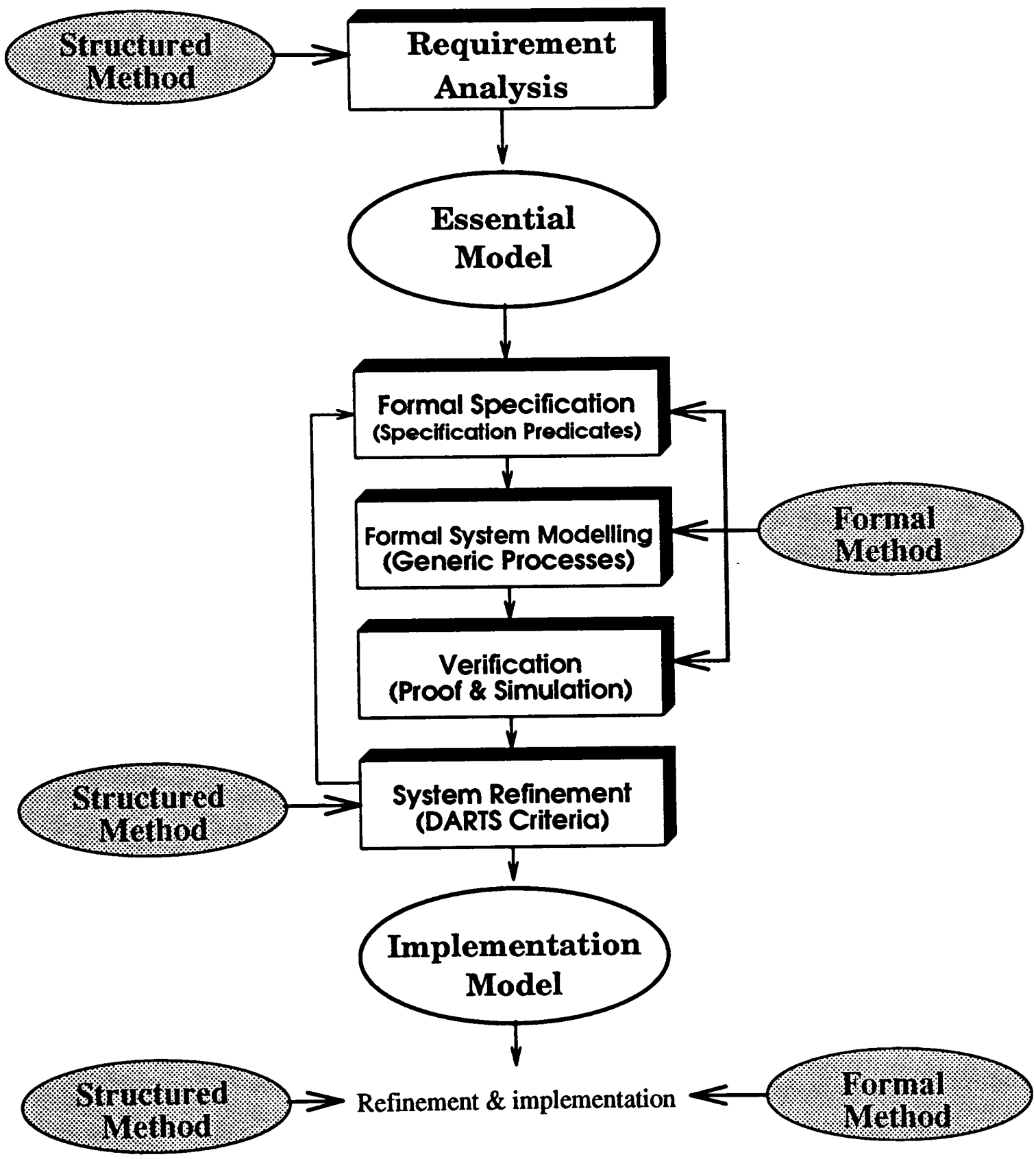


Figure 4.15: A unified approach between formal and structured methods

4.5.2 Two General Proof Rules

To prove that the overall properties of a system composed of generic processes satisfy their properties, the following two theorems due to Jackson [1989] are used to describe the effects of placing processes in parallel.

Theorem 4.1 This theorem corresponds to the intuitive principle that placing a single process which restricts the system traces in a certain manner in parallel with a valid system, the overall system (i.e. valid system and the process) meets the specification, that is if S holds for a set of traces, it must also holds for a subset of traces which is restricted by the alphabet set of the same process.

If the specification of a process P is S, and an alphabet set A such that:

- *S does not constrain the refusal set $\Rightarrow$ ref not free in $S(tr)$.*
- *S holds for a trace exactly when it holds for a restriction to the set A.*

$$\forall tr \mid S(tr) \Leftrightarrow S(tr \upharpoonright A)$$

and the non-diverging process P satisfies the following equation.

$$P \textbf{ sat } S(tr) \wedge A \subseteq \alpha P$$

then for any non-diverging system Q

$$(Q \parallel P) \textbf{ sat } S(tr)$$

Theorem 4.2 As for traces properties, the refusals properties of a system that is composed of a number of parallel processes can be considered separately for each of the component processes. This is related to the intuitive reasoning that the overall system will only refuse an event if at least one of its components will.

For some event e, and non-diverging processes P_1 and P_2 with corresponding specifications $D_1(tr)$ and $D_2(tr)$, are such that for $i \in \{1, 2\}$, the overall system specification $C(tr)$ is given by:

$$(e \in P_i) \Rightarrow \left(\begin{array}{cc} & P_i \textbf{ sat } (D_i(tr) \Rightarrow e \notin ref) \\ \wedge & \forall tr \mid C(tr) \Rightarrow D_i(tr \upharpoonright \alpha P_i) \end{array} \right)$$

then

$$(P_1 \parallel P_2) \textbf{ sat } (C(tr) \Rightarrow e \notin ref)$$

For systems with more than two parallel processes, these two theorems can be applied repeatedly and allowing each component to be considered in turn. The proofs for these two theorems are given in Appendix B.

4.5.3 Remarks

By following the defined methodology, *correctness* of system transformation is enhanced through the philosophy of *"correctness by construction"* which is superior to that followed by structured techniques in general. Despite the existence of rules governing the structuring of transformations in a structured technique, only syntactic errors such as "a bubble has incompatible input and output channels" are spotted, there are no guarantee that a transformation will satisfy the essential model. Using generic processes for structuring a system and reasoning its *satisfies* relations with the original essential requirements at every stage of restructuring and decomposition, system behaviour can be plotted through proofs and simulations. This enables undesirable behaviours such as any logical inconsistency, broken communication or potential deadlock to be predicted and resolved at a high level.

In a pragmatic sense, the application of an abstraction tool set will not be at a level in which design ideas are captured and organised since context diagrams and event lists from structured methods are still the main techniques for these tasks. Nevertheless, a higher quality of software can be obtained in the stages of translating the system requirements to implementation using the specification predicates and generic processes. A number of possible applications of the specification predicates have already been shown in Sub-section 4.3.4.1 and the set of generic processes have already been discussed in Section 4.4; the detailed case study presented in Chapter 5 will demonstrate the proposed philosophy of integrating formal analysis through the applications of abstraction tool set with a structured technique.

4.6 Completeness of Tool Set

Before one can put one's faith in using the abstraction tool set to develop correct and reliable control software, it is necessary to investigate the *completeness* of the tool set. Questions such as whether the proposed set is sufficient to describe all the defined properties of control systems, how well a generic process maps onto a transformation in a structured technique, on what assumptions are the tool set being designed upon, and the completeness of generic processes over the subset of chosen CSP operators have to be considered.

An attempt to answer some of these questions suggests that, based on the assumptions made

in section 4.3.1, the CSP specification predicates and the corresponding generic processes model real world synchronous communication and parallelism. Its practical value is established by the implementation of the concurrent programming language OCCAM [INMOS, 1988a] on Transputers. Generic processes on the other hand is a close formalisation of structured transformations on the grounds of:

- both are building blocks for building up control systems software.
- they are finite state machines with behaviours characterised by their input and output activities.
- data flow is modelled through process communication.
- both represent entities in which their abstraction is controllable. This allows them to be applicable to different hierarchies within control systems.
- they describe behaviours such as: operations, data store, interactions, synchronisation and data flow.

Concerning the completeness of each of the generic process with respect to their specification and with the defined set of operators given by Equation 2.9, the first condition can be reasoned using mathematics such that, if there is an one-to-one correspondence between a process P and the duple (A, S), where A is the alphabet set of P and S is its specification, then it is a sufficient condition to identify the concept of P and (A, S). From the properties of all the defined generic processes described in Section 4.4, there exists a one-to-one correspondence between generic processes and their corresponding (A, S) pair, this is therefore sufficient to conclude the first condition.

For the second condition, in order to prove the completeness of operators on generic processes, it is necessary to show that an operations on two generic processes with a specific operator results in a generic process. Although in this thesis, this condition has not been verified for every generic process, the results from Proofs 4.3 and 4.4 show that the parallel operator is complete on the process FSM. Similar proofs can be constructed for other generic processes using the CSP laws. From these results and deductions, the completeness of generic processes with respect to the subset of CSP operators under appropriate assumptions is proven. These semi-formal reasonings on generic processes provide the basis for their formal applicability.

Nevertheless, the limit of applicability of formal techniques to describe and reason with real systems is still open ended at the moment. The methodology put forward in this thesis is only

a first attempt to bring about a partial solution to the problem. Furthermore, there are no well defined metrics to quantify how well a formal expression matches its physical counterpart, only the correctness with respect to what is being specified can be verified. An example of these questions is the problem of constructing a CSP model for a motor. Although an attempt to the problem is presented in Lau and Daniel [1989], it is nowhere near realistic, a number of real world complexities such as non-linearity and un-predictability are still very difficult tasks to be specified and verified. In the next chapter, a case study on a simplified robot controller software system using the proposed methodology is presented. Although the study only provides a partial solution, it is hoped that through this realistic example, achievements and difficulties of the method can be evaluated.

Chapter 5

A Real-time Robot Control System: Specification

5.1 Introduction

The primary aim of this chapter is to illustrate and to evaluate the unified methodology for the design of a real world control system. A robot system is chosen as the target system with its architecture described in Section 5.2. The starting point of the case study is an environment model of an existing robot system which is characterised by a bunch of informal design criteria and desirable requirements. These requirements are organised into subsets of safety and liveness conditions which are presented in Sub-sections 5.3.1.1 and 5.3.1.2 respectively. Following the methodology, formal specifications are then translated from these conditions and a Zeroth Level Model realisation for this robot system is proposed in Section 5.4. Based on this verifiable formal model, various aspects of application of the methodology are explored. Formalisation using the set of specification predicates and structuring the design with generic processes, verification of properties with the aid of the established laws and theorems, decomposition using DARTS criteria and structuring of decomposed sub-system using the abstraction tool set leads to a refined First Level Model realisation that is presented in Section 5.5.

In the course of the development of the First Level Model, a potential problem with the initial design is found and the task of re-designing the system so as to arrive at some possible solutions is discussed in Sub-section 5.5.4. The discussion of the applicability of the unified methodology carries on with a pointer to the development of Higher Level models through an iterative formal refinement process. A number of suggestions are given in Section 5.6 which aims to provide a way forward to facilitate a full implementation of the overall system. Finally, the chapter concludes with a brief evaluation of the methodology.

5.2 System Definition

The system that this case study addresses is a prototype robot system. The hardware consists of a three degree of freedom direct drive robot developed at the Robotics Research Group at Oxford [Daniel and Irving, 1988]. The robot hardware is interfaced to a VME-bus based Transputer [INMOS, 1988c] network and a PC through a M68000 based input/output sub-system. The overall organisation of the system is pictured in Figure 5.1. This set up was initially constructed to study the various control problems which involve the control of flexible structures using a direct drive mechanism, the design of force control algorithms and the development as well as the implementation of distributed parallel controllers under a truly concurrent hardware platform.

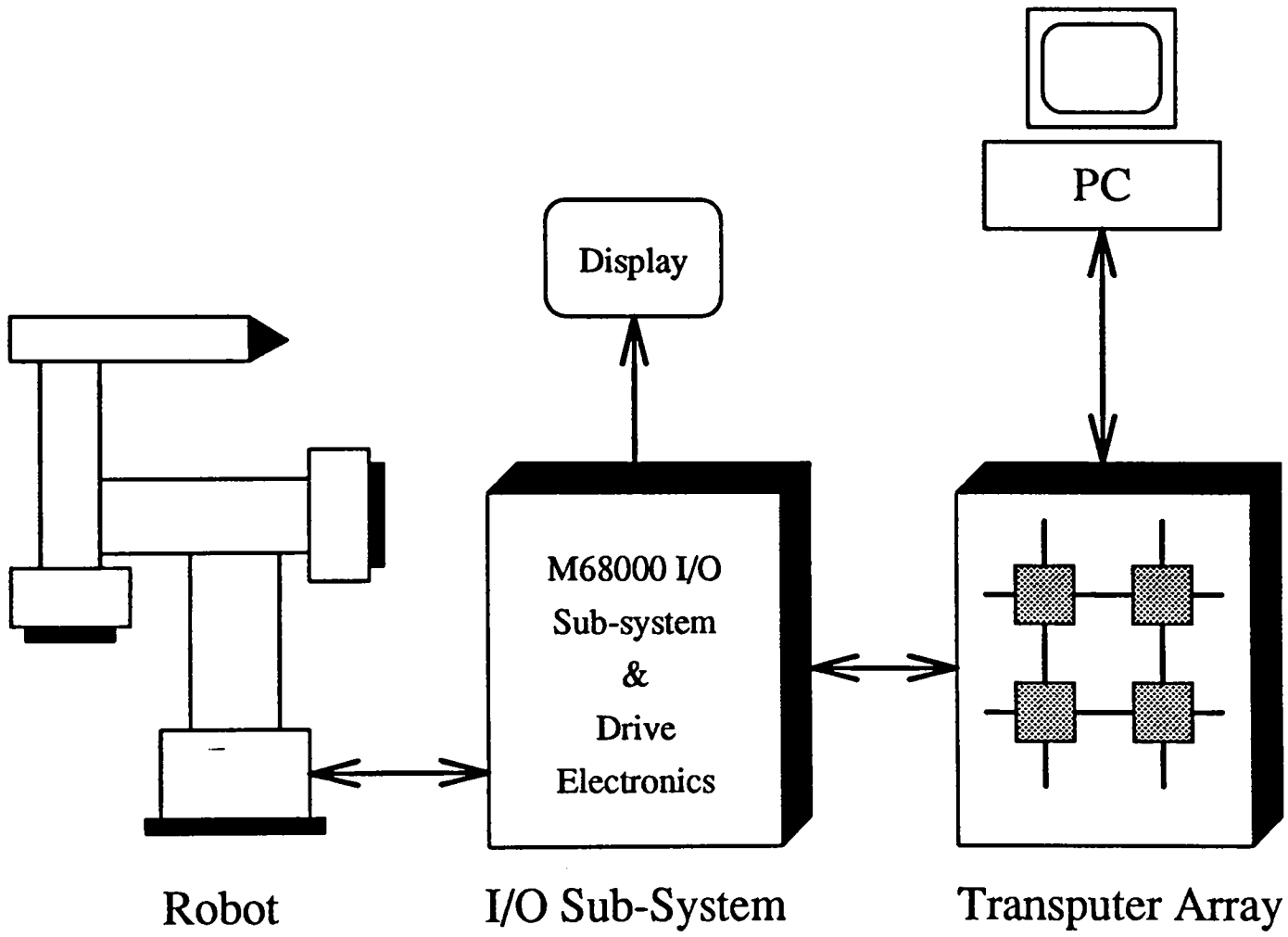


Figure 5.1: The organisation of the Oxford 3-DOF Robot

The described configuration of hardware plus the embedded concurrent force control software form the overall system of this case study. The system interacts with its environment, that is the robot work space and the operator. Such interactions are achieved via accepting commands and signals from the inputing devices such as a trajectory command entered via the PC keyboard by an operator. The output from this system includes the movement of the robot which may affect the configuration of its work space, information displays via output devices such as a VDU display

and hardcopies of robot parameters such as joint velocity that are logged onto a non-volatile media such as a hard disk.

However, in this case study, the application of the proposed methodology is not directed towards the development of control algorithms and robot hardware, but will be focussed onto the design and development of the concurrent distributed software that controls the robot. The reason for such a bias towards the software development is because of the very nature of the methodology which addresses the problem of performing a structured and formal software development as discussed in Chapter 4. Nevertheless, it is vital to emphasis that the qualities of both hardware and software remain equally important for the construction of a sound system and a strict discipline should always be followed when building both components.

5.3 An Environmental Model Design Requirements

Having described the robot system and identified the problem space for this case study, an environmental approach is adopted so as to lay down the essential system requirements. These requirements which form the *essential model* [Mellor and Ward, 1986] provides the basis for software requirement specifications. These requirements should be as full and precise as possible in describing the behaviours of the system so that they restrict the set of possible processes that satisfy them.

In achieving the desirable goals for requirements capturing, the following guidelines are proposed and followed. The answers prompted by the four requirement groups provide the underlying reasons for suggesting particular requirements and conditions.

R–I What can the system offer *initially*? What will be the *first* action or actions to take place once the system is started?

R–II For the set of input commands issued from the environment and not generated from the system itself, *conditions* are specified so that the system must not refuse to accept these commands.

R–III What are the *consequences* of these commands? Such requirements give the conditions for the corresponding *reactions* to be carried out.

R–IV In view of completeness, it is also desirable to specify what *must* follow a particular command and what *may* follow a reaction.

5.3.1 The Requirements

Following the environment modelling technique [Mellor and Ward, 1986], a system boundary is defined for the robot controller, with interactions between the system and its environment occurring solely along the two *channels* as illustrated by Figure 5.2. All systems interactions such as data flow and control signals are generalised as discrete events that enter and leave the system boundary through these two channels.

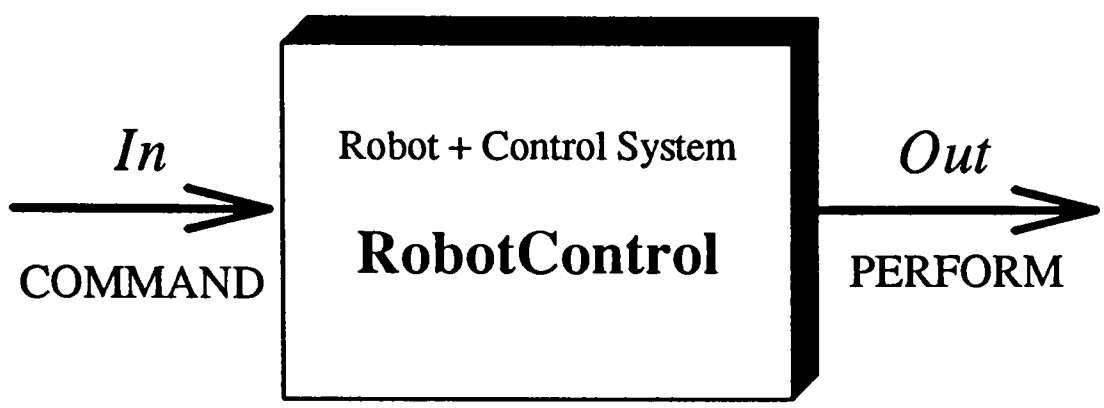


Figure 5.2: An environment model of the robot control system

It is known that the complexity of the system will result in a large number of actions performed by the robot system in respond to a large command set. However, with the aim of illustrating and evaluating the viability of the proposed methodology, a subset of the commands and actions will be considered and modelled. They are chosen to be as representative as possible to characterise specific behaviours, and yet circumscribing adequate information to portray the system in general.

In the study, two events are defined to represent the set of commands input from the *In* channel. The event *action* corresponds to an element from the set of all instructions input by an operator in order to move the robot. A more specific event *stop* is defined to represent a signal to terminate all the actions performed by the robot control system. Formally, the set of inputs are defined according to Equation 5.1 and are called *COMMAND*.

Under similar simplification, a restricted set of responses has been defined which represents the actions performed by the robot system. These events are consequences of *COMMAND* and are output from the channel named *Out*. The set of output events, which are called *PERFORM*, are designated to reflect the reactions undertaken by the robot. *Start* indicates the initialisation of an action, *complete* signifies a completion of a task and *fail* denotes a failure in performing an instruction. All three events are generalisations of the set of possible actions that are performed by the complex system. Equation 5.2 gives the definition for the set of actions.

$$COMMAND \;\widehat{=}\; \alpha In\,(\textsc{RobotControl}) = \{action,\ stop\} \tag{5.1}$$

$$PERFORM \;\widehat{=}\; \alpha Out\,(\textsc{RobotControl}) = \{start,\ complete,\ fail\} \tag{5.2}$$

According to the classification of properties in CSP, system requirements can be grouped into two distinct classes: the *safety* (what a system is allowed to do), and *liveness* (what a system must do) properties. In the following, the requirements from the essential model are listed under these two different classes.

5.3.1.1 Liveness Properties

L–1 All events in *COMMAND* must be acknowledged by the robot system. However, this requirement is implicitly satisfied as a consequence of the other requirements.

L–2 All events in *COMMAND* must respond with events in *PERFORM*, such that *no* unexpected actions should take place.

L–3 All actions in *PERFORM* must be initiated by the event *start*. This is a stronger requirement imposed onto the system for ensuring its liveliness.

L–4 The event *complete* must be the next action issued after a *stop* from the input. This ensures that the robot must be able to stop successfully.

L–5 The robot system must complete its task by issuing a *complete* or signals *fail* once started. However, the choice between the two responses may not be determined by the operator or environment.

L–6 There must be some events from *COMMAND* that result in a *complete* response, indicating a successful completion of a task.

L–7 The particular event *action* in *COMMAND* must not be refused initially or after completion of a designated task. This means that the robot is capable of performing tasks repeatedly.

5.3.1.2 Safety Properties

S–1 Events in *COMMAND* can only occur initially or following actions from *PERFORM*.

S–2 The event *stop* can only occur immediately after the event *action* or an event in *PERFORM*. This requirement is laid down in order to ensure that the robot can be stopped once started.

S–3 Events in *PERFORM* can only be started by the events in *COMMAND*, this requirement constraints the freedom of response made by the robot and makes sure that the robot will not "wander" off unpredictably by itself. This requirement is similar to the liveness condition L–2 but with a different implication.

S–4 A successful performance of a task by the robot is indicated by the sequence of events: *action* and then *start*, directly followed by the event *complete*.

S–5 Parallel to the last requirement, failure to perform an input command as signified by the event *fail* will only occur after the sequence of events: *action* and then *start*.

S–6 A general requirement states that *COMMAND* must be followed by *PERFORM*, specifying the strict sequences of input commands and reactions.

S–7 An auxiliary requirement in order to track the various actions performed by the robot insists that all reactions except *start* in *PERFORM* must be initiated by the event *start*.

5.3.2 Formal Specifications

In the last section, care has been taken to lay down the safety and liveness requirements for the robot control system with respect to the chosen sets of event given by Equation 5.1 and 5.2 such that these requirements describe the desired behaviours of the robot system as completely and unambiguously as possible. They are then translated into specifications using the set of predicates given in Sub-section 4.3.4.1 so as to facilitate formal specification, verification and realisation according to the proposed methodology. These system requirements are first organised under the four main requirement groups suggested in Section 5.3 and are translated into formal specifications. These four groups of formal software requirement specifications form the basis of this design study. They are referred to in the subsequent sections of this thesis as the formal document of the essential system behaviours.

In order to minimise the ambiguity, all the given specifications are constructed according to the following convention:

- Names in capital letters are used to define alphabet sets such as "*COMMAND*".

- Small capitals are used for naming systems and their subsidiaries such as sub-systems, processes; for example "RobotControl".
- Channel names begin with upper case letters, for example the input channel is written as "*In*".
- All events including synchronisation signals and messages are written in lower case letters.

In this case study, since an environmental model for specifications construction is adopted, it is convenient to define a predicate to describe the readiness of the environment to accept an output event from the robot system under certain conditions. The following predicate given by Equation 5.3 specifies this readiness condition of the environment and is used as an abbreviation.

$$\begin{aligned} Ready\ (PERFORM) \ \widehat{=}\ & s\ END\ (complete,\ start) \\ & \Rightarrow ref \cap (PERFORM - \{start\}) = \varnothing \\ & \Rightarrow ref \cap \{fail,\ complete\} = \varnothing \end{aligned} \tag{5.3}$$

5.3.2.1 What is Offered Initially?

The first part of the liveness property L–7 corresponds to the requirement group R–I. In addition, L–7 specifies what actions the system must not refuse to accept after completing a designated task.

L–7

$$\begin{array}{ll} & (\#s = 0 \wedge action\ NOTREF\ (s)) \\ \vee & \left(\begin{array}{l} last\ (s) = complete \\ \quad \Rightarrow (action\ NOTREF\ (s)) \end{array} \right) \end{array}$$

5.3.2.2 Conditions of Acceptance

According to the requirement group R–II, the liveness properties L–3 and L–4 specify these conditions.

L–3

$$\begin{array}{ll} & ORDER\ (action,\ start,\ 1) \\ \wedge & \left(\begin{array}{ll} & \exists s \mid s = u \frown v \\ \wedge & last\ (u \upharpoonright COMMAND) = action \\ \wedge & (v \upharpoonright PERFORM)_0 = start \end{array} \right) \end{array}$$

L–4

$$\left(\begin{array}{ll} & Ready\ (PERFORM) \\ \wedge & s = u \frown \langle stop \rangle \frown v \end{array} \right) \ \Rightarrow \ v_0 = complete$$

5.3.2.3 Conditions to Offer

The requirements for a particular event to occur as a consequence of an event from the set *COMMAND* are captured under the group R–III which consist of the liveness properties L–1, L–2 and L–5 together with the safety property S–6.

L–1/2

$$\begin{aligned} & (s\ START\ (action,\ e) \wedge e \in (PERFORM - \{start\})) \\ \Rightarrow\ & (PERFORM - \{complete\}) \not\subseteq ref \end{aligned}$$

L–5

$$\begin{pmatrix} & Ready\ (PERFORM) \\ \wedge & s = u \frown v \\ \wedge & last\ u = start \end{pmatrix} \Rightarrow \begin{pmatrix} & fail\ NOTREF\ (v) \\ \vee & complete\ NOTREF\ (v) \end{pmatrix}$$

S–6

$$\begin{aligned} ORDER \quad & (e_1,\ e_2,\ n) \\ \text{where} \quad & e_1 \in COMMAND \\ & e_2 \in PERFORM - \{start\} \\ & 1 \leq n \leq 2 \end{aligned}$$

5.3.2.4 Causes and Effects

All but one safety property constitute the group R–IV requirement which specifies the causes and effects of the reactive robot control system.

S–1

$$\begin{aligned} s\ NOTFAIL \quad & (e_1,\ e_2) \\ \text{where} \quad & e_1 \in PERFORM - \{start\} \\ & e_2 \in COMMAND \end{aligned}$$

S–2

$$stop\ IFEND\ (stop,\ start)$$

S–3

$$s = u \frown \langle action \rangle \frown v \Rightarrow \begin{pmatrix} (v_0 \in PERFORM) \\ \wedge \ (last\,(u) \in PERFORM) \end{pmatrix}$$

S–4

$$\begin{pmatrix} ORDER\,(action,\ start,\ 1) \\ \wedge \ ORDER\,(action,\ complete,\ 1) \\ \wedge \ ORDER\,(start,\ complete,\ 1) \end{pmatrix}$$

S–5

$$\begin{pmatrix} ORDER\,(action,\ start,\ 1) \\ \wedge \ ORDER\,(action,\ fail,\ 1) \\ \wedge \ ORDER\,(start,\ fail,\ 1) \end{pmatrix}$$

S–7

$$\begin{pmatrix} complete \ \ IFEND \ \ (complete,\ start) \\ \vee \quad fail \ \ IFEND \ \ (fail,\ start) \end{pmatrix}$$

Going through all the safety and liveness properties of the simplified system, it is seen that all but L–6 are specified formally as CSP specifications. This is because the particular liveness property L–6 cannot be expressed as a behavioural specification as it does not assert conditions about *all* the possible behaviours. As a matter of liveness, it is not desirable to have a process having the potential to be *a priori* unsuccessful.

5.4 Zeroth Level Model

Having obtained the system specification, a top-down development strategy is followed. A *Zeroth Level Model* of the system realisation is designed which is based on these specifications, and is described in the following sub-sections.

5.4.1 Tool Set Specification

At the highest level of abstraction, the system can be realised as a single generic process *FSM* with an input channel *In* for the communication of the set of events defined by *COMMAND*; and an output channel *Out* for *PERFORM*. The corresponding process definition for this highest level realisation is given by Equation 5.4 which corresponds to the pictorial representation given in Figure 5.2.

$$\textsc{RobotControl} \ \widehat{=} \ FSM\,(In.e_1,\ Out.e_2) \tag{5.4}$$

$$\equiv \quad \mu X \bullet (In\ ?\ e_1 \rightarrow Out\ !\ e_2 \rightarrow X)$$

where $e_1 \in COMMAND$

and $e_2 \in PERFORM$

From this general realisation, it is seen that a number of the described safety and liveness requirements are satisfied. The following proof justifications that are based on the properties of *FSM* establish these satisfies relations.

Proof 5.1 *Partial satisfies relations of the Single Process Model*

RobotControl	**sat**	L–7	Property 4.3
		first conjunct of L–3	Property 4.2
		L–1/2	Property 4.3
		2nd and 3rd conjuncts with implication of L–5	Properties 4.3 and 4.4
		S–6	Property 4.2
		S–1	Property 4.3
		S–3	Properties 4.2, 4.3 and 4.4
		1st and 2nd conjuncts of S–4	Property 4.2
		1st and 2nd conjuncts of S–5	Property 4.2

However, it is apparent that such a realisation is not specific enough to satisfy all the formal specifications defined in Sub-section 5.3.2. The realisation is not strong enough to reflect the properties L–3, L–1/2, S–4 and S–5 completely. Also the satisfies relations of properties L–4, S–2, S–7 as well as the environmental predicate *Ready* cannot be deduced using this general model. Therefore, a more refined realisation has to be designed in order to satisfy all the defined specifications. Hence the following design, which is called the *Zeroth Level Model* of the robot control system, by which a complete realisation of all the specifications, thus bearing the full set of safety and liveness properties at its highest abstraction is introduced. Equation 5.5 gives the design, and for clarity the channel names are abstracted. Nevertheless, the correspondence between channels and events as defined by Equations 5.1 and 5.2 still holds.

$$\text{RobotControl}_0 \ \widehat{=} \ FSM_0 \ || \ (\mu X \bullet (start \rightarrow PERM\ (stop, P, c, Q)\ ;\ X)) \tag{5.5}$$

$$FSM_0 \ = \ FSM\ (action,\ start) \tag{5.6}$$

$$P \;\widehat{=}\; stop \rightarrow complete \rightarrow SKIP \tag{5.7}$$

$$Q \;\widehat{=}\; c \rightarrow ((complete \rightarrow SKIP) \sqcap (fail \rightarrow SKIP)) \tag{5.8}$$

In this model, two different classes of generic process are used. Process *FSM* is used to reflect the recursive behaviour of the control software and *PERM* introduces the possibilities of alternative reactions to be taken by the system. The additional event c which is not a member of the external events given by the sets *COMMAND* and *PERFORM*, is introduced to act as a place holder to reflect the non-deterministic behaviour within the *PERM* generic process and is considered as an internal transition which will occur automatically without changing the behaviour of the overall system.

5.4.2 Verification Against Design Requirements

Although it is claimed that the proposed Zeroth Level Model is a correct and complete realisation of the system specifications, it is necessary to establish the remaining satisfies relations of its safety and liveness properties that the original single process model fails to establish. The satisfying of the original model by the Zeroth Level Model is also essential to justify this claim and to put forward that this model is a valid design.

Abstracting the channel names and expanding Equation 5.5 results in the following definition of the Zeroth Level Model.

$$\begin{aligned} \text{RobotControl}_0 \;\widehat{=}\; \mu X \bullet (action \rightarrow start \rightarrow \;& (\; stop \rightarrow complete \rightarrow X \\ & \Box\, c \rightarrow (\; complete \rightarrow X \\ & \sqcap\, fail \rightarrow X)) \end{aligned} \tag{5.9}$$

To consider whether the Zeroth Level Model satisfies the properties of *FSM* such that the *satisfies* relations given in Proof 5.1 hold, the following has to be established:

$$\text{RobotControl}_0 \quad \textbf{sat} \quad \left(\begin{array}{ll} & ORDER\,(In,\ Out,\ 1) \\ \wedge & s\ NOTFAIL\,(e_1,\ e_2) \\ \wedge & s\ NOTFAIL\,(e_2,\ e_1) \end{array} \right) \tag{5.10}$$

$$\text{where} \quad e_1 \in COMMAND$$

$$\text{and} \quad e_2 \in PERFORM$$

Proof 5.2 *Proof that* RobotControl$_0$ **sat** *properties of FSM*

The proof is constructed by considering the three possible recursive processes $P(X)$, $Q(X)$ and $R(X)$ given by Equations 5.11, 5.12 and 5.13 representing the three possible scenarios of the system in its Zeroth Level Model which are identified from Equation 5.9. Here, the channel names as well as the internal transition represented by the event c are abstracted from these definitions and channel communications are implicitly defined by Equations 5.1 and 5.2.

$$P(X) \;\widehat{=}\; action \rightarrow start \rightarrow stop \rightarrow complete \rightarrow X \tag{5.11}$$

$$Q(X) \;\widehat{=}\; action \rightarrow start \rightarrow complete \rightarrow X \tag{5.12}$$

$$R(X) \;\widehat{=}\; action \rightarrow start \rightarrow fail \rightarrow X \tag{5.13}$$

By considering the specifications for each of the three processes in turn, the satisfaction of the properties of *FSM* by ROBOTCONTROL$_0$ is deduced. The specifications for each of the three processes can be written down without proof as these proofs parallel Proofs 4.1 and 4.2 which are given in Sub-section 4.4.1.

Starting with $P(X)$, its specifications are given by:

$$P(X) \quad \textbf{sat} \quad \left(\bigcup_{n \geq 0} \{s \mid s \leq \langle action,\ start,\ stop,\ complete \rangle^n\}\right) \tag{5.14}$$

$$\wedge\, (ORDER\ (action,\ start,\ 1)) \tag{5.15}$$

$$\wedge\, (ORDER\ (action,\ stop,\ 1)) \tag{5.16}$$

$$\wedge\, (ORDER\ (action,\ complete,\ 1)) \tag{5.17}$$

$$\wedge\, (ORDER\ (start,\ stop,\ 1)) \tag{5.18}$$

$$\wedge\, (ORDER\ (start,\ complete,\ 1)) \tag{5.19}$$

$$\wedge\, (ORDER\ (stop,\ complete,\ 1)) \tag{5.20}$$

$$\vee \left(\begin{array}{l} action\ NOTREF\ (s) \wedge n \geq 0 \\ \wedge\, s \leq \langle action,\ start,\ stop,\ complete \rangle^n \end{array} \right) \tag{5.21}$$

$$\vee \left(\begin{array}{l} start\ NOTREF\ (s) \wedge n \geq 0 \\ \wedge\, s \leq \langle action \rangle \frown \langle start,\ stop,\ complete,\ action \rangle^n \end{array} \right) \tag{5.22}$$

$$\vee \left(\begin{array}{l} stop\ NOTREF\ (s) \wedge n \geq 0 \\ \wedge\, s \leq \langle action,\ start \rangle \frown \\ \qquad \langle stop,\ complete,\ action,\ start \rangle^n \end{array} \right) \tag{5.23}$$

$$\vee \left(\begin{array}{l} complete\ NOTREF\ (s) \wedge n \geq 0 \\ \wedge\, s \leq \langle action,\ start,\ stop \rangle \frown \\ \qquad \langle complete,\ action,\ start,\ stop \rangle^n \end{array} \right) \tag{5.24}$$

Considering Assertions 5.15 and 5.20:

$$\left(\begin{array}{lll} & ORDER\,(action,\ start,\ 1) & \Rightarrow\ ORDER\,(e_1,\ e_2,\ 1) \\ \wedge & ORDER\,(stop,\ complete,\ 1) & \Rightarrow\ ORDER\,(e_1,\ e_2,\ 1) \end{array}\right)$$

$$\Rightarrow \quad \left(\begin{array}{ll} & ORDER\,(action,\ start,\ 1) \\ \wedge & ORDER\,(stop,\ complete,\ 1) \end{array}\right) \Rightarrow\ ORDER\,(In,\ Out,\ 1)$$

And for assertions 5.17 together with 5.21 and 5.24:

$$\left(\begin{array}{ll} & action\ NOTREF\ (s) \\ \wedge & n \geq 0 \\ \wedge & s \leq \langle action,\ start,\ stop,\ complete\rangle^n \end{array}\right)$$

$$\Rightarrow \quad s\ NOTFAIL\,(complete,\ action)$$

$$\Rightarrow \quad s\ NOTFAIL\,(e_2,\ e_1)$$

$$\left(\begin{array}{ll} & complete\ NOTREF\ (s) \\ \wedge & n \geq 0 \\ \wedge & s \leq \langle action,\ start,\ stop\rangle \frown \langle complete,\ action,\ start,\ stop\rangle^n \end{array}\right)$$

$$\Rightarrow \quad s\ NOTFAIL\,(action,\ complete)$$

$$\Rightarrow \quad s\ NOTFAIL\,(e_1,\ e_2)$$

where $e_1 \in COMMAND$

and $e_2 \in PERFORM$

Similar argument applies to Assertions 5.22 and 5.23 with identical deductions which concludes that these assertions satisfy the weaker predicates: $s\ NOTFAIL\,(e_2,\ e_1)$ and $s\ NOTFAIL\,(e_1,\ e_2)$ of the generic process FSM. Applying the rule of consequence to the above deductions:

$$\Rightarrow P(X) \quad \textbf{sat} \quad \mathrm{H}\ (tr, ref)$$

$$\text{where} \quad \Pi\ (tr, ref) \mathrel{\hat{=}} \left(\begin{array}{ll} & ORDER\,(In,\ Out,\ 1) \\ \wedge & s\ NOTFAIL\,(e_2,\ e_1) \\ \wedge & s\ NOTFAIL\,(e_1,\ e_2) \end{array}\right)$$

where $e_1 \in COMMAND$

and $e_2 \in PERFORM$

This establishes the satisfies relation $P(X)$ **sat** $\Pi\ (tr,\ rf)$, and hence the satisfying of all the properties of FSM by $P(X)$. □

And for the other two processes: $Q(X)$ and $R(X)$, similar deductions are made with similar arguments as in the case of $P(X)$.

$$\begin{array}{ll} \text{If} & \left(\begin{array}{ll} & P(X) \textbf{ sat } \Pi\ (tr, ref) \\ \wedge & Q(X) \textbf{ sat } \Pi\ (tr, ref) \\ \wedge & R(X) \textbf{ sat } \Pi\ (tr, ref) \end{array}\right) \\ \Rightarrow & \textsc{RobotControl}_0 \textbf{ sat } \Pi\ (tr, ref) \end{array}$$

This completes the proof. □

The set of assertions given by Equations 5.15 to 5.24 which forms part of the specifications for the process $\textsc{RobotControl}_0$, represents a stronger set than the one given by the single process model. On the other hand, the Zeroth Level Model being a more refined model, system liveness and safety conditions that are not satisfied by the single process model are now satisfied.

Satisfies relations that were not previously established are:

$$\textsc{RobotControl}_0 \quad \textbf{sat} \quad \left(\begin{array}{ll} & \mathit{Ready}\ (\mathit{PERFORM}) \\ \wedge & s = u \frown v \\ \wedge & \mathit{last}\ (u \restriction \mathit{COMMAND}) = \mathit{action} \\ \wedge & (v \restriction \mathit{PERFORM})_0 = \mathit{start} \end{array}\right) \qquad \text{L–3}$$

$$\wedge\ (s = u \frown \langle \mathit{stop} \rangle \frown v \Rightarrow v_0 = \mathit{complete}) \qquad \text{L–4}$$

$$\wedge\ \mathit{stop}\ \mathit{IFEND}\ (\mathit{stop},\ \mathit{start}) \qquad \text{S–2}$$

$$\wedge\ \mathit{ORDER}\ (\mathit{start},\ \mathit{complete},\ 1) \qquad \text{S–4c}$$

$$\wedge\ \mathit{ORDER}\ (\mathit{start},\ \mathit{fail},\ 1) \qquad \text{S–5c}$$

$$\wedge \left(\begin{array}{ll} & \mathit{complete}\ \mathit{IFEND}\ (\mathit{complete},\ \mathit{start}) \\ \vee & \mathit{fail}\ \mathit{IFEND}\ (\mathit{fail},\ \mathit{start}) \end{array}\right) \qquad \text{S–7}$$

To justify each of these satisfies relations, proofs have to be constructed that are based on the stronger set of specifications of $\textsc{RobotControl}_0$. The following three specifications: S–2, S–7 and *Ready* can be verified using the corresponding properties of *FSM* and *PERM* together with Theorem 4.1. The proof strategies are outlined in the following proofs:

Proof 5.3 *Proof of* $\textsc{RobotControl}_0$ *satisfying specifications S–2 and S–7*

Consider the Zeroth Level Model of the system in the form given by Equation 5.5, which is written as:

$$\textsc{RobotControl}_0 \ \widehat{=}\ (\mathit{FSM}_0 \parallel \mathit{PERM}_0) \qquad (5.25)$$

$$\text{where} \quad PERM_0 \mathrel{\widehat{=}} (\mu\, X \bullet (start \rightarrow PERM_2\ ;\ X))$$

$$\text{and} \quad PERM_2 \mathrel{\widehat{=}} PERM\ (stop, P, c, Q)$$

To Prove:

$$\text{ROBOTCONTROL}_0 \quad \textbf{sat} \quad S-7$$
$$\Rightarrow \begin{pmatrix} & complete & IFEND & (complete,\ start) \\ \vee & fail & IFEND & (fail,\ start) \end{pmatrix}$$

Proof:

$$PERM_2 \mathrel{\widehat{=}} (\ stop \rightarrow complete \rightarrow SKIP) \qquad [1]$$
$$\Box\ (\ complete \rightarrow SKIP \qquad [2]$$
$$\sqcap\ fail \rightarrow SKIP) \qquad [3]$$

def of $PERM_2$

$\Rightarrow$ $(start \rightarrow PERM_2)$ **sat**
$complete\ IFEND\ (complete,\ start)$ — Property 4.2 and [1,2]

$\Rightarrow$ $(start \rightarrow PERM_2)$ **sat**
$fail\ IFEND\ (fail,\ start)$ — Property 4.2 and [3]

$\Rightarrow$ $(start \rightarrow PERM_2)$ **sat**
$$\begin{pmatrix} & complete\ IFEND\ (complete,\ start) \\ \vee & fail\ IFEND\ (fail,\ start) \end{pmatrix}$$ law of $\sqcap$ and $\Box$

$\Rightarrow$ $PERM$ **sat** $S-7$ — def of S-7 and PERM

$\Rightarrow$ $(FSM \parallel PERM)$ **sat** $S-7$ — Theorem 4.1

$\Rightarrow$ ROBOTCONTROL_0 **sat** $S-7$ — def of ROBOTCONTROL_0

And for specification S–2, the proof follows the strategy of Proof 5.2, that is to consider the three branches of the system separately. Taking the first branch of ROBOTCONTROL_o, $P(X)$ given by Equation 5.11, the proof follows:

To Prove:

$$\text{ROBOTCONTROL}_0 \quad \textbf{sat} \quad S-2$$
$$\Rightarrow \quad stop\ IFEND\ (stop,\ start)$$

Proof:

Define $\mathrm{H}(P) \;\widehat{=}$ as the specification of process P

$P(X)$ **sat** $(stop\ IFEND\ (stop,\ start))$ assertions 5.18 and 5.23

ROBOTCONTROL$_2$ **sat**
$\Pi(P(X)) \vee \Pi(Q(X)) \vee \Pi(R(X))$ laws of $\sqcap$ and $\Box$

$\Rightarrow$ ROBOTCONTROL$_0$ **sat**
$stop\ IFEND\ (stop,\ start)$

This completes the proof. $\Box$

For the case of the environment predicate *Ready*, a possible proof strategy is given as follows:

Proof 5.4 *Proof of* ROBOTCONTROL$_0$ **sat** *Ready* $(PERFORM)$

To Prove:

$$\text{ROBOTCONTROL}_0 \quad \textbf{sat} \quad Ready\ (PERFORM) \Rightarrow \begin{pmatrix} s\ END\ (complete,\ start) \\ \Rightarrow \quad ref \cap \{fail,\ complete\} = \varnothing \end{pmatrix}$$

Proof:

Here, the definition of ROBOTCONTROL$_0$ as given by Equation 5.25 is considered which describes the overall system as a parallel composition of processes FSM_0 and $PERM_0$.

$PERM_0$ **sat** $(s\ END(complete,\ start) \Rightarrow$
$\{fail,\ complete\} \not\subseteq ref$ def of $PERM_0$
$\{complete,\ fail\} \not\subseteq \alpha FSM_0$ def of FSM_0

$\Rightarrow$ $\{complete,\ fail\} \notin refusals\ (FSM_0)$ property of refusals
$\Rightarrow$ $(FSM_0 \parallel PERM_0)$ **sat**
$$\begin{pmatrix} s\ END\ (complete,\ start) \\ \Rightarrow \quad \{fail,\ complete\} \not\subseteq ref \end{pmatrix}$$ Theorem 4.2

$\Rightarrow$ ROBOTCONTROL$_0$ **sat** *Ready* $(PERFORM)$ def of ROBOTCONTROL$_0$ and the *Ready* predicate

Finally, for specifications L–3, S–4c and S–5c, the justifications for the satisfies relations are based on the various assertions defining the *ORDER* properties of the three processes $P(X)$, $Q(X)$ and $R(X)$ given by Equations 5.11 to 5.13, and then follows by an inference on a system with choices.

Proof 5.5 *Proof of* ROBOTCONTROL$_0$ **sat** *L–3, S–4c and S–5c*

For specification S–4c and S–5c, the relevant proof strategies are:

	$Q(X)$ **sat** *ORDER* (*start*, *complete*, 1)	definition of $Q(X)$
$\Rightarrow$	ROBOTCONTROL$_2$ **sat** $S-4c$	laws of $\Box$ and $\sqcap$
	$R(X)$ **sat** *ORDER* (*fail*, *complete*, 1)	definition of $R(X)$
$\Rightarrow$	ROBOTCONTROL$_2$ **sat** $S-5c$	laws of $\Box$ and $\sqcap$

And for S–3 all *ORDER* predicates from $P(X)$, $Q(X)$ and $R(X)$ are used.

Summarising the various deductions from Proof 5.2 which implies the results of Proofs 5.1 and from Proofs 5.3, 5.4 and 5.5, the satisfies relations between all the essential system requirements specifications and the Zeroth Level Model are proved. $\Box$

5.4.3 Deduction

From the expanded process definition of the Zeroth Level Model given by Equation 5.9, various salient system behaviours are captured. According to the proof results given in the previous section, it is seen that the Zeroth Level Model satisfies the set of essential system specifications. This results in this model being a complete realisation of the defined specifications. Although the more general single process model has already captured a number of these system behaviours, its specifications are weaker than those from the Zeroth Level Model.

At this level of abstraction, the process definition of the system reflects its *behaviour* pattern. The overall realisation is recursive by which inputs in the form of discrete events follow discrete outputs. The deterministic choice between the two processes P and Q insists that the environment, in this case the operator, has control over the actions of the system. Whereas the non-deterministic behaviour, that is the set formed by the two sub-processes of Q, indicates that *internal* transitions of the robot system are not affected by the environment. This complies with the intuitive thoughts that the internal progression of the robot software are solely determined by the different parameters local to the hardware platform which are not controllable by any events form the environment.

5.5 First Level Model

The previous sections have discussed in detail the initial application of the unified methodology in order to obtain a highly abstracted model of the robot control system which is based on the full set of behaviour specifications from the essential model. In this section, the aspects of system

refinement and further applications of the abstraction tool set for formal system structuring are illustrated. The intention is to demonstrate the applicability of the methodology on the grounds of: more involved refinements, transformation from a behavioural system description into a more functional description, formal reasoning on parallel processes and prediction of design problems resulting from the defined system architecture.

As such, going on from the Zeroth Level Model, the focus of system development is narrowed down onto considering of the control software architecture rather than the robot itself. The set of essential system specifications which describe the system behaviours are invariant and through system decomposition following the DARTS criteria given in Section 3.4.3, a *First Level Model* is obtained.

5.5.1 System Decomposition

To focus onto the problem space, the system as pictured by Figure 5.2 is logically decomposed into two separate elements: the robot hardware and the control software. The First Level Model which is a partial refinement of the Zeroth Level Model specifies the software architecture of the controller that is run on an array of Transputers and the VME-based sub-system. The first stage of the refinement is illustrated by Figure 5.3. In this figure, the control system CONTROL is defined as a system that inherits *all* the properties of the essential model and interacts with its environment through *three* communication channels. A system boundary is defined in which robot hardware has now become the environment of the system of interest.

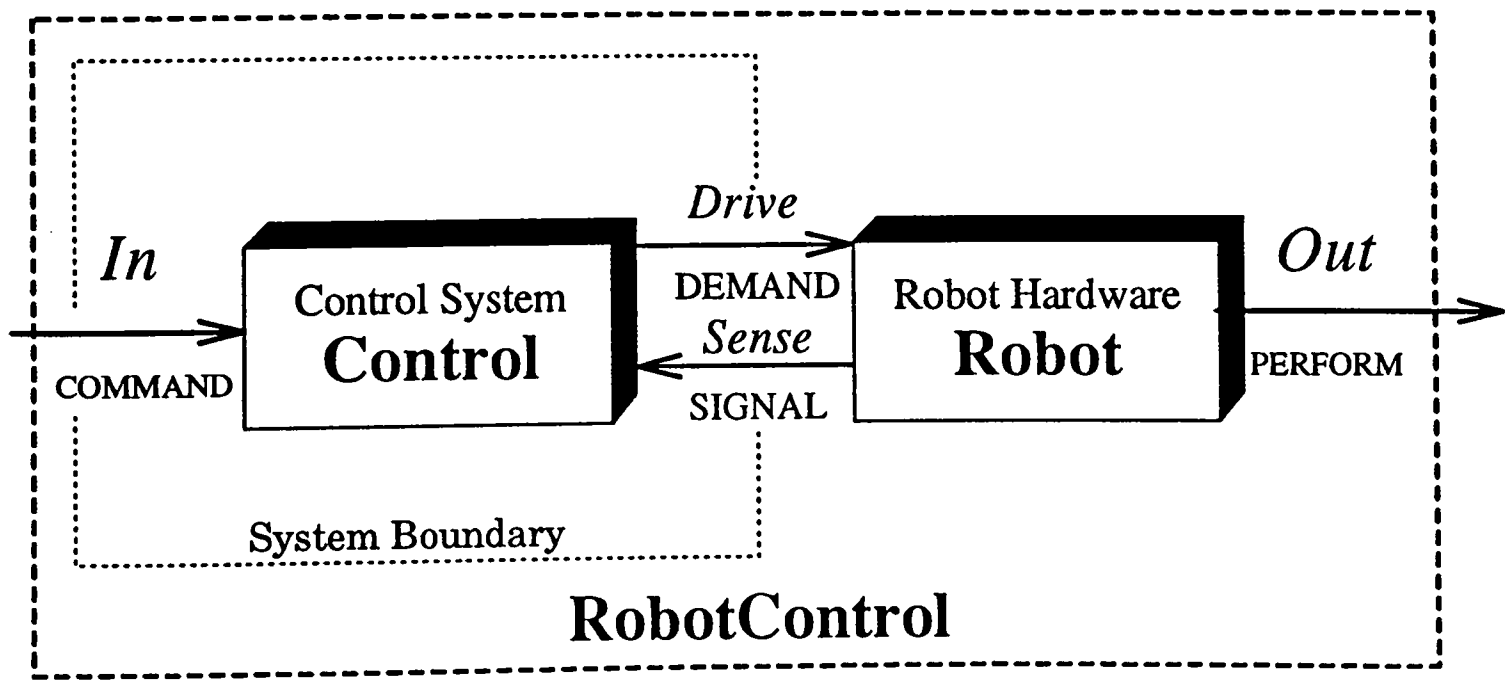


Figure 5.3: Decomposition of the robot system

The *In* channel is identical to the one defined in the Zeroth Level Model which takes inputs

within the set *COMMAND* from the operator. Two other channels, namely *Drive* and *Sense* are introduced to interface with the robot. Channel *Drive* sends *DEMAND*, which is the set of valid robot commands whereas *Sense* channels in the various robot parameters such as joint data from the input/output sub-system in the form of discrete message packages that are summarised by the set called *SIGNAL*. The relationships between these two channels and their corresponding alphabet sets are formally defined by Equations 5.26 and 5.27.

$$SIGNAL \;\widehat{=}\; \alpha Sense\ (\textsc{Control}) \tag{5.26}$$

$$DEMAND \;\widehat{=}\; \alpha Drive\ (\textsc{Control}) \tag{5.27}$$

It is noted that the set of events *DEMAND* by which the control actions are represented, is a subset of *PERFORM* which reflects the actions of the robot system. In particular, the event *start* which is an element of *PERFORM*, is also a member of the set *DEMAND*. These relationships are given by Equations 5.28 and 5.29.

$$DEMAND \subset PERFORM \tag{5.28}$$

$$start \in DEMAND \tag{5.29}$$

The next stage of the design is to apply the set of DARTS criteria given in Section 3.4.3 to decompose the control system — CONTROL into sub-systems. Since the idea of a *jacketed* system as described in Sub-section 4.2.1 is adopted, the decomposition should explicitly reflect operations such as data logging, trajectory calculations, control and interfacing with the environment. As a result, three functional groups of sub-system are identified according to the following criteria.

1. Intense interaction with the environment such as in the cases of input/output devices. Their identification follows to the DARTS criteria: DC–1, & DC–2.

2. Functions with a high computation and communication overheads. Their identification follows to the DARTS criteria: DC–3 & DC–5.

3. Functional cohesion which corresponding to DARTS criteria: DC–4.

According to these criteria, five sub-systems have been identified, with each of them encompassing a set of more specific functionalities. Altogether they form the First Level Model which reflects the *steady state* behaviours (in which initialisation is assumed to have completed successfully) as described by the essential requirements as well as more detailed behaviours given by the jacketed system. These five sub-systems are TRAJECTORY GENERATOR, DATA LOGGER, CONTROLLER, OPERATOR INTERFACE and DATA PROCESSOR. In the following sections, each of their functionalities are described together with their set of design requirements.

5.5.1.1 Trajectory Generator

The TRAJECTORY GENERATOR is decomposed from the Zeroth Level Model under the functional cohesion criteria. As the name of this sub-system suggests, its primary function is to calculate the appropriate set of control parameters for the CONTROLLER so that the robot will act accordingly. More specifically, the TRAJECTORY GENERATOR must operate with the following requirements.

TG–1 The TRAJECTORY GENERATOR generates a new set of control parameters for a specific trajectory upon receiving a validated operator input command.

TG–2 On the other hand, if no validated input is received, control parameters are calculated from the previous logged robot data which provides information about the current status of the robot.

TG–3 All the output from the TRAJECTORY GENERATOR are fed directly to the CONTROLLER so that the robot is controlled to perform the desired trajectories.

TG–4 Newly formed control parameters as a result of validated operator's input must be logged by the DATA LOGGER so as to facilitate subsequent analysis on grounds of performance and subsequent tuning of various controller parameters.

TG–5 The operation of the TRAJECTORY GENERATOR is recursive.

5.5.1.2 Data Logger

The DATA LOGGER is a sub-system decomposed from the Zeroth Level Model under the concept of a jacketed system. Although for the robot control system, the main issue is to control the robot to perform some desirable tasks, the ability to capture the robot data so that its performance can be assessed is equally important. The sub-system DATA LOGGER is designed to achieve this. Since this sub-system has a completely different functionality to CONTROLLER, under similar criteria as in the case of the TRAJECTORY GENERATOR, it is identified as a separate sub-system by the functional cohesion criterion. The DATA LOGGER in a simple sense, represents a set of functions that manage the task of data storage. The following list gives the specific requirements that the DATA LOGGER must follow.

DL–1 The main function of the DATA LOGGER is a data store which captures the sets of system parameters defined by the designer.

DL–2 More specifically, it receives current robot data from the DATA PROCESSOR and/or the newly formed set of control parameters from the TRAJECTORY GENERATOR.

DL–3 As a data manager, it works closely with the TRAJECTORY GENERATOR. Only in the case if a new set of control parameters has not been received as a result of no validated input commands, the most recently logged set of robot data is passed to the TRAJECTORY GENERATOR.

DL–4 Initially, if no data is being logged, the sub-system must not output any stray data to the TRAJECTORY GENERATOR.

DL–5 The sub-system is recursive and even in the case when valid control parameters from TRAJECTORY GENERATOR is being logged, the robot data must also be logged for every control cycle.

5.5.1.3 Controller

The CONTROLLER is the main body of the robot control system. Its operations are precisely defined by a set of control algorithms which are designed according to various factors such as the robot hardware and performance issues. However, at the level of abstraction defined by the First Level Model, the exact control algorithm is abstracted from the analysis. Moreover, the study of the design of various control algorithms for flexible robot systems is not the subject of this thesis.

Apart from the fact that the overall system is a control system, the identification of the CONTROLLER sub-system is a result of the DARTS criteria concerning sub-systems with a high computation and communication overheads. This sub-system implements a group of algorithms that are both computationally intensive and with a high communication overhead between computation modules, an example is the inverse dynamics controller [Paul, 1981]. The requirements for the CONTROLLER sub-system are:

CTL–1 CONTROLLER receives control parameters from the TRAJECTORY GENERATOR and current robot data form the DATA PROCESSOR.

CTL–2 Corresponding control signals, *DEMAND*, which drives the robot are computed from the inputs according to the embedded algorithms.

CTL–3 Control signals will only be calculated for every update of control parameters and robot data.

CTL–4 Its operation is recursive in which control signals are continuously computed after receiving each successive sets of inputs, independent of the order of receipt.

5.5.1.4 Operator Interface

For the system CONTROL, there are three external channels conveying messages between the system and its environment, it is therefore desirable to have a dedicated functional unit to cater for these interactions. One is the sub-system OPERATOR INTERFACE. Its identification has arisen from the first criteria described in Sub-section 5.5.1. Although the functions of the OPERATOR INTERFACE as characterised by the requirements below is relatively simple, it is this sub-system that interfaces between the environment — the operator, and the control system.

OI–1 It responses to the set of elements from *COMMAND* issued by the environment and interprets them to produce validated output commands set, *VCOMMAND* to the TRAJECTORY GENERATOR.

OI–2 Every input command form the operator must follow an interpreted output command.

OI–3 This sub-system is recursive and is always ready to receive external commands.

5.5.1.5 Data Processor

Under similar criteria as in the case of OPERATOR INTERFACE, the DATA PROCESSOR is identified as a sub-system which deals with robot data such as the joint positions and joint velocities of the robot. This sub-system captures the various data from the robot system, organises the data and then distributes them to a number of sub-systems described in the previous sections. Having similar properties as the OPERATOR INTERFACE, the DATA PROCESSOR is an interface between the environment and the control system. The various requirements for this sub-system are listed as follows:

DP–1 The sub-system is an interface between the set of events given by *SIGNAL* and two other sub-systems, namely the DATA LOGGER and the CONTROLLER.

DP–2 The robot data captured by DATA PROCESSOR are logged by the DATA LOGGER and fed to the CONTROLLER. The order of data distribution between the two

sub-systems is arbitrary.

DP–3 Only data captured from the robot system will be distributed to the various sub-systems. Before data is distributed, it must be captured by the DATA PROCESSOR.

DP–4 The overall sub-system is recursive in which the synchronisation is determined by the external events.

5.5.2 Tool Set Specification

Following the identification of sub-systems, according to the methodology, the next phase of system development is to express the behaviours of each of the sub-systems formally and to suggest corresponding implementations of their functionalities using generic processes. This involves the translation of the requirements into the specification predicates. These predicates are then matched with the sets of specification predicates which characterise the generic processes such that formal realisations with generic processes of these sub-systems are obtained.

At the level of abstraction defined by the First Level Model, the primary design issue is centred onto obtaining a correct system realisation on the grounds of the topology of sub-systems which is dependent on their communication patterns and parallel interactions. These considerations are reflected by the specifications of the sub-systems and their realisation with generic processes. Because of this emphasis, only interactions in the form of channel communication with other sub-systems are specified, whereas their internal operations and data dependencies are abstracted from this model. Moreover, these sub-systems are structured concurrently in which maximum parallelism is assumed and studied. The following sections discuss the use of the abstraction tool set to specify each of these sub-systems. The convention for distinguishing between alphabet sets, channel names and processes follows strictly to those set out in Sub-section 5.3.2.

Before each sub-system is specified, three sets of external events are defined that are common to all the sub-systems, given by Equations 5.30, 5.31 and 5.32, and a list of abbreviations for sub-system nomenclature are given in Table 5.1 so that more concise specifications are generated.

$$COMMAND \mathrel{\widehat{=}} \alpha Opin\ (\text{CONTROL}) \tag{5.30}$$

$$SIGNAL \mathrel{\widehat{=}} \alpha Sense\ (\text{CONTROL}) \tag{5.31}$$

$$DEMAND \mathrel{\widehat{=}} \alpha Drive\ (\text{CONTROL}) \tag{5.32}$$

Sub-system	Abbreviations
TRAJECTORY GENERATOR	TRAJGEN
DATA LOGGER	DATALOG
CONTROLLER	CTL
OPERATOR INTERFACE	OPINT
DATA PROCESSOR	DATAPROC

Table 5.1: List of abbreviations of sub-systems that are used in writing specifications

5.5.2.1 Trajectory Generator: Formal Specification

From the requirements given in Sub-section 5.5.1.1, the TRAJECTORY GENERATOR interacts with the sub-system OPERATOR INTERFACE, CONTROLLER and DATA LOGGER. In order to facilitate the modelling of the interactions between the TRAJECTORY GENERATOR and other sub-systems, channels names with their corresponding events set are given in Table 5.2 with the relevant justifications from the requirements.

Interacting Sub-systems	**Channel name**	**Alphabet set**	**Justification**
TRAJGEN & OPINT	Opout	VCOMMAND	TG–1
DATALOG & TRAJGEN	Logdata	LDATA	TG–2
TRAJGEN & CTL	Traj	TRAJ	TG–3
TRAJGEN & DATALOG	Trajdata	TDATA	TG–4

Table 5.2: Formal parameters for specifying TRAJECTORY GENERATOR

According to the requirements, the following specifications are written using the described formal parameters. Analysing these requirements, all implicitly define the safety properties about the sub-system which can be formalised using the *ORDER* predicate. Also, from requirements TG–1 and TG–2, an external choice is made to either communicate with OPINT or DATALOG using the corresponding channels. These specifications, $\Pi_{TG}[i]\ (tr,\ rf)$, are summarised as follows:

$$\begin{pmatrix} \quad ORDER\,(Opout, Traj, 1) \\ \vee \quad ORDER\,(Logdata, Traj, 1) \end{pmatrix} \qquad \text{by TG–1, 2 and 3: } \Pi_{TG}[1]$$

$$ORDER\,(Opout, Trajdata, 1) \qquad \text{by TG–4: } \Pi_{TG}[2]$$

$$s = \langle\rangle \wedge \{e_1, e_2\} \not\subseteq ref \qquad \text{by TG–1 and 2: } \Pi_{TG}[3]$$

$$\begin{pmatrix} \quad s_0 = \langle Opout.e_1 \rangle \wedge \begin{pmatrix} \quad Trajdata.e_3 \notin ref \\ \vee\ Traj.e_4 \notin ref \end{pmatrix} \\ \vee \quad s_0 = \langle Logdata.e_2 \rangle \wedge Traj.e_4 \notin ref \end{pmatrix} \qquad \text{by TG–1 and 2: } \Pi_{TG}[4]$$

$$\text{where} \quad \begin{pmatrix} \quad e_1 \in VCOMMAND \\ \wedge \quad e_2 \in LDATA \\ \wedge \quad e_3 \in TDATA \\ \wedge \quad e_4 \in TRAJ \end{pmatrix}$$

To realise the set of specifications describing the actions of TRAJGEN, a recursive form of generic process *PERM* is proposed. The choice of this particular class of generic process is justified by the satisfying of Property 4.17 of the process *PERM* by all the predicates: $\Pi_{TG}[1]$ to $\Pi_{TG}[4]$ with the extra constraints given by these predicates defining the parameters of *PERM*. The realisation of the sub-systems TRAJGEN is given by Equation 5.33.

$$\text{TRAJGEN} \;\widehat{=}\; \mu X \bullet (PERM\ (Opout.e_1,\ P,\ Logdata.e_2,\ Q)\ ;\ X) \tag{5.33}$$

$$P \;\widehat{=}\; Traj\ !\ e_4 \rightarrow Trajdata\ !\ e_3 \rightarrow SKIP \tag{5.34}$$

$$Q \;\widehat{=}\; Traj\ !\ e_4 \rightarrow SKIP \tag{5.35}$$

where e_1 to e_4 are defined in the specifications of TRAJGEN

The recursive form of *PERM* has a unique solution as *PERM* is itself a guarded process and a pictorial representation of the sub-system is given in Figure 5.4.

5.5.2.2 Data Logger: Formal Specification

In Sub-section 5.5.1.2, the requirements for the sub-system DATA LOGGER require that sub-systems TRAJECTORY GENERATOR and DATA PROCESSOR interact with it. Following the analysis from Sub-section 5.5.2.1, Table 5.3 gives the various abbreviations for channel names and alphabet sets for constructing the specifications.

According to the requirements, the following specifications are written using the abbreviated

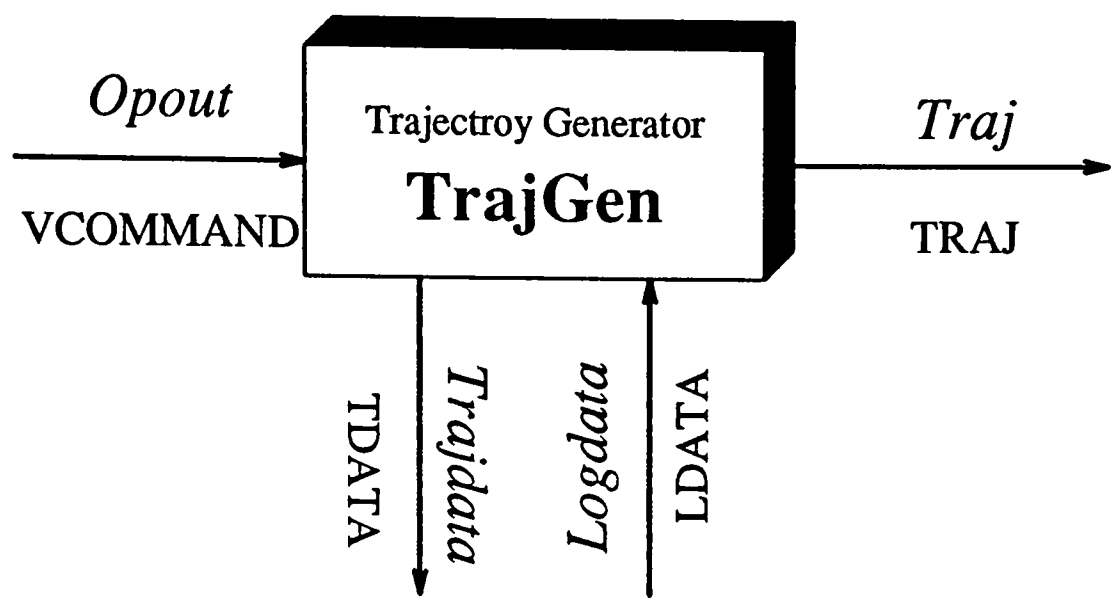


Figure 5.4: A pictorial view of TRAJGEN

Interacting Sub-systems	Channel name	Alphabet set	Justification
DATALOG & TRAJGEN	Logdata	LDATA	DL–2, 3
TRAJGEN & DATALOG	Trajdata	TDATA	DL–2, 3
DATAPROC & DATALOG	Ddata	DDATA	DL–2

Table 5.3: Formal parameters for specifying DATA LOGGER

parameters. By analysing the requirements, it is seen that the communication between the sub-systems DATALOG and TRAJGEN, that is the flow of information *from* DATALOG to TRAJGEN is constrained by the following requirements.

C–1 From requirement DL–4, the communication from DATALOG to TRAJGEN must not occur initially.

C–2 Only if a communication between DATAPROC and DATALOG happens more recently than a communication from DATALOG to TRAJGEN, communication from DATALOG to TRAJGEN is allowed. This requirement is filtered from requirement DL–3.

Taking into account the above requirement, the specifications for the sub-system, $\Pi_{DL}[i]\ (tr, ref)$, written as functions of traces and refusals are summarised as follows:

$$\left(\begin{array}{ll} & ORDER\,(Trajgen.e_1,\ Logdata.e_2,\ n) \\ \vee & ORDER\,(Ddata.e_3,\ Logdata.e_2,\ n) \\ \vee & ORDER\,(Ddata.e_3,\ Logdata.e_2,\ 1) \end{array}\right) \qquad \text{by DL–1 and 3: } \Pi_{DL}[1]$$

$$s = \langle\rangle \wedge \{Ddata.e_3,\ Trajdata.e_1\} \not\subseteq ref \qquad \text{by DL–2/4 \& C–1: } \Pi_{DL}[2]$$

$$\left(\begin{array}{ll} & ORDER\,(Ddata.e_3,\ Logdata.e_2,\ 1) \\ \vee & ORDER\,(Trajdata.e_1,\ Ddata.e_3, 1) \end{array}\right) \qquad \text{by DL–3/5 \& C–2: } \Pi_{DL}[3]$$

$$Logdata.e_2\ IFEND\ (Trajdata.e_1,\ Ddata.e_3) \qquad \text{by DL–3: } \Pi_{DL}[4]$$

$$\text{where} \quad \left(\begin{array}{ll} & e_1 \in TDATA \\ \wedge & e_2 \in LDATA \\ \wedge & e_3 \in DDATA \end{array}\right)$$

In designing possible generic processes to realise the set of specifications, a simplification has been made. Since the emphasis at this level of design is to focus onto process communication, internal functionality such as the *buffering* action given by DL–1 is abstracted, although a possible generic process realisation for predicate Π_{DL} [1] is given by Equation 5.36.

$$BUFFER\,(Trajgen.e_1,\ Logdata.e_2, n)\ \textbf{sat}\ \Pi_{DL}[1a] \tag{5.36}$$

In this case, it is assumed that the buffering action is embedded and the size of storage is infinite. By matching the specifications with the properties of the set of generic processes, the ordering properties given by $\Pi_{DL}[3]$ and $\Pi_{DL}[4]$ together with the the possible alternative actions indicate by $\Pi_{DL}[2]$ and $\Pi_{DL}[3]$ results in the use of a *PERM* class generic process for realising the sub-system. A general *PERM* process is used in a recursive form with the corresponding parameterised processes P and Q assigned according to these ordering properties.

$$\textsc{DataLog} \ \widehat{=}\ \mu X \bullet (PERM\,(Ddata.e_3,\ P,\ Trajdata.e_1,\ Q)\ ;\ X) \tag{5.37}$$

$$P \ \widehat{=}\ Logdata\,!\,e_2 \rightarrow SKIP \tag{5.38}$$

$$Q \ \widehat{=}\ Ddata\,?\,e_3 \rightarrow SKIP \tag{5.39}$$

where e_1 to e_3 are defined in the specifications of DataLog

A pictorial representation of DataLog is given in Figure 5.5.

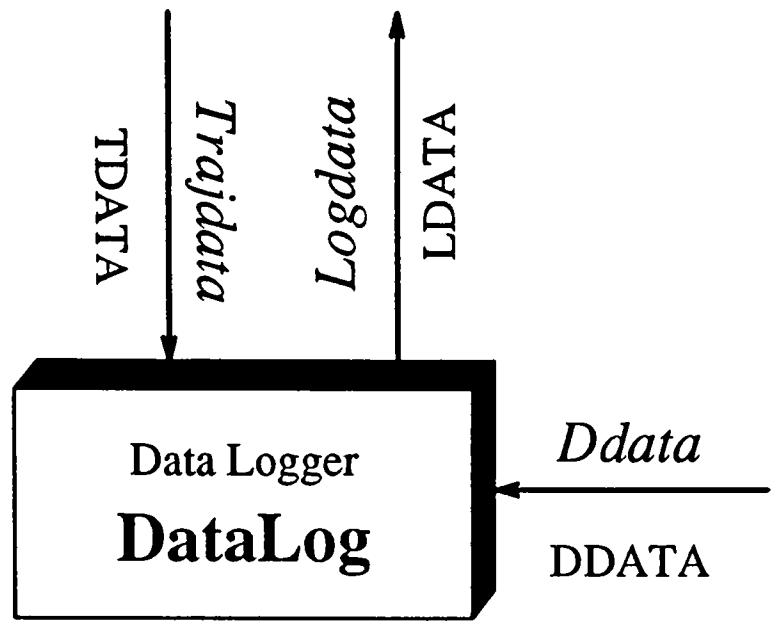


Figure 5.5: A pictorial view of DATALOG

5.5.2.3 Controller: Formal Specification

The requirements for the sub-system CONTROLLER given in Sub-section 5.5.1.3, suggest that it interacts with both TRAJECTORY GENERATOR and DATA PROCESSOR. Also, as an interfacing sub-system with the environment, it communicates with the ROBOT[1] via the channel *Drive* with the computed control demands. Table 5.4 gives the various abbreviations for channel names and alphabet sets for the construction of specifications for CTL.

Interacting Sub-systems	Channel name	Alphabet set	Justification
TRAJGEN & CTL	Traj	TRAJ	CTL–1
DATAPROC & CTL	Cdata	CDATA	CTL–1
CTL & ROBOT	Drive	DEMAND	CTL–2

Table 5.4: Formal parameters for specifying CONTROLLER

Corresponding to each of the requirements, formal specifications for the sub-system, $\Pi_{CTL}[i]\ (tr, ref)$, are translated and summarised as follows:

[1] The system ROBOT is considered as an element from the environment and will not be specified formally in the context of this thesis.

$$\begin{pmatrix} \quad ORDER\,(Traj,\ Drive,\ 1) \\ \vee \quad ORDER\,(Cdata,\ Drive,\ 1) \end{pmatrix} \qquad \text{by CTL–1/ 2: } \Pi_{CTL}[1]$$

$$\begin{pmatrix} \quad ORDER\,(Traj,\ Cdata,\ 1) \\ \vee \quad ORDER\,(Cdata,\ Traj,\ 1) \end{pmatrix} \qquad \text{by CTL–4: } \Pi_{CTL}[2]$$

$$s = \langle\rangle \wedge \{Traj.e_1,\ Cdata.e_2\} \not\subseteq ref \qquad \text{by CTL–2: } \Pi_{CTL}[3]$$

$$last\ s = Drive.e_3 \Rightarrow \begin{pmatrix} \quad s\ END\,(Traj.e_1,\ Cdata.e_2) \\ \vee\ s\ END\,(Cdata.e_2,\ Traj.e_1) \end{pmatrix}$$
$$\Rightarrow \begin{pmatrix} \quad Drive.e_3\ IFEND\,(Traj.e_1,\ Cdata.e_2) \\ \vee \quad Drive.e_3\ IFEND\,(Cdata.e_2,\ Traj.e_1) \end{pmatrix} \qquad \text{by CTL–3: } \Pi_{CTL}[4]$$

$$\text{where} \quad \begin{pmatrix} \quad e_1 \in TRAJ \\ \wedge \quad e_2 \in CDATA \\ \wedge \quad e_3 \in DEMAND \end{pmatrix}$$

It is seen that the set of specifications presented satisfied Property 4.20 of generic process *MUX*. Also, the requirement CTL–4 states that the sub-system CTL is recursive and whose input actions are toggled between two distinctive events, this fact has strengthened the use of *MUX*. The corresponding generic process realisation is given by Equation 5.40 together with a pictorial representation given in Figure 5.6.

$$\text{CTL} \ \widehat{=}\ MUX\,(Traj.e_1,\ Cdata.e_2,\ Drive.e_3) \qquad (5.40)$$

where e_1 to e_3 are defined in the specifications of CTL

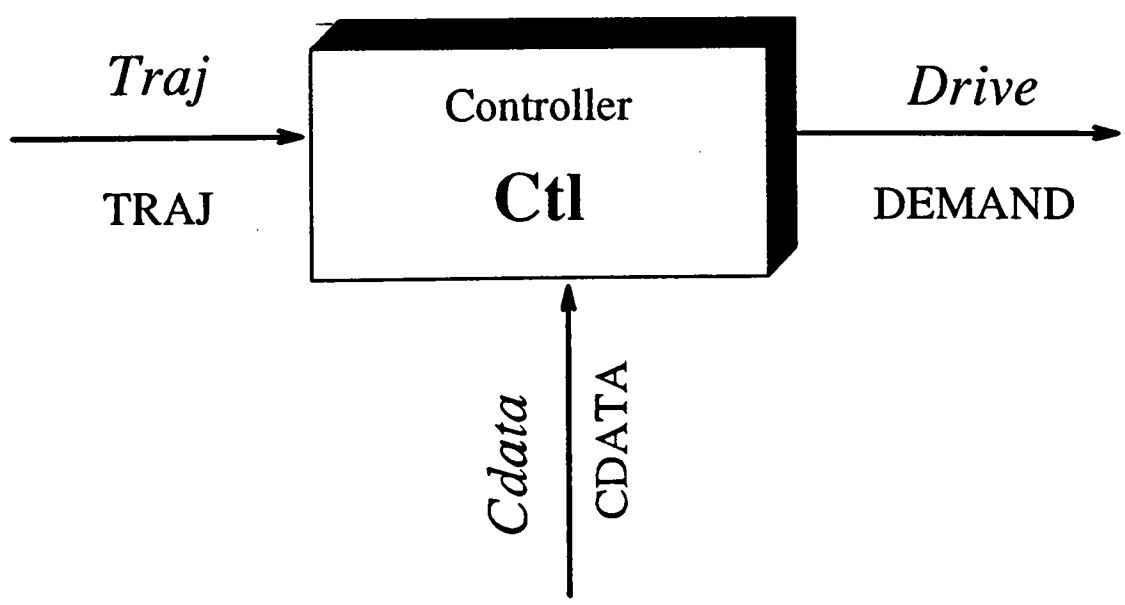


Figure 5.6: A pictorial view of CTL

5.5.2.4 Operator Interface: Formal Specification

As an interface between the environment and the system, the OPERATOR INTERFACE interacts with the OPERATOR[2] and the TRAJECTORY GENERATOR. The two sets of abbreviation parameters for specifications are given in Table 5.5.

Interacting Sub-systems	Channel name	Alphabet set	Justification
OPINT & TRAJGEN	Opout	VCOMMAND	OI–1
OPERATOR & OPINT	Opin	COMMAND	OI–1

Table 5.5: Formal parameters for specifying OPERATOR INTERFACE

Being a sub-system with a well defined input/output relationship and a function of command validation, the corresponding specifications, $\Pi_{OI}[i]\ (tr, ref)$, are translated from the requirements and summarised as follows:

$$ORDER\ (Opin.e_1,\ Opout.e_2,\ 1) \qquad \text{by OI–1: } \Pi_{OI}[1]$$

$$s = \langle\rangle \wedge \{Opin.e_1\} \not\subseteq ref \qquad \text{by OI–3: } \Pi_{OI}[2]$$

$$s\ NOTFAIL\ (Opin.e_1,\ Opout.e_2) \qquad \text{by OI–2: } \Pi_{OI}[3]$$

$$s\ NOTFAIL\ (Opout.e_2,\ Opin.e_1) \qquad \text{by OI–2: } \Pi_{OI}[4]$$

$$\text{where} \quad \left(\begin{array}{ll} & e_1 \in COMMAND \\ \wedge & e_2 \in VCOMMAND \end{array} \right)$$

From these specifications, it is concluded that the generic process *FSM* is most suitable for the realisation of the sub-system OPERATOR INTERFACE. This is justified by the satisfying of Property 4.2 by $\Pi_{OI}[1]$, Properties 4.3 and 4.4 by predicates $\Pi_{OI}[3]$ and $\Pi_{OI}[4]$ together with $H_{OI}[2]$. Equation 5.41 gives the corresponding generic process realisation of OPINT. Figure 5.7 gives a pictorial representation.

$$\text{OPINT} \ \widehat{=}\ FSM\ (Opin.e_1,\ Opout.e_2) \qquad (5.41)$$

where e_1 and e_2 are defined in the specifications of OPINT

5.5.2.5 Data Processor: Formal Specification

The DATA PROCESSOR is the other interfacing sub-system between the environment and the system. It interacts with the SENSOR[3], the DATA LOGGER and the CONTROLLER. The various

[2] OPERATOR is an element from the environment by which commands are generated, usually this is an human users and will not be specified formally in the context of the thesis.

[3] SENSOR is a device for capturing robot data such as joint positions and joint velocities, which is implemented as the input/output sub-system using hardware. Also, it is a member of the environment which is not specified formally.

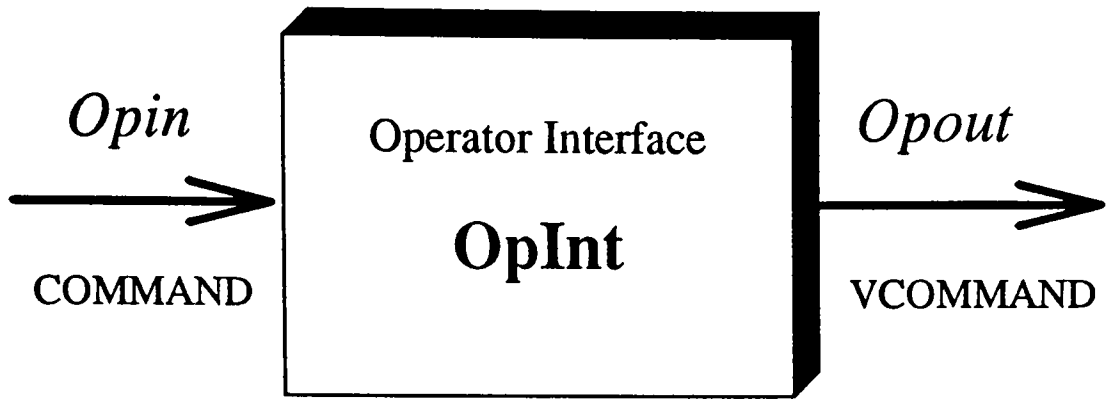


Figure 5.7: A pictorial representation of OPINT

abbreviations for specifications are given in Table 5.6

Interacting Sub-systems	Channel name	Alphabet set	Justification
SENSOR & DATAPROC	Sense	SIGNAL	DP–1
DATAPROC & DATALOG	Ddata	DDATA	DP–2
DATAPROC & CTL	Cdata	CDATA	DP–2

Table 5.6: Formal parameters for specifying DATA PROCESSOR

From the requirements given in Sub-section 5.5.1.5, specifications describing the *ordering* of events and possible *choices* between alternative system behaviours are translated. The following predicates $\Pi_{DP}[i]\ (tr, ref)$ summarises the behaviours of DATAPROC and are given as follows:

$$\left(\begin{array}{ll} & ORDER\,(Sense.e_1,\ Ddata.e_2,\ 1) \\ \vee & ORDER\,(Sense.e_1,\ Cdata.e_3,\ 1) \end{array}\right) \qquad \text{by DP–2/3: } \Pi_{DP}[1]$$

$$\left(\begin{array}{ll} & ORDER\,(Cdata.e_3,\ Ddata.e_2,\ 1) \\ \vee & ORDER\,(Ddata.e_2,\ Cdata.e_3,\ 1) \end{array}\right) \qquad \text{by DP–2: } \Pi_{DP}[2]$$

$$\left(\begin{array}{ll} & s\ NOTFAIL\,(Sense.e_1,\ Ddata.e_2) \\ \vee & s\ NOTFAIL\,(Sense.e_1,\ Cdata.e_3) \end{array}\right) \qquad \text{by DP–3: } \Pi_{DP}[3]$$

$$\left(\begin{array}{ll} & s\ NOTFAIL\,(Cdata.e_3,\ Ddata.e_2) \\ \vee & s\ NOTFAIL\,(Ddata.e_2,\ Cdata.e_3) \end{array}\right) \qquad \text{by DP–3: } \Pi_{DP}[4]$$

$$\left(\begin{array}{ll} & s\ NOTFAIL\,(Cdata.e_3,\ Sense.e_1) \\ \vee & s\ NOTFAIL\,(Ddata.e_2,\ Sense.e_1) \end{array}\right) \qquad \text{by DP–3: } \Pi_{DP}[5]$$

$$s = \langle\rangle \wedge \{Sense.e_1\} \not\subseteq ref \qquad \text{by DP–3: } \Pi_{DP}[6]$$

$$\left(\begin{array}{ll} & Ddata.e_2\ NOTREF\ (\langle Sense.e_1\rangle) \\ \vee & Cdata.e_3\ NOTREF\ (\langle Sense.e_1\rangle) \end{array}\right) \qquad \text{by DP–2: } \Pi_{DP}[7]$$

$$\text{where} \quad \left(\begin{array}{ll} & e_1 \in SIGNAL \\ \wedge & e_2 \in DDATA \\ \wedge & e_3 \in CDATA \end{array}\right)$$

These specifications of DATAPROC suggest that its behaviours inherit both from the generic processes *FSM* and *PERM*. This conclusion is made by comparing the above predicates and Properties 4.2 to 4.4 of *FSM* and Property 4.1 of *TPERM*. It is therefore possible to realise DATAPROC using a combination of *FSM* and *TPERM*. However, at the current level of abstraction, it is more appropriate to aim at the simplest realisation of DATAPROC. Hence, the corresponding generic process given by Equation 5.42, which is a recursive form of *TPERM* with a prefix action given by the communication across the channel *Sense* is adopted. Figure 5.8 gives a pictorial representation of this sub-system.

$$\text{DATAPROC} \ \widehat{=}\ \mu X \bullet (Sense\ ?\ e_1 \rightarrow TPREM\ (Cdata.e_3,\ Ddata.e_2)\ ;\ X) \qquad (5.42)$$

where $TPERM\ (Cdata.e_3,\ Ddata.e_2)$

$$\widehat{=}\ (\ Cdata\ !\ e_3 \rightarrow Ddata\ !\ e_2 \rightarrow SKIP$$

$$\Box\ Ddata\ !\ e_2 \rightarrow Cdata\ !\ e_3 \rightarrow SKIP)$$

and e_1 to e_3 are defined in the specifications of DATAPROC

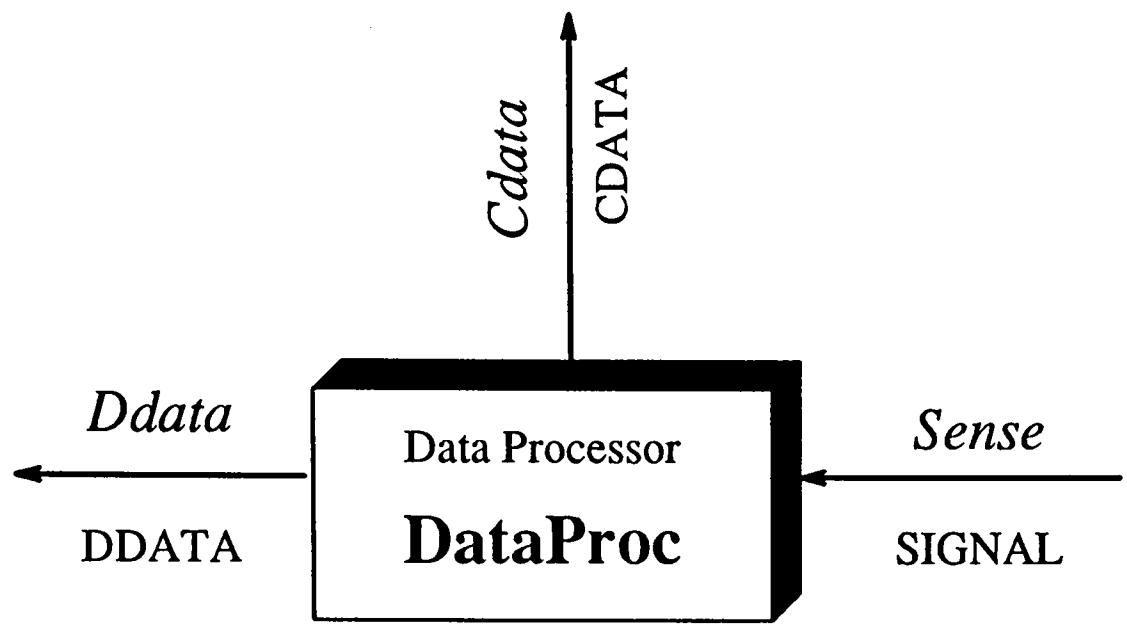


Figure 5.8: A pictorial view of DATAPROC

5.5.2.6 CONTROL System: Formal Specification

The overall CONTROL system specifications is derived from the sets of sub-system specifications by combining each of its specifications with appropriate conjunctions and disjunctions. The sub-systems are composed *concurrently* to form the overall system as maximum parallelism is assumed at this level of system development. A formal realisation of the overall system is given by Equation 5.43 with its corresponding alphabet set given by Equation 5.44.

$$\begin{aligned} \text{CONTROL} \;\widehat{=}\; & (\text{TRAJGEN} \parallel \text{DATALOG} \parallel \text{CTL} \parallel \\ & \text{OPINT} \parallel \text{DATAPROC}) \end{aligned} \tag{5.43}$$

$$\begin{aligned} \alpha\text{CONTROL} \;\widehat{=}\; & (\alpha\text{TRAJGEN} \cup \alpha\text{DATALOG} \cup \alpha\text{CTL} \cup \\ & \alpha\text{OPINT} \cup \alpha\text{DATAPROC}) \end{aligned} \tag{5.44}$$

However, the specifications given by the First Level Model must satisfy the essential system requirement specifications as set out in Sub-section 5.3.2, in the following sections, verification of the refined CONTROL system satisfying the essential specifications is investigated for a number of salient aspects. Here, the overall system specification is taken to be the union of each of the sub-system specifications according to the Rule of Combination (IR–2) discussed in Sub-section 2.3.3.

At this stage, a pictorial representation of the overall system composing of the five concurrent sub-systems expressed in generic processes together with the names of the communication channels are given in Figure 5.9. The advantages of having an overall view of the system architecture expressed in generic processes is not so much for formal reasoning, but as an illustrative means to provide designers and engineers a visual representation of the communication topology between

sub-systems and a structured view of the functionalities of the system. Moreover, such visual representations are based on mappings of formal objects and is therefore formal and unambiguous.

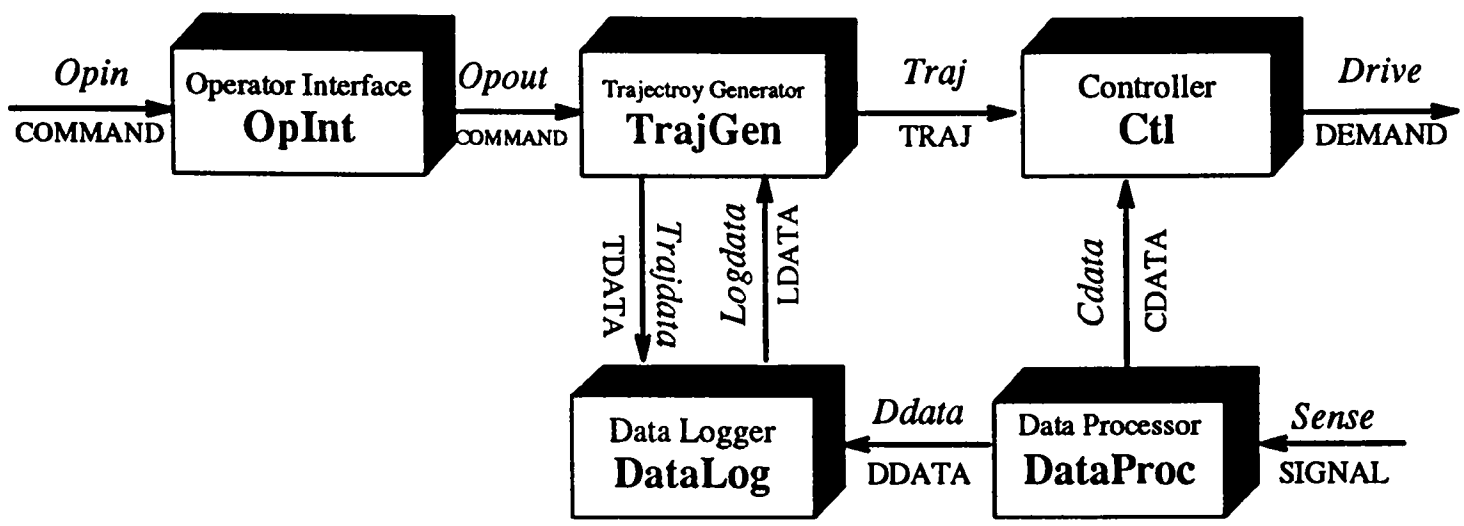


Figure 5.9: A generic process representation of the communication topology of the overall CONTROL system

5.5.3 Verification

In this section, the application of formal mathematics for proving the satisfies relations with respect to the original system specifications and consistency among parallel sub-systems in the First Level Model realisation is undertaken. It is intended that through these proofs, the role of formal verification within the unified methodology is illustrated and a general proof strategy can be developed so that a systematic approach can be followed for more complex systems. Nevertheless, the reasonings and proofs presented in the following sections do not represent a complete or full proof-of-correctness of the overall system that is realised according to the First Level Model. Due to the limitation in space, although a partial verification to the problem is presented, it is considered adequate to illustrate the point in this context.

It is suggested that a *constructive* approach to proof development is to be followed by which the proofs of the correctness of the overall system are built up from a number of smaller proofs on the combinations of sub-systems, and each of these proofs are based on the correctness of each sub-system satisfying their corresponding specifications. Following this strategy, the approach is roughly divided into *three* successive stages: the *atomic verification*, the *composite verification* and the *system verification*.

5.5.3.1 Atomic Verification

The atomic verification involves the establishing of the truthfulness of an individual sub-system realisation satisfying its corresponding specifications. The results obtained from this stage form a basis for further formal analysis on systems which are made up from these proven sub-systems of greater complexity. Therefore, the atomic stage should not be over-looked.

In this case study, each of the sub-systems is realised using one or more generic processes which are mapped from the sets of formal specifications describing its behaviours. From the discussion of the design of specification predicates and generic processes in Sub-sections 4.3.4.1 and 4.3.4.2, the main property between specification predicates and a particular generic process is the one-to-one correspondence between the two parties. Equation 5.45 defines the property which all generic processes hold. As a consequence, given a set of specifications describing the behaviours of a process which satisfies *all* the properties of a generic process, the rule of consequence infers that the particular generic process is at least a valid implementation of a subset of these specifications.

$$GP_i \textbf{ sat } \Pi_i\ (tr, ref) \tag{5.45}$$

where GP_i is a generic process defined in Section 4.3.4.2

and $\Pi_i(tr, ref)$ is the corresponding set of specifications

According to the formalisation discussed in Sub-sections 5.5.2.1 to 5.5.2.5, each sub-system is itself realised by one or more generic process, therefore its corresponding set of specifications are completely defined by *all* the properties of the particular generic process used. This implies using the rule of consequence that each sub-system that is expressed using generic processes satisfies its corresponding set of specifications. Formally, each of these satisfies relations are summarised in the followings.

$$\textsc{TrajGen} \quad \textbf{sat} \quad \begin{pmatrix} \Pi_{TG}[3] \\ \vee \begin{pmatrix} \Pi_{TG}[4] \\ \wedge\ \Pi_{TG}[1] \\ \wedge\ \Pi_{TG}[2] \end{pmatrix} \end{pmatrix} \qquad \begin{array}{r} \textit{PERM} \textbf{ sat } \textit{Properties } 4.17 \\ \textit{Properties } 4.17 \Rightarrow \Pi_{TG}[1] \textit{ to } \Pi_{TG}[4] \end{array} \tag{5.46}$$

$$\textsc{DataLog} \quad \textbf{sat} \quad \begin{pmatrix} \Pi_{DL}[2] \\ \vee \begin{pmatrix} \Pi_{DL}[4] \\ \wedge\ \Pi_{DL}[1] \\ \wedge\ \Pi_{DL}[3] \end{pmatrix} \end{pmatrix} \qquad \begin{array}{r} \textit{PERM} \textbf{ sat } \textit{Properties } 4.17 \\ \textit{Properties } 4.17 \Rightarrow \Pi_{DL}[3,4] \\ \textit{Properties } 4.2 \Rightarrow \Pi_{DL}[1,2,3] \end{array} \tag{5.47}$$

$$\textsc{Ctl} \quad \textbf{sat} \quad \left(\begin{array}{cc} & \Pi_{CTL}[3] \\ \vee & \left(\begin{array}{cc} & \Pi_{CTL}[4] \\ \wedge & \Pi_{CTL}[1] \\ \wedge & \Pi_{CTL}[2] \end{array} \right) \end{array} \right) \qquad \begin{array}{r} \textit{MUX} \textbf{ sat } \textit{Properties } 4.20 \\ \textit{Properties } 4.20 \Rightarrow \Pi_{CTL}[1,2,3,4] \end{array} \quad (5.48)$$

$$\textsc{OpInt} \quad \textbf{sat} \quad \left(\begin{array}{cc} & \Pi_{OI}[2] \\ \vee & \left(\begin{array}{cc} & \Pi_{OI}[3] \\ \wedge & \Pi_{OI}[4] \\ \wedge & \Pi_{OI}[1] \end{array} \right) \end{array} \right) \qquad \begin{array}{r} \textit{FSM} \textbf{ sat } \textit{Properties } 4.2 \textit{to } 4.4 \\ \textit{Properties } 4.3 \Rightarrow \Pi_{OI}[2,3] \\ \textit{Properties } 4.4 \Rightarrow \Pi_{OI}[4] \\ \textit{Properties } 4.2 \Rightarrow \Pi_{OI}[1] \end{array} \quad (5.49)$$

In the case of DataProc, since a basic form of generic process has not been used to realise its behaviours, therefore the proof of satisfies relations by such realisation is proceeded in two stages. The strategy is to show that a subset of specifications presented in Sub-section 5.5.2.5 is satisfied by the generic process *TPERM* which is a component of DataProc. Then the prefix event, that is the communication along channel *Sense* together with *TPERM* which is expressed in a recursive form is considered, and proved to satisfy all the specifications of DataProc. An outline of the proof strategy is presented as follows:

$$\textbf{TPERM} \quad \textbf{sat} \quad \left(\begin{array}{cc} & \Pi_{DP}[4] \\ \vee & \left(\begin{array}{cc} & \Pi_{DP}[7] \\ \wedge & \Pi_{DP}[2] \end{array} \right) \end{array} \right) \qquad \begin{array}{r} \textit{TPERM} \textbf{ sat } \textit{Properties } 4.18 \\ \textit{Properties } 4.18 \Rightarrow \Pi_{DP}[4,7,2] \end{array} \quad (5.50)$$

And for the overall sub-system DataProc:

$(\textit{Sense} \; ? \; e_1 \rightarrow \textit{TPERM})$ **sat**

$$\begin{array}{ll} \left(\begin{array}{cc} & \Pi_{DP}[6] \\ \vee & \left(\begin{array}{cc} & \Pi_{DP}[1] \\ \wedge & \Pi_{DP}[3] \\ \wedge & \Pi_{DP}[5] \end{array} \right) \end{array} \right) & \begin{array}{l} \text{prefix process} \\ \text{Property 4.2} \\ \text{Property 4.3} \\ \text{Property 4.4} \end{array} \\ \vee \left(\begin{array}{ll} & s = \langle \textit{Sense}.e_1 \rangle \wedge \textit{TPERM} \\ \Rightarrow & s = \langle \textit{Sense}.e_1 \rangle \wedge \left(\begin{array}{cc} & \Pi_{DP}[4] \\ \vee & \left(\begin{array}{cc} & \Pi_{DP}[7] \\ \wedge & \Pi_{DP}[2] \end{array} \right) \end{array} \right) \end{array} \right) & \begin{array}{l} \text{def DataProc} \\ \text{Assertion 5.50} \end{array} \end{array} \quad (5.51)$$

where $e_1 \in \textit{SIGNAL}$

From these results, it is concluded that each sub-system satisfies its corresponding set of specifications and this completes the proof-of-correctness in the atomic verification stage. □

5.5.3.2 Composite Verification

In this stage of verification, sub-systems are combined using the parallel operator and proofs are constructed to verify that these *composite* sub-systems satisfy the combined or CONTROL system specifications. Initially, two communicating sub-systems can be combined and verified, and the number of communicating sub-systems that are combined is increased as verification progresses. This is an intermediate stage of verification by which it takes on the form of the results obtained from the atomic verification and the subsequent results from this verification stage will support the following system verification. In this section, a number of cases are investigated.

Proof 5.6 *Proof of* (OPINT || TRAJGEN || CTL) *satisfying system properties*

The strategy is to prove that the three sub-systems running in parallel satisfies the predicates $\Pi_{OI}[1]$, $H_{TG}[1]$ and $\Pi_{CTL}[1]$ which implies that the overall CONTROL system's liveness property L–3 given in Sub-section 5.5.2.6.

To Prove:

(OPINT || TRAJGEN || CTL) **sat** $ORDER\ (Opin.e_1,\ Opout.e_2,\ 1)$

where $e_1 \in DEMAND$

and $e_2 \in VDEMAND$

Proof:

OPINT	**sat**	$ORDER\ (Opin.e_1,\ Opout.e_2,\ 1)$	proof result 5.49
		where $e_1 \in DEMAND$	
		and $e_2 \in VDEMAND$	
$ORDER\ (a, b, 1)$	$\Leftrightarrow$	$ORDER\ (a, b, 1)\ [s \restriction \{a, b\}/s]$	Proof 4.2
$\{Opin.e_1,\ Opout.e_2\}$	$\subseteq$	αOPINT	def of OPINT
$\{Opin.e_1,\ Opout.e_2\}$	$\subseteq$	α(OPINT || TRAJGEN || CTL)	property of ||

By Theorem 4.1 for parallel process composition:

(OPINT || TRAJGEN || CTL) **sat** $ORDER\ (Opin.e_1,\ Opout.e_2,\ 1)$

To Prove:

(OPINT || TRAJGEN || CTL) **sat** $ORDER\ (Opout.e_3,\ Traj.e_4,\ 1)$

where $e_3 \in VDEMAND$

and $e_4 \in TRAJ$

Proof:

$$\begin{array}{rcll} \textsc{TrajGen} & \textbf{sat} & ORDER\,(Opout.e_1,\ Traj.e_2,\ 1) & \text{proof result 5.46} \\ & & \text{where} \quad e_3 \in VDEMAND & \\ & & \text{and} \quad e_4 \in TRAJ & \\ \\ ORDER\,(a,b,1) & \Leftrightarrow & ORDER\,(a,b,1)\,[s \restriction \{a,b\}/s] & \text{Proof 4.2} \\ \{Opout.e_3,\ Traj.e_4\} & \subseteq & \alpha\textsc{TrajGen} & \text{def of TrajGen} \\ \{Opout.e_3,\ Traj.e_4\} & \subseteq & \alpha(\textsc{OpInt} \parallel \textsc{TrajGen} \parallel \textsc{Ctl}) & \text{property of } \parallel \end{array}$$

By Theorem 4.1 for parallel process composition:

$$(\textsc{OpInt} \parallel \textsc{TrajGen} \parallel \textsc{Ctl}) \quad \textbf{sat} \quad ORDER\,(Opout.e_3,\ Traj.e_4,\ 1)$$

To Prove:

$$(\textsc{OpInt} \parallel \textsc{TrajGen} \parallel \textsc{Ctl}) \ \textbf{sat}\ ORDER\,(Traj.e_5,\ Drive.e_6,\ 1)$$

where $e_5 \in TRAJ$

and $e_6 \in DEMAND$

Proof:

The proof follows identically to the two given above and by proof results 5.48, Proof 4.2, the definition of Ctl and Theorem 4.1, the deduction follows:

$$\Rightarrow (\textsc{OpInt} \parallel \textsc{TrajGen} \parallel \textsc{Ctl}) \quad \textbf{sat} \quad ORDER\,(Traj.e_5,\ Drive.e_6,\ 1)$$

Finally, from these three results, the following deduction is made:

$$\begin{array}{rl} & (\textsc{OpInt} \parallel \textsc{TrajGen} \parallel \textsc{Ctl})\ \textbf{sat} \\ & \left(\begin{array}{ll} & ORDER\,(Opin.e_1,\ Opout.e_2,\ 1) \\ \wedge & ORDER\,(Opout.e_3,\ Traj.e_4,\ 1) \\ \wedge & ORDER\,(Traj.e_5,\ Drive.e_6,\ 1) \end{array}\right) \qquad \text{proof results} \\ \\ \Rightarrow & ORDER\,(Opin.e_1,\ Drive.e_6,\ 1) \qquad \text{property of traces} \\ \Rightarrow & ORDER\,(action,\ start,\ 1) \qquad action \in DEMAND,\ start \in PERFORM \\ \Rightarrow & \text{L–3} \qquad \text{def of L–3} \end{array}$$

This completes the proof showing that the three sub-systems running in parallel do not only satisfy their combined properties but also the global liveness property L–3. □

The second case to investigate is the interactions between the sub-systems TrajGen and DataLog. The proof determines the sequentiality of communication between the two parallel

sub-systems. According to Figure 5.9, it is seen that there are two other sub-systems interacting with TRAJGEN and DATALOG, in the following proof, it is assumed that their communications with these sub-systems are *external* events which are ready to occur at any moment. On grounds of safety and liveness, the following proof tries to uncover a number of their properties.

Proof 5.7 *Proof of* (DATALOG || TRAJGEN) *satisfying their combined specifications*

This proof investigates the combined properties that are defined by the specifications from both DATALOG and TRAJGEN. The parallel combination of the two sub-systems must satisfy predicates $\Pi_{TG}[1]$ to $\Pi_{TG}[4]$ of TRAJGEN and $\Pi_{DL}[2]$ to $\Pi_{DL}[4]$ of DATALOG. These predicates are grouped and are considered in the following cases.

Case: (DATALOG || TRAJGEN) **sat** $(\Pi_{TG}[1] \wedge \Pi_{TG}[2])$

Proof:

$$\text{TRAJGEN} \quad \textbf{sat} \quad \left(\begin{array}{cc} & ORDER\,(Opout, Traj, 1) \\ \vee & \left(\begin{array}{cc} & ORDER\,(Logdata, Traj, 1) \\ \wedge & ORDER\,(Opout, Trajdata, 1) \end{array} \right) \end{array} \right) \qquad \text{result 5.46}$$

$$\begin{array}{rcll} ORDER\,(a, b, 1) & \Leftrightarrow & ORDER\,(a, b, 1)\,[s \upharpoonright \{a, b\}/s] & \text{Proof 4.2} \\ \{Opout,\ Traj\} & \subseteq & \alpha\text{TRAJGEN} & \text{def. TRAJGEN} \\ \{Opout,\ Traj\} & \subseteq & \alpha(\text{DATALOG} \parallel \text{TRAJGEN}) & \text{property of } \parallel \end{array}$$

By Theorem 4.1 for composite parallel processes:

$$(\text{DATALOG} \parallel \text{TRAJGEN})\ \textbf{sat} \left(\begin{array}{cc} & ORDER\,(Opout, Traj, 1) \\ \vee & \left(\begin{array}{cc} & ORDER\,(Logdata, Traj, 1) \\ \wedge & ORDER\,(Opout, Trajdata, 1) \end{array} \right) \end{array} \right)$$

Case: (DATALOG || TRAJGEN) **sat** $(\Pi_{DL}[2, 3] \wedge \Pi_{TG}[3, 4])$

In this case, the proof considers the consequences *after* the occurrence of the two external communications via channels *Ddata* on receiving robot data and via channel *Opout* on receiving operator's commands. Although the communication via channel *Traj* is also an external communication, it is assumed that their environment, that is formed from the other sub-systems, is always ready to accept this communication. As a result, communication via *Traj* is abstracted from the process definitions. Moreover, since the primary objective in this proof is to investigate the ordering properties of channel communications, the actual messages that communicate across

a particular channel is also abstracted and only channel names are used to indicate the occurrence of communications.

According to Equation 5.33 for the implementation of TRAJGEN and Equation 5.37 for DATALOG, the two simplified processes given by Equations 5.52 and 5.53 defining the behaviours of these sub-systems are considered with communication across *Traj* abstracted from TRAJGEN (X).

$$\begin{aligned} \text{TrajGen}\,(X) \;\widehat{=}\; & (\mathit{Opout} \rightarrow \mathit{Trajdata} \rightarrow X \\ & \mid \mathit{Logdata} \rightarrow X) \end{aligned} \tag{5.52}$$

$$\begin{aligned} \text{DataLog}\,(X) \;\widehat{=}\; & (\mathit{Ddata} \rightarrow \mathit{Logdata} \rightarrow X \\ & \mid \mathit{Trajdata} \rightarrow \mathit{Ddata} \rightarrow X) \end{aligned} \tag{5.53}$$

And from the specifications given by the four predicates $\Pi_{DL}[2,3]$ and $\Pi_{TG}[3,4]$, the proof assumptions become:

$$\begin{aligned} &(\text{DATALOG}(X) \parallel \text{TRAJGEN}(X)) \text{ sat} \\ &\left(\begin{array}{ll} & s = \langle\rangle \wedge \{\mathit{Ddata},\ \mathit{Opout}\} \not\subseteq \mathit{ref} \\ \vee & s_0 = \langle \mathit{Opout} \rangle \wedge s' = \langle\rangle \wedge \{\mathit{Trajdata},\ \mathit{Ddata}\} \not\subseteq \mathit{ref} \\ \vee & s_0 = \langle \mathit{Ddata} \rangle \wedge s' = \langle\rangle \wedge \{\mathit{Trajdata},\ \mathit{Opout}\} \not\subseteq \mathit{ref} \\ \\ \vee & s \geq \langle \mathit{Opout}, \mathit{Ddata} \rangle \wedge \left(\begin{array}{ll} & \mathit{ORDER}\,(\mathit{Ddata},\ \mathit{Logdata},\ 1)[s/s''] \\ \vee & \mathit{ORDER}\,(\mathit{Opout},\ \mathit{Trajdata},\ 1)[s/s''] \end{array} \right) \\ \vee & s \geq \langle \mathit{Ddata}, \mathit{Opout} \rangle \wedge \left(\begin{array}{ll} & \mathit{ORDER}\,(\mathit{Ddata},\ \mathit{Logdata},\ 1)[s/s''] \\ \vee & \mathit{ORDER}\,(\mathit{Opout},\ \mathit{Trajdata},\ 1)[s/s''] \end{array} \right) \\ \\ \vee & s \geq \langle \mathit{Opout}, \mathit{Trajdata} \rangle \wedge \mathit{ORDER}\,(\mathit{Opout},\ \mathit{Trajdata},\ 1)[s/s''] \\ \vee & s \geq \langle \mathit{Ddata},\ \mathit{Logdata} \rangle \wedge \mathit{ORDER}\,(\mathit{Ddata},\ \mathit{Logdata},\ 1)[s/s''] \end{array} \right) \end{aligned} \tag{5.54}$$

Proof:

The proof is constructed using a case analysis by which each predicate of the disjunction is considered.

Case: $s = \langle\rangle$

$$\begin{aligned} & (s \downarrow \mathit{Trajdata}) = (s \downarrow \mathit{Opout}) = \langle\rangle \\ & (s \downarrow \mathit{Logdata}) = (s \downarrow \mathit{Ddata}) = \langle\rangle && \text{from assumption} \\ \Rightarrow\ & \left(\begin{array}{l} \mathit{ORDER}\,(\mathit{Ddata},\ \mathit{Logdata},\ 1) \\ \vee\ \mathit{ORDER}\,(\mathit{Opout},\ \mathit{Trajdata},\ 1) \end{array} \right) && \text{def of TRAJGEN \& DATALOG} \\ & \wedge\ (\{\mathit{Ddata},\ \mathit{Opout}\} \not\subseteq \mathit{ref}) && \text{refusals of processes} \end{aligned}$$

Case: $s_0 = \langle Opout \rangle$

$$s_0 = \langle Opout \rangle \wedge s' = \langle\rangle$$ from assumption

$$\Rightarrow \quad (s \downarrow Trajdata \overset{1}{\leq} s \downarrow Opout)$$
$$\wedge\, ((s \downarrow Logdata) = (s \downarrow Ddata) = \langle\rangle)$$ def of TRAJGEN & DATALOG

$$\Rightarrow \quad (ORDER\,(Opout,\ Trajdata,\ 1)$$
$$\wedge \begin{pmatrix} \{Trajdata\} \not\subseteq ref \\ \vee \quad \{Ddata\} \not\subseteq ref \end{pmatrix}$$ def of TRAJGEN & DATALOG, and general choice on processes

Case: $s_0 = \langle Ddata \rangle$

$$s_0 = \langle Ddata \rangle \wedge s' = \langle\rangle$$ from assumption

$$\Rightarrow \quad (s \downarrow Logdata \overset{1}{\leq} s \downarrow Ddata)$$
$$\wedge\, ((s \downarrow Opout) = (s \downarrow Trajdata) = \langle\rangle)$$ def of TRAJGEN & DATALOG

$$\Rightarrow \quad (ORDER\,(Ddata,\ Logdata,\ 1))$$
$$\wedge \begin{pmatrix} \{Logdata\} \not\subseteq ref \\ \vee \quad \{Opout\} \not\subseteq ref \end{pmatrix}$$ def of DATALOG & TRAJGEN, and general choice on processes

Case: $s = \langle Opout, Ddata \rangle \frown s'' \wedge s'' \neq \langle\rangle$

Let $TGDL\,(X) =$
$$(\text{DATALOG}(X) \parallel \text{TRAJGEN}(X))$$

$$s_0 = \langle Opout \rangle \wedge s_0' = \langle\rangle$$
$$\Rightarrow \quad \{Ddata,\ Trajdata\} \not\subseteq ref$$ from previous cases

$$s_0 = \langle Opout \rangle \wedge s_0' = \langle Ddata \rangle \wedge s''$$ from assumption
$$\Rightarrow \quad \{Trajdata,\ Logdata\} \not\subseteq ref$$ def of DATALOG & TRAJGEN [I]

However $\{Logdata\} \subseteq refusals\,(\text{TRAJGEN}/\langle Opout \rangle)$ def of TRAJGEN
$$\Rightarrow \quad \{Logdata\} \subseteq refusals\,(TGDL\,(X)/\langle Opout \rangle)$$ and Theorem 4.2
$$\Rightarrow \quad \{Logdata\} \subseteq refusals\,(TGDL\,(X)/\langle Opout, Ddata \rangle)$$ $Ddata \notin \alpha\text{TRAJGEN}$

and $\{Trajdata\} \subseteq refusals\,(\text{DataLog}/\langle Ddata \rangle)$ def of DATALOG
$$\Rightarrow \quad \{Trajdata\} \subseteq refusals\,(TGDL\,(X))/\langle Ddata \rangle)$$ and Theorem 4.2
$$\Rightarrow \quad \{Trajdata\} \subseteq refusals\,(TGDL\,(X)/\langle Opout, Ddata \rangle)$$ $Opout \notin \alpha\text{DATALOG}$

$$\Rightarrow \quad \{Trajdata, Logdata\} \subseteq refusals\,(TGDL\,(X)/\langle Opout, Ddata \rangle)$$ from above 2 cases [II]
$$\Rightarrow \quad s'' = \langle\rangle$$ from [II]
$$\Rightarrow \quad \textbf{deadlock}$$ from [I] & [II]

Case: $s = \langle Ddata, Opout \rangle \frown s'' \wedge s'' \neq \langle\rangle$

The proof follows that of the last case where $s = \langle Opout, Ddata \rangle \frown s'' \wedge s'' \neq \langle\rangle$. The implication in the proof contradicts with the initial assumption that $s'' \neq \langle\rangle$ according to the following proof conclusion:

$$\{Trajdata, Logdata\} \subseteq refusals\ (TGDL\ (X)/\langle Ddata, Opout \rangle)$$

From the proof results of the last two cases, it is seen that the proposed realisations of the sub-systems TRAJGEN and DATALOG running in parallel has a potential hazard of *deadlock* under particular external conditions. The same conclusion can be obtained from an analysis using process algebra given by Equation 5.55 on the two sub-systems written in the form given by Equations 5.52 and 5.53.

$$\begin{aligned}
(\text{TRAJGEN} \parallel \text{DATALOG}) \equiv\ & (\mu X \bullet \quad (Opout \rightarrow Trajdata \rightarrow X \mid Logdata \rightarrow X)) \\
\parallel\ & (\mu X \bullet \quad (Ddata \rightarrow Logdata \rightarrow X \\
& \qquad\quad \mid Trajdata \rightarrow Ddata \rightarrow X)) \\
\\
\equiv\ & \mu X \bullet \quad (Opout \rightarrow Ddata \rightarrow \mathbf{STOP} \\
& \qquad\quad \mid Opout \rightarrow Trajdata \rightarrow Ddata \rightarrow X \\
& \qquad\quad \mid Ddata \rightarrow Logdata \rightarrow X \\
& \qquad\quad \mid Ddata \rightarrow Opout \rightarrow \mathbf{STOP}) \qquad (5.55)
\end{aligned}$$

From the expanded process definitions of (TRAJGEN $\parallel$ DATALOG), there exists two alternative branches in which a deadlock results. It is therefore necessary to re-design the realisation of either one or both of these sub-systems so as to eliminate this potential deadlock. In Sub-section 5.5.4, two re-design strategies are discussed. At this point, although the realisation of the sub-systems has found to be potentially hazardous, the correctness of the remaining two cases given by the last two predicates in Equation 5.54 can be verified using the result of Proof 4.2 given in Section 4.4.1.

5.5.3.3 System Verification

System verification is the final verification process for a particular system implementation under a defined realisation model. The process sums up all the deductions from the previous two stages

of verification, namely the atomic and composite verifications, and concludes the correctness of the overall system realisation with respect to the original formal essential specifications. Since the results from an atomic and composite verification are atomic deductions from the overall system in different aspects and are being considered under various conditions, they are linked in a logical way at the system verification stage so that *global properties* are proved based on these specifications.

In system verification, the two theorems given in Sub-section 4.5.2 are found to be useful in building proofs with a number of parallel sub-systems. In Proof 5.7, it is seen that the designs of the sub-systems DATALOG and TRAJGEN have resulted in a local deadlock which may lead to an eventual global deadlock when *all* the sub-systems are combined. In order to avoid this predicted mishap, the sub-system DATALOG is taken out from the system in the following proofs. Sub-systems TRAJGEN and DATAPROC are modified such that robot data are always available to TRAJGEN and that are directly fed from DATAPROC. As a result, the system is simplified to one which does not have a Data Logger. The following modified TRAJGEN and DATAPROC can be proved to satisfy their corresponding specifications according to the strategies described in Sub-section 5.5.3.1.

$$\text{TRAJGEN}_1 \;\widehat{=}\; \mu X \bullet (PERM\ (Opout.e_1,\ P,\ Logdata.e_2,\ Q)\ ;\ X) \tag{5.56}$$

$$P \;\widehat{=}\; Traj\ !\ e_3 \rightarrow Logdata\ ?\ e_2 \rightarrow SKIP$$

$$Q \;\widehat{=}\; Traj\ !\ e_3 \rightarrow SKIP$$

and

$$\text{DATAPROC}_1 \;\widehat{=}\; \mu X \bullet (Sense\ ?\ e_4 \rightarrow TPERM\ (Cdata.e_5,\ Logdata.e_2)\ ;\ X) \tag{5.57}$$

where $TPERM\ (Cdata.e_5,\ Logdata.e_2)$

$$\widehat{=} (\ \ Cdata\ !\ e_5 \rightarrow Logdata\ !\ e_2 \rightarrow SKIP$$

$$\Box\ Logdata\ !\ e_2 \rightarrow Cdata\ !\ e_5 \rightarrow SKIP)$$

and
$$\left(\begin{array}{ll} & e_1 \in VCOMMAND \\ \wedge & e_2 \in LDATA \\ \wedge & e_3 \in TRAJ \\ \wedge & e_4 \in SIGNAL \\ \wedge & e_5 \in CDATA \end{array}\right)$$

The following proof strategies are used to establish the correctness of the modified overall system by satisfying its liveness properties L–1/2.

Proof 5.8 *Proof of overall system correctness with respect to Properties L-1/2*

To Prove:

$$(\text{OPINT} \parallel \text{TRAJGEN}_1 \parallel \text{CTL} \parallel \text{DATAPROC}_1) \textbf{ sat } \left(\begin{array}{rl} & s\ START\ (Opin,\ Drive) \\ \Rightarrow & \{Drive\} \not\subseteq ref \end{array} \right)$$

Proof:

If	OPINT	sat	$\Pi_{OI}\ [i]$	from proof result 5.49
and	TRAJGEN_1	sat	$\Pi_{TG_1}\ [i]$	parallel to proof result 5.46
and	CTL	sat	$\Pi_{CTL}\ [i]$	from proof result 5.48
and	DATAPROC_1	sat	$\Pi_{DP_1}\ [i]$	parallel to proof results 5.50 and 5.51

Let $\text{CONTROL}_1 =$
$(\text{OPINT} \parallel \text{TRAJGEN}_1 \parallel \text{CTL} \parallel \text{DATAPROC}_1)$
and $\text{CONTROL}_2 =$
$(\text{OPINT} \parallel \text{TRAJGEN}_1 \parallel \text{CTL})$

also CONTROL_2 will not deadlock — from Proof 5.6 and Appendix D

$$\text{and} \quad \left(\begin{array}{rl} & \text{OPINT sat } ORDER\ (Opin,\ Opout,\ 1) \\ \wedge & \text{TRAJGEN}_1 \text{ sat } ORDER\ (Opout,\ Traj,\ 1) \\ \wedge & \text{CTL sat } ORDER\ (Traj,\ Drive,\ 1) \end{array} \right) \quad \text{from atomic verification}$$

$$\Rightarrow \quad (\text{OPINT} \parallel \text{TRAJGEN}_1 \parallel \text{CTL}) \textbf{ sat } ORDER\ (Opin,\ Drive,\ 1) \quad \text{parallel processes}$$

$$\Rightarrow \quad \text{CONTROL}_2 \textbf{ sat } \left(\begin{array}{rl} & s\ START\ (Opin,\ Drive) \\ \Rightarrow & \{Drive\} \not\subseteq ref \end{array} \right) \quad s \in traces\ (\text{CONTROL}_2)$$

If $\{Drive\} \not\subseteq \alpha$ DATAPROC — def of DATAPROC
$\Rightarrow$ $\{Drive\} \notin refusals\ (\text{DATAPROC})$
$\Rightarrow$ $\{Drive\} \not\subseteq ref$
where $ref \in refusals\ (\text{DATAPROC} \parallel \text{CONTROL}_2)$ — Theorem 4.2

$$\Rightarrow \quad \text{CONTROL}_1 \textbf{ sat } \left(\begin{array}{rl} & s\ START\ (Opin,\ Drive) \\ \Rightarrow & \{Drive\} \not\subseteq ref \end{array} \right) \quad s \in traces\ (\text{CONTROL}_1)$$

The result establishes the satisfies relationship of the original liveness properties L–1/2 by the modified system realisation CONTROL_1. □

5.5.4 Potential Problems and Re-design Strategies

It is seen from Proof 5.7 that the realisations of sub-systems TRAJGEN and DATALOG in the First Level Model have resulted in a potential deadlock. Although in the evolution of CONTROL,

such deadlocking situations may not occur, it is necessary to eliminate all possible occurrences of deadlock as they violate the system liveness requirements.

To this extend, two re-designs of the realisation of one of the sub-systems are described in an effort to eliminate the predicted deadlocks. Sub-system DATALOG is arbitrary chosen and two separate re-design strategies have been followed. The first one is based on the *mutual exclusion principal* on shared resources due to Dijkstra [1976] and the other is a total re-design of the sub-system through the re-interpretation of the requirements with the aid of the set of generic processes. The later re-design strategy has followed a novel path in seeking a solution and forms part of this thesis' contribution.

5.5.4.1 The Data Master

The introduction of the DATA MASTER has a primary role of restricting the sequence of communication between the two sub-systems in a way such that shared resources are requested and are granted in a pre-defined sequence. The solution is parallel to Dijkstra's solution to the dining philosophers problem where a *footman* is introduced.

The function of the DATA MASTER is to ensure that a pre-determined sequence of communication with the sub-systems (DATALOG || TRAJGEN) is obeyed such that the input of robot data from DATAPROC must occur exclusively with the input of validated operator's commands from sub-system OPINT or vice versa. To elaborate the description of its function, once an input is taken by the two sub-systems, their subsequent actions as defined by their corresponding specifications have to be completed before another input is allowed. Following this requirement, a DATA MASTER with *gating events* that are used to impose the pre-defined sequences of communication through synchronisation is designed. A pair of gating events corresponding to the particular sequence of actions initiated via channel communications *Opout* and *Ddata* is introduced. A possible realisation of the design is given by Equation 5.58.

$$\begin{aligned} \text{DATAMASTER} \;\widehat{=}\; \mu X \bullet \;\; & (Ddata\_g \rightarrow g_{10} \rightarrow g_{11} \rightarrow X \\ & \mid Opout\_g \rightarrow g_{20} \rightarrow g_{21} \rightarrow X) \end{aligned} \tag{5.58}$$

This implementation forces the choice between a communication with either OPINT or DATAPROC to be made exclusively by their environment through the extra synchronising events. The two synchronising events are simple events that involve no message passing, *Opout_g* signals a valid command is present whereas *Ddata_g* indicates that robot data is ready for input. These two events are amalgamated into the modified sub-systems OPINT and DATAPROC which is given by

Equations 5.59 and 5.60. For reasons explained previously, the messages that are communicated through all the channels are abstracted from these implementations. Nonetheless, the input/output operators are maintained to indicate the directions of data flow.

$$\text{OPINT}_{DM} \;\widehat{=}\; FSM_{DM}\,(Opin,\ Opout\_g,\ Opout) \tag{5.59}$$

where $FSM_{DM}\,(Opin,\ Opout\_g,\ Opout) \;\widehat{=}$

$$\mu\, X \bullet (Opin\,? \rightarrow Opout\_g \rightarrow Opout\,! \rightarrow X)$$

$$\text{DATAPROC}_{DM} \;\widehat{=}\; \mu\, X \bullet (Sense\,? \rightarrow TPREM_{DM}\,(Cdata,\ Ddata)\ ;\ X) \tag{5.60}$$

where $TPERM_{DM}\,(Cdata,\ Ddata) \;\widehat{=}$

$$(\ \ Cdata\,! \rightarrow Ddata\_g \rightarrow Ddata\,! \rightarrow SKIP$$

$$\Box\ Ddata\_g \rightarrow Ddata\,! \rightarrow Cdata\,! \rightarrow SKIP)$$

Having described these modifications made on the two *external* sub-systems, re-designs of the sub-systems TRAJGEN and DATALOG are then carried out. With a view to retaining the satisfies conditions with the corresponding specifications, the structures of the two re-designed sub-systems are maintained but with the introduction of extra gating events. These gating events correspond to those defined in the alphabet set of DATAMASTER and are used for synchronising with the DATAMASTER. The proposed re-designed TRAJGEN and DATALOG are given by Equations 5.61 and 5.62.

$$\text{TRAJGEN}_2 \;\widehat{=}\; \mu\, X \bullet \ (g_{20} \rightarrow Opout\,? \rightarrow Trajdata\,! \rightarrow X$$

$$\mid Logdata\,? \rightarrow g_{11} \rightarrow X) \tag{5.61}$$

$$\text{DATALOG}_2 \;\widehat{=}\; \mu\, X \bullet \ (g_{10} \rightarrow Ddata\,? \rightarrow Logdata\,! \rightarrow X$$

$$\mid Trajdata\,? \rightarrow Ddata\,? \rightarrow g_{21} \rightarrow X) \tag{5.62}$$

To show that the re-design works without the hazard of a potential deadlock, process algebra is used to expand the overall system CONTROL_{DM}, which is made up from the two re-designed sub-systems and the DATAMASTER running in parallel.

$$(\text{TRAJGEN}_2 \;||\; \text{DATALOG}_2 \;||\; \text{DATAMASTER}) \;\widehat{=}$$

$$(\mu\, X \bullet (Ddata\_g \rightarrow g_{10} \rightarrow g_{11} \rightarrow X$$

$$
\begin{aligned}
&\quad | \; Opout\_g \rightarrow g_{20} \rightarrow g_{21} \rightarrow X)) \\
&|| \, (\mu\, Y \bullet (g_{20} \rightarrow Opout \rightarrow Trajdata \rightarrow Y \\
&\quad | \; Logdata \rightarrow g_{11} \rightarrow Y)) \\
&|| \, (\mu\, Z \bullet (g_{10} \rightarrow Ddata \rightarrow Logdata \rightarrow Z \\
&\quad | \; Trajdata \rightarrow Ddata \rightarrow g_{21} \rightarrow Z))
\end{aligned}
$$

$$
\begin{aligned}
\equiv \mu\, X \bullet \quad & (\; Ddata\_g \rightarrow g_{10} \rightarrow Ddata \rightarrow Logdata \rightarrow g_{11} \rightarrow X \\
& | \; Opout\_g \rightarrow g_{20} \rightarrow Opout \rightarrow Trajdata \rightarrow Ddata \rightarrow g_{21} \rightarrow X)
\end{aligned} \tag{5.63}
$$

The final expression is a recursive process with two mutually exclusive sequences of actions which is free from deadlock. Hence the re-design is valid. □

5.5.4.2 A Total Re-design

A different strategy to avoid the occurrence of potential deadlocks is to re-interpret the requirements and specifications of the problematic sub-systems such that novel realisations using different generic processes are proposed. Following this strategy, the sub-system DATALOG is chosen to be the sub-system for a novel realisation.

By re-examining the specifications of DATALOG and matching each predicates with the properties of generic processes, it is concluded that the generic process *INIT* is also capable of reflecting the actions describe by the specifications of DATALOG. In particular, the conditional behaviours expressed in specifications $\Pi_{DL}[3]$ matches exactly with Property 4.7. Although, specification $\Pi_{DL}[2]$ can be realised using a *PERM* class generic process, such a realisation is in-sufficient in implementing $\Pi_{DL}[3]$. Also, the recursive nature of the sub-system means that the mutual recursive processes *INIT* and *UPDATE* are viable candidates. Hence, a possible realisation of DATA LOGGER is given by Equation 5.64.

$$
\text{DATALOG}_3 \;\widehat{=}\; INIT\,(Logdata.e_2,\; Ddata.e_3,\; Trajdata.e_1) \tag{5.64}
$$

$$
\begin{aligned}
&\text{with} \quad UPDATE\,(Logdata.e_2,\; Ddata.e_3,\; Trajdata.e_1) \\
&\text{and} \quad \left(\begin{array}{ll} & e_1 \in TDATA \\ \wedge & e_2 \in LDATA \\ \wedge & e_3 \in DDATA \end{array} \right)
\end{aligned}
$$

To verify that this novel realisation satisfies its specifications, an atomic verification is carried out and an outline of the proof is given below with the corresponding results.

Proof 5.9 *Proof of* DATALOG$_3$ *satisfies its specifications*

$$\text{DATALOG}_3 \quad \textbf{sat} \quad \left(\begin{array}{cc} & \Pi_{DL}[2] \\ \vee & \left(\begin{array}{cc} & \Pi_{DL}[4] \\ \wedge & \Pi_{DL}[3] \end{array} \right) \end{array} \right) \qquad \begin{array}{r} \textit{INIT} \textbf{ sat } \textit{Properties } 4.7 \textit{and } 4.8 \\ \textit{Properties } 4.7 \Rightarrow \Pi_{DL}[2] \\ \textit{Properties } 4.8 \Rightarrow \Pi_{DL}[3] \end{array}$$

To determine the correctness of the design when running concurrently with TRAJGEN, the following proof is considered.

Proof 5.10 *Proof of deadlock freedom in* (DATALOG$_3$ || TRAJGEN)

The proof investigates the correctness of the combined sub-system (DATALOG$_3$ || TRAJGEN). The aim is to predict whether the novel design of DATALOG$_3$ is free from deadlock. Writing DATALOG$_3$ in an expanded form given in Section 4.4.2 and the sub-system TRAJGEN as given by Equation 5.53, the followings are defined:

$$\begin{aligned} \text{CONTROL}_3 \quad &\widehat{=} \quad (\text{DATALOG}_3 \,||\, \text{TRAJGEN}) \\ \\ \text{DATALOG}_3 \quad &\widehat{=} \quad \mu Z \bullet (\mu X \bullet (\mathit{Trajdata} \rightarrow X \;\Box\; \mathit{Ddata} \rightarrow \mathit{SKIP}))\,; \\ &\qquad\qquad (\mu Y \bullet (\mathit{Logdata} \rightarrow Y \;\Box\; \mathit{Ddata} \rightarrow Y \;\Box\; \mathit{Trajdata} \rightarrow \mathit{SKIP}))\,; \\ &\qquad\qquad Z \\ \\ \text{TRAJGEN} \quad &\widehat{=} \quad \mu X \bullet (\mathit{Opout} \rightarrow \mathit{Trajdata} \rightarrow X \\ &\qquad\qquad \mid \mathit{Logdata} \rightarrow X) \end{aligned}$$

To Prove:

CONTROL$_3$ is deadlock free

Proof:

The proof strategy follows that from Proof 5.7 by considering the refusals of CONTROL$_3$ in a number of cases. In order to apply the two theorems derived in Sub-section 4.5.2 for parallel processes, the relationship between the alphabet set from each of the two sub-systems are laid down as follows:

$$\alpha\,(\text{TRAJGEN}) \;\widehat{=}\; \{\mathit{Opout},\ \mathit{Trajdata},\ \mathit{Logdata}\} \tag{5.65}$$

$$\alpha\,(\text{DATALOG}_3) \;\widehat{=}\; \{\mathit{Trajdata},\ \mathit{Ddata},\ \mathit{Logdata}\} \tag{5.66}$$

$$\alpha\,(\text{TRAJGEN}) \cap \alpha\,(\text{DATALOG}_3) \;\widehat{=}\; \{\mathit{Trajdata},\ \mathit{Logdata}\} \tag{5.67}$$

Case: $s = \langle\rangle$

$$
\begin{array}{lll}
 & \textsc{Control}_3 \textbf{ sat } S_1(s, ref) & \\
\text{where} & S_1 \mathrel{\widehat{=}} \{ref = X \cup Y \mid X \in refusals\,(\textsc{TrajGen}) & \\
 & \qquad \wedge\ Y \in refusals\,(\textsc{DataLog}_3) & \text{refusals for } \| \\
 & \qquad \wedge\ \{Opout,\ Logdata\} \not\subseteq X & \text{def of } \textsc{TrajGen} \\
 & \qquad \wedge\ \{Trajdata,\ Ddata\} \not\subseteq Y & \text{def of } \textsc{DataLog}_3 \\
 & \qquad \wedge\ s = \langle\rangle\} & \\
\Rightarrow & \{Opout,\ Ddata\} \not\subseteq refusals\,(\textsc{Control}_3) & \text{Theorem 4.2 and 5.67} \\
\Rightarrow & S_1(s, ref) &
\end{array}
$$

Case: $s_0 = \langle Opout \rangle$

$$
\begin{array}{lll}
 & s_0 = \langle Opout \rangle \wedge s' = \langle\rangle & \text{from assumption} \\
 & \textsc{Control}_3 \textbf{ sat } S_2(s, ref) & \\
\text{where} & S_2 \mathrel{\widehat{=}} \{ref = X \cup Y \mid X \in refusals\,(\textsc{TrajGen}) & \\
 & \qquad \wedge\ Y \in refusals\,(\textsc{DataLog}_3) & \text{refusals for } \| \\
 & \qquad \wedge\ \{Trajdata\} \not\subseteq X & \text{def of } \textsc{TrajGen} \\
 & \qquad \wedge\ \{Trajdata,\ Ddata\} \not\subseteq Y & \text{def of } \textsc{DataLog}_3 \\
 & \qquad \wedge\ s = \langle Opout \rangle\} & \\
\Rightarrow & \{Ddata,\ Trajdata\} \not\subseteq refusals\,(\textsc{Control}_3) & \text{Theorem 4.2 and 5.67} \\
\Rightarrow & S_2(s, ref) &
\end{array}
$$

Case: $s_0 = \langle Ddata \rangle$

$$
\begin{array}{lll}
 & s_0 = \langle Ddata \rangle \wedge s' = \langle\rangle & \text{from assumption} \\
 & \textsc{Control}_3 \textbf{ sat } S_3(s, ref) & \\
\text{where} & S_3 \mathrel{\widehat{=}} \{ref = X \cup Y \mid X \in refusals\,(\textsc{TrajGen}) & \\
 & \qquad \wedge\ Y \in refusals\,(\textsc{DataLog}_3) & \text{refusals for } \| \\
 & \qquad \wedge\ \{Opout,\ Logdata\} \not\subseteq X & \text{def of } \textsc{TrajGen} \\
 & \qquad \wedge\ \{Trajdata,\ Ddata,\ Logdata\} \not\subseteq Y & \text{def of } \textsc{DataLog}_3 \\
 & \qquad \wedge\ s = \langle Ddata \rangle\} & \\
\Rightarrow & \{Opout,\ Ddata,\ Logdata\} \not\subseteq & \\
 & \qquad refusals\,(\textsc{Control}_3) & \text{Theorem 4.2 and 5.67} \\
\Rightarrow & S_3(s, ref) &
\end{array}
$$

The following two cases are similar in nature, and only the proof for the first case is presented here.

Case: $s = \langle Opout,\ Ddata \rangle \frown s''$

Case: $s = \langle Ddata,\ Opout \rangle \frown s''$

$s \geq \langle Opout,\ Ddata \rangle$ — first case
CONTROL$_3$ **sat** $S_4(s, ref)$
where $S_4 \mathrel{\widehat{=}} \{ref = X \cup Y \mid X \in refusals\ (\text{TRAJGEN})$
$\wedge\ Y \in refusals\ (\text{DATALOG}_3)$ — refusals for ||
$\wedge\ \{Trajdata\} \not\subseteq X$ — def of TRAJGEN
$\wedge\ \{Trajdata,\ Ddata,\ Logdata\} \not\subseteq Y$ — def of DATALOG$_3$
$\wedge\ s = \langle Ddata,\ Opout \rangle \frown s''\}$
$\Rightarrow$ $\{Ddata,\ Trajdata\} \not\subseteq refusals\ (\text{CONTROL}_3)$ — Theorem 4.2 and 5.67
$\Rightarrow$ $S_2(s, ref)\ [s/s'']$

Case: $s = \langle Ddata,\ Logdata \rangle \frown s''$

$s \geq \langle Ddata,\ Logdata \rangle$
CONTROL$_3$ **sat** $S_5(s, ref)$
where $S_5 \mathrel{\widehat{=}} \{ref = X \cup Y \mid X \in refusals\ (\text{TRAJGEN})$
$\wedge\ Y \in refusals\ (\text{DATALOG}_3)$ — refusals for ||
$\wedge\ \{Opout,\ Logdata\} \not\subseteq X$ — def of TRAJGEN
$\wedge\ \{Trajdata,\ Ddata,\ Logdata\} \not\subseteq Y$ — def of DATALOG$_3$
$\wedge\ s = \langle Ddata,\ Logdata \rangle \frown s''\}$
$\Rightarrow$ $\{Opout,\ Ddata,\ Logdata\} \not\subseteq$
$refusals\ (\text{CONTROL}_3)$ — Theorem 4.2 and 5.67
$\Rightarrow$ $S_3(s, ref)\ [s/s'']$

Case: $s = \langle Opout,\ Trajdata \rangle \frown s''$

$s \geq \langle Opout,\ Trajdata \rangle$
CONTROL$_3$ **sat** $S_6(s, ref)$
where $S_6 \mathrel{\widehat{=}} \{ref = X \cup Y \mid X \in refusals\ (\text{TRAJGEN})$
$\wedge\ Y \in refusals\ (\text{DATALOG}_3)$ — refusals for ||
$\wedge\ \{Opout,\ Logdata\} \not\subseteq X$ — def of TRAJGEN
$\wedge\ \{Trajdata,\ Ddata\} \not\subseteq Y$ — def of DATALOG$_3$
$\wedge\ s = \langle Opout,\ Trajdata \rangle \frown s''\}$
$\Rightarrow$ $\{Opout,\ Ddata\} \not\subseteq refusals\ (\text{CONTROL}_3)$ — Theorem 4.2 and 5.67
$\Rightarrow$ $S_1(s, ref)\ [s/s'']$

And finally for the following case with non-repeating events:

Case: $s = \langle Opout,\ Ddata,\ Trajdata \rangle \frown s'''$

$$
\begin{array}{lll}
 & s \geq \langle Opout,\ Ddata,\ Trajdata \rangle & \\
 & \textsc{ControlL}_3 \textbf{ sat } S_7(s, ref) & \\
\text{where} & S_7 \mathrel{\widehat{=}} \{ ref = X \cup Y \mid & \\
 & \qquad X \in refusals\ (\textsc{TrajGen}) & \\
 & \qquad \wedge\ Y \in refusals\ (\textsc{DataLog}_3) & \text{refusals for } \| \\
 & \qquad \wedge\ \{Opout,\ Logdata\} \not\subseteq X & \text{def of } \textsc{TrajGen} \\
 & \qquad \wedge\ \{Logdata,\ Trajdata,\ Ddata\} \not\subseteq Y & \text{def of } \textsc{DataLog}_3 \\
 & \qquad \wedge\ s = \langle Opout,\ Ddata,\ Trajdata \rangle \frown s'' \} & \\
\Rightarrow & \{Opout,\ Ddata,\ Logdata\} \not\subseteq & \\
 & \qquad refusals\ (\textsc{ControlL}_3) & \text{Theorem 4.2 and 5.67} \\
\Rightarrow & S_3(s, ref)\ [s/s''] &
\end{array}
$$

The results from the case analysis has established deadlock freedom in the re-designed sub-system ($\textsc{DataLog}_3$ || $\textsc{TrajGen}$), and this completes the proof. □

5.5.4.3 Remarks

It is important to note that the two solutions presented in this section are by no means the only solutions for solving the problem of deadlock. Nevertheless, the examples given have illustrated the following:

- The utilisation of the unified methodology and its corresponding tool set in face of problem solving, and in this case — the avoidance of potential deadlocks.
- Two of the system re-design techniques which are based on both formal mathematics as well as the creative re-interpretation of original system requirements.
- The use of *refusals* properties in the verification of deadlock freedom.

5.6 Higher Level Models

As the unified methodology supports the philosophy of a top-down hierarchical design and development, higher level models of the robot system describing different hierarchies of the control architecture with different levels of abstraction can be obtained by repeating the steps of refinement, formalisation and verification on the sub-systems presented in this case study. Each of the higher level models is a refinement from a lower level model which contains more information on the final implementable form and yet preserving the following two properties:

- They are logically consistent with the original design so that system integrity is maintained.

- These models must satisfy the original essential system specifications.

In the previous sections, an in-depth development from the essential model to the Zeroth Level Model; and then to the First Level Model of the robot control system has been discussed. The operations have outlined and illustrated the basic principals of system design and development of the unified methodology. Although system development does not stop at this stage, in the context of the thesis, the development of higher level models for this particular robot control system will not be further elaborated. Nonetheless, a few suggestions for further refinement and development are given briefly for the First Level sub-systems in the following sections so as to pave the way for a full and correct implementation.

5.6.1 Trajectory Generator: Refinement

Since the primary function of the TRAJECTORY GENERATOR is to generate trajectories for the controller, a possible refinement of TRAJGEN is to specify the trajectory generating function TRAJGENFUNC as a recurrence equation which will lead to a logical realisation using the generic process *FSM*, such as the one given in Equation 5.68.

$$\text{TRAJGENFUNC} \;\widehat{=}\; FSM\ (Indata.e,\ Outdata.f(e)) \tag{5.68}$$

where *Indata*, *Outdata* are channels for data input and output
and $f(e)$ is the trajectory generating function for the given set of input data e

Further application of DARTS system structuring criteria suggests that TRAJGEN can be broken down into a number of modularised processes, each with a well defined functionality. These decomposed processes are consequences of the functional cohesion and input/output orientation criteria. A pictorial representation of this refinement is shown in Figure 5.10.

Each of these functional blocks is drawn according to a corresponding generic process which formalises the realisation which is amenable to subsequent verification. These corresponding generic processes are given by the following equations with their refinement criteria given in Table 5.7.

$$\begin{aligned}\text{TRAJGEN} \;\widehat{=}\; &(FSM_{TG-1} \;||\; FSM_{TG-2} \;||\\ &PERM_{TG-INT} \;||\; FSM_{TG-TGF})\end{aligned} \tag{5.69}$$

$$FSM_{TG-1} \;\widehat{=}\; FSM\ (Logdata.e_1,\ Inchan_1.e_1') \tag{5.70}$$

$$FSM_{TG-2} \;\widehat{=}\; FSM\ (Opout.e_2,\ Trajdata.e_3,\ Inchan_2.e_2') \tag{5.71}$$

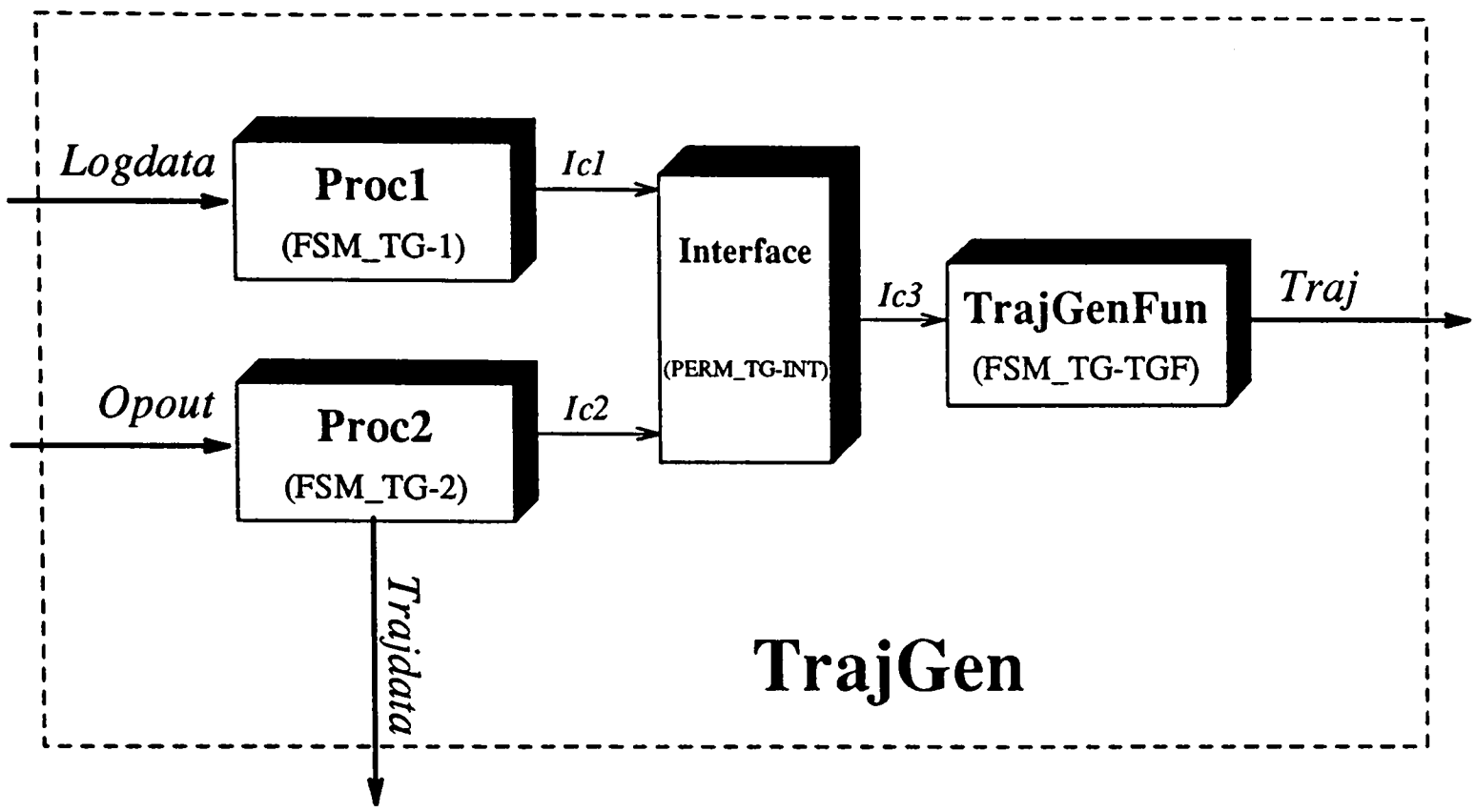


Figure 5.10: A refinement of the First Level sub-system TRAJGEN

$$\equiv \quad \mu X \bullet (Opout\,?\ e_2 \rightarrow Trajdata\,!\ e_3 \rightarrow Inchan_2\,!\ e_2' \rightarrow X)$$

$$PERM_{TG-INT} \ \widehat{=}\ PERM\ (Inchan_1.e_1',\ F,\ Inchan_2.e_2',\ F) \tag{5.72}$$

$$F \ \widehat{=}\ Inchan_3.e_3' \rightarrow SKIP \tag{5.73}$$

$$FSM_{TG-TGF} \ \widehat{=}\ FSM\ (Inchan_3.e_3',\ Traj.f(e_3')) \tag{5.74}$$

where $\begin{pmatrix} & e_1 \in LDATA \\ \wedge & e_2 \in VCOMMAND \\ \wedge & e_3 \in TDATA \end{pmatrix}$

and $Inchan_1$, $Inchan_2$ and $Inchan_3$ are arbitrary internal communication channels
e_1', e_2' and e_3' are messages corresponding to these arbitrary channels

5.6.2 Controller: Refinement

In this sub-section, a multi-level refinement for the sub-system CTL is described. From the corresponding specifications, an initial refinement is taken to separate the operation into one described by *MUX* and with the output from the *MUX* feeding into a recursive control algorithm specified using *FSM*. Such refinement will have a principle effect of separating the two functionalities within

Generic Process	DARTS Criteria
FSM_{TG-1}	I/O interface (DC–1 & DC–5)
FSM_{TG-2}	I/O interface (DC–1 & DC–5)
$PERM_{TG-INT}$	Functional cohesion (DC–4)
FSM_{TG-TGF}	Functional cohesion & I/O interface (DC–1, DC–4 & DC–5)

Table 5.7: Justifications by DARTS criteria on the corresponding identified processes

CTL, which might be useful during the stage of implementation. The two processes, DATAORG — a Data Organiser which is realised using *MUX* and CONTROLPROC, the Control Process, which is realised using *FSM* are shown pictorially in Figure 5.11.

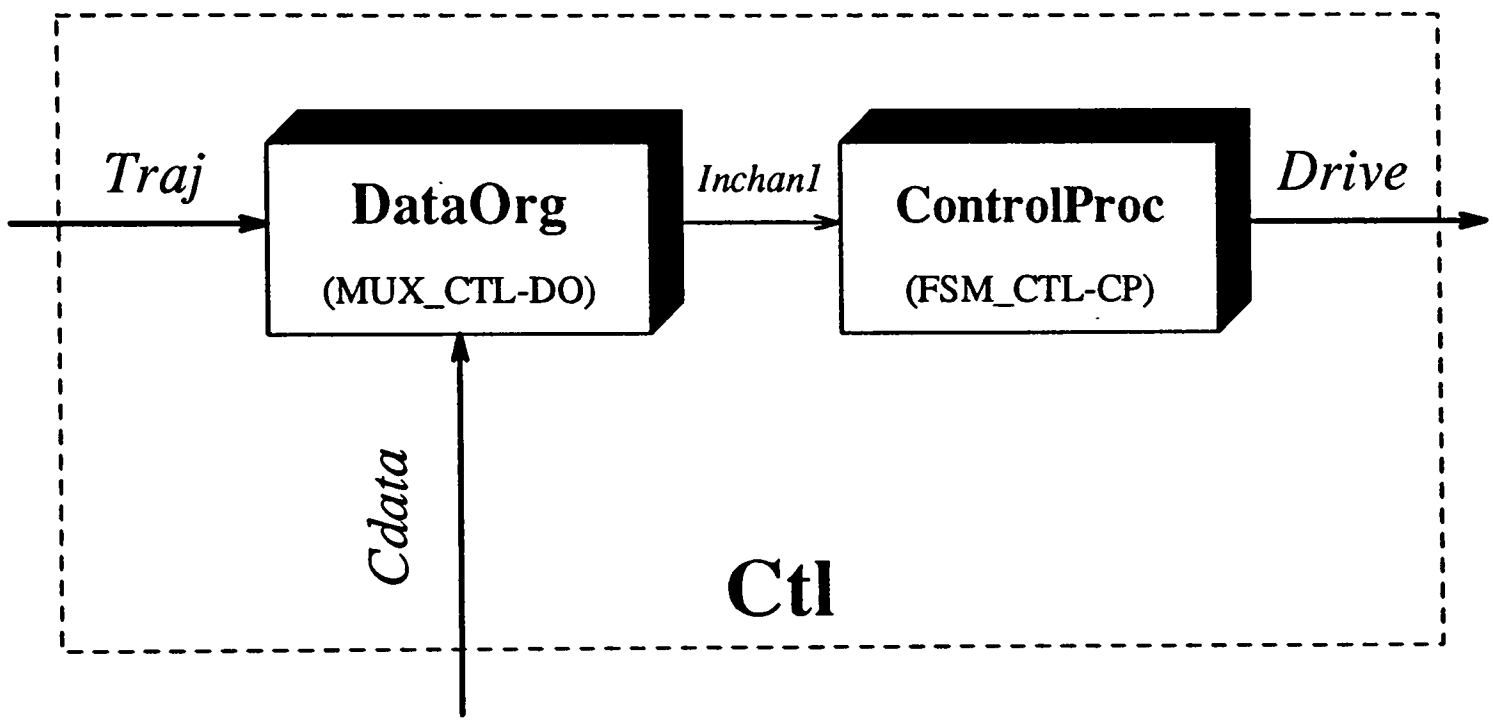


Figure 5.11: An initial refinement of the First Level sub-system CTL

The corresponding process definitions for the two parallel processes expressed as generic processes are given by Equations 5.76 and 5.77.

$$\text{CTL} \;\widehat{=}\; (MUX_{CTL-DO} \;||\; FSM_{CTL-CP}) \tag{5.75}$$

$$MUX_{CTL-DO} \;\widehat{=}\; MUX\ (Traj.e_1,\ Cdata.e_2,\ Inchan_1.f_1(e_1, e_2)) \tag{5.76}$$

$$FSM_{CTL-CP} \;\widehat{=}\; FSM\ (Inchan_1.f_2^{-1}(e_3),\ Drive.e_3) \tag{5.77}$$

$$\text{where} \quad \boxed{\begin{pmatrix} & e_1 \in TRAJ \\ \wedge & e_2 \in CDATA \\ \wedge & e_3 \in DEMAND \end{pmatrix}}$$

and $Inchan_1$ is an internal channel
f_1 is a data organisation function
f_2 is the control algorithm

Having considered the initial refinement of CTL, further development suggests that the process which formalises the control algorithm corresponding to the generic process FSM_{CTL-CP} consists of two main functions: a recursive controller by which new demands are evaluated; and an inverse dynamic generator by which robot parameters that are used in the control algorithm are updated [Paul, 1981]. On top of the two main functions, an additional desirable function of this control process is status reporting. As the requirement for the status reporting facility is non *time-critical*, which means that its communications with other sub-systems can proceed in an asynchronous way. Following these requirements, process FSM_{CTL-CP} is decomposed into three independent sub-processes, each with a unique functionality. In Table 5.8, the justification for the decomposition and formalisation using the corresponding generic processes are summarised.

Sub-process	Generic process realisation	DARTS criteria
FORCECTL	$FSM_{CTL-CP-FC}$	Functional cohesion (DC–4)
STATUSREP	$BUFFER_{CTL-CP-SR}$	Temporal consideration (DC–2)
INVDYNGEN	$PERM_{CTL-CP-IDG}$	Computation Intensity (DC–3)

Table 5.8: A summary of the corresponding identified and formalised sub-process for the process CONTROLPROC

An abstraction from the real controller requirements defines the functionalities and behaviours of the followings three decomposed sub-processes.

1. FORCECTL inputs the generated trajectory, initiates a request for parameter update for the robot model, receives the updated robot parameters from INVDYNGEN, then calculates the demands and stores them with STATUSREP.

2. STATUSREP is always ready to receive the calculated demands from FORCECTL unless its storage is full, and communicates asynchronously with the external sub-systems.

3. INVDYNGEN computes the robot parameters from the input robot data. Once a request is received from FORCECTL, the computed parameters are sent. When no request is received, the sub-process is always ready to receive robot data.

According to these requirements, corresponding generic processes are assigned, and are given in the following equations with the corresponding messages communicated across channels being abstracted from the process algebra.

$$\text{CONTROLPROC} \;\widehat{=}\; (\text{FORCECTL} \,||\, \text{INVDYNGEN} \,||\, \text{STATUSREP}) \tag{5.78}$$

$$\textsc{ForceCtl} \;\widehat{=}\; FSM_{CTL-CP-FC}\,(Traj,\ Req,\ Mat,\ Drive,\ Sin) \tag{5.79}$$

$$FSM_{CTL-CP-FC} \;\widehat{=}\; \mu X \bullet (Traj \rightarrow Req \rightarrow Mat \rightarrow Drive \rightarrow Sin \rightarrow X) \tag{5.80}$$

$$\textsc{InvDynGen} \;\widehat{=}\; \mu X \bullet (PERM_{CTL-CP-IDG}\,(Cdata,\ P,\ Req,\ Q)\ ;\ X) \tag{5.81}$$

$$P \;\widehat{=}\; \text{“}store\text{”} \rightarrow SKIP \tag{5.82}$$

$$Q \;\widehat{=}\; \text{“}calc\text{”} \rightarrow Mat \rightarrow SKIP \tag{5.83}$$

$$\textsc{StatusRep} \;\widehat{=}\; BUFFER_{CTL-CP-SR}\,(Sin,\ Sout,\ n) \tag{5.84}$$

where “*store*” and “*calc*” are intrinsic operations which are unspecified

A pictorial mapping of this refinement of the process ControlProc is shown in Figure 5.12.

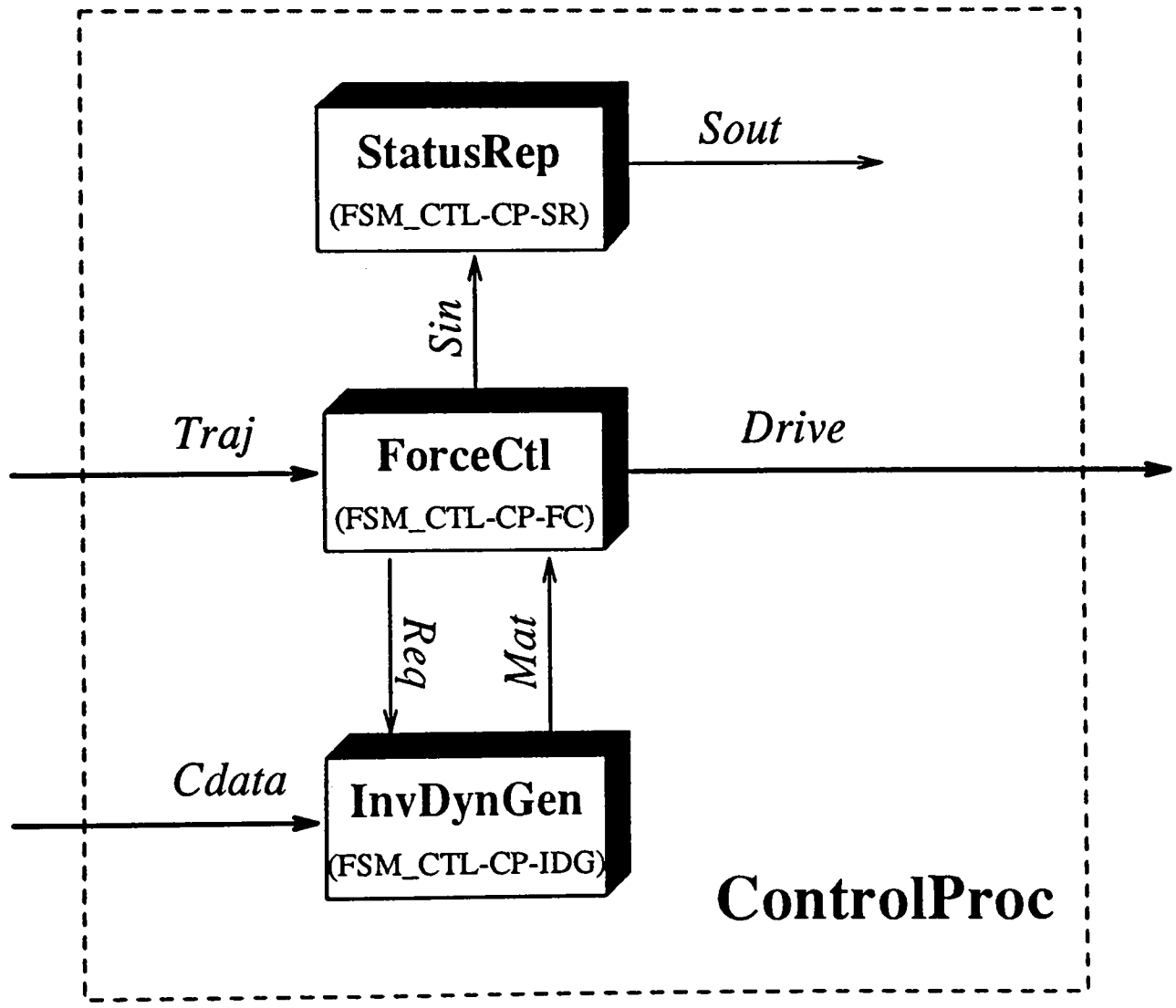


Figure 5.12: A refinement of the process ControlProc

Having described the two stages of refinement for the sub-system Ctl, further refinements of the sub-process ForceCtl is possible which will have the effect of formalising the controller down to an algorithmic level. Explicit recurrence equations defining the exact functionalities for the

generation of control parameters can be modelled from which a direct implementation is possible. To illustrate this development, Equation 5.79 is decomposed into a number of modules which can be realised using the class of generic process *FSM* as described in the following equations. Here, only the particular generic process $GP_{CTL-C-FC-i}$, which describes the control algorithm is considered. According to the utilisation of generic process for formalising the particular PID control algorithm discussed in Section 4.4.3.4, a similar approach is applied to this case and a possible refinement of the control algorithm used is given by Equation 5.87.

$$\text{FORCECTL} \;\widehat{=}\; FSM_{CTL-CP-FC}\,(Traj,\ Req,\ Mat,\ Drive,\ Sin) \tag{5.85}$$

$$\equiv\; (GP_{CTL-CP-FC-1} \,||\, GP_{CTL-CP-FC-2} \,||\, \cdots \,||\, GP_{CTL-CP-FC-(n-1)} \,||\, GP_{CTL-CP-FC-n}) \tag{5.86}$$

$$GP_{CTL-CP-FC-i} \;\widehat{=}\; FSM\ (x.e,\ y.f(e)) \tag{5.87}$$

where $f(e)$ is a function for the set of input data e which describes the action of the control algorithm expressed in the form of Equation 4.20

5.6.3 Remarks

The best way to conclude the process of system development through refinement is by referring to Figure 5.13. In this figure, the process of a step-wise refinement of the sub-system CTL is summarised by an "implosion" of the original entity into finer modules of different levels of abstraction. Through this process, the highly abstracted system realisation has continuously grown into a less abstracted system description with the introduction of an increasing amount of implementation detail. However, at every stage of system refinement, logical consistency and system integrity are upheld so that a correct implementation with respect to the original requirements is obtained. It is also noted that the hierarchical nomenclature of generic processes in all process definitions for system, sub-systems and all the way down to modules has the advantage of an unambiguous representation of the inter-relationship between generic processes used in the different control hierarchies. Furthermore, such a nomenclature scheme has exemplified the essence of an Object Orientated representation [Goguen and Tardo, 1979].

5.7 Properties with respect to Software Architecture

In this case study, a detailed discussion of the design and development of a real robot control system is presented. The unified methodology presented in Sub-section 4.5.1 has been followed

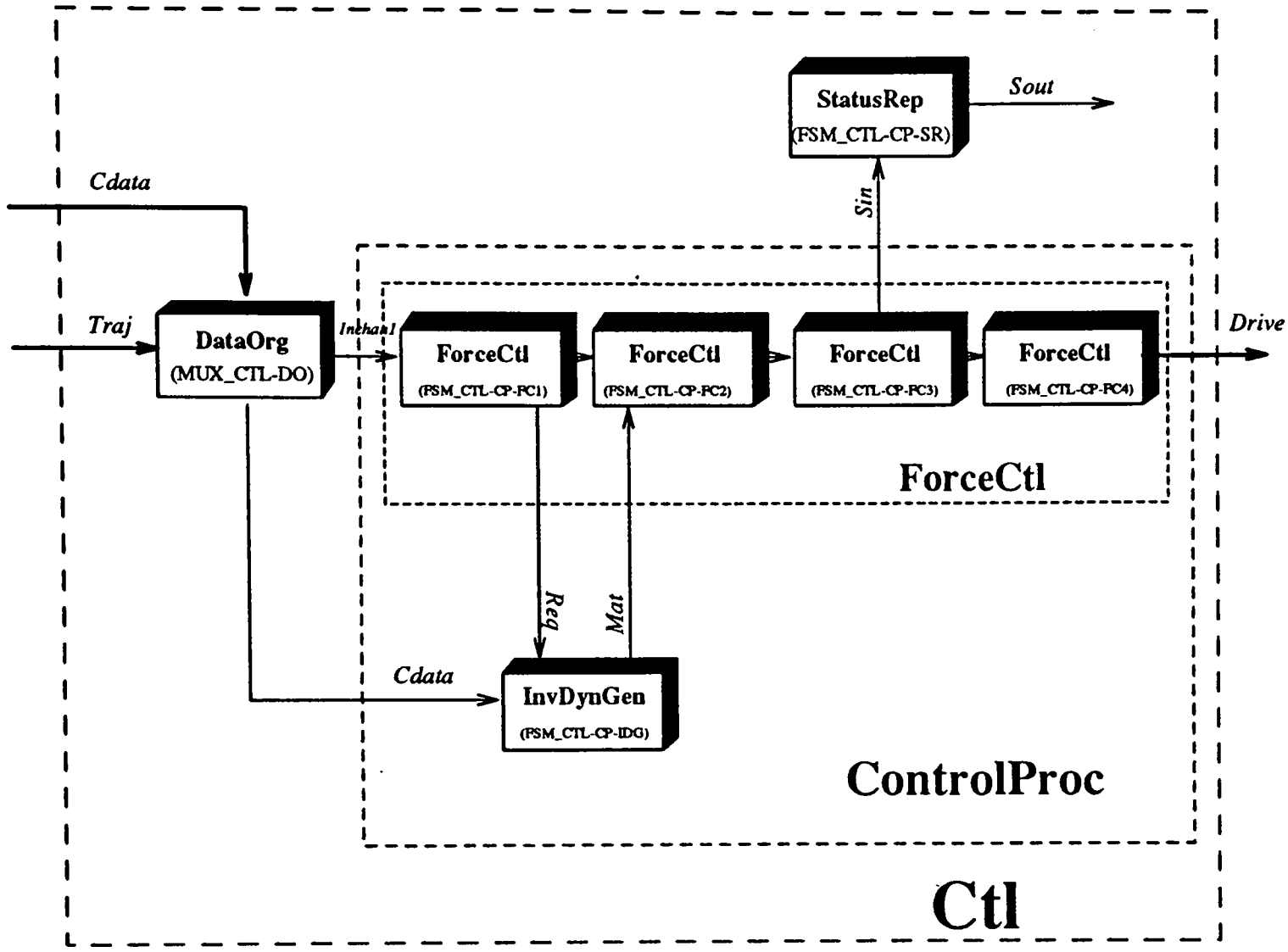


Figure 5.13: An "imploded" view of system refinement

closely in the various stages of the software development starting from a set of informal essential model requirements. Through the detailed transformation from requirements to a formal Zeroth Level Model and then via subsequent refinements into the First Level Model, the following aspects of the methodology are illustrated:

- Formalisation of essential requirements into safety and liveness conditions using the specification predicates as described in Section 4.3.4.1.
- Realisation using the set of generic processes from the corresponding specifications in various levels of abstraction.
- Formal mapping of generic process descriptions into pictorial representations such that structural information of the evolving system designs are captured.
- Reasoning with the correctness of the realised systems with respect to the laid down specifications using the defined laws and theorems established in this thesis and in Hoare [1985b].

- Refinement and structuring of the system and its components by following the criteria of DARTS within different levels of the control hierarchies.

- Strategies for system re-design in order to circumvent design errors such as the case of a deadlock, which is based on the application of formal mathematics as well as and a more intuitive approach.

By taking a closer look at the different software models developed, it is observed that there is a gradual transformation in the nature of system realisations. It is seen that the Zeroth Level Model presented in Section 5.4 is a *behavioural orientated* description of the system, and as the process of refinement proceeds, such an orientation is gradually shifting towards a more *functional* description in which system operations and functionalities are specified. This observation is reflected in the Higher Level Models of system realisation such as the examples given in Section 5.6.

Although the proposed methodology is built on the philosophy of "correctness by construction", in the design of the First Level Model, a potential deadlock between the composite sub-systems (TRAJGEN || DATALOG) was identified in the process of verification. However, the scenario is not due the lack of formality in the abstraction tool set, but is a result of the creative nature of the design process. Nevertheless, the use of the abstraction tool set to formalise the design enabled these design errors, such as communication deadlocks, to be predicted, identified and investigated at an early stage. Appropriate corrections and modifications of the design can then be carried out at a level of abstraction such that both time and cost are minimised. However, it is necessary to emphasis that in reality, the process of system design and development is an iterative process by which subsequent designs and the safety part of the requirements are sometimes modified continuously as the process proceeds so that improvements on both the designs and requirement can be made. All in all, formal mathematics can only be used to prove the correctness of what is being specified.

Finally, it is seen that the process of proof construction is vital to predict properties of formally specified systems, and to a more rigorous extent, to establish the correctness of a system realisation. However, from the proofs constructed in this chapter, it is shown that this process is non-trivial, lengthy and sometimes repetitive. It is therefore beneficial to have some form of automated or semi-automated proof assistant to lessen the burden on designers in proof construction when using formal mathematics. In the next chapter, a simple yet novel proof assistant is presented for the simulation and display of CSP processes in action. To illustrate the usefulness of this automated proof assistant, a number of the sub-systems and process realisations discussed in this chapter are

investigated using this proof assistant. A number of these results are presented in the next chapter and a comparison with those obtained in this chapter is made.

Chapter 6

Simulation and Implementation

6.1 Introduction

From the discussion in Chapter 2 it is known that many formal methods exist for proving the correctness of specifications and software so as to establish logical consistency among parallel processes, to identify system properties and to predict system behaviours. However, little is available to help an engineer to evaluate high level specifications during the earliest stages of a design; such as those found in the Zeroth and First Level Models expressing the essential system requirements of the robot control software.

Being part of the unified methodology and aimed at assisting formal reasoning in general, this chapter presents the **Causality Diagram Evaluation Tool (CADET)** which is primarily designed for the prototyping of highly abstracted system descriptions constructed using the CSP abstraction tool set. This evaluation tool is designed with a view to simulate the interactions between concurrent processes and to extract properties from these co-operating processes through records of their traces and refusals in both textual and graphical forms. As such, this evaluation tool can be used as a proof assistant in the unified methodology for reasoning about the correctness of concurrent processes, and as a consequence, relieve the burden of verification from engineers. In the following sections, the design and utilisation of this tool on the robot control systems software described in the last chapter is discussed.

As said, the ultimate goal of software development is to produce correct executable software, the chapter concludes with the discussion of the possibility of a direct implementation of the set of generic processes which forms the formal building blocks for concurrent systems. The transformation of these processes from highly abstracted definitions to the practical parallel programming language OCCAM is demonstrated which is contrasted with the simulation using CADET. All in all, with the already specified aim, the provision of a direct implementation of generic processes

with parallel languages makes a fully proven system closer to reality.

6.2 The Need for Machine Assisted Verification Tools

From of the various proofs constructed in the last chapter, formal verification of CSP specifications is a non-trivial, complex and laborious task. This is clearly reflected in the length and incompleteness in some of the cases. Most often, the process requires a thorough understanding and knowledge of the subject and verifying the correctness of a simple system can often be a major undertaking, examples such as those given in Sub-section 4.4.1 and Appendix D are common. However, the outcome of a proof is often needed to determine the correctness and to predict the behaviours of a specified system. Moreover, constructing formal proofs manually is susceptible to human error. These disadvantages have caused the unpopularity and reluctance of use expressed by engineers when asked to carry out their designs using a technique which is related to formal specification and/or verification.

One solution to overcome these problems is to investigate the possibility of developing proof assistants. Systems such as HOL [Gordon, 1987; Camilieri *et al.*, 1986] have been developed for machine assisted and interactive proof construction mainly for hardware designs. Theorem provers such as the B Tool [Abrial, 1988] was developed with applications to systems specified in Z. Graphical proof assistants such as the Edinburgh Concurrent Workbench [Cleaveland *et al.*, 1989] which is a system designed for the manipulation and analysis of concurrent systems based on the formalism of CCS [Milner, 1980]; the INRIA tools AUTO [de Simonbe and Vergamini, 1989] for machine assisted analysis of concurrent finite state machines and AUTOGRAPH [Roy and de Simone, 1989] which provides a graphical interface to facilitate process input and displaying results for the AUTO system.

Other automated or semi-automated verification systems developed for different levels of abstraction of software include the *SpecBox* [Bloomfield *et al.*, 1989] and other supporting systems for MASCOT [Simpson and Jackson, 1979; Foulkes and Deitch, 1982]. These automated tools enable designers to explore the properties and behaviours of high level specifications and to identify a few promising approaches for subsequent more detailed and rigorous development. Their judicious use offers a relatively cheap and quick way of testing initial ideas and locating errors at an early stage which often saves time and money [Luk, 1988]. The techniques of using automated proof assistants and simulation aids that are based on some formal systems for software development are called — **automated software prototyping**.

The novelty of automated software prototyping is in most cases, rather than to directly prove that a particular implementation is correct with respect to a set of specifications, its utilisation provides a preliminary semi-formal reasoning technique for the implementation and specifications concerned; more formal arguments can then follow. Automated software prototyping can also increase the accuracy of a design and helps to iron out logical in-consistencies as well as syntactical errors.

There have already been some suggestions by Roscoe and Dathi [1986], Jeffrey [1988a] and Davies [1987] on how to produce theorem provers for CSP. With this need, a simple but useful simulation tool for prototyping CSP process has been developed to complement the use of formal reasoning in the application of the unified methodology. In the following sections, the design and application of the Causality Diagram Evaluation Tool is discussed.

6.3 The Design of the Causality Diagram Evaluation Tool

The *Causality Diagram Evaluation Tool* or CADET has been developed to allow CSP processes, both sequential and concurrent, to be built, simulated and displayed. This tool in particular, provides a way for a designer to extract and display properties of a design structure using generic processes at a high level of abstraction. The tool is capable of extracting the following information from a formally specified system:

- Time histories of a system.
- Sequentiality of communicating events among a number of interacting processes.
- Parallel interactions between processes.
- Failures of processes; such as deadlocks.

Using the simulation results obtained from CADET as a prelude together with more formal reasoning on system properties, the correctness of behaviours for a given system with respect to the original specification can be checked. Before illustrating the usages of this tool, the background of the design and structure of the tool is discussed.

CADET is a prototype of a window based Computer Aided Software Engineering (CASE) tool. Structurally, it consists of two main functional sub-systems: a process algebra re-writing machine called the CSP interpreter and a user interface which provides a reasonably user friendly inputting device, a trace analyser and a graphical front end for displaying process interactions. Figure 6.1

shows the architecture of the tool. The two basic functional sub-systems as shown in this figure communicates with each other using a file protocol by which information such as traces, refusals and process definitions are communicated. The main window of the tool in operation is shown in Figure 6.2. In the following sections, the design of these two functional sub-systems are described.

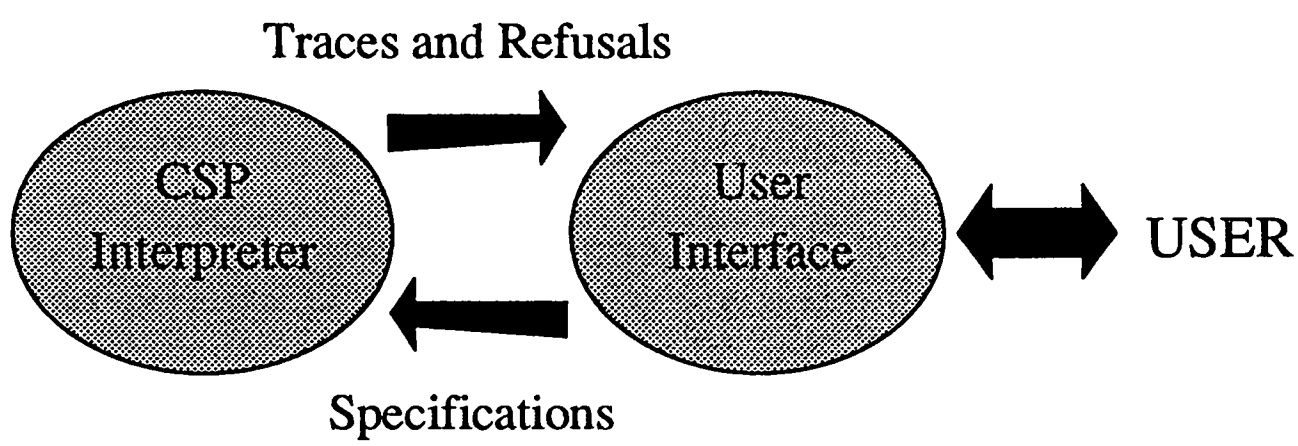


Figure 6.1: The architecture of the causality evaluation tool

6.3.1 The CSP Interpreter

The CSP interpreter is an automatic re-writing machine which attempts to implement CSP processes dynamically so that systems expressed in CSP notations such as those structured using generic processes given in Chapter 5 can be simulated and 'run'. The present system is built on top of the original implementation by Jeffrey [1988b] with the semantics following very closely those presented by Hoare [1985b], Brookes and Hoare [1984]. It is implemented in *Orwell* [Wadler and Miller, 1988; Miller, 1990] — a functional language which has the advantage of simple definitions for recursion as most of the work has been taken care of by the language itself [Bird and Walder, 1988]. This implementation allows recursive processes to be built up easily and their traces to be examined interactively. Also, the nature of the language makes it possible for the interpreter to carry out symbolic manipulation efficiently.

It is important to note that the current implementation of the interpreter handles non-determinism by generating all the possible paths that a process may choose using a permutation on choices. Such an implementation allows the pursuit of deadlock with the aid of logged information such as "whether a process will deadlock on a particular trace" by the interpreter. However, with this implementation, as an exponential amount of non-determinism can be built up easily and will inevitably sacrifice the speed of execution. In order to mitigate against the effects of this drawback, the tool has to be applied in a judicious way and at an appropriate level of abstraction

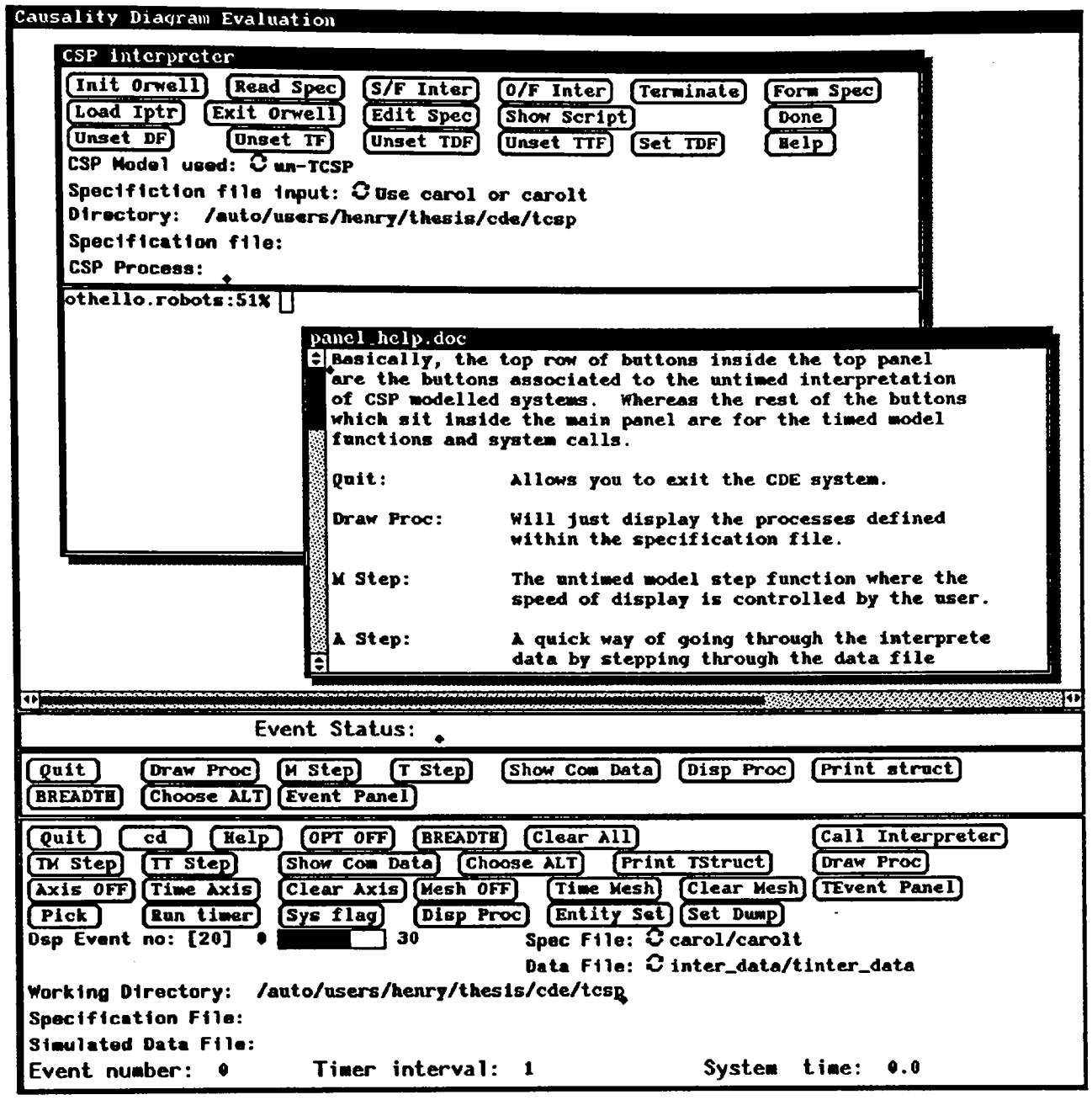


Figure 6.2: The main window of the Causality Diagram Evaluation Tool

so as to minimise the amount of redundant information produced. Nonetheless, the design of the trace analyser which will be discussed in the next section and as part of the user interface provides useful functions to assist users to examine the generated traces in a more efficient way. From a pragmatic point of view, it is known that the Orwell based interpreter should be fairly portable to commercial functional languages such as Miranda [Turner, 1987] for better efficiency.

Figure 6.3 shows the internal architecture of the CSP interpreter. The inputs to the interpreter are CSP processes written using the notations defined in Hoare [1985b] and the outputs are *traces* and *refusal sets*. Every output produced by the interpreter can be logged and subsequently communicated to the user interface for further analysis.

From Figure 6.3, the main body of the interpreter is a *non-alphabetised processor*. This processor is made up of a group of functions that perform process algebra manipulation onto non-alphabetised CSP processes according to the laws of CSP. A subset of the operators on CSP

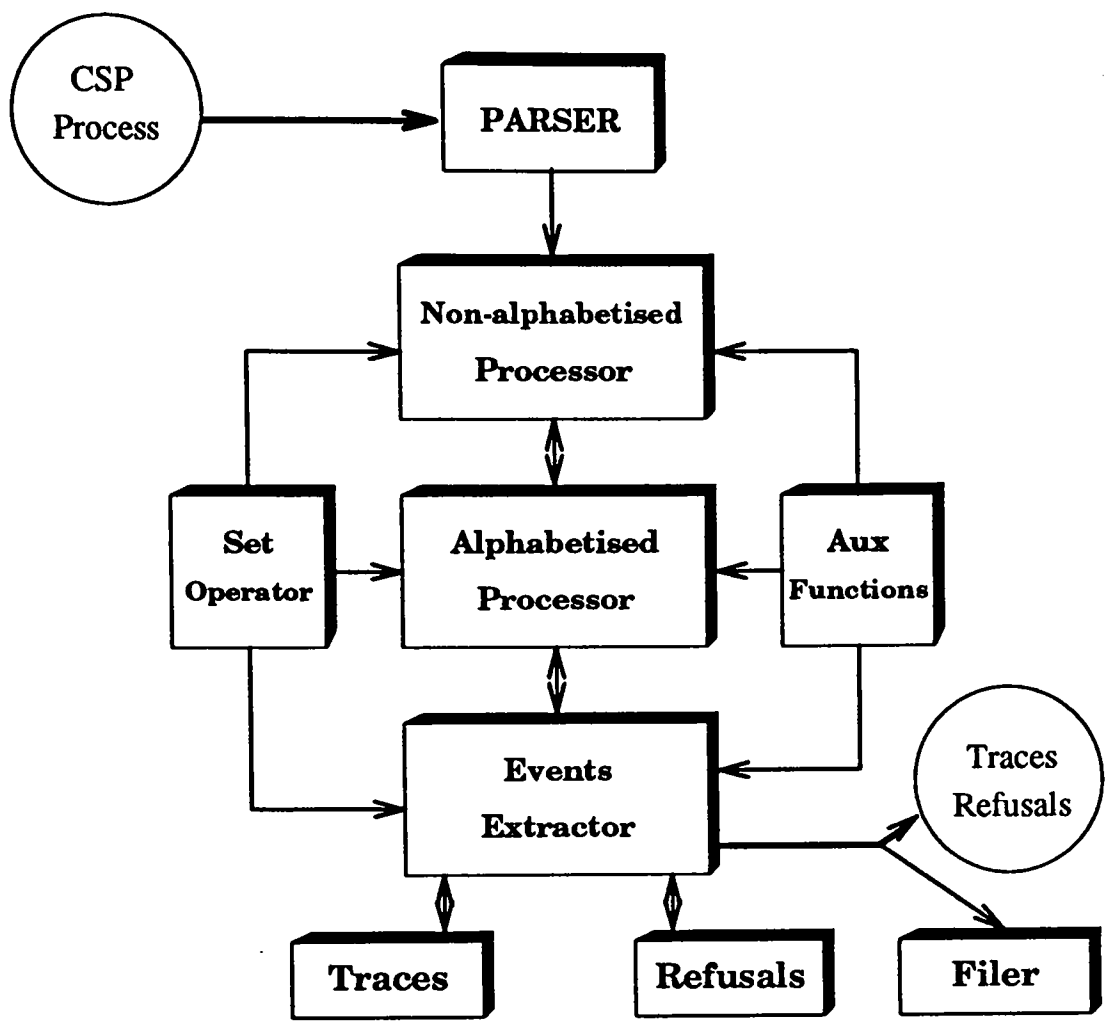


Figure 6.3: The structure of the CSP interpreter

processes given by Equation 2.9 are implemented by the interpreter and their syntax are given as follows:

$$P ::= \mathbf{STOP} \mid \mathbf{SKIP} \mid a \rightarrow P \mid c?x \rightarrow P \mid c!e \rightarrow P \mid$$
$$P_1 \sqcap P_2 \mid P_1 \Box P_2 \mid P_1 \|_A P_2 \mid P \backslash C \mid P/s \mid \mu X \bullet P$$

The functions that implement these operators operate on the data type "`proc' a`" which is written in Orwell as:

```
> proc' a ::= Nondet' (proc' a) (proc' a) |
>             Prefix' (set a) (a -> proc' a)
```

the expression has been translated from the definition of a non-diverging process that is written in the form:

$$P ::= P \sqcap P \mid a : A \rightarrow P(a)$$

Actions of a system which are modelled by discrete events are represented by the data type "`event a`" which is structured using the polymorphic data type [Bird and Walder, 1988] in Orwell:

```
> event a  :: =  Tick | Do [a]
```

of which `Tick` is the event which represents a successful termination and an ordinary communicating event is maintained using a *list of characters* preceded by the *type constructor* `Do`. The non-alphabetised processor therefore consists of a set of functions that operate on the polymorphic "`event a`" data type. These functions not only automate the manipulation of process algebra, but also take care of message passing between communicating events.

For the manipulation of alphabetised CSP processes such as the defined generic processes, the *alphabetised processor* is used. This consists of a set of higher order functions which operates on a new data type "`proc a`". The data type "`proc a`" is a pair made up of a *set of events* which represents the alphabet set of a particular process and the non-alphabetised process "`proc' a`". This new data type is written in Orwell as:

```
> proc a == (set a, proc' a)
```

This set of higher order functions operates on both the alphabet set as well as calling the appropriate non-alphabetised processor's functions to perform process operations. Since the functions form the alphabetised processor requires the manipulation of sets, a *set operator* is incorporated which consists of a group of functions designed to carry out set operations such as the union and intersection of sets of events.

These two processors together with the set operator make up the heart of the interpreter. In order to observe the interactions between the interpreted processes, information such as the following have to be extracted from the interacting processes:

- The alphabet set of processes.
- The initial action of a process.
- Traces and refusals.

To achieve this, a number of functions which form the *events extractor* is designed to extract these informations from the processes under interpretation for further analysis. These functions

also allow the extracted traces and refusals information to be displayed textually and to re-format these informations so that they are compatible with those used by the user interface.

In addition to the four main functional units, a *parser* has also been incorporated. This serves as a syntax checker for the input process definitions. Syntax errors such as communication contention as a result of more than one process trying to input or output on the same channel simultaneously is reported. Another functional group that the interpreter has in-built is the *filer*. The filer is primarily used for storing the traces and refusals informations produced by the interpreter into the Unix file system so that hardcopy of this information is made available to both designers as well as the user interface. Finally, a set of *auxiliary functions* is designed to handle miscellaneous operations such as the operation on data structures within the interpreter. As the main aim of the interpreter is to automate the manipulation of process algebra to assist verification, the resultant traces and refusals produced by the interpreter will serve as vital information for subsequent analysis.

6.3.2 The User Interface

The primary motivation of the development of the user interface is to assist designers to analysis the bulk of information produced by the interpreter. It is found that using the CSP interpreter alone, a designer will already be armed with a powerful tool to automate the manipulation of CSP processes. However, the output of the interpreter is textual and further analysis will become inefficient and laborious especially in cases when processes involve a large amount of non-determinism. This prompts the design of a *trace analyser* in order to present and analyse the interpreter outputs in a more manageable way. Besides the trace analyser which forms the main part of the user interface, the user interface consists of an inputting device which facilitates the input and editing of processes to be interpreted as well as translating the CSP processes written as in Hoare [1985b] to a form which is readable by the interpreter. Additionally, a graphical front end is incorporated to display the traces so that process interactions through synchronous communication are shown. Figure 6.4 shows the structure of the user interface. The interface is written in C which communicates with the CSP interpreter via a file protocol. By using the interpreter in conjunction with the user interface a designer is provided with an instant pictorial view of concurrent process interactions.

The functionalities for the three functional units shown by the figure are described briefly in the following sub-sections.

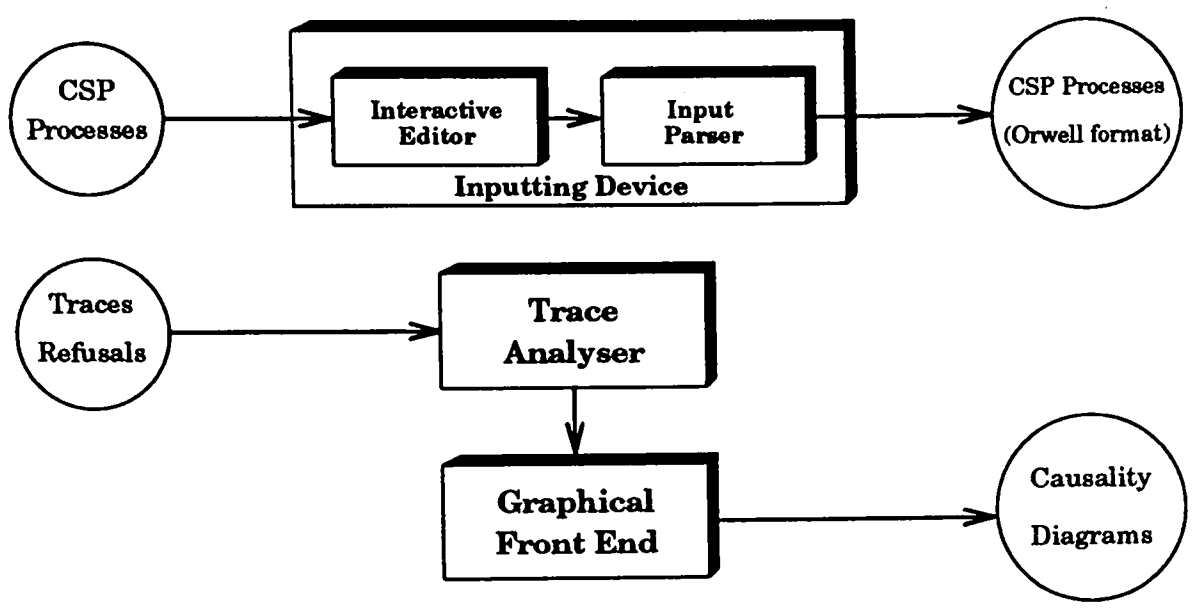


Figure 6.4: The structure of the User Interface

6.3.2.1 The Inputting Device

The inputting device provides a media for a designer to enter CSP processes with different levels of abstraction into CADET. The editor front end allows a designer to modify the design iteratively whereas the input parser translates the standard CSP processes into a format which is readable by the functions within the interpreter.

6.3.2.2 The Trace Analyser

The main function of the trace analyser is the provision of both *depth* and *breadth* first analysis on the logged traces resulting from deterministic and non-deterministic choices. A *breadth* first analysis allows all the possible evolutions for a process to be evaluated after a particular action. Whereas the *depth* first analysis enables a designer to trace specific time histories out of a set of concurrent processes by choosing particular guarding events from the corresponding prefixed processes. To provide such functions, the analyser organises the traces produced from the interpreter into a *tree* of events so as to allow both forward and backward referencing of events from a chosen origin. In a pragmatic point of view, any previous actions taken by a system which leads to a particular occurrence of a salient event can be tracked using the depth and breadth first analysis. As an example of use, a deadlock from a particular trace can be located and its cause is back tracked for correction.

6.3.2.3 The Graphical Front End

Traces produced by the interpreter are represented as *Causality Diagrams.* These are timing diagram like figures which are partial, and yet direct mappings of CSP processes and events. Each trace generated using the interpreter is represented uniquely by a causality diagram. A typical causality diagram which is generated from CADET corresponding to the CSP processes given by Equations 6.1 and 6.2 is shown in Figure 6.5.

$$P \;\widehat{=}\; \mu X \bullet (a\,?\,x \rightarrow b\,?\,y \rightarrow c\,!\,z \rightarrow X) \tag{6.1}$$

$$Q \;\widehat{=}\; \mu X \bullet (b\,!\,y \rightarrow c\,?\,z \rightarrow d\,!\,t \rightarrow X) \tag{6.2}$$

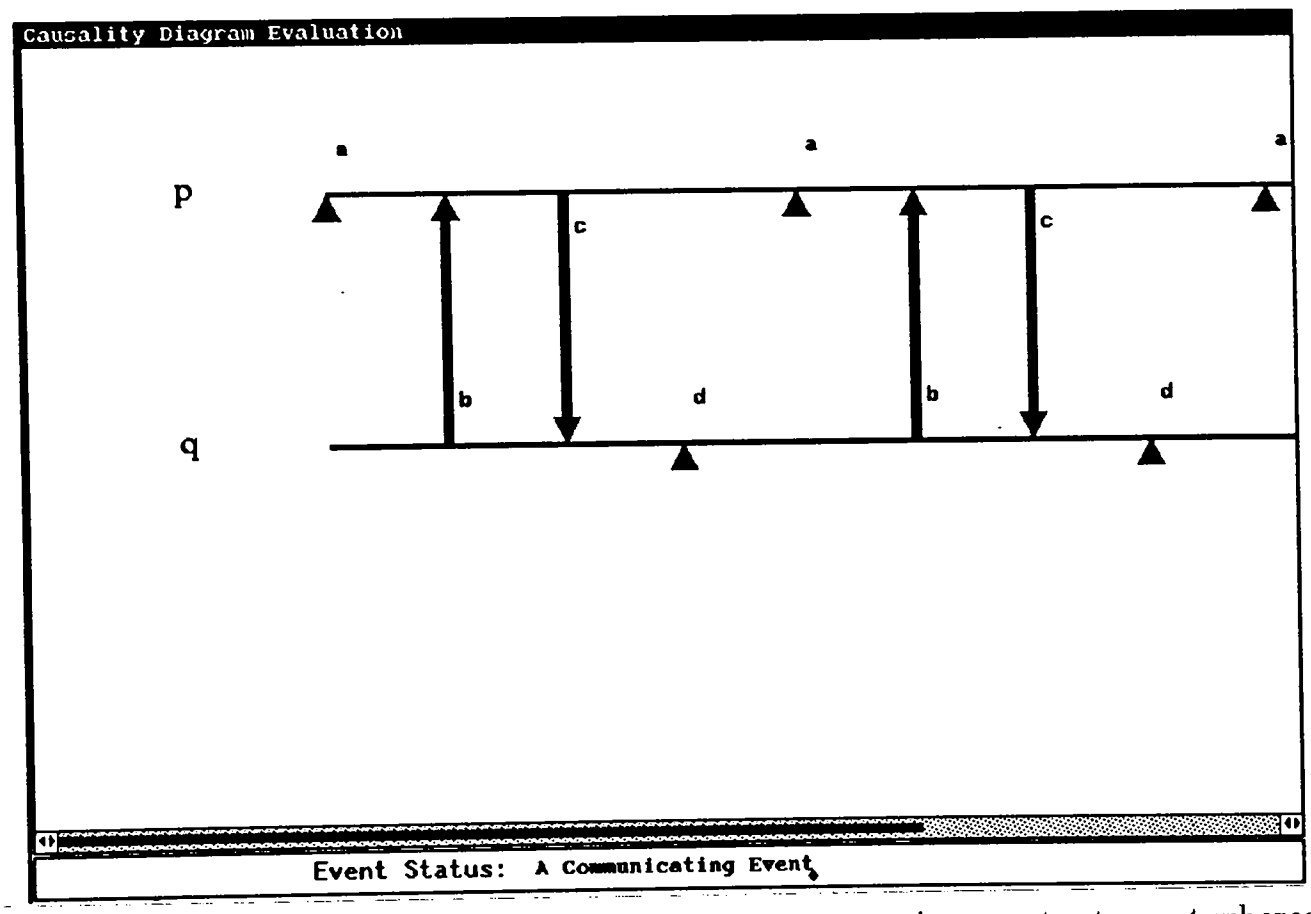


Figure 6.5: A typical Causality Diagram. Note that the event d is an output event whereas even a is an input event, and the two is not explicitly distinguished in the figure.

From this causality diagram, a number of main features can be identified. The time evolution of process P and Q are represented as horizontal axes with the progression of events starting from the left, showing the temporal ordering of communications. Each arrow on the diagram represents a discrete event and the direction of the arrow indicates the direction of data flow. An arrow between two processes signifies a synchronous communication has taken place between the two processes such as in the case of a communication along channel b. Whereas the free arrow-head labelled a on process P indicates this process has engaged in an interaction with its environment, and in this case an external input via channel a.

The notion of deadlock is represented as a possible "backward time dependency" between any

two communicating processes. For example, processes R and S given by Equations 6.3 and 6.4 which have resulted in a deadlock due to the existence of a backward time dependency in their communications. The notion of a backward time dependency can be visualised in a fictitious causality diagram as drawn in Figure 6.6.

$$R \;\widehat{=}\; \mu X \bullet (a\,?\,x \rightarrow b\,!\,y \rightarrow X) \tag{6.3}$$

$$S \;\widehat{=}\; \mu X \bullet (b\,?\,y \rightarrow a\,!\,x \rightarrow X) \tag{6.4}$$

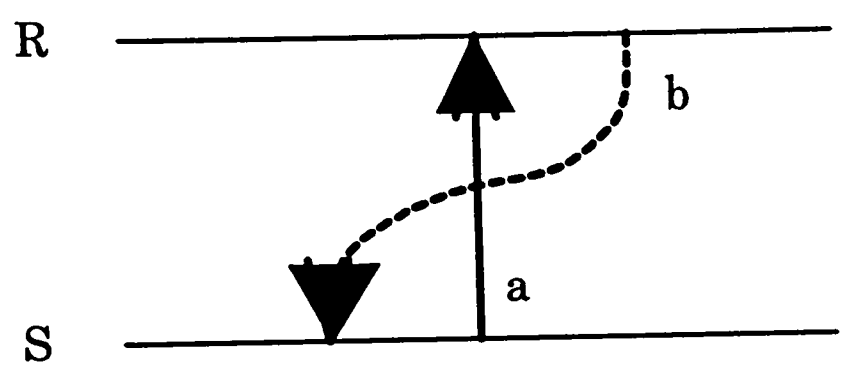


Figure 6.6: The notion of deadlock, represented using a fictitious causality diagram

Using CADET, deadlock detection is automatic should a mistake is made in the original design. Although the tool cannot identify the cause of a possible deadlock by itself and present its cause graphically such as the one shown in Figure 6.6, a signal is generated as soon as a deadlock occurs during the evolution of the communicating processes. Furthermore, the logged traces which represent the time histories prior to deadlock, contain information of how the deadlock occurred is available to the designer. Such information enables a designer to carry out appropriate modifications to the original high level system definition in order to eliminate the deadlock.

In the next section, the use of CADET is illustrated, firstly on two concurrent systems and then on a number of cases discussed in the robot control case study presented in Chapter 5.

6.4 Utilisation of CADET

This section presents some of the results obtained from using the Causality Evaluation Tool on two simple systems. Through these results, a number of useful functionalities of the tool are revealed. Moreover, some of the results obtained from the case study are complementary to the proofs produced in Chapter 2 and 5 which have the effect of strengthening the formality of the analysis given by CADET.

6.4.1 Application to a Simple Concurrent System

To illustrate how CADET can be used to simulate, display and extract information from concurrent processes expressed in CSP, a system with three communicating processes are chosen as an example. Figure 6.7 shows the system under consideration. The three concurrent processes $P1$, $P2$ and $P3$ communicate with one another via channels A, B and C.

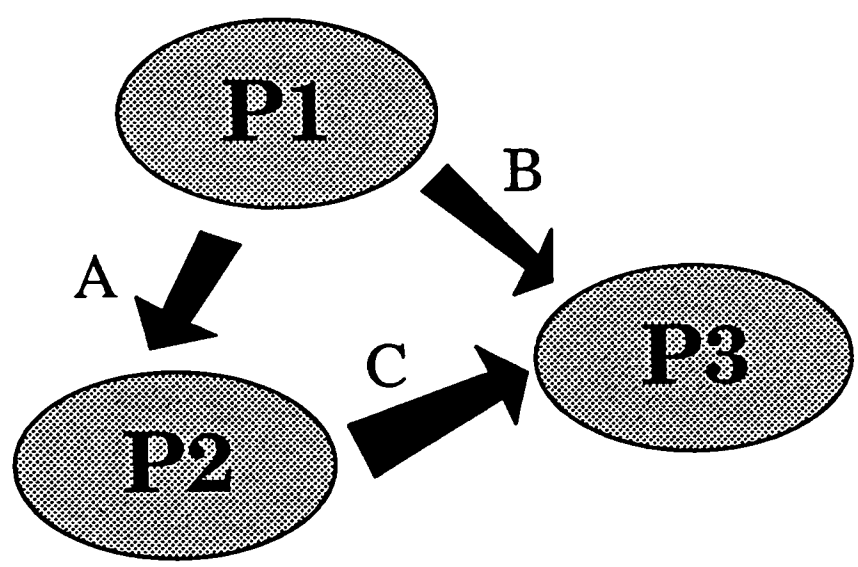


Figure 6.7: A simple three process system

The behaviour of the three processes are specified using the set of specification predicates and their actions are realised using the corresponding generic process *FSM*. Since CADET is an automatic process algebra manipulating machine, for the purpose of evaluation, the process algebra of each of these processes forms the inputs. Whereas the outputs from the tool provide the relevant information for the deduction of correctness in behaviour with respect to the defined realizations.

The corresponding CSP process definitions for the three processes are given by Equations 6.5, 6.6 and 6.7 and the overall system by Equation 6.8.

$$Proc1 \;\widehat{=}\; FSM^{*1}\,(a.x,\ b.y) = \mu X \bullet (a\,!\,x \rightarrow b\,!\,y \rightarrow X) \tag{6.5}$$

$$Proc2 \;\widehat{=}\; FSM\,(a.x,\ c.x) = \mu X \bullet (a\,?\,x \rightarrow c\,!\,z \rightarrow X) \tag{6.6}$$

$$Proc3 \;\widehat{=}\; FSM^{*}\,(b.y,\ c.z) = \mu X \bullet (b\,?\,y \rightarrow c\,?\,z \rightarrow X) \tag{6.7}$$

$$\text{SYSTEM} \;\widehat{=}\; (P1 \parallel P2 \parallel P3) \tag{6.8}$$

To verify that this system is free from deadlock with the given process definitions, expansion using process algebra on *SYSTEM* results in the recursive process given by Equation 6.9. The

[1] The generic process FSM^{*} has a different communication patterns as Process 4.8, nevertheless, they have identical structures and are classified under the same generic process class *FSM*

properties of a prefixed recursive process states that the overall system will work without deadlock.

$$\text{SYSTEM} \mathrel{\widehat{=}} \mu X \bullet (a\,!\,x \rightarrow b\,!\,y \rightarrow c\,!\,z \rightarrow X) \tag{6.9}$$

By entering the process definitions of the three processes together with the definition of the overall system into CADET, a series of causality diagrams are generated which reflect the subsequent interactions between the three concurrent processes. Figure 6.8 freezes an instant of the communication pattern after the system has completed a particular trace.

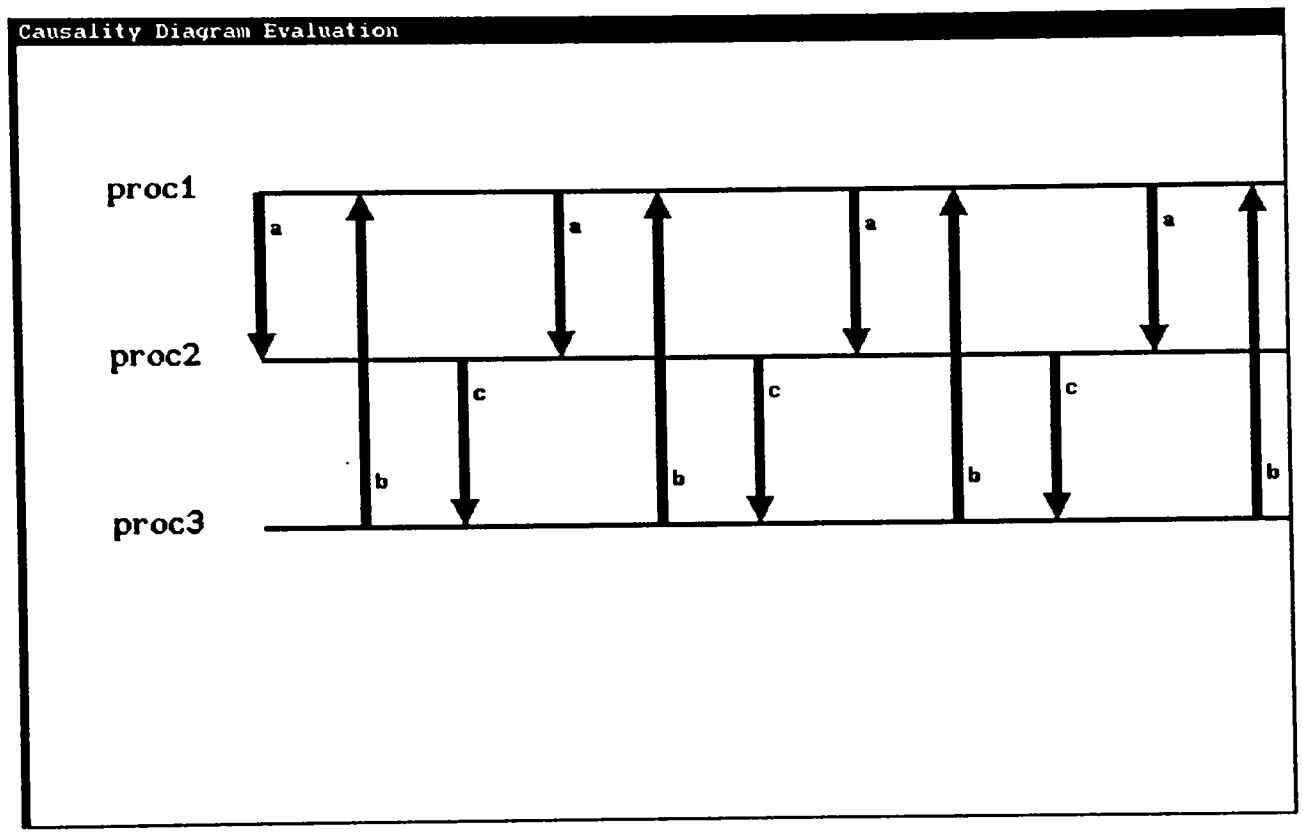


Figure 6.8: An instance of the communication between the three process system

From this causality diagram, the following deductions are made.

- The initial action that this system is capable of performing is the communication along channel a between processes $Proc1$ and $Proc2$.
- There exists a fixed temporal ordering in communication between the three concurrent processes which implies that a general trace of this system has a form: $\langle a.x,\ b.y,\ c.z\rangle^n$.
- The sequence of communication is repetitive implying that the resultant system is a recursive process which is free from deadlock.

From this simple case, it is shown that the use of CADET gives a quick and clear picture of the interactions between the three parallel processes and the results agree with that given by Equation 6.9.

6.4.2 A Communication Protocol

In this case, three slightly different models of the communication protocol discussed in Chapter 2 is studied using the tool. In Sub-section 2.5.5.5, it is shown formally that by varying the ordering of communicating events within a constituent process, the system behaviours is changed accordingly. Here, the tool is used to investigate these changes in the ordering properties.

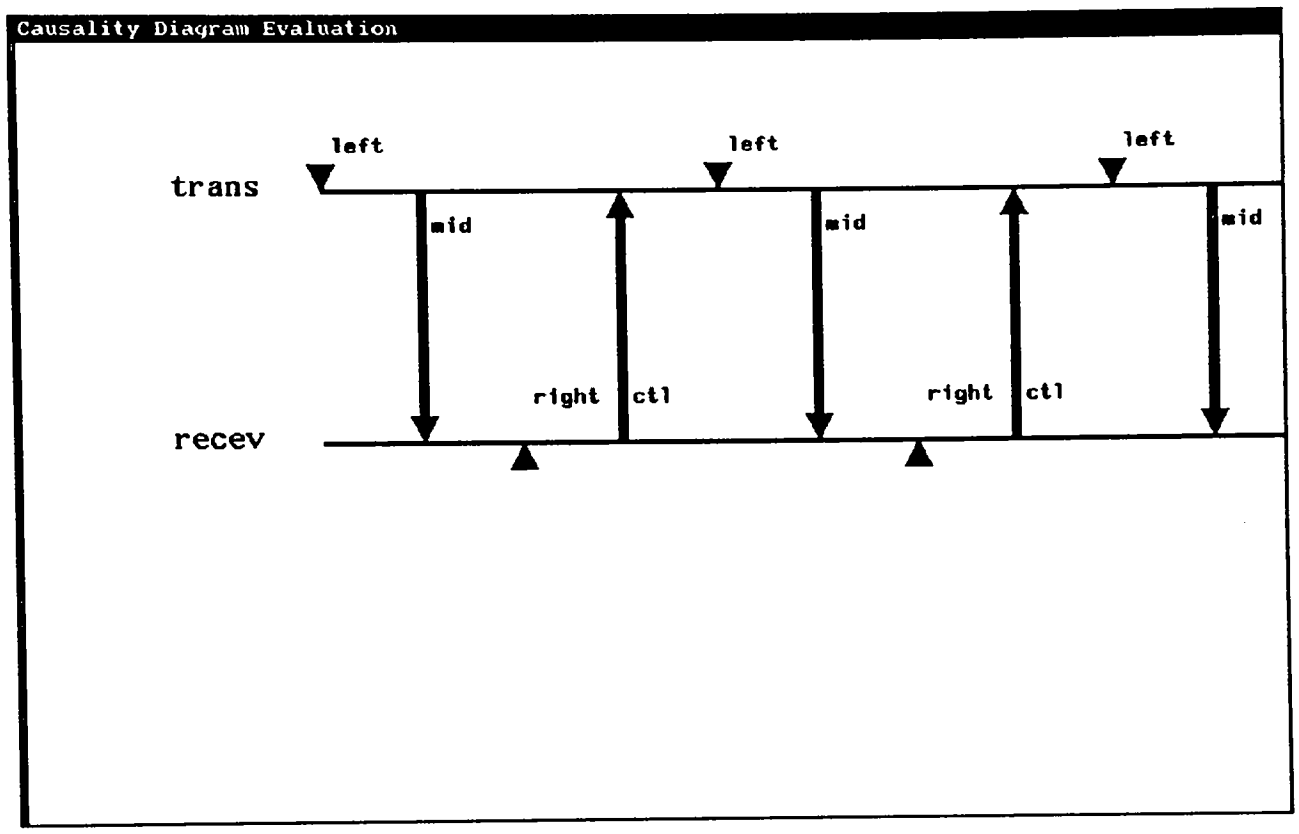


Figure 6.9: A trace from the original model of the communication protocol showing the uniqueness in the ordering of communicating events

Figure 6.9 is a causality diagram that corresponds to the original realization of the *Receiver* and *Transmitter* processes given by Equations 2.27 and 2.26. In this figure, a unique and recursive communication sequence is observed which agrees with the expanded process definition of (*Receiver* || *Transmiter*). On the other hand, a different set of causality diagrams are obtained with the modified process $Receiver_2$ given by Equation 2.29 as the input. Figures 6.10 and 6.11 are two typical causality diagrams showing that there exists an alternative evolution of the modified protocol.

Finally, with the process $Receiver_3$ given by Equation 2.33 taken as one of the components of the model of the communication protocol, the causality diagram given in Figure 6.12 displays only a single event and then signifies the existence of a deadlock.

It is seen that the observations given by the various causality diagrams provide identical information about the differences in behaviour of the three models, and these results not only agree with those obtained from formal analysis, they also provide visual impressions on process interactions.

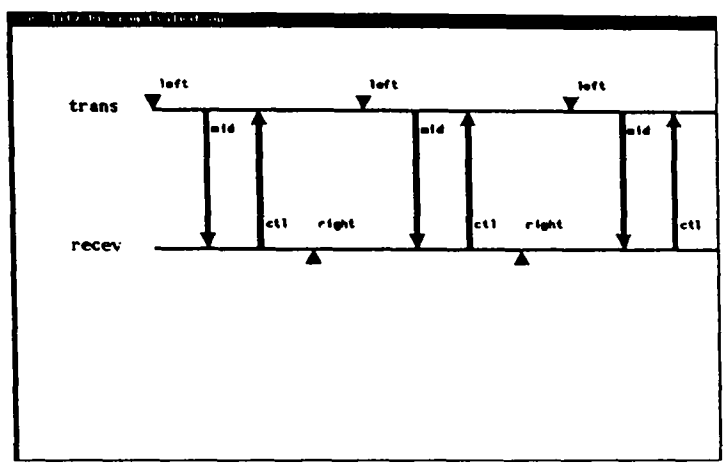


Figure 6.10: A particular trace of the communication protocol with a modified receiving process $Receiver_2$

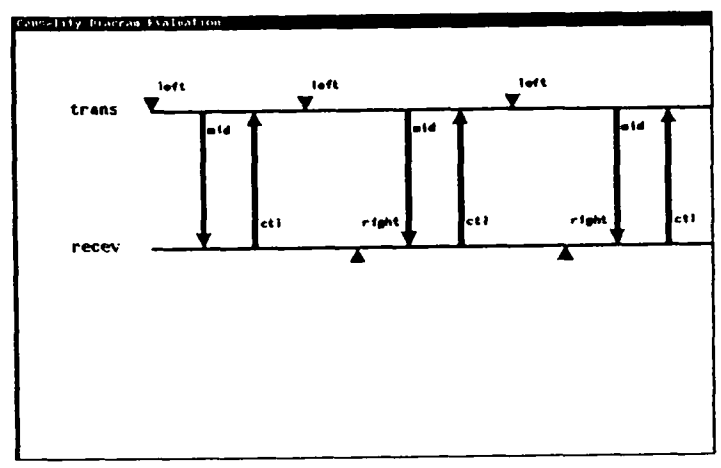


Figure 6.11: Another trace of the communication protocol with a modified receiving process $Receiver_2$

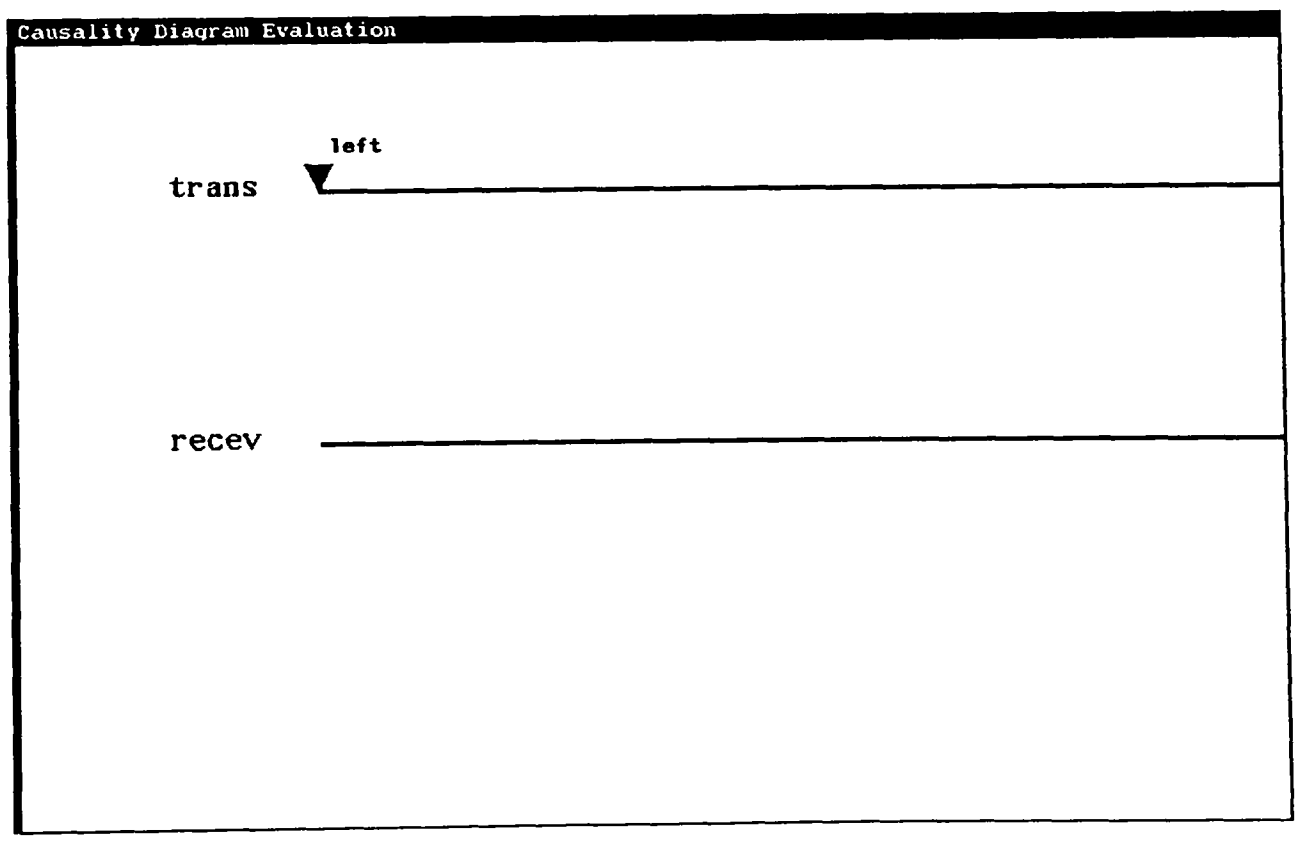


Figure 6.12: Only a single communicating event is displayed and there after the protocol fails to proceed, a deadlock is detected.

6.4.3 Application to the Robot Control System

To illustrate a number of uses of CADET in addition to those illustrated in the previous examples, and to assess the usefulness of the results it produces, the control software models of the robot system is studied using CADET. In this sub-section, the following six cases of the robot software components are investigated.

1. The parallel sub-systems (OpInt || TrajGen || Ctl).
2. The parallel sub-systems (DataLog || TrajGen).
3. The parallel sub-systems (DataLog || TrajGen || DataMaster)
4. The realization of the modified First Level Model
5. The time-critical refinement of sub-system Ctl with the parallel processes (ForceCtl || InvDynGen || StatusRep)
6. A single loop PID control process

It is seen that these cases are directly taken from the realizations of the robot system software in Chapter 5 and these CSP processes are written at different levels of abstraction, ranging from the realizations given in the Zeroth Level Model down to a single loop PID control process. Through such a wide spectrum of cases, the usefulness and applicability of CADET is evaluated.

6.4.3.1 Sub-systems (OpInt || TrajGen || Ctl)

In this analysis, the three concurrent process definitions given by Equation 5.41 for OpInt, Equation 5.33 for TrajGen and Equation 5.40 for Ctl are investigated using CADET, a *breadth* first analysis is conducted on the generated traces following by a *depth* first analysis. Through the depth first analysis, specific behaviours of the sub-systems are observed. From Figure 6.13, it is seen that the communicating sub-systems are always ready to received message from their environment through channels *Opin*, *Logdata* and *Cdata* initially, implying that when $tr = \langle\rangle$, the refusals of the three concurrent sub-systems must exclude these three communications. The same conclusion can be deduced by the law on refusals of the three processes running in parallel.

The time history given in Figure 6.14 displays the subsequent actions performed by the parallel system *after* an input form the operator whereas Figure 6.15 shows a sequence of actions performed when there is no current inputs form the operator. The communications between the three sub-systems given by the two diagrams reveal the recursive nature of the sequences of events that are

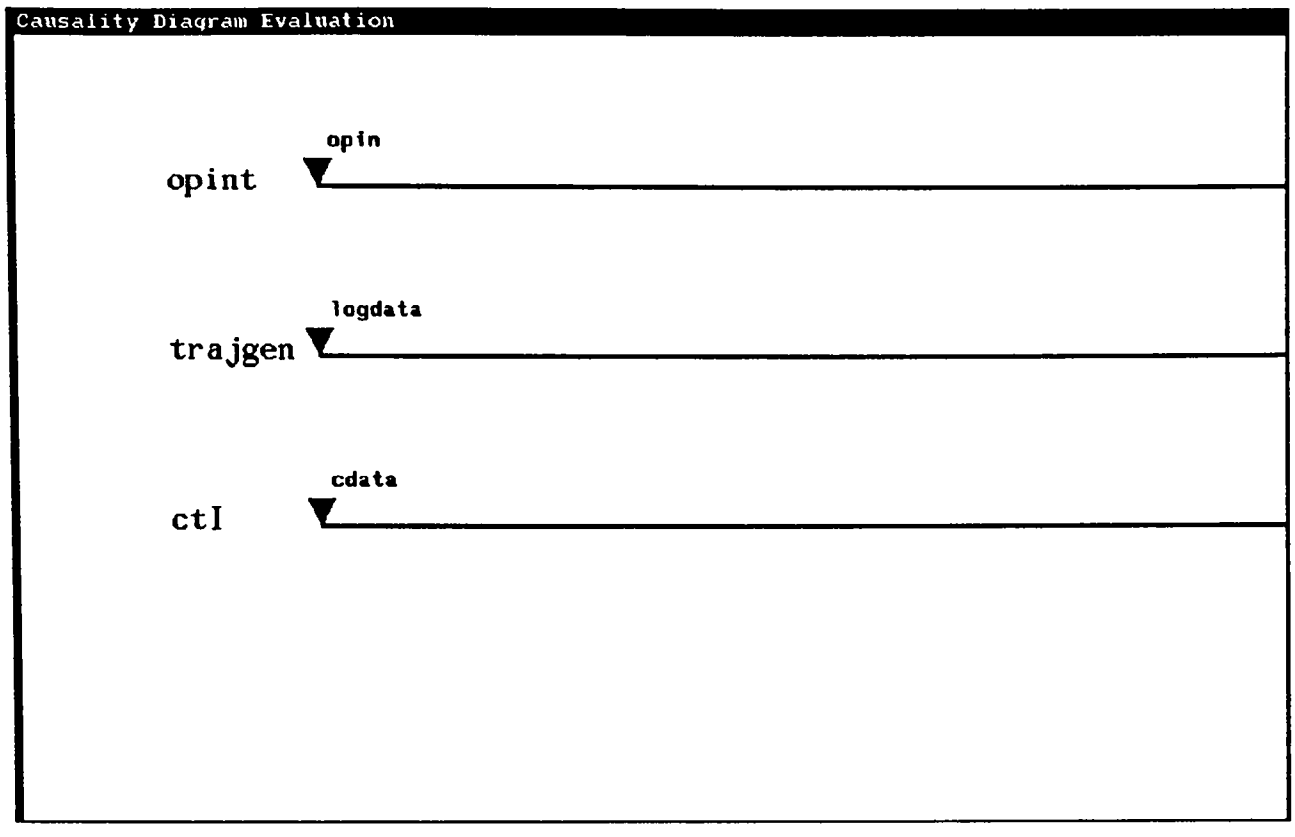


Figure 6.13: The initial actions offered to the environment by the three sub-systems OPINT, TRAJGEN and CTL running in parallel

occurring under the abstract realizations of the sub-systems. From the ordering behaviours of the various communicating events, the following properties are observed:

1. *ORDER* (*Opin*, *Opout*, 1)

2. *ORDER* (*Opout*, *Traj*, 1)

3. *ORDER* (*Traj*, *Drive*, 1)

These properties have been investigated in Proof 5.6 given in Sub-section 5.5.3.2 which confirmed these observations.

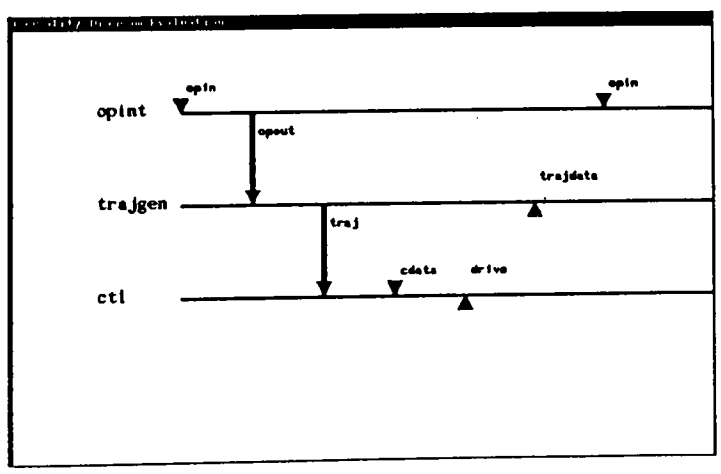


Figure 6.14: A typical traces *after* an operator's input

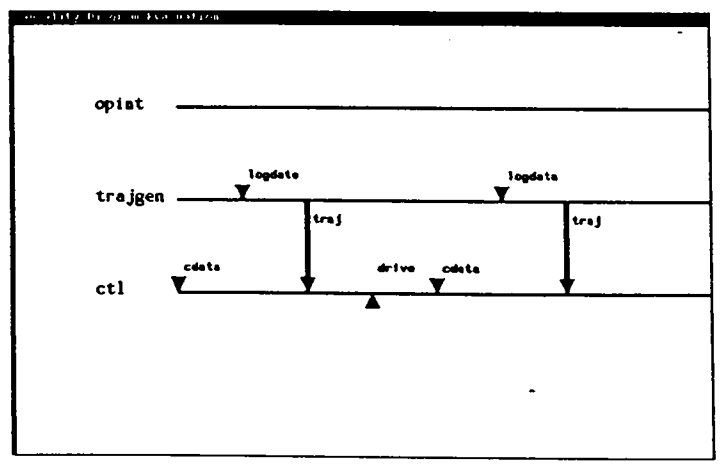


Figure 6.15: A typical traces in the absence of operator's input

Further investigation of the sub-system interactions using the interpreter and analysis of their traces indicates that the high level implementation has no potential problems of deadlock and *livelock* [Hoare, 1985b]. From Figures 6.16 and 6.17, it is seen that despite the inputs from the sub-system OPINT have dominated the interactions, overall properties such as control demands (indicated by the communication across channel *Drive*) must follow inputs from the operator are preserved. This results suggest that the implementation is free from deadlock and livelock. The problem of livelock among communicating processes will again be brought to our attention in Sub-section 6.4.3.5.

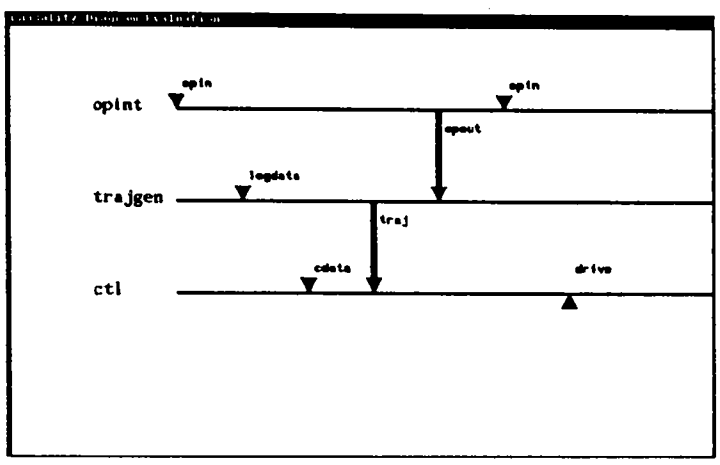


Figure 6.16: Behaviour of the three concurrent sub-systems under abnormal input conditions

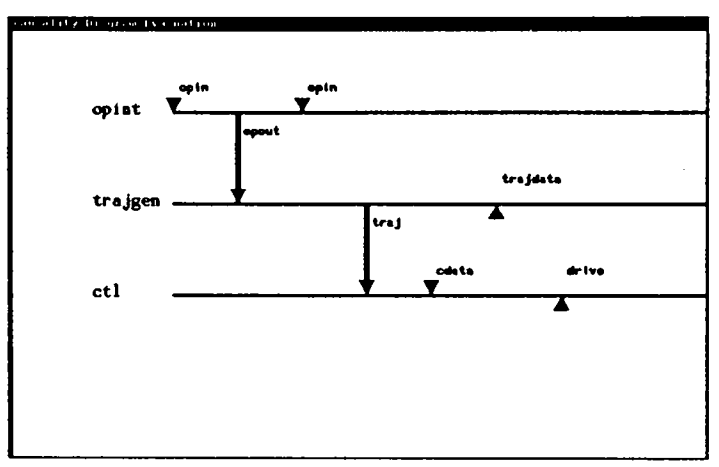


Figure 6.17: Behaviour of the three concurrent sub-systems under abnormal input conditions

6.4.3.2 Sub-systems (DATALOG || TRAJGEN)

According to the formal analysis presented in Sub-section 5.5.3.2, it is shown in Proof 5.7 that potential deadlocks exist between the two communicating sub-systems DATALOG and TRAJGEN under the realizations given by Equations 5.37 and 5.33. Analysing the occurrence of deadlock using CADET shows that if the environment of these two sub-systems has initiated the sequences of events that are shown in Figures 6.18 and 6.19, the predicted deadlocking situations are resulted. CADET reports the occurrence of a deadlock by giving the appropriate warning messages.

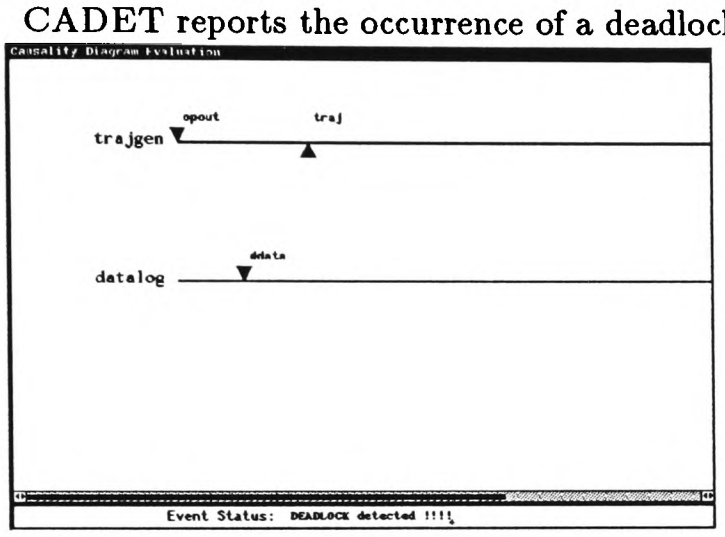


Figure 6.18: The sequence of actions taken by TRAJGEN and DATALOG which leads to the scenario of a deadlock

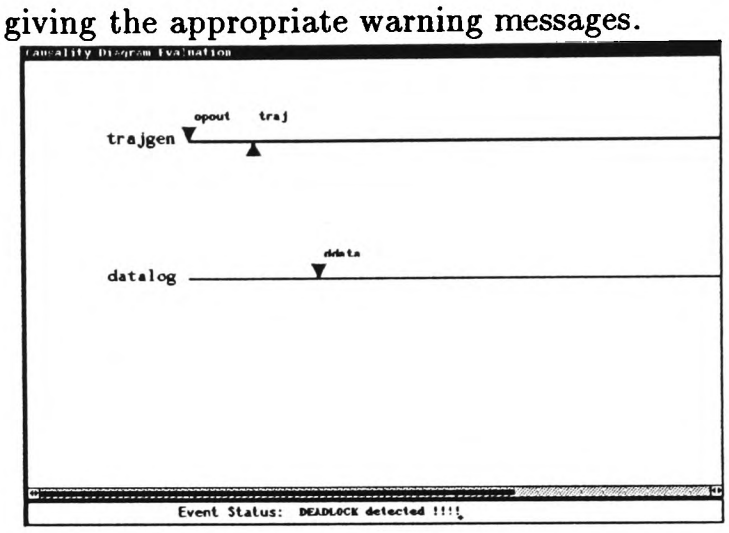


Figure 6.19: An alternative sequence of actions taken by TRAJGEN and DATALOG which leads to the scenario of a deadlock

From the two cases, it is seen that the detection of deadlock by the tool is automatic once a deadlock has occurred. The use of a depth first analysis provides the means for back-tracking the cause of the deadlock. Apart from the two initial deadlocking conditions, the two parallel sub-systems will enter a locking state once these two communication patterns have occurred within their time histories. The causality diagram in Figure 6.20 displays a particular traces which lead to a subsequent deadlock.

In the next sub-section, the result of introducing the DATA MASTER in order to eliminate the occurrence of deadlocks is presented. Nevertheless, according to Equation 5.55, the two parallel sub-systems will not deadlock under certain situations, and these are reflected from the two causality diagrams given by Figures 6.21 and 6.22.

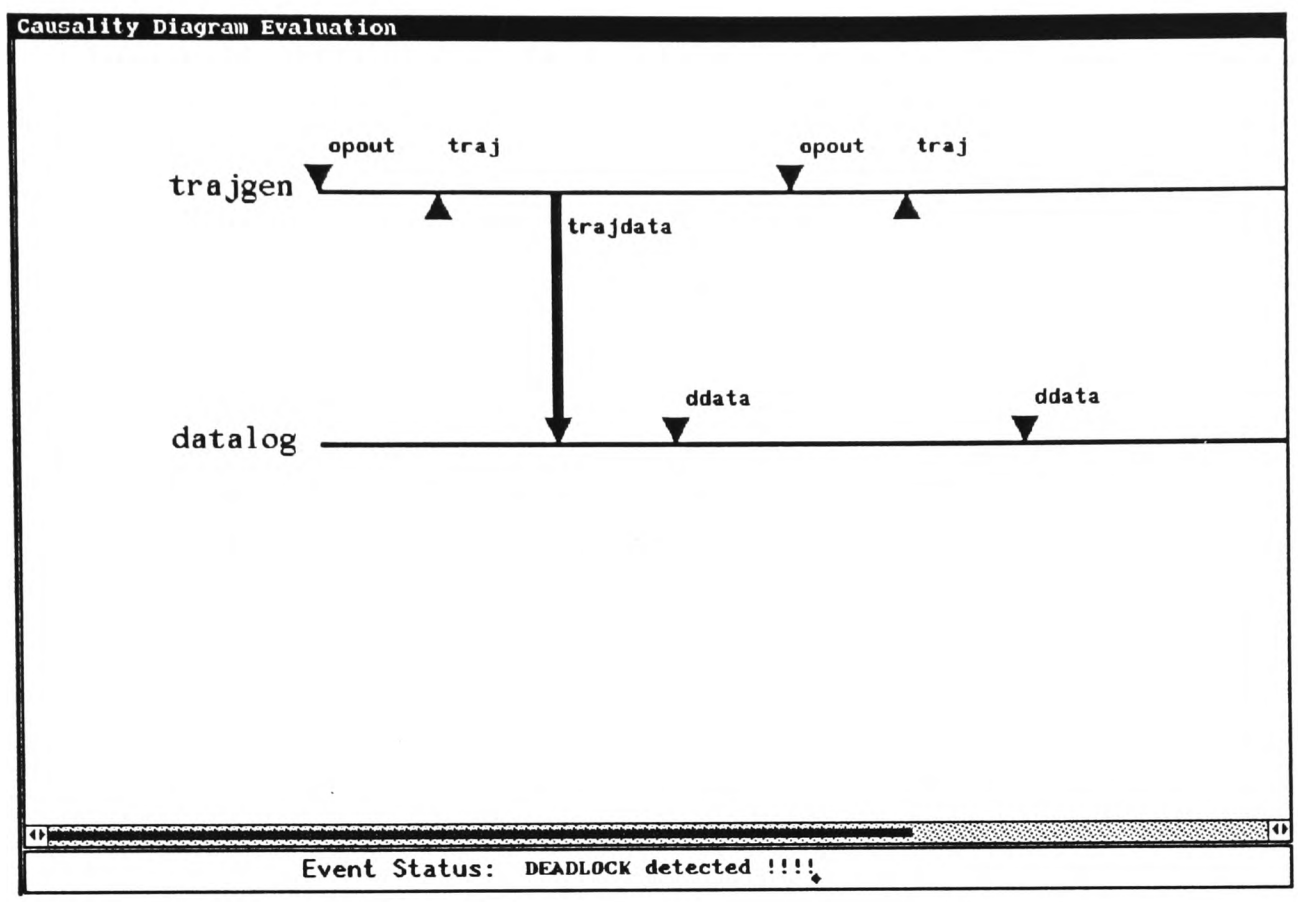


Figure 6.20: A trace which leads to a subsequent deadlock

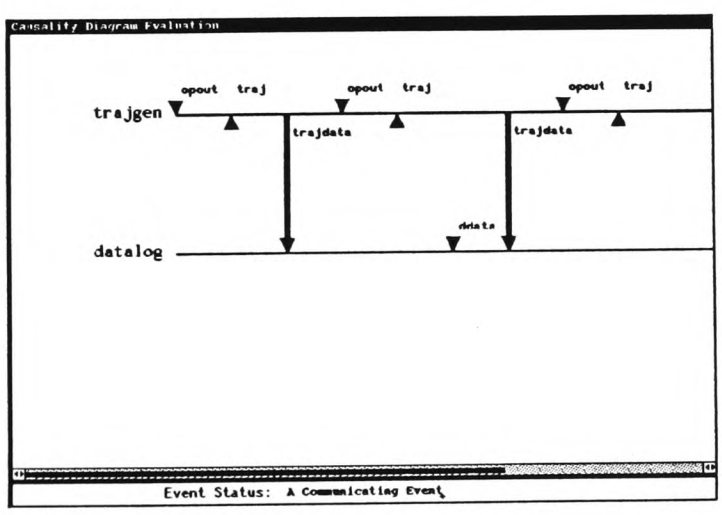


Figure 6.21: The sequence of actions taken by the parallel sub-systems TRAJGEN and DATALOG *after* an output from the sub-system OPINT

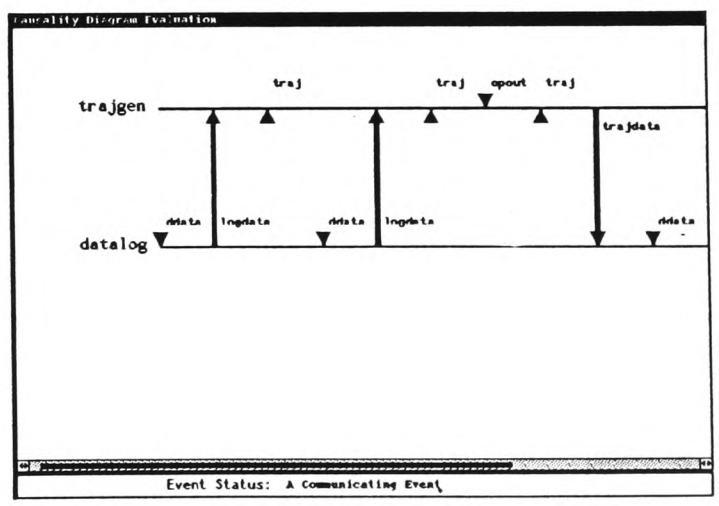


Figure 6.22: The sequence of actions taken by the parallel sub-systems TRAJGEN and DATALOG *after* an input from the sub-system DATALOG

6.4.3.3 Sub-systems (DATALOG || TRAJGEN || DATAMASTER)

In order to observe the corresponding corrective measures taken by the sub-system DATAMASTER onto the two concurrent sub-systems DATALOG and TRAJGEN, the process realizations of DATAMASTER according to Equation 5.58, the two re-designed sub-systems TRAJGEN_2 and DATALOG_2 specified by Equations 5.61 and 5.62 are input to CADET.

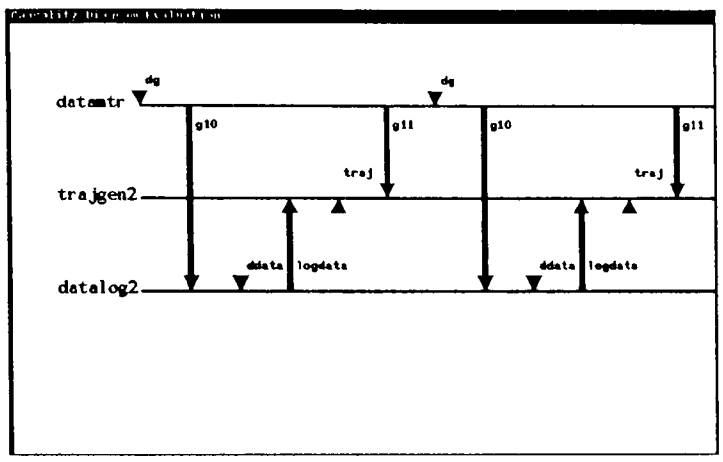


Figure 6.23: Subsequent communication pattern of the three parallel sub-systems: DATAMASTER, TRAJGEN_2 and DATALOG_2 *after* communications across channel *Ddata* are signalled

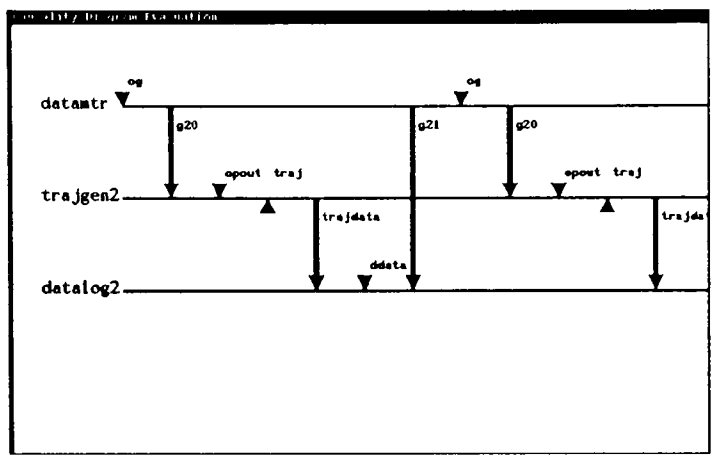


Figure 6.24: Subsequent communication pattern of the three parallel sub-systems: DATAMASTER, TRAJGEN_2 and DATALOG_2 *after* communications across channel *Opout* are signalled

Figures 6.23 and 6.24 show the two possible evolutions of the three concurrent sub-systems. It is seen that the introduction of the DATAMASTER has restricted the interactions between the two sub-systems TRAJGEN_2 and DATALOG_2 to follow the pre-defined orders such that the occurrence of communication along channels *Opout* and *Ddata* are made mutually exclusive. Furthermore, it is seen that the two sequences of actions are recursive. In addition to the two diagrams, Figure 6.25 shows that the two mutually exclusive sequences are chosen freely by their environment indicating that the occurrence of potential deadlock are avoided. All the results agree with the process algebra analysis given in Sub-section 5.5.4.1 and in particular, the mutual exclusion between the two communications: *Ddata* and *Opout* as displayed by Figure 6.25, is a direct representation of the process given by Equation 5.63.

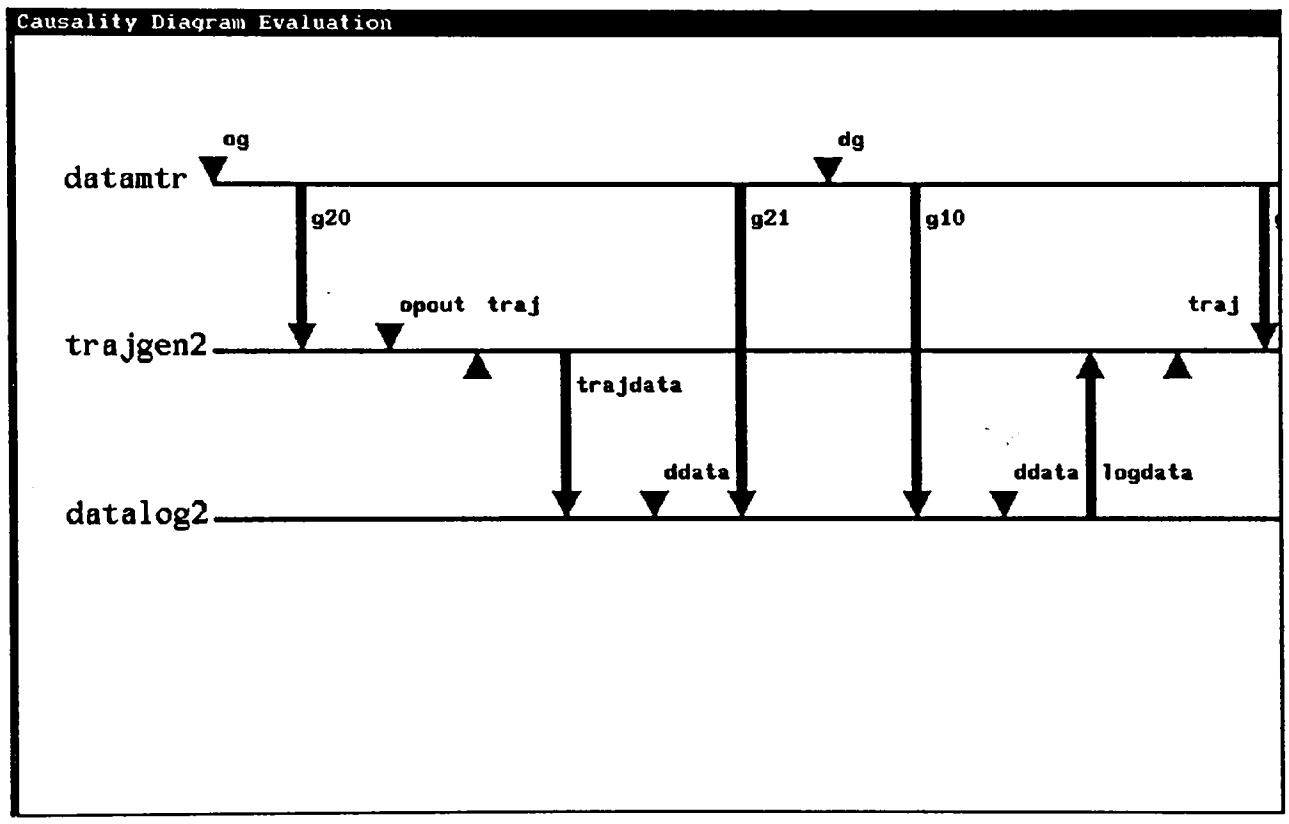


Figure 6.25: Communication pattern of the three parallel sub-systems: DATAMASTER, TRAJGEN$_2$ and DATALOG$_2$, showing that the two sequences of communication are mutually exclusive to one another, of which their initiations are chosen freely by their environment

6.4.3.4 The Modified First Level Model

Here, the interactions between all the sub-systems from the re-designed First Level Model of the robot system are studied simultaneously. The process definition for TRAJGEN$_2$, DATALOG$_2$ and DATAMASTER are taken directly from the previous case whereas the modified sub-systems definitions of OPINT$_2$ and DATAPROC$_2$ are given by Equations 5.59 and 5.60, and finally sub-system CTL takes the original process definition given by Equation 5.40. Figure 6.26 shows precisely the two events that are offered initially by the six parallel sub-systems to their environment.

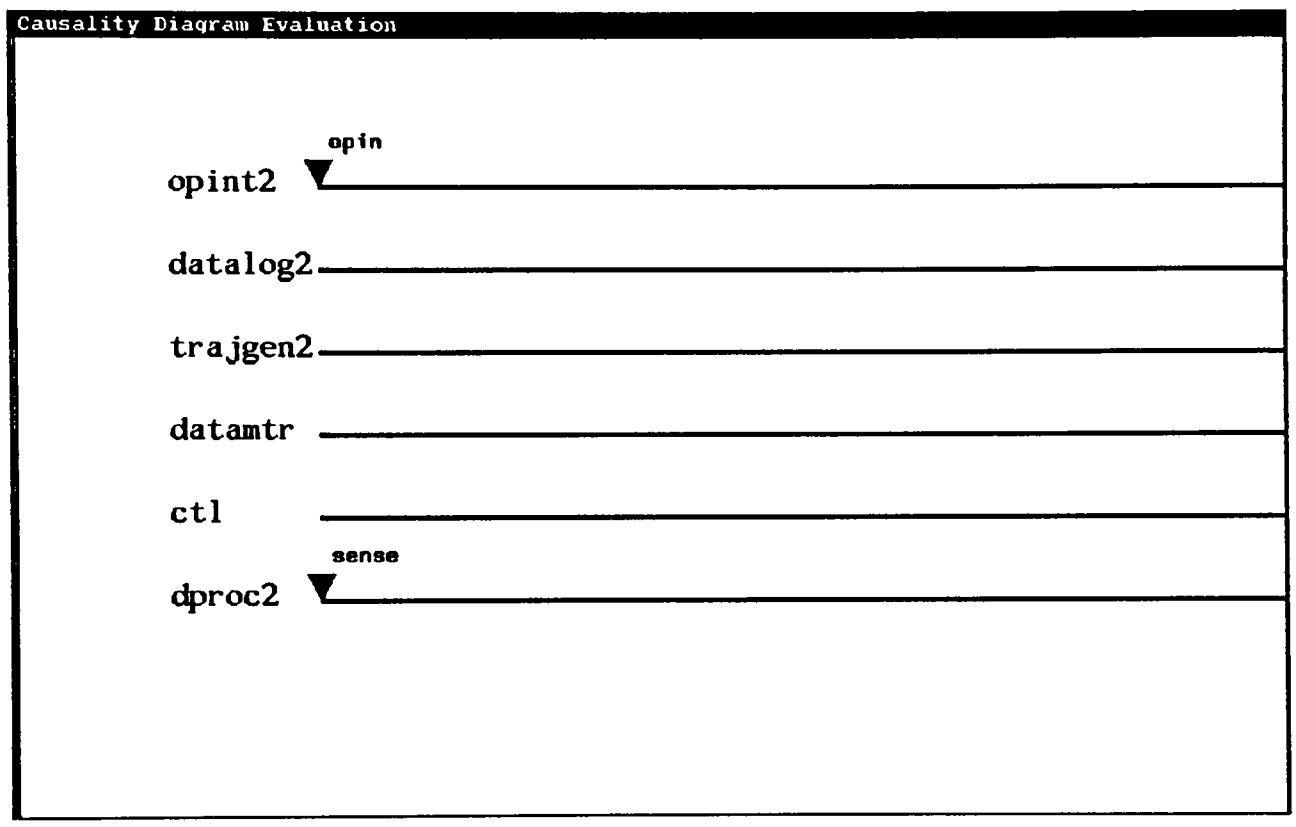


Figure 6.26: Initial actions offer by the overall system to its environment

By investigating the subsequent actions taken by the overall system CONTROL specified by the First Level Model after each of these initial actions, it is found that each of these subsequent traces are recursive. Figures 6.27 and 6.28 display the two possible traces. The diagrams show that the behaviours of the system CONTROL agree with that already established in Proof 5.8.

Two other general traces of the overall system with the scenario of interleaving operator input and sensor data are displayed in Figure 6.29 and 6.30. These represent more realistic situations for which the interactions from the environment such as the communications through channel *Opin* and *Sense* can occur at any order during the evolution of the controlling actions.

Finally, in order to visualise the corrective actions perform by the sub-system DATAMASTER and its interactions with the overall system, CADET is steered to investigate the otherwise deadlocking condition, that is to observe what will follow the trace $\langle Opin,\ Sense,\ Cdata \rangle$. The consequence is

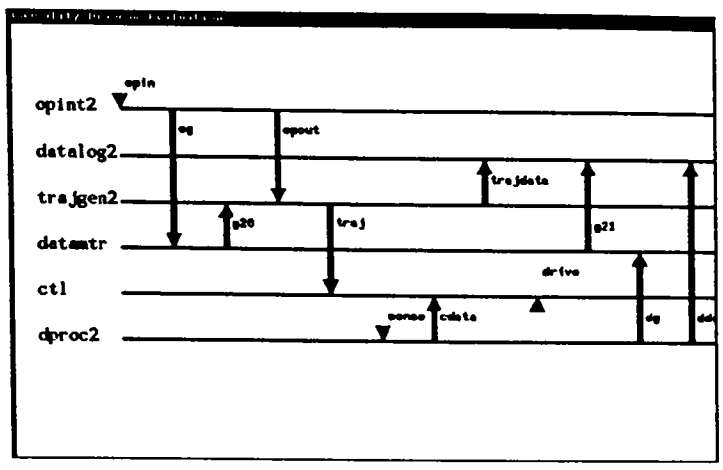


Figure 6.27: The subsequent actions taken by the overall system CONTROL *after* an operator's input

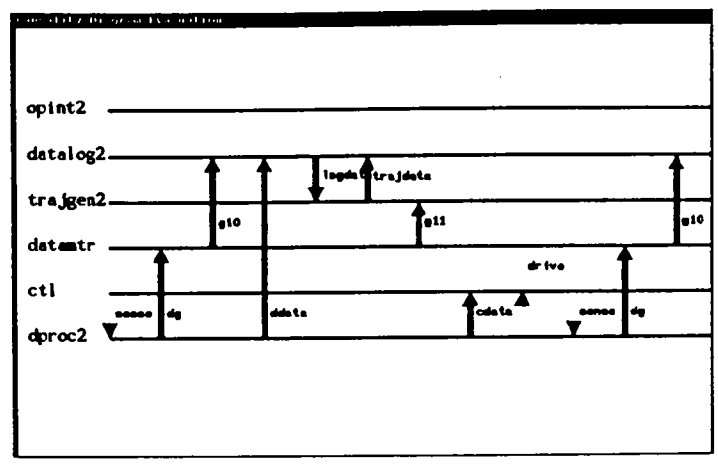


Figure 6.28: The subsequent actions taken by the overall system CONTROL *after* receiving the set of sensor data from the robot

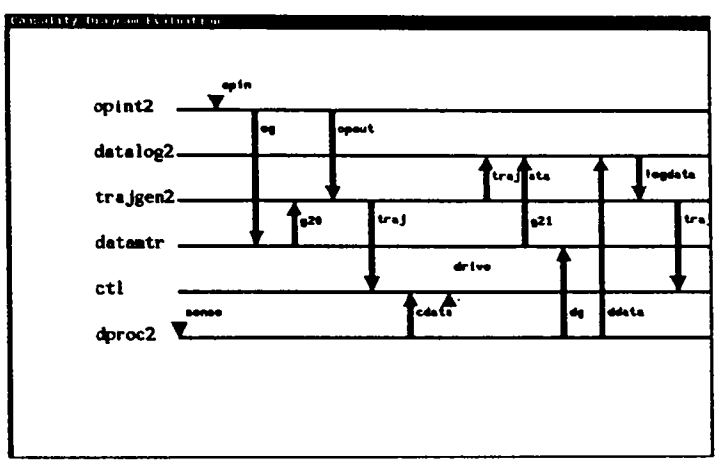


Figure 6.29: A typical traces of the system as specified by the First Level Model

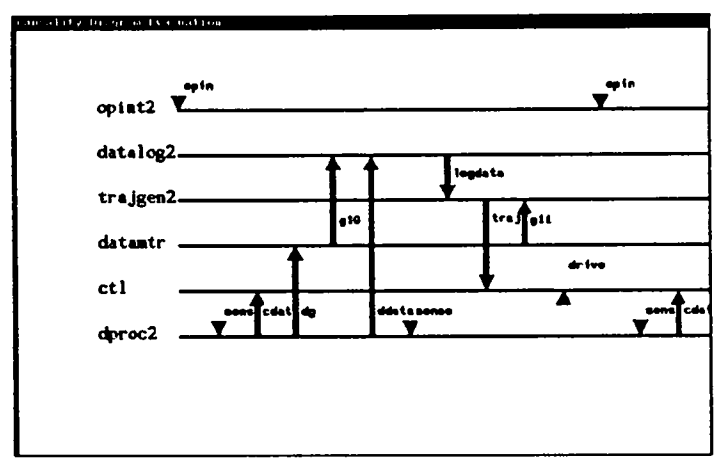


Figure 6.30: Another typical trace of the system as specified by the First Level Model

captured by the causality diagram shown in Figure 6.31. In this figure, it is seen that there is a choice given to the internal transition of either to proceed and take the validated operator's input which is initiated by the synchronising guard *og*; or to process the sensor data as initiated by the other synchronising guard *dg*. From the representation, these two guards are ***mutually exclusive*** implying that the two cannot take place simultaneously. The restrictive measure imposed by the sub-system DATAMASTER onto the internal actions of the overall system has avoided the occurrence of deadlock originally introduced by TRAJGEN and DATALOG.

In the course of investigation of using CADET on the re-designed overall system, the following deductions are made based on the observations of the properties of the causality diagrams given in this sub-section. These deductions are summarised as follows:

- The recursive nature of the communication sequences indicates that safety properties of the system can be concluded. This means that the re-designed First Level Model will work for at least some environment conditions. However, using these causality diagrams alone, the liveness properties such as deadlock freedom of the overall realization cannot be established.

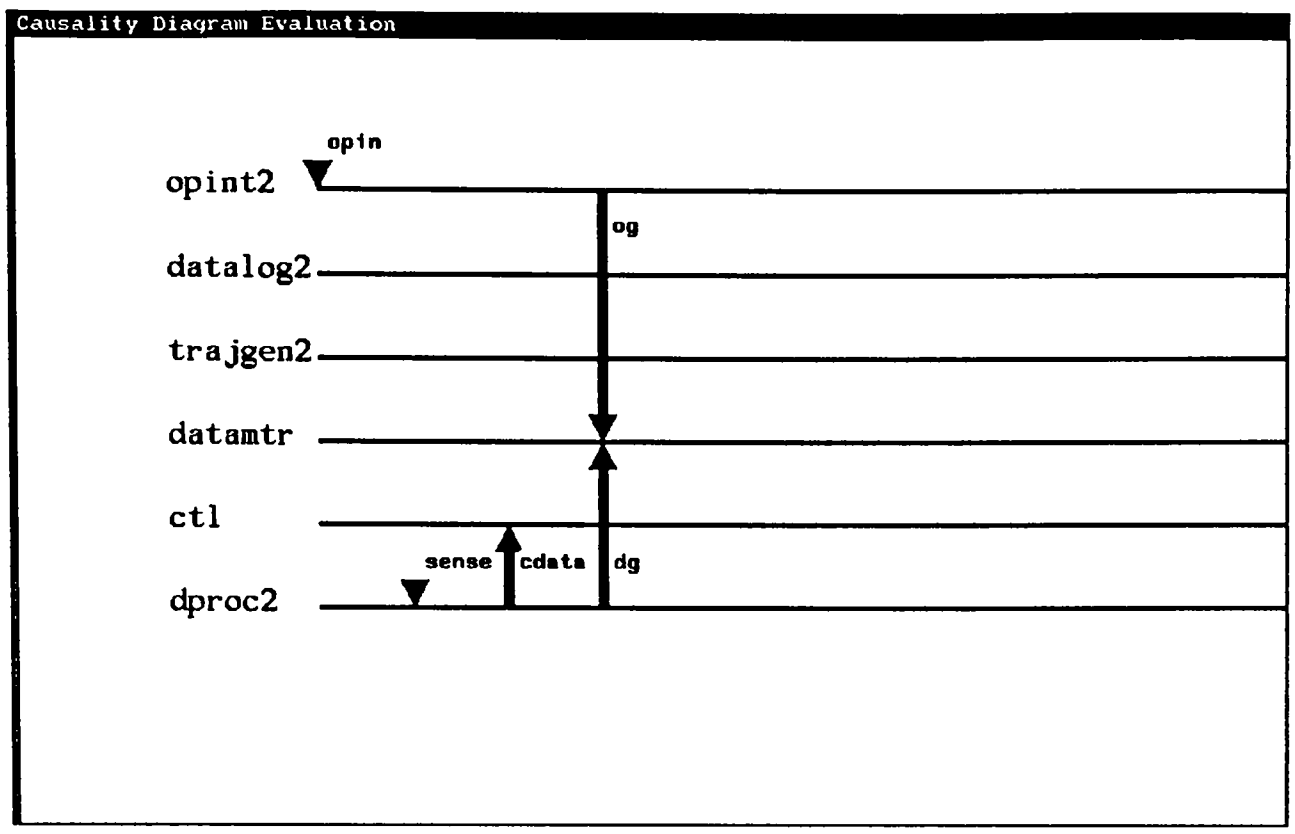


Figure 6.31: The restrictive measures introduced by the sub-system DATAMASTER to internal transitions so as to avoid possible deadlocks. Guarding events *og* and *dg* are mutually exclusive events which are displayed on the same time step, appearing as alternative actions.

- The ordering properties of salient events such as the initialisation of demand signals, *Drive* by input commands issued by the operator via *Opin*, as well as the robot data from the sensors, *Sense*, provide information for the incorporation of additional jacketed sub-systems. An example is the emergency handling sub-systems for safety critical applications.

  The information of sequentiality between discrete events enables a correct sequence of emergency signals to be injected to the robot control system so as to stop the robot in a pre-defined manner and in a predictable state. Although the study of designing an emergency handler for a distributed control system is not discussed in this thesis, the idea will be brought to our attention in the Conclusion.

- It is found that a system such as the one given by the First Level Model has resulted in a large amount of non-determinism which grows exponentially. As a result, the analysis of system interactions has become inefficient and time consuming even with the assistance of the trace analyser. In this particular example, to obtain a causality diagram given by Figure 6.26, generating the required traces alone would have taken raw CPU time in the order of hours. This observation has further re-iterated the importance of applying the evaluation tool judiciously and sensibly on system definitions that are constructed at a right level of abstraction.

6.4.3.5 Time-critical refinement of the sub-system CTL with (FORCECTL || INVDYNGEN || STATUSREP)

In order to investigate the feasibility of using CADET for simulating systems and processes at different levels of abstraction, the refinement of the sub-system CTL is studied in this section. In the next section, a further refined module within the control sub-system, the PID control process which is specified in detail is studied.

In this section, a refinement of the time critical section of the controller CTL which is described in Sub-section 5.6.2 is chosen to study of the communications as well as ordering and topology of interactions between processes within the sub-system. The r zations for the refined processes are adopted from Equations 5.79 and 5.81. In Equation 5.82 and 5.83, the actions *"store"* and *"calc"* are internal transitions and are not implemented. The corresponding process algebra for the two time critical processes are given by:

$$\begin{aligned} \text{FORCECTL} \;\widehat{=}\; \mu X \bullet \;\; & (Traj\,? \rightarrow Req\,! \rightarrow \\ & Mat\,? \rightarrow Drive\,! \rightarrow Sin\,! \rightarrow X) \end{aligned} \tag{6.10}$$

$$\begin{aligned} \text{INVDYNGEN} \;\widehat{=}\; \mu X \bullet \;\; & (Cdata\,? \rightarrow X \\ & \mid Req\,? \rightarrow Mat\,! \rightarrow X) \end{aligned} \tag{6.11}$$

And for the process STATUSREP for status reporting which is given by Equation 5.84, the buffering process is simplified as an *one place buffer*. Hence, its corresponding realization is:

$$\text{STATUSREP} \;\widehat{=}\; \mu X \bullet (Sin\,? \rightarrow Sout\,! \rightarrow X) \tag{6.12}$$

and the overall sub-system is given by a parallel composition of the three sequential processes:

$$\text{CONTROLPROC} \;\widehat{=}\; (\text{FORCECTL} \,||\, \text{INVDYNGEN} \,||\, \text{STATUSREP})$$

The causality diagrams showing the subsequent evolutions that are obtained are displayed in Figures 6.32, 6.33 and 6.34. In this case, a depth first analysis is chosen.

Figure 6.32 shows that there exists an initial *choice* between two alternative events and in reality, the a choice is made by the environment. In this analysis, the communication through channel *Cdata* is chosen. Subsequent actions taking place are observed and are examined using the tool.

Individual events can be examined for details on process communication such as the one illustrated by Figure 6.33. Figure 6.34 shows a complete cycle of events that have taken place which corresponds to the implementation given by Equation 6.13. From these causality diagrams,

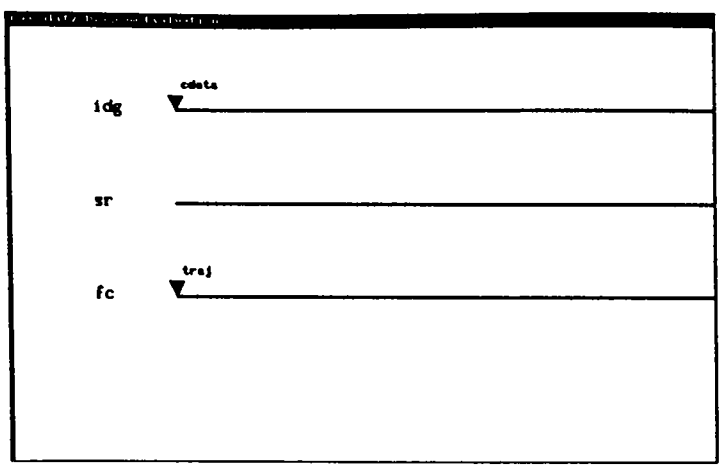


Figure 6.32: The initial communications, *Traj* and *Cdata* that are offered to the environment by the refined time critical sub-system CTL

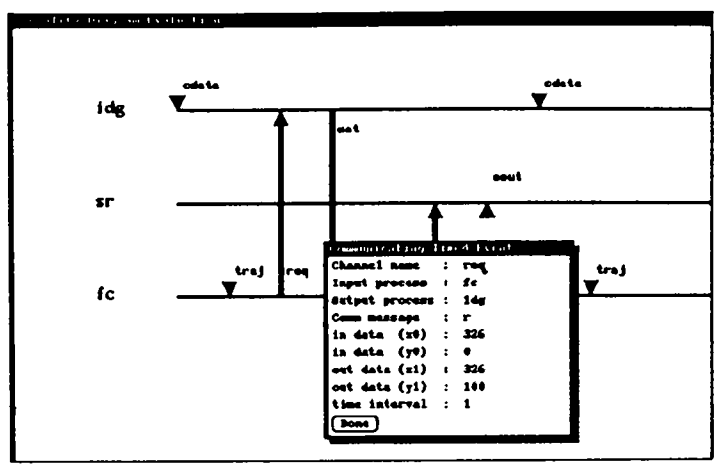


Figure 6.33: Each event can be examined for details

information on the possible ordering of communicating events occurring in this design are obtained. Also, the behaviours of the corresponding software structure can be deduced from the inter-process communications and most importantly, the recursive nature of the events suggests that at this level of abstraction, in which data dependency is immaterial, the system will not deadlock. The refined model of process realization forms the basis for subsequent development and more rigorous verification.

$$\begin{aligned} \text{CONTROLPROC} \;\widehat{=}\;\; & (\text{FORCECTL} \,||\, \text{INVDYNGEN} \,||\, \text{STATUSREP}) \\ \equiv \mu X \bullet \;\; & (\mathit{Cdata}\,? \rightarrow \mathit{Traj}\,? \rightarrow \mathit{Req}\,! \rightarrow \mathit{Mat}\,! \rightarrow \\ & \mathit{Drive}\,! \rightarrow \mathit{Sin}\,! \rightarrow \mathit{Sout}\,! \rightarrow X) \end{aligned} \tag{6.13}$$

During the investigation, it is found that there exists a subset of the evolutions of the three parallel processes which is characterised by the domination of the inter-process communication via channel *Cdata* with the process INVDYNGEN and the environment. Figure 6.35 shows a typical trace from this subset. In this case, the parallel processes enter an infinite communication loop with the environment after completing the trace $\langle \mathit{Cdata}, \mathit{Traj}, \mathit{Req}, \mathit{Mat} \rangle$.

Such a scenario can be explained by the fact that under the current realization of the three processes, the communication through channel *Cdata* is always offered as an alternative to the communication with channel *Traj* to the environment, and if the environment just happens to be ready to communicate through *Cdata* all the time, communication through channel *Cdata* will be the "dominating" event which suppresses all the other pending communications, both internal and external. The scenario is commonly known as *livelock*. The occurrence of livelock is as bad as, if not worse than the situation of a deadlock. A livelocked system consumes an infinite amount of resources over time without making any constructive progress. The trace shown in Figure 6.36 is

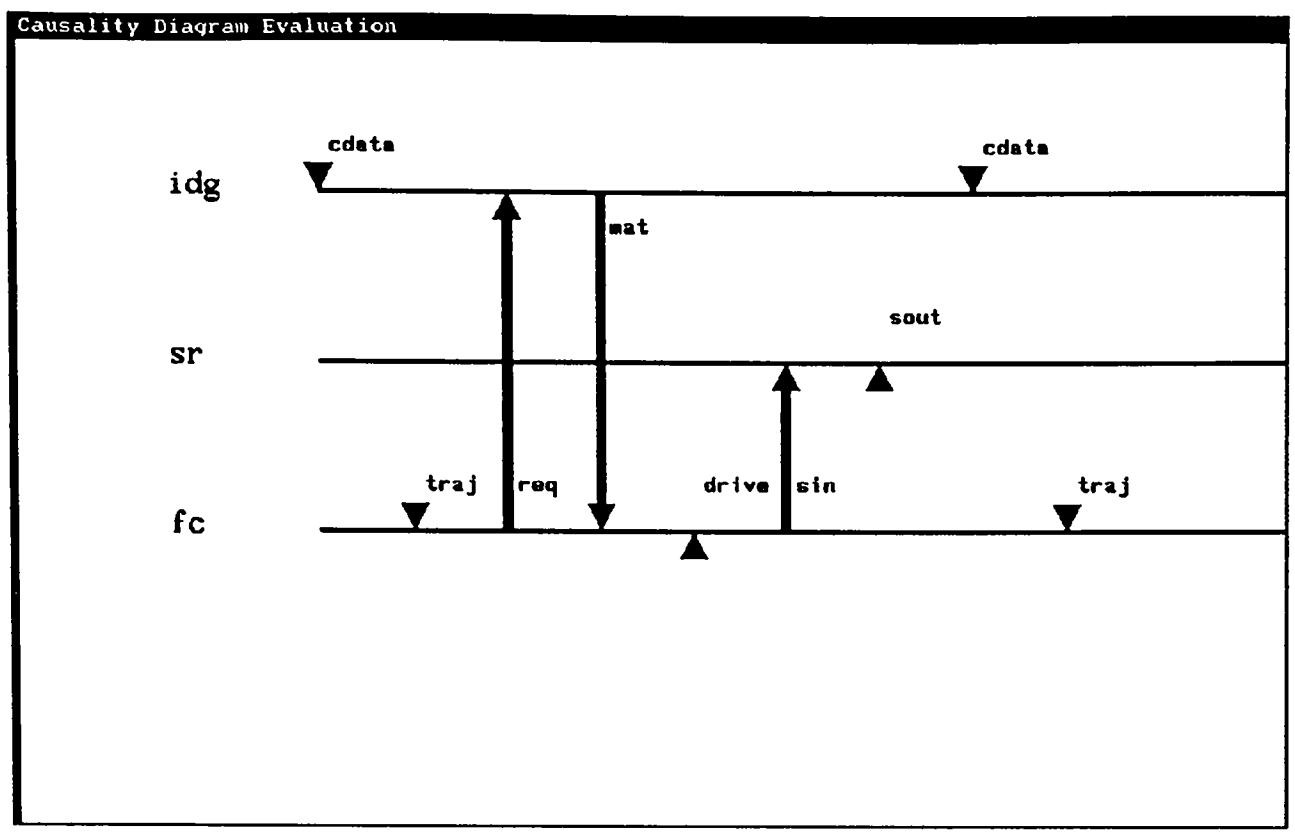


Figure 6.34: A display of the inter-process communications of the refined sub-system CTL

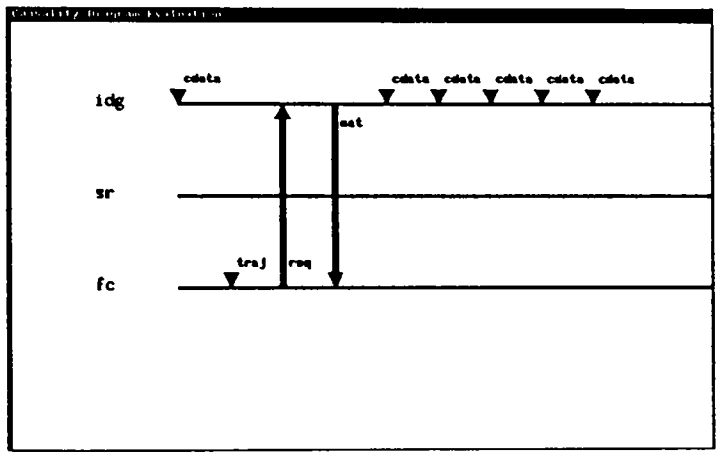


Figure 6.35: An infinite loop of communication across channel *Cdata* between process INVDYNGEN and its environment

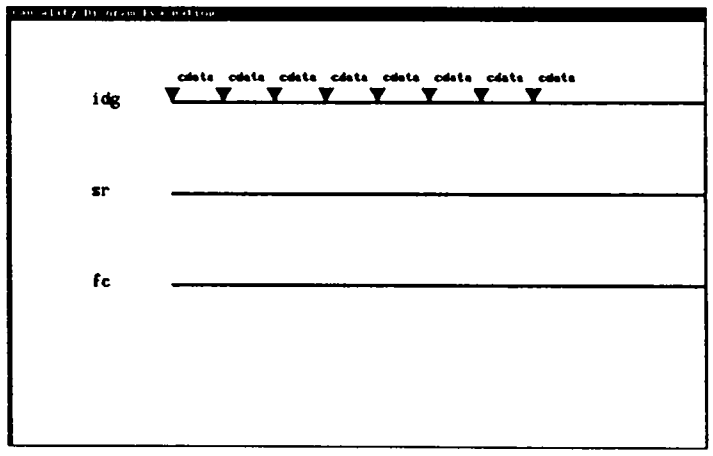


Figure 6.36: A "Livelock" between communicating processes

an extreme case of the scenario. In this figure, it is seen that despite the existence of a contesting event which is a communication through *Traj* after the seventh consecutive occurrence of *Cdata*, the communication through *Cdata* will still be chosen under the assumption.

In order to reason about a system with potential livelock, the adopted traces and refusals models of CSP is not sufficient to prove its absence, instead the *divergence* model [Hoare, 1985b] or the "fairness" of processes has to be considered. However, in the context of this thesis, this problem will not be addressed. Nonetheless, the pursuit of livelock within concurrent processes remains an active area of current research.

6.4.3.6 A single loop PID control process

In this final example, a PID controlled process is chosen. By formalising a simple compensated unity feedback control process given by Figure 4.5 in Chapter 4 using generic processes. The corresponding process implementations for the three sub-processes are given by the following equations. In effect, these definitions are similar to those already given by Equation 4.94.

$$PID \;\widehat{=}\; \mu X \bullet \; (a\,?\,e \rightarrow b\,!\,u \rightarrow X) \tag{6.14}$$

$$PLANT \;\widehat{=}\; \mu X \bullet \; (b\,?\,u \rightarrow c\,!\,y \rightarrow X) \tag{6.15}$$

$$OP \;\widehat{=}\; \mu X \bullet \; (\,i\,?\,x \rightarrow (\,c\,?\,y \rightarrow a\,!\,e \rightarrow X \;\; |\; a\,!\,e \rightarrow c\,?\,y \rightarrow X)) \tag{6.16}$$

$$PROCESS \;\widehat{=}\; (PID \parallel PLANT \parallel OP) \tag{6.17}$$

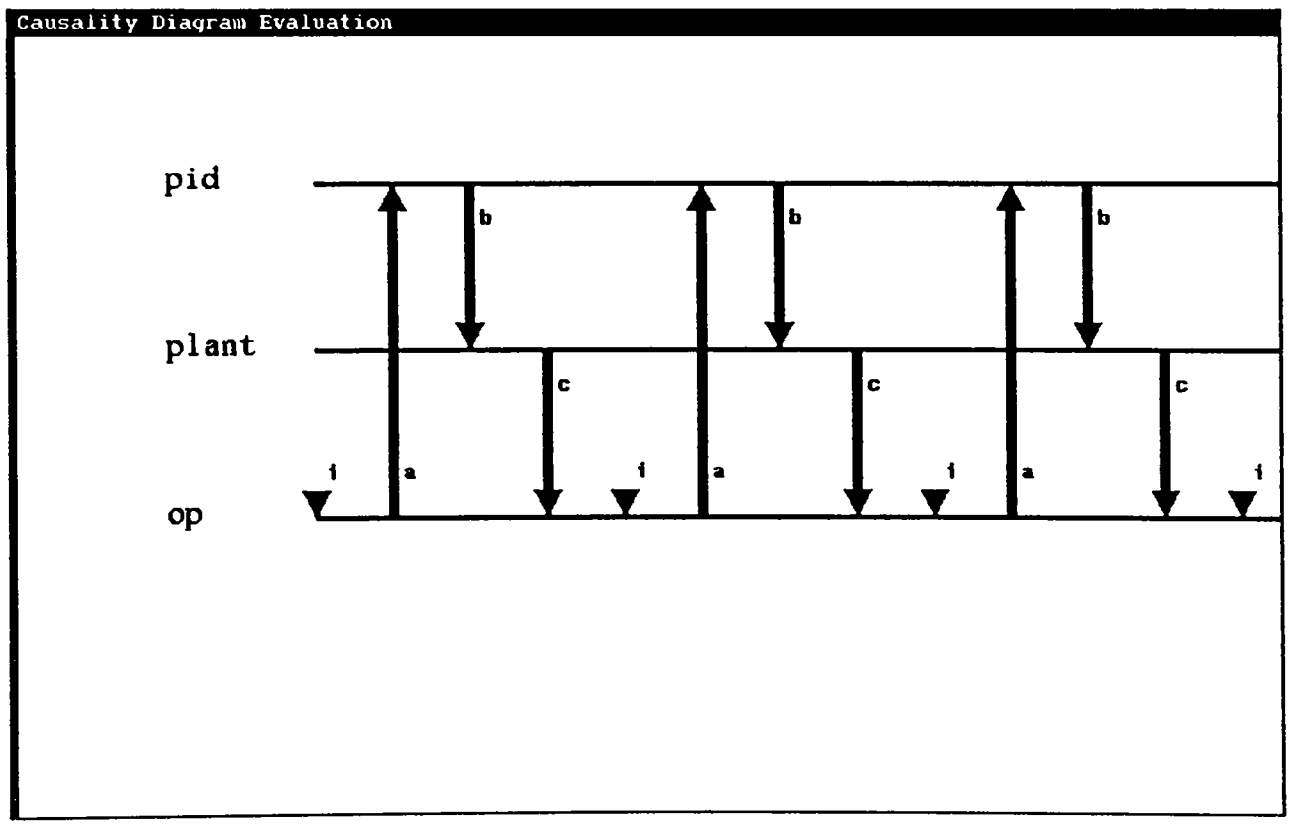


Figure 6.37: The internal and external communications of the PID control process

Figure 6.37 shows the internal as well as external communications of the PID controlled process. It is seen from the figure that the interactions between the three sub-processes follow a unique sequence and repeat indefinitely over time. Since it is already known that a typical PID algorithm is well defined by a recurrence equation in a form given by Equation 4.20, therefore the results obtained from the causality diagram is as expected.

6.4.4 The Value of CADET

Through these case studies, the applicability of CADET is evaluated. It is found that the tool is capable of simulating the interactions of process communication at different levels of abstraction; from trying out design ideas at a very high level to more rigorous uses such as tracking of deadlock and, determination of communication sequences.

Moreover, the application of CADET at the stage of software development where highly abstracted system realizations are being considered is most suitable as it helps to provide information about the following:

- temporal ordering of communications, both inter-process and process to environment communications. Using this temporal ordering information, properties about concurrent processes can be deduced.

- the structuring of the concurrent software, which includes information such as process utilisation and processor loading.

- the highlighting of recursive interactions which can be used to show that a group of concurrent processes will at least work under some conditions without the hazard of deadlock.

As such, the automation of process algebra manipulation enables process evolution to be monitored dynamically and the resultant time history is displayed visually for easy analysis. Although the present design only allows high level process definitions to be studied for their concurrent interactions through synchronous communications and the identification of deadlock is preliminary, the tool allows a designer to select a few promising approaches in a relatively quick and inexpensive manner. The correctly simulated high level processes can then be followed by more rigorous and detailed development. It is also found that the use of the graphical front-end for non-deterministic process analysis through the use of the depth first search of the process evolution tree has enhanced the efficiency of system analysis.

However, it is important to note that although the utilisation of machine assisted prototyping and simulation have revealed a number of appealing properties over manual verification, it is essential to use this automatic verification aids judiciously and intelligently in order to obtain sensible results. Nevertheless, the tool allows CSP processes to be manipulated automatically in the course of software development.

6.5 Implementation of Generic Processes

6.5.1 A High Level Implementation

In this chapter, the implementation of distributed control systems using the automatic software prototyping aid — the Causality Diagram Evaluation Tool is presented. Systems that are structured using highly abstracted generic processes are directly executed using the interpreter with their behaviours captured and displayed by the graphical interface. In particular, a simple system with three concurrent processes and a communication protocol are studied which is followed by an in depth investigation of the robot control system described in Chapter 5 using CADET. Properties of these systems are extracted from their traces and refusals with the assistance of the trace analyser which forms part of the interface. A number of these results are presented using the generated causality diagrams which themselves agree with and support the more formal analysis given in the last chapter. Moreover, the observation of possible livelocks among communicating processes in the study of the time critical refinement of the control sub-system given in Sub-section 6.4.3.5 was made, however the investigation of divergence or "fairness" is out of the scope of the thesis.

6.5.2 Practical Implementations

Despite the use of CADET being most suitable in executing highly abstracted system definitions directly, providing a powerful means of reasoning with design ideas, considerations such as data structuring and task allocation have to be taken care of in software for real systems. In these aspects, the CSP interpreter is handicapped, making the approach impractical for the implementation of real distributed control systems. Moreover, the large memory requirements and slow execution speeds of functional languages are among the other major drawbacks. This fact was observed during the use of the interpreter to investigate the parallel interactions of the six sub-systems defined in the First Level Model presented in Sub-section 6.4.3.4.

As the aim of software development is to produce executable programs that can be run on a concurrent hardware platform, practical parallel languages such as OCCAM [INMOS, 1988a; Jones and Goldsmith, 1988] and Parallel C are more promising candidates. In particular, since the language OCCAM is built on the formalism of CSP and was also treated as the assembly language of the INMOS Transputer [INMOS, 1988c], a direct transformation of the set of generic processes into OCCAM is feasible.

6.5.3 Mapping CSP into OCCAM

As the design of the language OCCAM is based on the theoretical model of CSP, which describes parallel computation using only *synchronisation*: no process engaged in a synchronising communication may proceed until its partners are simultaneously doing so, fundamental concepts such as the followings are shared between CSP and OCCAM.

- Processes can be combined in sequence using the SEQ construct.
- Processes can be combined in parallel using the PAR construct.
- Synchronised communication through channels which are declared with CHAN.
- Deterministic choice of input channels to the environment is provided using the ALT construct.
- Simple recursion with WHILE loops.
- Others common features includes timing and special process such as SKIP.

The similarities allows the combination of generic processes using the parallel, sequential and choice constructs which are parallel to the CSP counterparts. To illustrate this direct mapping from generic processes to OCCAM processes, the generic process *FSM* given by Equation 6.18 is implemented in OCCAM which is given by Figure 6.38.

$$FSM(a, b) \quad \widehat{=} \quad \mu X \bullet (a \rightarrow b \rightarrow X) \tag{6.18}$$

where the parameters of this generic process
a and b are simple synchronisation signals.

However, there exists a number of limitations in the implementation of OCCAM such that the streamlining of the translation from CSP to OCCAM is not as straight forward as it seems. The two main limitations are that OCCAM does not support the definition of mutual recursive processes and output guards for choices. The following sub-sections try to shed light into these problems.

6.5.3.1 Mutual Recursion

As OCCAM only supports simple recursion, processes such as *BUFFER*, *INIT* and *UPDATE* which involve mutual recursions, which is not supported, cannot be implemented in OCCAM directly. A possible solution to this problem is to convert the mutual recursive form into a *tail-recursive* form as described in Sub-section 4.4.2.2. The generic process *UPDATE* which is given by Equation 6.19

```
-- This is an Occam implementation of the
-- generic process FSM (a,b)
-- FSM (a,b) = uX. (a --> b --> X)
-- as in CSP


#USE  userio
BOOL  terminate:

WHILE  TRUE
    SEQ
        write.full.string (screen, "a")
        -- or any appropriate events

        write.full.string (screen, "b")
        -- or any appropriate events
```

Figure 6.38: An OCCAM implementation of the generic process *FSM*

can be implemented as an OCCAM process with nested WHILE loops. Such implementation is given by Figures 6.39.

$$\begin{aligned} UPDATE\,(c,a,b) \quad \equiv \quad & \mu\, Z \bullet (\mu\, X \bullet (c \rightarrow X \;\Box\; a \rightarrow X \;\Box\; b \rightarrow SKIP))\;; \\ & (\mu\, Y \bullet (b \rightarrow Y \;\Box\; a \rightarrow SKIP))\;; \\ & Z \end{aligned} \tag{6.19}$$

6.5.3.2 Deterministic Output Choice

Another limitation of OCCAM is that the deterministic choice ALT construct does not allow output from channels to be considered. This limitation is due to the difference in the notion of a CSP and an OCCAM channel. An OCCAM channel, unlike a CSP channel, is a mapping of a physical communication link which has a single "transmitting" and "receiving" ends. The implementation of the ALT construct only caters for a deterministic choice for input.

In cases when a deterministic choice between outputs is desirable such as allowing the use of a single output device — a terminal to display outputs from a number of parallel processes, measures in order to overcome this limitation have to be derived. In Jackson [1989], possible methods on how to replace a single output event with a handshake protocol so as to cater for deterministic choices for output is given.

6.5.4 Remarks on CSP to OCCAM Transformation

In this last section, although the discussion of a full implementation of the robot controller with generic processes in OCCAM is not given. The two modular OCCAM implementations of the generic processes *FSM* and *UPDATE* have be executed successfully on a parallel hardware platform using a BOO4 Transputer board in a PC. Although the two practical implementations are only prototypes, the feasibility of a direct transformation from CSP into OCCAM is established. The main advantage of the provision of such a direct mapping is to facilitate the refinement of high level specifications which are formally verified, all the way to practical executable programs. Despite the work on this front being preliminary, the provision has made the future development of a truly unified software development methodology more promising.

```
-- This is an Occam implementation of the
-- generic process UPDATE (c,a,b)
-- UPDATE ( c, a, b ) =
-- u Z . ( u X . ( c --> X | a --> X | b --> SKIP ));
--       ( u Y . ( b --> Y | a --> SKIP )) ; Z

#USE userio
BOOL terminate, looping:
CHAN of ANY a, b, c:

WHILE TRUE
  SEQ
    looping := TRUE
    WHILE looping
      ALT
        c ? ANY
          SEQ
            write.full.string (screen, "c")
            looping := TRUE
        a ? ANY
          SEQ
            write.full.string (screen, "a")
            looping := TRUE
        b ? ANY
          SEQ
            write.full.string (screen, "b")
            looping := FALSE
    looping := TRUE
    WHILE looping
      ALT
        b ? ANY
          SEQ
            write.full.string (screen, "b")
            looping := TRUE
        a ? ANY
          SEQ
            write.full.string (screen, "a")
            looping := FLASE
```

Figure 6.39: An OCCAM implementation of the generic process *UPDATE*

Chapter 7

Conclusion

7.1 An Overview

This thesis focusses on the application of formal methods to control software design and development through specification, verification and documentation. Its application is complementary to the established structured techniques resulting in an enhanced methodology. The unified method is aimed at benefiting from both approaches and is more complete in supporting the task of software development. With this novel methodology, various aspects of a design are addressed which are illustrated by a detailed study of a robot control system software design problem conducted at different levels of abstraction.

It is shown that by integrating formal mathematics into a structured development environment, vague design ideas are organised, system requirements are laid down according to the design and are expressed unambiguously with well-defined notations. Modular formalism with controllable abstraction is used to realize distributed system behaviour and the correctness of such realizations with respect to the specified properties is assured via formal reasoning. Different levels of abstract implementation are obtained via a step-by-step refinement through the application of structured criteria and formal generic building blocks, and the salient behaviours as defined by these models are investigated using mathematical rigour as well as by automatic execution of high level specifications using the custom-built Causality Diagram Evaluation Tool. Furthermore, the feasibility of mapping the formal generic processes to a parallel language and visual entities that are similar to those used by the structured techniques is demonstrated. It is concluded that the significance of a direct code mapping from the modular formalism is that of facilitating the streamlining of a correct implementation, while a precise and unambiguous visual system representation enables easy assimilation and the exchange of ideas.

This work has shown that adopting a formal approach to software development is not of mere

theoretical interest, the extra discipline and constraints that are imposed transforms a hand-waving system development problem into a methodical approach and provides a firmer under-pinning for a structured method, and at least enables the production of quality software in a systematic way which is of sound engineering significance.

7.2 Formal Methods in Control Software Design: An Evaluation

As the core of the investigation is centred on the applicability of an integrated method for producing reliable distributed control software, and with one of its components — the formal method (CSP) being a frontier approach as compared to its well established structured counterpart (Yourdon and DARTS), its contributions and benefit is often more ambivalent and imaginary. Through the studies in this thesis, a fair evaluation of formal methods in general and the applicability of CSP to distributed control software design is plotted.

In general terms, it is evident that the process of formal specification by which system behaviours are modelled mathematically, places additional constraints on a design because verification is the added objective. This extra discipline on software development facilitates a systematic approach and with the aid of mathematical reasoning: omission, ambiguity and contradiction can be identified so that the original design is rectified before a full implementation. Although conciseness of a specification is often a relative term, a formally specified design will benefit from its clarity in expressing concepts and relations as compared to using natural languages. As such, with a formal approach, correctness and hence reliability of software products is enhanced.

In the field of control, the value of applying a formal technique to software design varies markedly within the different hierarchies of a system. At an algorithmic level, the language of control is already that of sophisticated mathematics in which formal specification has little to add to what is already expressed unambiguously. It is seen from the example of a single loop control system in Section 4.2 that applying formal specification and verification at this level is both trivial and uninteresting.

However, taking a system point of view, since almost any complex control system can be decomposed into a number of inter-connected, cooperating functional blocks, modelling such systems with an appropriate formal semantic model (CSP) provides a metric for its design and evaluation. In this thesis, the use of CSP in the expressing and reasoning of distributed control system software at a high level, in which a jacketed control system model is adopted, was found to be

fruitful. It is well known that CSP already provides a sound mathematical basis for reasoning with concurrent communicating processes, and with the introduction of the abstraction tool set, specification predicates and generic processes, the level of abstraction is controlled via parameterisation and hence the modelling and reasoning with the components within different control hierarchies is demonstrated.

In the robot control case study, the use of specification predicates allows the behaviour of a system to be expressed in a more vivid and enlightening form which is easy to understand, such as the *ORDER* predicate for specifying the ordering property of the events of a process using trace semantics. Such a form has the advantage of being more concise and revealing than a full sentence of CSP notation while preserving its formality and expressive power. While the specification predicates are amenable to expressing behaviours, generic processes are introduced which mirror the relatively abstract concepts and implement directly the action described by the specification predicates. They form a set of formal building blocks for generating abstract system realizations based on the formally specified system requirements. From the two system realizations: Zeroth and First Level Models in Chapter 5, it is demonstrated that generic processes provide a set of scalable entities to be reused, modified and extended in building high level system implementations due to their parameterised and object orientated nature.

In CSP, verification follows an axiomatic approach by which predicate transformation is the basic operation. It is seen that proofs constructed in this manner establish a number of *satisfies relations* in the robot case study, for example the First Level Model is consistent with respect to the Zeroth Level Model. Also, system behaviours are predicted through the results of proofs such as the potential deadlock problem between the two cooperating sub-systems in the First Level Model. In proof construction, the property of a one-to-one mapping between a set of specification predicates and a generic process has simplified the process of verification at the atomic level, nevertheless, CSP proofs are generally complex and lengthy. On the other hand, the use of process algebra to re-write abstract process definitions is found to be more manageable as the operation is governed by well-defined laws, which are comparable to ordinary algebraic laws that are more familiar to engineers. Also, their strength of inferring system correctness is demonstrated from the various cases discussed in Chapters 2, 4 and 5, and is therefore of equal merit to the more elaborate axiomatic verification, once system definitions are available.

As the problem of deadlock is one of the major causes of software failure, special attention is paid to its detection in the various system models via verification. Using the two verification

strategies: process algebra and axiomatic proof, a particular occurrence of deadlock is identified among the communicating sub-systems in the robot case study which is due to the inter-dependency of shared resources. In conclusion, the following conditions are identified which leads to a possible deadlock:

- non-exclusive accessing of shared resources;
- circular waiting on communicating events; and
- incompatible or missing communication channels.

In addition to the limited analysis of deadlock discussed in this thesis, various authors have suggested alternative formal treatments for deadlock which address more specifically their identification. In Clark [1952], the topology of communication channels is mapped on to directed graphs and deadlock detection is achieved through the identification of cyclic behaviour within a graph. Other more theoretical approaches include the analysis of the execution of concurrent communication, such as those that share resources, are either accessed exclusively (critical path) at a particular transition or via shared communication. As in the robot control system, the introduction of the DATAMASTER in order to ensure that a system is free from deadlock has restricted the use of shared resources which in effect has limited the versatility of the system and introduced extra control overheads for enforcing such restrictions. This very fact has also been inspired by Tsutsui and Fujimoto [1987].

Even though the application of CSP to control software development has with it these benefits, for a designer who is not mathematically skilled, the initial training required should not be underestimated. Most importantly, before becoming committed deep into a particular verification, it is vital to decide *what* is the actual aim of a proof and *what* is really established as formal mathematics can only be used to prove the correctness of what is specified and not what a designer wants. Furthermore, there exist no well-defined metrics to quantify how well a formal expression matches its physical counterparts and hence, to claim that a system specification is 'complete' with respect to a design is often subjective. Moreover, formal models are theoretical and ideal concepts, their departure from real world scenarios are sometimes inevitable and in some circumstances, a physical interpretation is difficult, if not impossible to grasp. Apart from these philosophical views, a number of unique properties of control systems such as non-linearity, non-uniqueness and real-time dependency have presented difficulties to today's formal methods when attempting to formalise their analysis. It seems that even a practical formal method, such as CSP, can only

reason with system safety and liveness properties and its expressiveness and verifiability has fallen short on time-related properties, e.g. system performance which is often most interesting to a control system designer.

7.3 The Unified Approach: A Rational

From the foregoing discussion on structured and formal methods for producing software, it is learnt that both methods have their own weaknesses and strengths. The fact that both methods are complementary and mutually beneficial to one another prompts their integration with a view to achieving a complete unified method. As seen, the method described in Chapter 4 is a tight integration in which the two classes of methods cooperate in concert. An early definition of a system boundary concretised the 'system' requirements, and CSP formalism is introduced after convergence on an initial essential model. Formal refinement with the aid of mathematical reasoning, structured decomposition, automated simulation and formalised graphical assimilation takes a step-by-step transformation from a behavioural orientated description to a functional description. With this methodology, a design is verified and structured prior to implementation which reduces the risk of errors and re-design at a later stage.

During system refinement, an incremental transformation from essential requirements to implementation detail is adopted which is shown to take advantage of an iterative and incremental building of both sets of information so that modification and consistency checking are carried out at all stages. Although it is argued that in the process of software development, essential and implementation details should always be separated, however it is not the case for producing a realistic and practical piece of software. It is vital to realize that 'external knowledge' such as physical knowledge off the target system, past experience and previous implementations help to construct and improve a system description, and hence are the essential requirements. In addition, this 'external knowledge' also enables fundamental questions on system design and development, e.g. "what is actually needed to be specified?" and "what should be verified?" to be answered.

Separating the development of an implementation and a specification is perhaps more imaginary than real, and such separation can often lead to an incomplete and erroneous specification. The unified approach which incorporates an incremental model transformation is perhaps the answer to these radical questions: If everything is not known about the final system, does a designer know *what* to specify? Does such a specification clarify *how* this system is going to behave? Is there any practical *value* for specification and verification if the two issues are really separated?

Moreover, in Chapter 5, it is seen that the refinement from the Zeroth to the First Level Model and the re-design of high level system realisations are conducted in an methodical and systematical way as suggested by the unified method. However, it is undeniable that the mapping of specification-to-implementation is *one-to-many* and the process cannot be fully automated. So long as the designer follows the criteria of DARTS and using the formality of CSP in refinement, a correct and unambiguous system design will result.

Within the unified environment, abstract process simulation using CADET plays an important role in testing ideas and early designs. It is found that syntax error detection and type checking are competently undertaken by the Orwell interpreter and the visual representation of process interactions provided by the interface has a number of revealing features which are already summarised in Chapter 6. However, it is essential to emphasis that the usages of the tool should always be directed at the right level of abstraction and in a sensible manner. Otherwise, executing specifications through interpretation is actually a high-level form of code testing with similar pitfalls as software testing which is no replacement for formal proof.

All in all, although concise proof strategies are constructed on a number of salient cases and CADET developed is of limited functionality due to space constraints, they can be easily extended. Nonetheless, they have already illustrated the idea behind this thesis and further elaboration will have little to add to the argument.

7.4 Suggestions for Future Work

At this stage, the limit of applicability of formal methods such as CSP for expressing and reasoning with a distributed control system is still an open-ended question. As the work carried out in this thesis is only by an individual, it is perhaps not the best context to draw a definitive conclusion on this issue. Nonetheless, the methodology put forward in this thesis is a first attempt to bring about a partial solution to the problem.

In order to abet the development of formal methods for practical application, it is necessary to encourage their use by engineers on projects and design studies so that appropriate formal systems are derived which suit their needs. Work on providing an integrated environment for software design and development which follows the idea of the unified methodology proposed in this thesis will be much appreciated by the engineering community. The Causality Diagram Evaluation Tool is just an initial approach to producing a simple prototype of this idea. A more complete supporting environment, which includes automated theorem provers, syntax checkers, integrated databases

for axioms, laws and theorems and graphical tools which aim at maintaining the mechanistic and book-keeping aspects of formal methods at a minimum will lift some of the burden off a designer and help cater for the less mathematically trained engineer. In order to preserve the continuity of software development, not only the formal systems need to evolve, related programming languages which cater for real-time implementation and parallelism must also be developed. The provision of a direct mapping from CSP generic processes to a subset of OCCAM which was demonstrated in this context is just one of the may possibilities.

More specifically, the extension of the abstraction tool set to model systems in the time domain so as to enable formal reasoning of timed properties and constraints will be a big plus for tighter integration of the two classes of method for real-time systems. Possible enhancement to the graphical tool for the analysis of real-time systems includes: the incorporation of the timed interpreter based on the semantic model of timed CSP and a multi-dimensional representation of causality diagrams. Moreover, there are still properties that the existing model of CSP is capable of reasoning with. One of the properties is discussed briefly in Chapter 6 which might lead to system failure due to an infinite internal transition — livelock. Current research on stability and process divergence has shown that there is much potential in using CSP to investigate such properties. Last but surely not the least, the investigation of designing emergency handling processes for safety-critical systems with the aid of the causality diagrams could lead to more pragmatic areas of research.

Appendix A

CSP Notations

A subset of CSP notations used in this thesis for the construction of specifications and process definitions is given. The original set of CSP notations is given in Hoare [1985b].

Traces

In order to describe the sequence of events happening across a communicating channel or within a process, a trace is used. The following is a subset of notations for traces.

$\langle\rangle$ empty trace with no elements.

$\langle a, b, \ldots, z\rangle$ a trace containing the elements from a to z

$\#s$ the length of a trace s or the number of elements that the trace s contains.
If $s = \langle a,\ b,\ c,\ d\rangle$
then $\#s = 4$

$s \frown t$ if s and t are traces, the operation $\frown$ joins up the two traces in a specific order.
If $s = \langle a,\ b,\ c\rangle$
and $t = \langle d,\ e\rangle$
then $s \frown t = \langle a,\ b,\ c,\ d,\ e\rangle$

s_0 the first element of the trace s.
If $s \mathrel{\widehat{=}} \langle a, b, c, d\rangle$
then $s_0 = a$

s' the rest of the trace s with the first element being removed, it is also called the *tail* of s.
If $s = \langle a,\ b,\ c,\ d\rangle$
then $s' = \langle b,\ c,\ d\rangle$

$last\ (s)$ the last element of the trace s.
If $s = \langle a,\ b,\ c,\ d \rangle$
then $last\ (s) = d$

s^+ the initial trace of s, that is the trace s with the last element being removed.
If $s = \langle a,\ b,\ c,\ d \rangle$
then $s^+ = \langle a,\ b,\ c \rangle$

To summarise the above four trace operators, the following identities are given:

$$\begin{aligned} s &\equiv \langle s_0 \rangle \frown s' \\ &\equiv s^+ \frown \langle last\ (s) \rangle \end{aligned}$$

$s \leq t$ s is an initial trace of t; or a trace t begins with s. The relation is formally defined as: $\exists\, u \mid s \frown u = t$
If $t = \langle a,\ b,\ c,\ d,\ e \rangle$
and $s = \langle a,\ b \rangle$
then $s \leq t$ is satisfied.

$s \overset{n}{\leq} t$ The notation gives a precise relationship between the traces t and s:
$\exists\, n : \mathbb{N} \mid s \leq t \wedge (\#t \leq \#s + n)$.
For example if $t \mathrel{\widehat{=}} \langle a,\ b,\ c,\ d,\ e \rangle$
and $s \mathrel{\widehat{=}} \langle a,\ b \rangle$
then in this case, n = 3

s **in** t s is a sub-trace of t.
If $t = \langle a,\ b,\ c,\ d,\ e \rangle$
then $s \in \{\langle a \rangle, \langle b,\ c \rangle, \langle c,\ d,\ e \rangle\}$ are all sub-traces of t.

$s \downarrow b$ Given that s is a finite trace and b is an event, the notation gives the number of occurrence of event b within s.

$s \downarrow c$ Given that s is a finite trace and c is a communication channel, the notation gives a record of messages in form of a trace that are communicated acrossed channel c.
If $s \mathrel{\widehat{=}} \langle a.1,\ c.3,\ b.2,\ c.4 \rangle$
then $s \downarrow c = \langle 3, 4 \rangle$

$s \upharpoonright A$ The trace s is restricted to the set of elements A exclusively. It is formed by omitting all the elements in s which are not contained in the set A. The following two laws define this operations.

Law A.1

$$\langle\rangle \restriction A = \langle\rangle$$

Law A.2

$$(\langle a\rangle \frown s) \restriction A = \begin{cases} s \restriction A & \text{if } a \notin A \\ \langle a\rangle \frown (s \restriction A) & \text{if } a \in A \end{cases}$$

Alphabet

To describe the behaviour of a process , it is necessary to specify the relevant *events* that the process is capable of performing. The set of all possible events that are performed is the *alphabet* set. The following two notations specify the alphabet and a set of alphabet of a process.

A The Alphabet; it is the set of all symbols and channels that can happen in a process.

A^* The set of finite sequences which are formed from the elements in the set A.

Process

A process describes the behaviour pattern of an object. It could be regarded as a *black box* which is connected to the environment by named channels. As described in Hoare [1985b]:

> *"a process engages in an unbounded sequence of communication, each of which is either an input from the input wire or an output to an output wire. If the environment is not ready for communication, the process waits for it to become so, and vice versa. There is no buffering in the wire and the act of communication requires simultaneous synchronised participation of both sender and receiver."*

$\alpha\ P$ In addition to the described alphabet notations, the alphabet set of a process is defined by $\alpha\ P$. A communication is an atomic action and in the alphabet of a communicating process, a particular communication is defined by a compound symbol: $c.m$ which denotes a communication-of message m across channel c.

$\alpha\ c\ (P)$ The notation gives the set of messages that are communicated across a specified channel. In this case, the set of messages that are communicated across channel c is recorded.

traces (P) Traces play a central role in recording, describing and understanding the behaviour of a system. The sequence of events that is performed by process P is given by this notation. The following are the laws on traces of a process:

$$\langle\rangle \in traces\,(P) \tag{A.1}$$

$$(s \frown t) \in traces\,(P) \Rightarrow s \in traces\,(P) \tag{A.2}$$

$$traces\,(P) \subseteq (\alpha P)^* \tag{A.3}$$

Further notations and operations on processes are discussed in the context of the thesis.

Appendix B

Proofs of the General Proof Rules for Parallel Processes

The following proofs establish the validity of the two theorems for proving properties of placing generic processes in parallel.

Proof B.1 *Proof of theorem 4.1*

Given that P is a non-diverging process, Q is a valid system, A is an alphabet set and $S(tr)$ is a specification written in the traces semantics such that:

$$\begin{array}{ll} & \forall\, tr \mid S(tr) \Leftrightarrow S(tr \upharpoonright A) \\ \wedge & P \textbf{ sat } S(tr) \\ \wedge & A \subseteq \alpha P \\ \Rightarrow & (Q \parallel P) \textbf{ sat } S(tr) \end{array}$$

Proof:

as Q sat TRUE:

$$\begin{array}{ll} & (Q \parallel P) \textbf{ sat} \\ & \quad \exists\, X, Y \mid ref = (X \cup Y) \wedge (S(tr \upharpoonright \alpha P)) \wedge TRUE \\ \Rightarrow & \exists\, X, Y \mid ref = (X \cup Y) \wedge (S(tr \upharpoonright \alpha P) \upharpoonright A) \\ \Rightarrow & \exists\, X, Y \mid ref = (X \cup Y) \wedge (S(tr \upharpoonright A)) \\ \Rightarrow & S(tr \upharpoonright A) \wedge (\exists\, X, Y \mid ref = (X \cup Y)) \\ \Rightarrow & S(tr) \end{array}$$

this completes the proof for the first theorem. □

Proof B.2 *Proof of theorem 4.2*

For some event e, and non-diverging processes P_1 and P_2 with corresponding specifications $D_1(tr)$ and $D_2(tr)$, are such that for $i \in \{1,2\}$:

$$\begin{array}{ll} \text{If} & (e \in P_i) \Rightarrow \left(\begin{array}{c} P_i \textbf{ sat } (D_i(tr) \Rightarrow e \notin ref) \\ \wedge \quad \forall tr \mid C(tr) \Rightarrow D_i(tr \upharpoonright \alpha P_i) \end{array} \right) \\ \text{then} & (P_1 \parallel P_2) \textbf{ sat } (C(tr) \Rightarrow e \notin ref) \end{array}$$

Proof:

Three cases can be identified:

1. Neither αP_1 nor αP_2 contains e, this implies that $(P_1 \parallel P_2)$ cannot refuse e.
2. e is in either αP_1 or αP_2.
3. e is in both αP_1 and αP_2.

Using case analysis:

Case: $e \notin (\alpha P_1 \cup \alpha P_2$

$$\begin{array}{ll} \Rightarrow & e \notin \alpha(P_1 \parallel P_2) \\ \Rightarrow & e \notin ref \end{array}$$

Case: $(e \in \alpha P_1 \wedge e \notin \alpha P_2) \vee (e \notin \alpha P_1 \wedge e \in \alpha P_2)$

Subcase: $e \in \alpha P_1 \wedge e \notin \alpha P_2$

$$\begin{array}{lll} & & P_2 \textbf{ sat } ref \subseteq \alpha P_2 \\ \Rightarrow & & (P_1 \parallel P_2) \textbf{ sat } \exists X, Y \mid ref = X \cup Y \\ & \wedge & (D(tr \upharpoonright \alpha P_1) \Rightarrow e \notin X) \\ & \wedge & (Y \subseteq \alpha P_2) \end{array}$$

$$\begin{array}{lll} \text{Assume} & & C(tr) = TRUE \\ \Rightarrow & & \exists X, Y \mid ref = X \cup Y \\ & \wedge & (D(tr \upharpoonright \alpha P_1) \Rightarrow e \notin X) \end{array}$$

$$\wedge \quad (Y \subseteq \alpha P_2) \wedge (D(tr \restriction \alpha P_1))$$

$$\Rightarrow \qquad \exists X, Y \mid ref = X \cup Y$$

$$\wedge \quad (D(tr \restriction \alpha P_1) \Rightarrow e \notin X) \wedge (D(tr \restriction \alpha P_1))$$

$$\wedge \quad (Y \subseteq \alpha P_2)$$

$$\Rightarrow \qquad \exists X, Y \mid ref = X \cup Y$$

$$\wedge \quad e \notin X$$

$$\wedge \quad e \notin Y$$

$$\Rightarrow \qquad e \notin ref$$

Subcase: $e \notin \alpha P_1 \wedge e \in \alpha P_2$

The proof is similar to the above case;

$$\Rightarrow \qquad \exists X, Y \mid ref = X \cup Y$$

$$\wedge \quad e \notin X$$

$$\wedge \quad e \notin Y$$

$$\Rightarrow \qquad e \notin ref$$

Case: $e \in (\alpha P_1 \cap \alpha P_2)$

$$(P_1 \parallel P_2) \textbf{ sat}$$

$$\exists X, Y \mid ref = X \cup Y$$

$$\wedge \quad (D(tr \restriction \alpha P_1) \Rightarrow e \notin X)$$

$$\wedge \quad (D(tr \restriction \alpha P_2) \Rightarrow e \notin Y)$$

Assume $\qquad C(tr) = TRUE$

$$\Rightarrow \qquad \exists X, Y \mid ref = X \cup Y$$

$$\wedge \quad (D(tr \restriction \alpha P_1) \Rightarrow e \notin X) \wedge (D(tr \restriction \alpha P_1))$$

$$\wedge \quad (D(tr \restriction \alpha P_2) \Rightarrow e \notin Y) \wedge (D(tr \restriction \alpha P_2))$$

$$\Rightarrow \qquad \exists X, Y \mid ref = X \cup Y$$

$$\wedge \quad e \notin X$$

$$\wedge \quad e \notin Y$$

Property B.1

$\Rightarrow \quad e \notin ref$

The results from the three cases establish the theorem □

Property B.1

Given $(A \Rightarrow B) \wedge A \equiv B$

if $(A \Rightarrow B) = TRUE$

$\wedge \quad A = TRUE$

$\Rightarrow \quad B = TRUE$

Appendix C

Proofs for the Properties of Generic Process INIT and UPDATE

C.1 Properties 4.7 and 4.10

The two properties having a similar form are proved simultaneously by considering a two dimensional product space. It is shown in Jackson [1989] that a continuous specification on such a space can be obtained by *conjoining* the specifications for each process. It is seen that the recursions within each process is guarded and it is noted that the least element (STOP, STOP) satisfies the specifications.

Proof C.1 *Proof that INIT and UPDATE satisfy Properties 4.7 and 4.10*

The proof is carried out by recursive induction, and the following inductive hypothesis is made:

$$\left(\begin{array}{ll} & (INIT\ (c,\ a,\ b)\ \textbf{sat}\ c\ IFEND\ (b,\ a)) \\ \wedge & (UPDATE\ (c,\ a,\ b)\ \textbf{sat}\ c\ IFSTART\ (a,\ b)) \end{array}\right)$$

Firstly, consider the first conjunct in the assumption:

$$\begin{array}{lll} INIT & (c,\ a,\ b) \mathrel{\widehat{=}} & \\ & (a \rightarrow UPDATE \mid b \rightarrow INIT) & \text{def of Process 4.7} \\ \textbf{sat} & \left(\begin{array}{ll} & (tr = \langle\rangle \wedge \{a,\ b\} \not\subseteq ref) \\ \vee & (tr_0 = a \wedge (c\ IFSTART\ (a,\ b))[tr'/tr]) \\ \vee & (tr_0 = b \wedge (c\ IFEND\ (b,\ a))[tr'/tr]) \end{array}\right) & \text{Inductive Hypothesis} \end{array}$$

A case analysis then follows:

Case: $tr = \langle\rangle \wedge \{a, b\} \not\subseteq ref$

A trivial case, and the condition can be proved by the law on refusals for guarded processes.

Case: $tr = \langle a \rangle \frown tr'$

$$
\begin{array}{lll}
 & last\ (tr) = c & \text{from assumption} \\
\Rightarrow & last\ (\langle a \rangle \frown tr') = c & \\
\Rightarrow & last\ (tr') = c &
\end{array}
$$

$$
\begin{array}{lll}
 & (c\ IFSTART\ (a, b))[tr'/tr] & \text{Ind Hyp} \\
\Rightarrow & \left(\begin{array}{ll} & tr' \restriction \{a, b\} = \langle\rangle \\ \vee & (tr' \restriction \{a, b\} \neq \langle\rangle \wedge last\ (tr' \restriction \{a, b\}) = a) \end{array} \right) & \text{def of } IFSTART
\end{array}
$$

There are two subcases to follow:

Subcase: $tr' \restriction \{a, b\} = \langle\rangle$

$$
\begin{array}{lll}
 & tr = \langle a \rangle \frown tr' & \text{from assumption} \\
\Rightarrow & tr \restriction \{a, b\} = \langle a \rangle \frown (tr' \restriction \{a, b\}) & \text{Law A.2} \\
\Rightarrow & tr \restriction \{a, b\} = \langle a \rangle \frown \langle\rangle = \langle a \rangle & \text{Law A.1} \\
\Rightarrow & tr\ END\ (b, a) & \text{def of } END
\end{array}
$$

Subcase: $tr' \restriction \{a, b\} \neq \langle\rangle \wedge last\ (tr' \restriction \{a, b\}) = a$

$$
\begin{array}{lll}
 & tr \restriction \{a, b\} = \langle a \rangle \frown (tr' \restriction \{a, b\}) & \text{Law A.2} \\
 & last\ (tr \restriction \{a, b\}) = last\ (\langle a \rangle \frown (tr' \restriction \{a, b\})) & \\
 & \qquad = last\ (tr' \restriction \{a, b\}) & \\
 & \qquad = a & \\
\Rightarrow & tr\ END\ (b, a) & \text{def of } END
\end{array}
$$

Case: $tr = \langle b \rangle \frown tr'$

$$
\begin{array}{lll}
 & last\ (tr) = c & \text{from assumption} \\
\Rightarrow & last\ (tr') = c &
\end{array}
$$

$$
\begin{array}{lll}
 & (c\ IFEND\ (b, a))[tr'/tr] & \text{from assumption} \\
\Rightarrow & tr' \restriction \{a, b\} \neq \langle\rangle \wedge (last\ (tr' \restriction \{a, b\}) = a) & \text{def of } IFEND
\end{array}
$$

$$
\begin{array}{lll}
\text{if} & tr \restriction \{a, b\} = (\langle b \rangle \frown tr') \restriction \{a, b\} & \\
 & \qquad = \langle b \rangle \frown (tr' \restriction \{a, b\}) \neq \langle\rangle & \\
\text{and} & last\ (tr \restriction \{a, b\}) = last\ (\langle b \rangle \frown (tr' \restriction \{a, b\})) & \\
 & \qquad = last\ (tr' \restriction \{a, b\}) & \\
 & \qquad = a & \\
\Rightarrow & tr' \restriction \{a, b\} \neq \langle\rangle \wedge (last\ (tr' \restriction \{a, b\}) = a) & \\
\Rightarrow & tr\ END\ (b, a) & \text{def of } END
\end{array}
$$

Hence, $INIT$ **sat** $c\ IFEND\ (b, a)$ is established. □

The second conjunct is considered:

$$UPDATE\ (c,\ a,\ b) \mathrel{\widehat{=}} (a \rightarrow UPDATE \mid b \rightarrow INIT \mid c \rightarrow UPDATE) \qquad \text{def of Process 4.7}$$

$$\textbf{sat} \left(\begin{array}{ll} & (tr = \langle\rangle \wedge \{a,\ b,\ c\} \not\subseteq ref) \\ \vee & (tr_0 = a \wedge (c\ IFSTART\ (a,\ b))[tr'/tr]) \\ \vee & (tr_0 = b \wedge (c\ IFEND\ (b,\ a))[tr'/tr]) \\ \vee & (tr_0 = c \wedge (c\ IFSTART\ (a,\ b))[tr'/tr]) \end{array} \right) \qquad \text{Ind Hyp}$$

A case analysis then follows:

Case: $tr = \langle\rangle$

$$\begin{array}{lll} & refusals\ (UPDATE(c,\ a,\ b)) & \\ \equiv & refusals\ (a \rightarrow UPDATE \mid b \rightarrow INIT \mid c \rightarrow UPDATE) & \\ \equiv & (refusals\ (a \rightarrow UPDATE) \cup refusals\ (b \rightarrow INIT) \cup & \text{prop of refusals} \\ & refusals\ (c \rightarrow UPDATE)) & \text{when } tr \neq \langle\rangle \\ \Rightarrow & \{a,\ b,\ c\} \cap ref = \varnothing & \\ \Rightarrow & \{a,\ b,\ c\} \not\subseteq ref & \end{array}$$

Case: $tr = \langle a \rangle \frown tr'$

$$\begin{array}{lll} & last\ (tr) = c & \text{assumption} \\ \Rightarrow & last\ (tr') = c & \\ & (c\ IFSTART\ (a,\ b))[tr'/tr]) & \text{Ind Hyp} \\ \Rightarrow & \left(\begin{array}{ll} & tr' \restriction \{a,\ b\} = \langle\rangle \\ \vee & (tr' \restriction \{a,\ b\} \neq \langle\rangle \wedge last\ (tr' \restriction \{a,\ b\}) = a) \end{array} \right) & \text{def of } IFSTART \end{array}$$

There are two subcases to follow:

Subcase: $tr' \restriction \{a,\ b\} = \langle\rangle$

$$\begin{array}{lll} & tr = \langle a \rangle \frown tr' & \text{from assumption} \\ \Rightarrow & tr \restriction \{a,\ b\} = \langle a \rangle \frown (tr' \restriction \{a,\ b\}) & \text{Law A.2} \\ \Rightarrow & tr \restriction \{a,\ b\} = \langle a \rangle \frown \langle\rangle = \langle a \rangle & \text{Law A.1} \\ \Rightarrow & last\ (tr \restriction \{a,\ b\}) = a & \\ \Rightarrow & tr\ START\ (a,\ b) & \text{def of } START \end{array}$$

Subcase: $tr' \restriction \{a,\ b\} \neq \langle\rangle \wedge last\ (tr' \restriction \{a,\ b\}) = a$

$$\begin{array}{lll} & last\ (tr) = c & \text{assumption} \\ \Rightarrow & last\ (tr') = c & \end{array}$$

$$\begin{array}{lll} \text{if} & tr = \langle b \rangle \frown tr' & \\ & tr \restriction \{a,\ b\} \neq \langle\rangle & \\ \text{and} & last\ (tr \restriction \{a,\ b\}) = last\ (\langle a \rangle \frown (tr' \restriction \{a,\ b\})) & \\ & \qquad = last\ (tr' \restriction \{a,\ b\}) & \\ & \qquad = a & \\ \Rightarrow & tr' \restriction \{a,\ b\} \neq \langle\rangle \wedge (last\ (tr' \restriction \{a,\ b\}) = a) & \\ \Rightarrow & tr\ START\ (b,\ a) & \text{def of } START \end{array}$$

Case: $tr = \langle b \rangle \frown tr'$

$$\begin{array}{ll} & last\ (tr) = c & \text{assumption} \\ \Rightarrow & last\ (tr') = c & \end{array}$$

$$\begin{array}{ll} & (c\ IFEND\ (b,\ a))[tr'/tr] & \text{Ind Hyp} \\ \Rightarrow & tr' \restriction \{a,\ b\} \neq \langle\rangle \wedge (last\ (tr' \restriction \{a,\ b\}) = a) & \\ \Rightarrow & tr \restriction \{a,\ b\} \neq \langle\rangle \wedge (last\ (tr \restriction \{a,\ b\}) = a) & \\ \Rightarrow & tr\ START\ (a,\ b) & \text{def of } START \end{array}$$

Case: $tr = \langle c \rangle \frown tr'$

$$\begin{array}{ll} & last\ (tr) = c & \text{assumption} \\ \Rightarrow & last\ (\langle c \rangle \frown tr') = c & \\ \Rightarrow & tr' \restriction \{c\} = \langle\rangle \vee last\ (tr') = c & \end{array}$$

From the last expression, there are two cases to follow:

Subcase: $tr' \restriction \{c\} = \langle\rangle$

$$\begin{array}{ll} \Rightarrow & \text{the first occurrence of } c \\ \Rightarrow & tr' = \langle\rangle \\ \Rightarrow & tr \restriction \{a,\ b\} = \langle\rangle \\ \Rightarrow & tr\ START\ (a,\ b) \qquad \text{def of } START \end{array}$$

Subcase: $last\ (tr') = c$

$$\begin{array}{ll} & (c\ IFSTART\ (a,\ b))[tr'/tr] & \text{Ind Hyp} \\ \Rightarrow & \left(\begin{array}{ll} & tr' \restriction \{a,\ b\} = \langle\rangle \\ \vee & (tr' \restriction \{a,\ b\} \neq \langle\rangle \wedge last\ (tr' \restriction \{a,\ b\}) = a) \end{array} \right) & \text{def of } IFSTART \end{array}$$

There are two cases to follow:

Subsubcase: $tr' \restriction \{a,\ b\} = \langle\rangle$

$$\begin{array}{ll} \Rightarrow & tr \restriction \{a,\ b\} = \langle\rangle \\ \Rightarrow & tr\ START\ (a,\ b) \end{array}$$

Subsubcase: $tr' \restriction \{a,\ b\} \neq \langle\rangle \wedge last\ (tr' \restriction \{a,\ b\}) = a$

$$\begin{array}{ll} \Rightarrow & tr \restriction \{a,\ b\} \neq \langle\rangle \wedge last\ (tr \restriction \{a,\ b\}) = a \\ \Rightarrow & tr\ START\ (a,\ b) \end{array}$$

Hence, *UPDATE* **sat** $c\ IFSTART\ (a,\ b)$ is established and so is the original inductive hypothesis. □

C.2 Properties 4.8 and 4.11

Similar to the last proof, the two similar properties are proved simultaneously by considering a two dimensional product space.

Proof C.2 *Proof that INIT and UPDATE satisfy Properties 4.8 and 4.11*

The proof is carried out by recursion induction, and the following inductive hypothesis is made:

$$\left(\begin{array}{ll} & (INIT\ (c,\ a,\ b)\ \mathbf{sat}\ \{a,\ b\} \not\subseteq ref) \\ \wedge & (UPDATE\ (c,\ a,\ b)\ \mathbf{sat}\ \{a,\ b\} \not\subseteq ref) \end{array}\right)$$

Firstly, consider the first conjunct in the assumption:

$$\begin{array}{lll} INIT & (c,\ a,\ b) \mathrel{\widehat{=}} & \\ & (a \rightarrow UPDATE \mid b \rightarrow INIT) & \text{def of Process 4.7} \\ \mathbf{sat} & \left(\begin{array}{ll} & (tr = \langle\rangle \wedge \{a,\ b\} \not\subseteq ref) \\ \vee & (tr_0 = a \wedge \{a,\ b,\ c\} \not\subseteq ref) \\ \vee & (tr_0 = b \wedge \{a,\ b\} \not\subseteq ref) \end{array}\right) & \text{Inductive Hypothesis} \\ \\ refusals & (INIT) \equiv & \\ & (refusals\ (a \rightarrow UPDATE) \cup refusals\ (b \rightarrow INIT)) & \text{law of refusals} \\ & & \text{when } tr \neq \langle\rangle \\ \\ \Rightarrow & \{a,\ b\} \not\subseteq ref & \end{array}$$

The second conjunct is considered:

$$\begin{array}{lll} UPDATE & (c,\ a,\ b) \mathrel{\widehat{=}} & \\ & (a \rightarrow UPDATE \mid b \rightarrow INIT \mid c \rightarrow UPDATE) & \text{def of Process 4.7} \\ \\ \mathbf{sat} & \left(\begin{array}{ll} & (tr = \langle\rangle \wedge \{a,\ b,\ c\} \not\subseteq ref) \\ \vee & (tr_0 = a \wedge \{a,\ b,\ c\} \not\subseteq ref) \\ \vee & (tr_0 = b \wedge \{a,\ b\} \not\subseteq ref) \\ \vee & (tr_0 = c \wedge \{a,\ b,\ c\} \not\subseteq ref) \end{array}\right) & \text{Ind Hyp} \\ \\ refusals & (UPDATE) \equiv & \\ & (refusals\ (a \rightarrow UPDATE) \cup refusals\ (b \rightarrow INIT) \cup & \\ & refusals\ (c \rightarrow UPDATE)) & \text{law of refusals} \\ & & \text{when } tr \neq \langle\rangle \\ \\ \Rightarrow & \{a,\ b\} \not\subseteq ref & \end{array}$$

From the two results, the refusals properties are established. □

C.3 Properties 4.5 and 4.6

The proof of the closure properties of the two generic processes follows that from Proof 4.1 with the aid of laws on choices and is not repeated here.

Appendix D

Proof of Deadlock Freedom for the Process Control_2

The following process algebra establishes that the process CONTROL_2 is free from deadlock.

Proof D.1

Given the following:

$$\text{CONTROL}_2 \;\widehat{=}\; (\text{OPINT} \parallel \text{TRAJGEN}_1 \parallel \text{CTL}) \tag{D.1}$$

$$\text{OPINT} \;\widehat{=}\; FSM\ (Opin.e_0,\ Opout.e_1) \tag{D.2}$$

$$\text{TRAJGEN}_1 \;\widehat{=}\; \mu X \bullet (PERM\ (Opout.e_1,\ P,\ Logdata.e_2,\ Q)\ ;\ X) \tag{D.3}$$

$$P \;\widehat{=}\; Traj\ !\ e_3 \rightarrow Logdata\ ?\ e_2 \rightarrow SKIP$$

$$Q \;\widehat{=}\; Traj\ !\ e_3 \rightarrow SKIP$$

$$\text{CTL} \;\widehat{=}\; MUX\ (Traj.e_3,\ Cdata.e_5,\ Drive.e_4) \tag{D.4}$$

$$\text{where} \quad \left(\begin{array}{ll} & e_0 \in COMMAND \\ \wedge & e_1 \in VCOMMAND \\ \wedge & e_2 \in LDATA \\ \wedge & e_3 \in TRAJ \\ \wedge & e_4 \in DRIVE \\ \wedge & e_5 \in CDATA \end{array} \right) \tag{D.5}$$

To show that the three parallel processes are free from deadlock, it is assumed that the environment of these processes is ready to engage in any communications at all time. External

communications of CONTROL$_2$ will not affect the liveness of the overall system and can therefore be abstracted from the analysis. Also, at this level of design, data dependency is not considered and hence the messages given by Equation D.5 is immaterial in the process definitions. With these assumptions and simplifications, the reduced definitions of the three processes: OPINT$_R$, TRAJGEN$_{1R}$ and CTL$_R$ are given by the following process algebra.

$$\text{OPINT}_R \;\widehat{=}\; \mu X \bullet (\ \mathit{Opout}\,! \rightarrow X) \tag{D.6}$$

$$\begin{aligned} \text{TRAJGEN}_{1R} \;\widehat{=}\; & \mu Y \bullet (\ \mathit{Opout}\,? \rightarrow \mathit{Traj}\,! \rightarrow Y \\ & \quad \mid \mathit{Traj}\,! \rightarrow Y) \end{aligned} \tag{D.7}$$

$$\text{CTL}_R \;\widehat{=}\; \mu Z \bullet (\ \mathit{Traj}\,? \rightarrow Z) \tag{D.8}$$

Expanding the overall process CONTROL$_2$ using the three reduced definitions parallel components D.6, D.7 and D.8:

$$\begin{aligned} \text{CONTROL}_2 \;\widehat{=}\; & (\text{OPINT}_R \parallel \text{TRAJGEN}_{1R} \parallel \text{CTL}_R) \\ \equiv\; & (\mu X \bullet (\ \mathit{Opout}\,! \rightarrow X)) \\ & \parallel (\mu Y \bullet (\ \mathit{Opout}\,? \rightarrow \mathit{Traj}\,! \rightarrow Y \\ & \qquad \mid \mathit{Traj}\,! \rightarrow Y)) \\ & \parallel (\mu Z \bullet (\ \mathit{Traj}\,? \rightarrow Z)) \\ \equiv\; & (\ \mathit{Opout}\,! \rightarrow \mathit{Traj}\,! \rightarrow (\text{OPINT}_R \parallel \text{TRAJGEN}_{1R} \parallel \text{CTL}_R) \\ & \bar{\mid}\ \mathit{Traj}\,! \rightarrow (\text{OPINT}_R \parallel \text{TRAJGEN}_{1R} \parallel \text{CTL}_R)) \\ \equiv\; & \mu X \bullet (\ \mathit{Opout}\,! \rightarrow \mathit{Traj}\,! \rightarrow X \\ & \qquad \mid \mathit{Traj}\,! \rightarrow X) \end{aligned}$$

The final definition of CONTROL$_2$ is a *recursive* process. This abstract process definition clearly defines the two unique alternative scenarios that this process is capable of undertaking and establishes the freedom of deadlock among these three parallel processes. □

Bibliography

[Abrial, 1988] J. R. Abrial. *B User Manual.* Technical report, Programming Research Group, Oxford University, 1988.

[Alford, 1977] M. W. Alford. *A Requirements Engineering Methodology.* IEEE Transactions of Software Engineering, 1(SE-3):180–193, 1977.

[Ambler and et. la., 1976] A. L. Ambler and et. la. *Gypsy: A Language for Specification and Implementation of Verifiable programs.* In ACM Conference on Language Design for Reliable Software, 1976.

[Aström and Haggland, 1984] K. J. Aström and T. Haggland. *Automatic Tuning of PID Regulators.* Lund Institute, Sweden, 1984.

[Aström and Wittenmark, 1984] K. J. Aström and B. Wittenmark. *Computer Control Systems.* Prentics Hall International, Englewood Cliff, N.J., 1984.

[Banaszak, 1986] Z. Banaszak. *Perspectives of Automatic Real-time Program Synthesis.* In IFAC Real Time Programming, Proceeding of the 14th IFAC Workshop, pages 19 – 27, Lake Balaton, Hungary, 26 – 28 May 1986.

[Bejczy, 1974] A. K. Bejczy. *Robot Arm Dynamics and Control.* Technical Report 33–699, Jet Propulsion Laboratory, Pasadena, CA, 1974.

[Bennett, 1990] S. Bennett. *Computer Control of Real-time Processes — Design of software for real-time system.* Peter Peregrinus Ltd., 1990.

[Berg *et al.*, 1982] H. K. Berg, W. E. Boebert, W. R. Franta, and T. G. Moher. *Formal Methods of Program Verification and Specification.* Prentice Hall Software Series, 1982.

[Bird and Walder, 1988] R. Bird and P. Walder. *Introduction to Functional Programming.* Prentice Hall International, 1988.

[Blackhouse, 1986] Roland C. Blackhouse. *Program Construction and Verification.* Prentice Hall International, 1986.

[Blikle and Mazurkiewicz, 1972] A Blikle and A. Mazurkiewicz. *An Algebraic Approach to the Theory of Programs, Algorithms, Languages and Recursiveness.* Mathematical Foundation of Computer Science, Warsaw, Poland, 1972.

[Bloomfield and Froome, 1986] R. E. Bloomfield and P. K. D. Froome. *The Application of Formal Methods to the Development of High Integrity Software.* IEEE Transaction on Software Engineering, 12(9):988–993, 1986.

[Bloomfield *et al.*, 1989] Robin Bloomfield, Peter Froome, and Brian Monahan. *Specbox: A Toolkit for BSI-VDM.* SafetyNet, pages 4–6, Apr/May/June 1989.

[Boriani, 1987] Dario V. Boriani. *Software Engineering for Control (first year report).* Technical report, Department of Engineering Science, University of Oxford, Dec. 1987.

[Boriani, 1990] Dario V. Boriani. *Software Engineering for Control.* PhD thesis, University of Oxford, Department of Engineering Science, 1990.

[Bowen, 1987a] J. Bowen. *The Formal Specification of a Microprocessor Instruction Set.* Technical Report PRG – 60, Programming Research Group, Oxford University, Jan. 1987.

[Bowen, 1987b] J. Bowen. *The Specification of Network Services.* Technical report, Programming Research Group, Oxford University, 1987.

[Bowen, 1988] J. Bowen. *Specifying System Implementations in Z.* Technical report, Programming Research Group, Oxford University, Feb. 1988.

[Brookes and Hoare, 1984] S. D. Brookes and C. A. R. Hoare. *A Theory of Communicating Sequential Processes.* Journal of the Association for Computing and Machinery, 31(3):560 – 599, July 1984.

[Brookes, 1983] Stephen David Brookes. *A Model for Communicating Sequential Processes.* Technical report, Programming Research Group, Oxford University, January 1983.

[Brown *et al.*, 1988] D.W. Brown, C.D. Carson, W.A. Montgomery, and P. Zislis. *Software Specification and Prototyping Technology.* AT & T Technical Journal, July/August 1988.

[Burstall and Goguen, 1986] R. M. Burstall and J. A. Goguen. *The semantics of CLEAR.* In Proc. 1979 Copenhagen Winter School Abstract Software Specification, Lecture Notes in Compter Science 86, Springer-Verlag, 1986.

[Camilieri *et al.*, 1986] A. Camilieri, M. Gordon, and T. Melham. *H.D.L. Description to Guarantee Correct Circuit Designs.* In Proc. of the IFIP WG 10.2 Working Conference, Grenoble, France, September 1986.

[Chen and Hoare, 1981a] Zhou C. Chen and C. A. R. Hoare. *Partial Correctness of Communicating Sequental Processes.* Technical report, Programming Research Group, May 1981.

[Chen and Hoare, 1981b] Zhou C. Chen and C. A. R. Hoare. *Partial Correctness of Communication Protocol.* Technical report, Programming Research Group, Oxford University, May 1981.

[Chen, 1982] Zhou Chao Chen. *The Consistency of the Calculus of Total Correctness for Communicating Processes.* Technical report, Programming Research Group, Oxford University, Feb. 1982.

[Church, 1951] A. Church. *The Calculi of Lambda-Conversion*, in Annal of Mathematical Studies, volume 6. Princeton University Press, Princeton, 1951.

[Clark, 1952] W. Clark. *The Gantt Chart.* Pitman Press, London, 1952.

[Cleaveland *et al.*, 1989] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. *The Concurrent Workbench Operating Instructions.* Technical Report LFCS-TN-10, Department of Computer Science, University of Edinburgh., October 1989.

[Cohn, 1989] Avra Cohn. *The Notion of Proof in Hardware Verification.* Journal of Automated Reasoning, 5:127–139, 1989.

[Cox *et al.*, 1988] I. J. Cox, D. A. Kapilow, W. J. Kropfl, and J. E. Shopiro. *Real-time Software for Robotics.* AT and T Technical Journal, 67(2):61–72, March/April 1988.

[Craig, 1986] John J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison and Wesley, 1986.

[Cullyer, 1989] W. J. Cullyer. *Using VIPER in Railway Signalling.* Safety Net, 4:7 – 12, Jan/Feb/Mar 1989.

[Daniel and Irving, 1988] R. W. Daniel and M. A. Irving. *A Compliant Direct Drive Industrial Robot Arm.* In Proceedings of Romansy 88, September 1988.

[Darringer and King, 1978] J. A. Darringer and J. C. King. *Application of Symbolic Execution of Program Testing.* Computer, 11(4), 1978.

[Davies and Schneider, 1989a] Jim Davies and Steve Schneider. *Factorizing Proofs in Timed CSP.* Technical Report in PRG-75, Programming Research Group, Oxford University, 1989.

[Davies and Schneider, 1989b] Jim Davies and Steve Schneider. *An Introduction to Timed CSP.* Technical Report PRG-75, Programming Research Group, Oxford University, 1989.

[Davies, 1987] Jim Davies. *Assisted Proofs for Communicating Sequential Processes.* Technical report, Programming Research Group, Oxford University, 1987.

[de Simonbe and Vergamini, 1989] Robert de Simonbe and Didier Vergamini. *Aboard AUTO.* Technical Report 111, Institut National de Recherche en Informatique et en Automatique, October 1989.

[DeMarco, 1978] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, New York, 1978.

[Dickinson, 1988a] Mark Dickinson. *A Parallel Processing Controller Architecture for Fine Force Control of an Industrial Robot, CEGB research fellowship final report.* Technical report, Department of Engineering Science, Oxford University, July 1988.

[Dickinson, 1988b] Mark Dickinson. *A Parallel-processing Controller Architecture for Force Control of an Industrial Robot.* In IEE Second Workshop on Parallel Processing and Control, Bangor, Wales, 1988. IEE.

[Dijkstra, 1976] E. W. Dijkstra. *Discipline of Programming.* Englewood Cliffs: Prentice Hall, 1976.

[Ditt, 1986] W. Ditt. *Mapping an Algebraic Specification onto a Modula-2 Program.* In IFAC Real Time Programming, Proceeding of the 14th IFAC Workshop, pages 11 – 19, Lake Balaton, Hungary, 26 – 28 May 1986.

[Djian and Probert, 1990] D. Djian and P. J. Probert. *Case Study in Formal Specification for Distributed Sensor Integration.* Technical Report 1844/90, Oxford University Department of Engineering Science, 1990.

[Duce and Fielding, 1987] D. A. Duce and E. V. C. Fielding. *Formal Specification — a Comparison of Two Techniques.* The Computer Journal, 30(4), 1987.

[Floyd, 1967] R. W. Floyd. *Assigning Meaning to Programs.* In Proceeding of the Symposium on Applied Mathematics, volume 19, American Mathematic Society, 1967.

[Foulkes and Deitch, 1982] R. Foulkes and M. Deitch. *The Design of Software for Real-time Systems and Application of MASCOT.* YARD, YM3919, 1982.

[Gerber and Lee, 1989] Richard Gerber and Insup Lee. *Communicating Shared Resources: A Model for Distributed Real-time Systems.* In Proceeding of IEEE Real-time Systems Syn., 1989.

[Göedel, 1931] K. Göedel. *Über Formal Unentscheidbare Sä Tze der Principia Mathematica and Verwandter Systeme I.* Monatsheft für Mathematik and Physik, 38:173–198, 1931.

[Goedicke, 1986] M. Goedicke. *Development of Real-time System: Specifying Functional and Parallel Behaviour.* In IFAC Real Time Programming, Proceeding of the 14th IFAC Workshop, pages 67 – 81, Lake Balaton, Hungary, 26 – 28 May 1986.

[Goguen and Tardo, 1979] J. A. Goguen and J. J. Tardo. *An Introduction of OBJ: A Language for Writing and Testing formal Algebric Program Specifications.* In Proceeding Specification of Reliable Software Conference. Cambridge, Mass., 1979.

[Gomaa, 1984] H. Gomaa. *A Software Design Method for Real-time Systems.* Communication of the ACM, 27(9):938 – 949, Sept. 1984.

[Gomaa, 1986] H. Gomaa. *Software Development of Real-time Systems.* Communication of the ACM, 29(7):657 – 669, July 1986.

[Gordon, 1987] Mike Gordon. *HOL: A Proof Generating System for Higher-order Logic.* Technical report, University of Cambridge, Computing Laboratory, 1987.

[Gries, 1976] D. Gries. *An Introduction to Current Indeas on the Derivation of Correctness Proofs and Correct Programs.* IEEE Transaction on Software Engineering, 2(4), 1976.

[Gries, 1981] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Group, 1989] HOOD Working Group. *HOOD Reference Manual,* June 1989. Draft B issue 3.0.

[Guttag *et al.*, 1985] J. V. Guttag, J. J. Horning, and J. M. Wing. *The LARCH Family of Specification Languages.* IEEE Software, 2(5):24–36, September 1985.

[Guttag, 1975] J. V. Guttag. *The Specificaiton and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, Department of Computer Sciecne, 1975.

[Hayes, 1985] Ian Hayes. *Specification Directed Module Testing.* Technical report, Programming Research Group, Oxford University, July 1985.

[Hayes, 1987] Ian Hayes. *Specification Case Study.* Prentice Hall International, 1987.

[Hoare, 1969] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming.* Communication of the ACM, 12(10):576–583, October 1969.

[Hoare, 1978] C. A. R. Hoare. *Communicating Sequential Processes.* Communication of the ACM, 21(8):666–677, 1978.

[Hoare, 1981] C. A. R. Hoare. *A Calculus of Total Correctness for Communicating Processes.* Technical Report PRG – 23, Programming Research Group, Oxford University, April 1981.

[Hoare, 1982] C. A. R. Hoare. *Specification, Program and Implementation.* Technical Report PRG – 29, Programming Research Group, Oxford University, June 1982.

[Hoare, 1985a] C. A. R. Hoare. *Algebraic Specification and Proof of Properties of Communicating Sequential Processes.* Technical Report PRG – 52, Programming Research Group, Oxford University, Nov. 1985.

[Hoare, 1985b] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Hock, 1984] Goh Tung Hock. *Some Specifications Using CSP.* Master's thesis, Programming Research Group, Oxford University, 1984.

[Houpis and Lamont, 1985] C. H. Houpis and G. B. Lamont. *Digital Control Systems, Theory, Hardware, Software.* International Student Edition, 1st edition edition, 1985.

[Howard, 1976] J. H. Howard. *"Monitor".* Communication of the ACM, 19(5), 1976.

[INMOS, 1988a] Limited INMOS. *OCCAM 2 Reference Manual.* Prentice Hall International, 1988.

[INMOS, 1988c] Limited INMOS. *Transputer Reference Manual.* ISBN 0-13-929001-X. Prentice Hall International, 1988.

[Jackson, 1983] Michael Jackson. *System Development.* Prentice Hall International, 1983.

[Jackson, 1989] David M. Jackson. *The Specification of Aircraft Engine Control Software Using Timed CSP* . Master's thesis, Programming Research Group, Oxford University, September 1989.

[Jackson, 1990] Ken Jackson. *Design Methods for Real-time Software.* In IEE Colloquium on MASCOT and Related Issues. Digest No: 1990/174, 1990.

[Jacquot, 1981] Raymond G. Jacquot. *Modern Digital Control Systems.* Marcel Dekker, INC., 1981.

[Jeffrey, 1988a] Alan Jeffrey. *A Complete Set of Laws for Finite State CSP.* Technical Report Transferral Paper, Programming Research Group, Oxford University, 1988.

[Jeffrey, 1988b] Alan Jeffrey. *A CSP Interpreter.* Technical Report Internal Document, Programming Research Group, Oxford University, 1988.

[JIMCOM, 1987] JIMCOM. *The Official Handbook of MASCOT (Version 3.1).* her Majesty's Stationary Office, June 1987.

[Jones and Goldsmith, 1988] Geraint Jones and Michael Goldsmith. *Programming in OCCAM 2.* Prentice Hall International, 1988.

[Jones, 1986] C. B. Jones. *Systemic Software Development using VDM.* Prentice Hall International, 1986.

[Jones, 1990] G. Jones. *The Queen's Award for Technological Achievement.* Occam User Group Newsletter, (13):74–76, July 1990.

[Kemmerer, 1990] Richard A. Kemmerer. *Integrating Formal Methods into the Development Process.* IEEE Computer, pages 37–50, Sept. 1990.

[Lamport, 1983] L. Lamport. *Specifying Concurrent Program Modules.* ACM Transcation on Programming Languages and Systems, 5(2):190–222, April 1983.

[Lau and Daniel, 1989] Y. K. H. Lau and R. W. Daniel. *A CSP Model for Distributed Control Software Design.* Technical Report 1789/89, Oxford University Department of Engineering Science, 1989.

[Lau, 1990] Y. K. H. Lau. *The Design of Safety Critical Software Using CSP.* In IEE Colloquium of Safety Critical Software for Vehical and Traffic Control, 13th Feb. 1990.

[le Guernic, 1986] Paul le Guernic. *Real-time Synchronous, Data Flow Programming: the Language 'SIGNAL' and its Semantics.* Rapports de Recherche, 1(533), Juin 1986.

[Leveson, 1991] Nancy L. Leveson. *Software Safety in Embedded Computer Systems.* Communication of the ACM, 34(2):34–46, February 1991.

[Liu and Shyamasundar, 1990] Leo Yuhsiang Liu and R. K. Shyamasundar. *Static Analysis of Real-time Distributed Systems.* IEEE Transactions on Software Engineering, 16(4):373 - 388, April 1990.

[Looney, 1990] M. Looney. *Implementation of a MASCOT Design in Ada.* In IEE Colloquium on MASCOT and Related Issues. Digest No: 1990/174, 1990.

[Lotos, 1984a] Lotos. *Definition of the Temporal Ordering Specification Language LOTOS.* IOS, 1984. TC97/16n1984.

[Lotos, 1984b] Lotos. *Information System Processing - Open System Interconnection - LOTOS.* International Standard Organization, 1984. Technical Report.

[Luckham and von Henke, 1985] D. C. Luckham and F. W. von Henke. *An Overview of ANNA.* IEEE Software, 2(2):9–23, March 1985.

[Luk, 1988] W. Luk. *Parameterized Design of Regular Processor Arrays.* PhD thesis, Programming Research Group, Oxford University, October 1988.

[Marshall, 1986] R. M. Marshall. *Synthesis of Hardware Systems from very High Level Behavioural Specifications.* In Ph D thesis, University of Edinburgh, Dec. 1986.

[McNulty, 1990] Mark McNulty. *Making the Right Method Mix.* Microsystem Design, 48:14 - 15, February 1990.

[Mellor and Ward, 1986] S. J. Mellor and P.T. Ward. *Structured Development for Real-Time Systems.* Prentice Hall International, 1986.

[Miller, 1990] Quentin Miller. *An Introduction to Orwell 6.19 Pre-release.* Technical report, Programming Research Group, Oxford University Computing Laboratory, 1990.

[Milner, 1980] A. J. R. G. Milner. *A Calculus of Communicating Systems*, volume 92. Lecture Notes on Computer Science, Springly-Velag, 1980.

[Morgan and Sufrin, 1984] Carroll Morgan and Bernard Sufrin. *Specification of the UNIX Filing System.* IEEE Transaction on Software Engineering, 1984.

[Mullery, 1979] G. P. Mullery. *CORE — a Method for Controlled Required Specification.* In Proceedings of the Fourth International Conference on Software Engineering, pages 126–135, Sept. 1979.

[Musser, 1979] D. R. Musser. *Abstract Data Type Specification in AFFIRM System.* In Proceedings of the Specification of Reliable Software Conference. Cambridge, Mass., 1979.

[Nielsen, et. la., 1989] M. Nielsen, et. la. *The Raise Language, Method and Tools.* Formal Aspects Computing, 1:85–114, 1989.

[Olderog and Hoare, 1984] E-R Olderog and C.A.R. Hoare. *Specification-orientated Semantics for Communicating Processes.* Technical report, Programming Research Group, Oxford University, Feb. 1984.

[Ostroff, 1989] J. S. Ostroff. *Temporal Logic for Real Time System.* Research Studies Press, 1989.

[Owicki and Gries, 1976] Susan Owicki and David Gries. *Verifying Properties of Parallel Programs: An Axiomatic Approach.* Communication of the ACM, 19(5):279 – 285, May 1976.

[Owicki, 1975] S. Owicki. *Axiomatic Proof Technique for Parallel Programs.* PhD thesis, Department of Computer Science, Cornell University, 1975.

[Parnas and Clements, 1986] David L. Parnas and Paul C. Clements. *A Rational Design Process: How and Why to Fake It.* IEEE Transaction on Software Engineering, 12(2):251 – 257, February 1986.

[Parnas, 1972a] David L. Parnas. *On the Criteria to be Used in Decomposing Systems into Modules.* Communication of the ACM, 5(12):1053 – 1058, December 1972.

[Parnas, 1972b] David L. Parnas. *A Technique for Software Module Specification.* Communication of the ACM, 2(5), 1972.

[Paul, 1981] R. Paul. *Robot Manipulators: Mathematics Programming and Control.* MIT Press, Cambridge, MA, 1981.

[Peterson, 1977] J. L. Peterson. *Petri Nets.* Computing Surveys, 9(3), September 1977.

[Pnueli, 1977] A. Pnueli. *The Temporal Logic of Programs.* In Proceeding of the 18th IEEE Annual Symposium on the Foundation of Computer Sciecne, Providence, R.I., November 1977.

[Principato, 1987] R. N. Principato. *A Formalization of State Machine Specification Technique.* Technical Report MIT/hsc/TR-2-2, MIT Laboratory for Computer Science, 1987.

[Puente *et al.*, 1986] J. A. De La Puente, A. Crespo, and T. Perez. *Formal Specification of Real-time Systems: An Industrial Application.* In IFAC Real Time Programming, Proceeding of the 14th IFAC Workshop, pages 113 – 127, Lake Balaton, Hungary, 26 – 28 May 1986.

[Reed, 1984] David Reed. *Specification of Low Level Abitration Protocal Using CSP.* Master's thesis, Programming Research Group, Oxford University, 1984.

[Roscoe and Dathi, 1986] A. W. Roscoe and Naiem Dathi. *The Persuit of Deadlock Freedom.* Technical report, Programming Research Group, University of Oxford, Nov. 1986.

[Ross and Schoman, 1977] D. T. Ross and K. E. Schoman. *Structured Analysis for Requirements Definitions.* IEEE Transactions on Software Engineering, SE-3(5):6–15, January 1977.

[Roubine and Robinson, 1977] O. Roubine and L. Robinson. *SPECIAL (SPECification Assertion Language) Reference Manual.* TR-CSG-45. SRI International, Menlo Park, California, 1977.

[Roy and de Simone, 1989] Valerie Roy and Robert de Simone. *An AUTOGRAPH Primer.* Technical Report 112, Institut National de Recherche en Informatique et en Automatique, October 1989.

[Sanders, 1988] J. W. Sanders. *An Introduction to CSP.* Technical report, Programming Research Group, Oxford University, March 1988.

[Scheid and Holtsberg, May 1989] J. Scheid and S. Holtsberg. *INA JO Specification Language Reference Manual.* Technical report, Formal Method Group, Unisys, Culver City, California, May., 1989.

[Scheid, 1979] J. Scheid. *Presentation on INA JO.* Air Force Summer Study on System Security, 1979.

[Sheehan, 1985] Alan Sheehan. *A CSP Specification of a Fault Tolerant Pipeline of Processors.* Master's thesis, Programming Research Group, Oxford University, 1985.

[Simpson and Jackson, 1979] Simpson and Jackson. *MASCOT – a Modular Approach to Software Construction.* Computer Journal, 22(4):332 –345, 1979.

[Simpson, 1990a] H. R. Simpson. *A Data Interaction Architecture (DIA) for Real-time Embedded Multi-processor System.* In RAe Conference on Computing Techniques in Guided Flight, Boscombe Down, 1990.

[Simpson, 1990b] H. R. Simpson. *Four-slot Fully Asynchronous Communication Mechanism.* In IEE Proceedings on Computers and Digital Techniques, number 137 E in 1, pages 17–30, 1990.

[Spivey, 1989] J. M. Spivey. *The Z Notation, A Reference Manual.* Programming Research Group, OU, March 1989.

[Sridhar and Hoare, 1985] K. T. Sridhar and C. A. R. Hoare. *JSD Expressed in CSP.* Technical report, Programming Research Group, Oxford University, July 1985.

[Stamper, 1990] Richard Stamper. *The Specification of AGV Control Software Using Timed CSP.* Master's thesis, Programming Research Group, Oxford University, 1990.

[Sufrin (Ed.), 1986] B. Sufrin (Ed.). *Mathematics for System Specification.* MSc. Lecture Notes, 1986.

[Technologies, 1988] VIPER Technologies. *VIPER, Assassinate It or Use It?* SafetyNet, page-1, October 1988.

[Theaker, 1991] Colin J. Theaker. *Editorial on Methodologies (Systems and Software).* The Computer Journal, 34(2), April 1991.

[Tsutsui and Fujimoto, 1987] S. Tsutsui and Y. Fujimoto. *Deadlock Prevention in Process Control Computer Systems.* The Computer Journal, 30.1:20 – 26, 1987.

[Turner, 1987] D. A. Turner. *An Introduction to Miranda, In the* Implemenation of Functinoal Programming Languages (by S. L. Peyton Jones). Prentice Hall International Series in Computer Science, 1987.

[US Dod, 1983] US Dod. *Military Standard: Ada Programming Language.* US Ministry of Defence, 1983.

[Wadler and Miller, 1988] Philip Wadler and Quentin Miller. *Introduction to Orwell 5.0.* Programming Research Group, Computing Laboratory, Oxford University, 1988.

[Ward and Campbell, 1983] P. Ward and P. Campbell. *Structured Analysis for Process Control Systems.* In Presented at ISA, Houston, October 1983.

[Willson and Krogh, 1990] Reg G. Willson and Bruce H. Krogh. *Petri Net Tools for Specification and Analysis of Descrete Controllers.* IEEE Transactions on Software Engineering, 16(1):39 – 50, January 1990.

[Wing, 1990] Jeannette M. Wing. *A Specifier's Introduction to Formal Methods.* IEEE Computer, 23(9):8 – 24, September 1990.

[Wirth, 1983] N. Wirth. *Programming in Modula 2.* Springer Publishing, 1983.

[Wise, 1989] Jon Wise. *Progress with the VIPER 2 Design.* SafetyNet, page 14, Jan/Feb/Mar 1989.

[Wojcik and Wokcik, 1989] Barbara E. Wojcik and Zbigniew M. Wokcik. *Sufficient Condition for a Communication Deadlock and Distributed Deadlock Detection.* IEEE Transaction on Software Engineering, 15(12):1587 – 1595, December 1989.

[Woodcock, 1989] Jim C. P. Woodcock. *Using Z — Specification, Refinement and Proof.* Programming Research Group, Feb. 1989.

[Yourdon and Constantine, 1978] E. Yourdon and L. L. Constantine. *Structured Design.* Yourdon Press, 1978.

[Zave, 1982] P. Zave. *An Operational Approach to Specification for Embedded Systems.* IEEE Transactions of Software Engineering, 8(SE-8):250–269, 1982.

[Ziegler and Nichols, 1942] J. G. Ziegler and N. B. Nichols. *Optimal Setting for Automatic Controllers.* Transaction ASME, 64:759 – 768, 1942.

[Ziegler and Nichols, 1964] J. G. Ziegler and N. B. Nichols. *Optimal Setting for Controllers.* ISJE, pages 731 – 734, June 1964.