ROBERT GORDON
UNIVERSITY•ABERDEEN

OpenAIR@RGU

The Open Access Institutional Repository at Robert Gordon University

http://openair.rgu.ac.uk

Citation Details

Citation for the version of the work held in 'OpenAIR@RGU':

CORR, G. A., 1996. A formalism for describing and simulating systems with interacting components. Available from *OpenAIR@RGU*. [online]. Available from: http://openair.rgu.ac.uk

Copyright

Items in 'OpenAIR@RGU', Robert Gordon University Open Access Institutional Repository, are protected by copyright and intellectual property law. If you believe that any material held in 'OpenAIR@RGU' infringes copyright, please contact openair-help@rgu.ac.uk with details. The item will be removed from the repository while the claim is investigated.

A Formalism for Describing and Simulating Systems with Interacting Components

GLENN A CORR

A thesis submitted in partial fulfilment of the requirements of The Robert Gordon University for the degree of Doctor of Philosophy.

May 1996

The Robert Gordon University, Aberdeen.

Abstract

This thesis addresses the problem of descriptive complexity presented by systems involving a high number of interacting components. It investigates the evaluation measure of performability and its application to such systems.

A new description and simulation language, ICE and it's application to performability modelling is presented. ICE (Interacting ComponEnts) is based upon an earlier description language which was first proposed for defining reliability problems. ICE is declarative in style and has a limited number of keywords. The ethos in the development of the language has been to provide an intuitive formalism with a powerful descriptive space. The full syntax of the language is presented with discussion as to its philosophy. The implementation of a discrete event simulator using an ICE interface is described, with use being made of examples to illustrate the functionality of the code and the semantics of the language.

Random numbers are used to provide the required stochastic behaviour within the simulator. The behaviour of an industry standard generator within the simulator and different methods of number allocation are shown. A new generator is proposed that is a development of a fast hardware shift register generator and is demonstrated to possess good statistical properties and operational speed.

For the purpose of providing a rigorous description of the language and clarification of its semantics, a computational model is developed using the formalism of extended coloured Petri nets. This model also gives an indication of the language's descriptive power relative to that of a recognised and well developed technique. Some recognised temporal and structural problems of system event modelling are identified and ICE solutions given.

The growing research area of ATM communication networks is introduced and a sophisticated top down model of an ATM switch presented. This model is simulated and interesting results are given. A generic ICE framework for performability modelling is developed and demonstrated. This is considered as a positive contribution to the general field of performability research.

Declaration

I hereby declare that this thesis is a record of work undertaken by myself, that it has not been the subject of any previous application for a degree and that all sources of information have been duly acknowledged.

Glenn A Corr

May 1996

Acknowledgements

I wish to acknowledge the following people to whom I am indebted and without whom this project would not have been possible.

Dr Tony Miller as my main project supervisor, who has provided many helpful ideas and with whom I have held many fruitful discussions.

Wendy, my wife, for her unfailing belief and encouragement over the past three years.

I would also like to thank The Robert Gordon University for financially supporting this project.

This thesis is dedicated to the memory of my late grandfather, Andrew Smith,
"*A true Christian gentleman*"

Contents

1 Introduction **8**

1.1 Project objectives . . . 8

1.2 Historical background to project . . . 10

1.3 Project development . . . 11

1.4 What has been achieved ? . . . 13

1.5 Guide for the reader . . . 14

1.5.1 ICE for the first time . . . 17

2 Performability Modelling **18**

2.1 Introduction . . . 18

2.2 Performability . . . 19

2.3 Formal probability theoretic definition of performability . . . 21

2.4 Modelling techniques . . . 23

2.4.1 Non-state space . . . 24

2.4.2 Markov reward models . . . 26

2.4.2.1 Unified performability, performance and dependability framework . . . 28

2.4.3 Stochastic Petri nets . . . 33

2.4.3.1 High-level Petri nets 34
2.4.3.2 Modelling and analysis with stochastic reward nets 36
2.4.4 Hierarchical and hybrid 38
2.4.5 How should realistic systems be modelled ? 39
2.5 The evolution of performability 40
2.6 Tools 42
2.7 Problems encountered in modelling systems 44
2.7.2 Largeness 44
2.7.2 Stiffness 46
2.7.3 Non exponential behaviour 47
2.8 Conclusions 47

3 The ICE Language **48**
3.1 Introduction 49
3.2 Overview 50
3.3 Language Details 51
3.3.1 Counters 52
3.3.1.1 State integer attributes 53
3.3.1.2 Resources as state descriptors 54
3.3.1.3 Component integer attributes 55
3.3.1.3.1 Counter declaration 56
3.3.1.3.2 Counter initialisation 56
3.3.1.3.3 Counter modification 57
3.3.1.3.4 Counters as transition pre-conditions 59
3.3.2 The syntax of transitions 59
3.3.3 Timing with transition pre-conditions 60
3.3.4 Hierarchical editing of models 64
3.4 Conclusions 65

4 Implementation of the ICE simulator 66

4.1 Overview 66

4.1.1 I_SIM directory structure 68

4.2 Background 69

4.2.1 Simulation styles and languages 69

4.2.1.1 Examples of specific simulation languages 70

4.2.2 Features of a discrete event simulator 71

4.2.3 The development environment 72

4.3 Compilation of the language 73

4.3.1 The Precompiler 73

4.3.1.1 Error reporting 75

4.3.2 The Parser 76

4.3.3 The Compiler 78

4.3.3.1 Simulation objects 78

4.3.3.2 General compiler operation 78

4.3.3.3 The StateSpace 80

4.3.3.4 Consequence graphs 80

4.4 Simulation Phase 82

4.4.1 The event processing cycle 84

4.4.2 The event data file 88

4.5 Post processing 89

4.5.1 VIZ the visual post processor 89

4.5.2 TPP the statistical analyzer 91

4.6 Conclusions 99

5 Random Number Generation 100

5.1 Overview 100

5.2 The random number generator 102

5.2.1 The Lehmer generator 102

5.2.2 The Park and Miller generator 103

5.2.3 Statistical properties of random number sequences 105

5.3 Random number properties 105
5.3.1 Statistical tests 106
5.3.1.1 The Chi-square test 106
5.3.2 The testing procedure 109
5.4 Testing the Park and Miller generator 110
5.5 Random number allocation 112
5.6 Testing the sub-sequence produced by the Park and Miller generator within I_SIM 114
5.6.1 Data logging modification to I_SIM 116
5.6.2 Example results of I_SIM generated sub-sequences 115
5.6.3 Detection of any predictably statistically poor sub-sequences 116
5.6.4 The effect of statistically poor subsequence on I_SIM models 119
5.7 A revised scheme for random number allocation within I_SIM 123
5.8 The development of a novel pseudo-random number generator 126
5.8.1 Linear feedback shift register sequences 126
5.8.1 Decimation of m-sequences 128
5.8.2 The generation of pseudo-random numbers from linear feedback shift register sequences 129
5.8.3 The Tausworthe generator 129
5.8.4 The Lewis Payne generator 130
5.8.5 The split-up feedback shift register generator 131
5.8.6 The Barel generator 132
5.8.7 A proposed fast Tausworthe generator 133
5.8.7.1 Software implementation 134
5.8.7.2 Testing the fast Tausworthe generator 134
5.8.7.3 Timing of the generator 135
5.9 The generation of seeds to produce parallelized random number sources . . . 136
5.10 Conclusions 138

6 **Computational models for ICE** 139
6.1 Introduction 139
6.2 COMPONENT 140
6.3 CONSTANTS 141
6.4 STATE_SETs and COUNTERS 141
6.5 SYSTEM 143
6.6 RESOURCES 146
6.7 BEHAVIOUR 147
6.7.1 Timed Probabilistic Transitions 147
6.7.2 Transition Firing Policies 148
6.7.3 ON_EVENT 150
6.7.4 IF 152
6.8 WAIT_FOR 153
6.9 ATTRIBUTES 154
6.10 Behavioural models 157
6.10.1 Dependency 157
6.10.2 Concurrency 158
6.10.3 Synchronisation 159
6.10.4 Conflict 159
6.11 Conclusions 160

7 **An ICE performance model of an ATM switch** 162
7.1 Introduction 162
7.2 Asynchronous Transfer Mode 163
7.3 ATM switches 163
7.3.1 The architecture of Banyan networks 164
7.4 The ICE model 167
7.4.1 Overview 167
7.4.2 The input traffic 169
7.4.3 The input controllers 170
7.4.4 Operation of the cross-bar switches 171

7.4.5 The switching elements 173
7.4.6 The output controllers 175
7.5 Model validation 176
7.6 Results 176
7.6.1 Performance parameters 177
7.6.2 Simulation parameters 177
7.6.3 Simulation results and observations 178
7.7 Conclusions 180

8 Performability modelling with ICE 182
8.1 Introduction 182
8.2 Distribution of accumulated reward 183
8.3 The ICE reward model 184
8.4 Multiprocessor example 185
8.4.1 Performability measures 189
8.5 Conclusions 191

9 Discussion and conclusions 193
9.1 Overview 193
9.2 The ICE language 194
9.3 Implementation of the ICE simulator 198
9.4 Random number generation 200
9.5 Computational models for ICE 201
9.5.1 A macroscopic view 202
9.6 Performability modelling 203
9.6.1 Problems encountered in modelling systems 204
9.7 Suggested areas of further work 205
9.8 Conclusions 207

References 208

A Performability Specification Language 223

A.1 Introduction 224

A.2 Overview 224

A.3 Language Syntax 225

A.3.1 COMPONENT 225

A.3.2 STATE_SET 226

A.3.3 SYSTEM 227

A.3.4 RESOURCE 229

A.3.5 BEHAVIOUR 229

A.3.6 WAIT_FOR 232

B Data Structures, Objects and Files created during Simulation 234

B.1 An ICE program - 'logger.ice' 235

B.2 Parsing of 'logger.ice' 237

B.2.1 Simulation Data Structures 241

B.3 Compilation 253

B.3.1 Consequence Graphs 257

B.4 Simulation - The Event Data File 260

B.5 Statistical Analysis Data 260

B.5.1 Objects created from Event Data File information 260

B.5.2 Objects created from the Analysis File 263

C ICE listing of communications network sources model 266

D ICE listing of Delta-2 Banyan switch architecture model 270

E Reprint of published paper 278

Chapter 1

Introduction

1.1 Project objectives

The aim of this project was to research a new approach to describing and simulating complex systems involving multiple interacting components.

Complexity is a problem not only for computational analysis but for the system description itself. There is a need for a problem specification formalism which supports descriptions proportional to the size of the physical system rather than the overall system state space. This is an approach which addresses the problem of complexity as presented to the human modeller rather than the computing hardware. Whereas computing architecture continues to become increasingly more powerful, we can safely predict that human capability to intellectually grasp the operation of highly interconnected systems will remain relatively limited.

To facilitate the simplified modelling of systems the first objective was to develop a new approach of describing each component in a system separately. This was to include means of specifying inter-component relationships so that all component interaction could be left to simulation. This formalism has been called the ICE (Interacting ComponEnts) language.

The foundation for ICE is an earlier reliability description language (RDL) [6] which has been used to successfully model a number of reliability problems. While it is recognised that a state space approach such as that adopted by ICE is not as universally applicable to performance measures as it is to reliability measures, an objective of the language was to be able to model the performance of complex systems in an efficient manner. Further to this, to provide a rigorous testing ground for ICE's descriptive power, it was decided to focus on performability measures in modelling and to develop a generic performability modelling framework using the language. Performability is a composite measure of performance and dependability and as such presents a more significant challenge to the modeller and the modelling technique than either pure performance or dependability modelling. Here we take dependability to be a collective term for both reliability and availability measures.

When considering the field of performability it was important to recognise the existing formalisms and their relative merits. It was thus necessary to consider the popular approach of stochastic Petri nets in some detail and derive detailed comparisons between these and the ICE technique. There was also a recognised need for a rigorous mathematical definition of the languages semantics. Stochastic Petri nets were identified as a medium for achieving this.

ICE describes systems in a generic manner in terms or their functionality and performance, dependability and performability measures. A further objective was to develop a bespoke software package that could perform discrete event simulations on ICE descriptions.

A summary of what has been achieved and to what extent the objectives have been met is given in section 1.4.

The project specification was roughly followed though due to the interesting nature of the work a number of additional avenues of investigation were considered as they presented themselves. Particularly the evolution of the simulator and the simulation algorithms posed some interesting questions about the language's semantics and they accordingly underwent an iterative process of modification. Related issues arose as the performability framework was investigated and attractive and fruitful areas of further work were identified. An outline of the general development of the project is given in section 1.3.

1.2 Historical background to project

There has been research conducted into reliability modelling at the Robert Gordon University since the late seventies. Work within the School of Electronic and Electrical Engineering in this area started in the field of hardware reliability simulators [7]. These simulators were dedicated microprocessor implementations that facilitated the accelerated simulation of multi-component concurrent systems. It was possible to enter data as to the behaviour and reliability characteristics of individual components and their inter-dependant relationships. Simulations could then be conducted over the modelled systems life cycle and measures determined for the reliability of the complete system.

Prior to simulating a system on such hardware it was necessary to describe the functionality of the model in an appropriate manner. There are a number of ways to describe reliability problems such as task graphs, fault trees, mathematical statements. There was however no standard technique. System designers and evaluators may describe the same system in different ways and this can have a corresponding impact on the types of analysis possible and the results obtainable.

It was this situation that was the catalyst for the idea of a reliability description language (RDL). The language was to act both as an input for the hardware simulators and as a generic method for describing systems in terms of reliability. It was also to provide a method which would encourage greater coherence between the approaches of designers and evaluators.

The initial outline of RDL was first presented in 1986 [8]. It was identified by the European Commission in Brussels as a possible standard for formalised reliability descriptions. This interest lead to refinements in the language and this revised version was presented to the EEC in 1987 [9].

Up to this point the language was purely a descriptive tool and no simulator existed. In 1987 Scrase, a research assistant in the school, began work on further refining the language and writing a software simulator [10]. Once a simulator had been developed Scrase went on to apply the language to a number of standard reliability problems and investigated its use in the modelling of communications networks. The final version of RDL was documented in 1991 [6].

In 1988 Walker, a teaching company associate, started work on the simulator. His remit was to revise it to be of such a standard so as to be of use as an industry tool. At about the same time, Smith, another research assistant, applied RDL to flexible manufacturing systems (FMS) [11]. He developed a super-set of the language with added constructs unique to FMS modelling.

In 1993 it had been recognised that RDL was limited in that the ethos behind its development was focused on the reliability measures of a system. The first objective of this project was to develop a declarative language which would be a super-set of RDL facilitating generic modelling of universal systems.

1.3 Project development

This section describes the general progress of the project and the extent to which the initial specification in section 1.1 was followed.

Initial work focused on the development of the ICE language, with the main focus being on increasing its descriptive space and reviewing all means to facilitate component interaction. A general philosophy was adopted at this stage to make the language as intuitive as possible. The observation was made that often an expert is required to model a system due to the complexity of the modelling technique. We wished to remove this complexity and the involved level of abstraction to produce a tool which could be used directly by design and evaluation engineers.

As work progressed on the language development began on the simulator. It was decided to use the Tecsim simulator that had been written for RDL as a basis and build upon it. A suitable compiler writer was identified and the first stage of the software was implemented. Once the compiler had been completed, attention focused on the discrete event simulator and the simulation algorithms. From development of the algorithms it became apparent that although the language syntax was relatively simple, some of the semantics were necessarily complex and would be implementation dependant. The questions that arose during the implementation of the simulator lead to further reviews of the language, especially in relation to event priority and timing. It could be argued that these matters should have been finalised before work on the simulator began but it was the

experience of event scheduling within the simulator that directed our thinking towards these modifications.

Any model that is simulated must then be analysed to extract meaningful results. To ease this process it was decided to automate an analysis process and incorporate it into the simulation software. To this end two post processors for the simulator were developed. One post processor was implemented to provide textual analysis of a complete simulation event list or a subset there of. The other was designed to provide statistical analysis on the event list. For ease of use it was decided to make the user interface to the post processor resemble a spreadsheet.

In parallel to the development of the language and simulator, extensive literary research was conducted into the area of performability. The evolution of this field was investigated as was the variety of modelling techniques that have been presented. There was much evidence in the more recent literature of the suitability of reward type techniques to performability modelling. Further examination of these techniques highlighted some underlying principles that could be adopted within ICE and therefore suggested this to be a promising area of research and application for the language. This research also confirmed the suspected complexity of performability modelling showing it to be both a rigorous testing area and an area in which simplified approaches are required.

As the simulator was nearing completion a tangential but fruitful area of investigation presented itself. Initially an industry standard random number generator had been chosen for the simulator. Further consideration revealed that this generator was designed to provide a constant stream of numbers for use by a single consumer and that it had only been proven to be statistically good over runs of close to a full period. Our use of the generator was to provide numbers for multiple consumers and the run of numbers allocated to a consumer may be of any duration. Statistical tests were run on numbers generated during actual simulations and this revealed that the technique of randomly allocating numbers to different components from a single source was not always satisfactory. Prompted by these results we adopted a new approach to number allocation. It was also noted that while the generators used were statistically good they were computationally intensive. For this reason it was decided to consider the custom generator that had been developed for the preceding hardware reliability simulators. A software implementation based on this generator was developed which proved to be over four times faster in

operation and thus it was chosen as the preferred generator for the ICE simulator.

At this point, the uncovered complexity of the language's semantics and the requirement to compare ICE to existing performability formalisms lead to an extensive investigation of stochastic Petri nets. It was soon discovered that the descriptive power of basic stochastic Petri nets was too limited to properly describe all of the ICE constructs. Further research into recent high-level developments of Petri nets however uncovered additions that countered this problem and a rigorous description of ICE was derived.

Following the initial objectives, the next stage of the project was to apply ICE to complex performance problems. The application area of communications systems was chosen. Research revealed that the area of significant current interest was ATM networks. Other research in the school at the time was pursuing the implications of transferring video data across ATM networks. It was felt that we could add valuable input to this work and thus focused on ATM systems. Extensive investigation into ATM networks by literary searches and conference attendance narrowed our focus further to ATM switches. Different types of switches and their constituent parts were considered and sophisticated performance models were developed and simulated with ICE, showing its applicability to performance problems.

Once the performance models had been built and tested, attention was turned to the rigorous testing area of performability, and results from this were very promising. It was decided to research a generic performability modelling framework using ICE that could be applied to a variety of problems. This work has illuminated a number of fruitful areas of further work.

1.4 What has been achieved ?

In this section we summarise the main achievements of the project.

A formal descriptive language, ICE, has been developed and revised. ICE has been proven to be generic, transparent and inherently lucid. Although it is simple to learn and apply it has surprising power whilst maintaining a relatively limited state space.

An object oriented software modelling tool, I_SIM, has been built around the ICE language. I_SIM facilitates the compilation, discrete event simulation and analysis of ICE models. The simulator is appropriately powerful and flexible. All applications described in this thesis had simulation run times of less than 5 minutes. Analysis is achieved both by observation of a selected event list and by statistical analysis of this event list.

In relation to I_SIM, investigation has been conducted into the generation and allocation of pseudo-random numbers. A novel software implementation of a linear feedback shift register pseudo-random number generator has been proposed. The generator has been demonstrated to operate at a rate of over four times faster than an industry standard linear congruential generator whilst possessing guaranteed statistical properties when applied to concurrent systems.

A rigorous definition of ICE has been demonstrated by the manipulation of extended coloured generalised stochastic Petri nets. This has also given an indication of the comparable power of ICE to these nets which are a standard formalism applied to performability modelling.

ICE has been used to produce performance measures of complex concurrent systems. Whilst it has been recognised that a state space approach is not always the most suitable for performance measures, ICE has proven that in some instances it can give considerable insight into a systems behaviour by virtue of its low level of abstraction and inherent transparency.

A generic framework for the performability modelling of systems incorporating interacting components has been proposed. This framework uses the notion of stochastic reward models. It is of comparable power to a stochastic reward net approach but notably simpler to apply.

1.5 Guide for the reader

In this section we give a guide as to how to read the thesis. The document flows in a logical progression, though depending upon specific interest, the first time reader may wish to miss out some chapters.

Figure 1.1 is a schematic representation of the thesis. This is intended as a broad guide only and should not be perceived as being restrictive. Related themes are shown together and the links between the appendices and the chapters they relate to are detailed.

Chapter 2 gives a detailed look at performability modelling, introducing the different approaches and implications. For readers already familiar with this field or for those who are interested in ICE for another application this chapter may be missed on a first reading. A full listing of the constructs and syntax of ICE is presented in appendix A and chapter 3 discusses some of the significant aspects in greater detail. These sections should be read together as appropriate.

Chapter 4 tells of the development if I_SIM. This is rather a complex software tool and to aid in understanding a complete step by step example to its operation is listed in appendix B. Chapter 5 describes the work done in developing the random number generator. These chapters may be missed by those whose sole interest is in applying ICE.

Chapter 6 describes the computational models for ICE using Petri nets. This is a valuable guide for a full understanding of the semantics.

Chapter 7 presents a detailed ICE performance model of an ATM switch. A full listing of the model is given at appendix C and a published paper on this area is reprinted in appendix D. These chapters provide insight into the use of ICE for performance modelling but can be bypassed for those who wish to focus on performability modelling.

Chapter 8 introduces the ICE framework for performability modelling. A reasonable understanding of the work presented in chapter 3 and the techniques of performability modelling described in chapter 2 will be required before reading this chapter.

Chapter 9 brings our conclusions from the work together and discusses areas of possible further work.

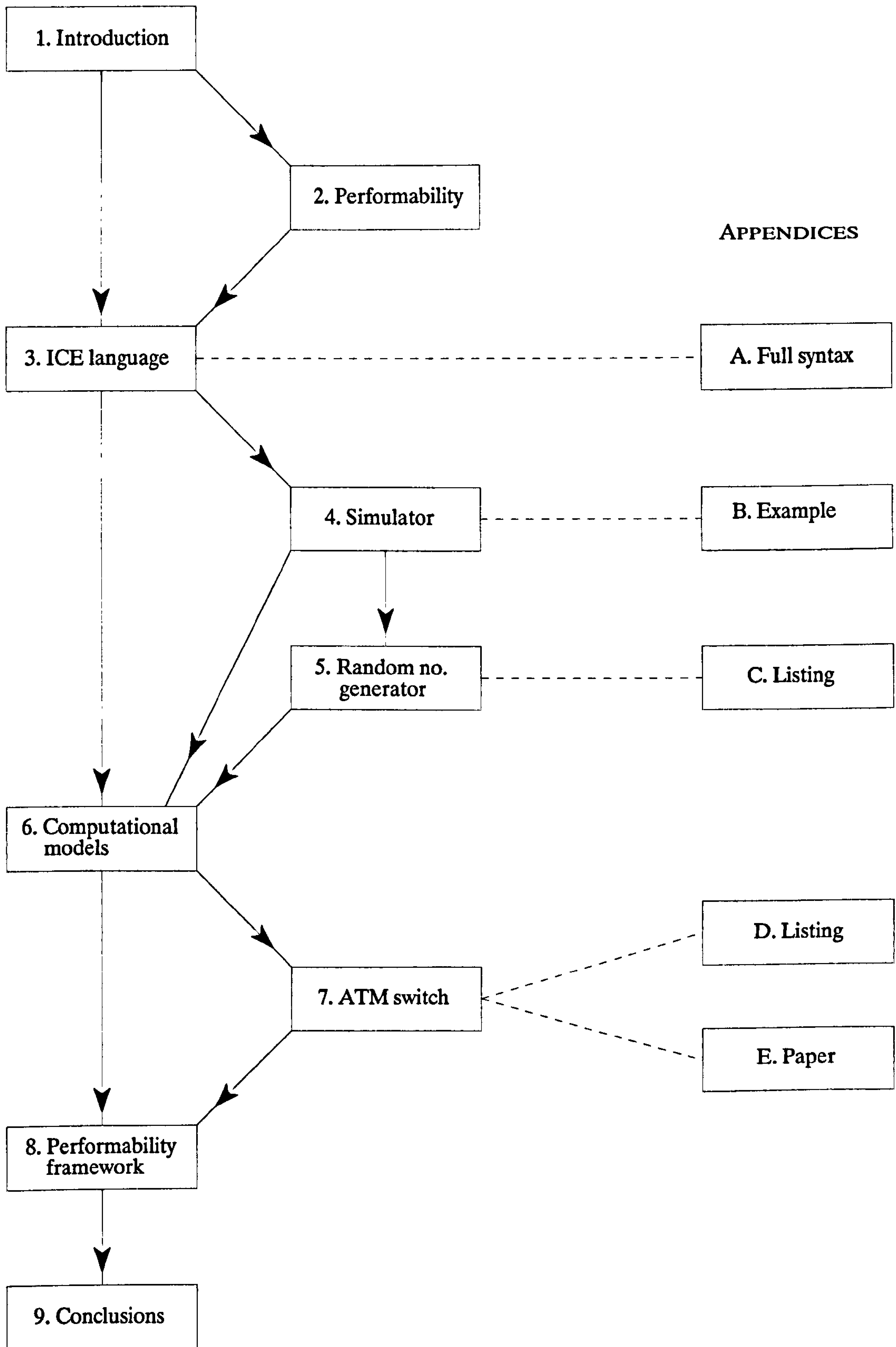


Figure 1.1 Schematic representation of the thesis

1.5.1 ICE for the first time

ICE has a relatively limited number of constructs and its structure is logical and should be quite straightforward to gain familiarity with the language. The apparently simple syntax does however hide some complex semantics which may at first be hard to grasp.

The starting point for learning the language should be sections 3.1 and 3.2 which provide a broad overview of the language's approach. The reader is then referred to appendix A which gives a thorough presentation and explanation of the entire syntax.

When this is understood, we would suggest the remainder of chapter 3 be read. This will give further insight into the operation and implications of some of the languages more sophisticated constructs. At this point a reasonable grasp of ICE should have been obtained. To enforce this, consideration of the performability example in section 8.3 is advised.

For a thorough insight into the semantics and an example of ICE's use in modelling complex systems comprising inter-acting components, study of the ATM switch model in section 7.4 is recommended. This example takes the reader through the modelling process from conceptual system diagrams, via state transition diagrams for the constituent components to the ICE code for each component. A full listing of the model is given in appendix D.

Chapter 2

Performability Modelling

2.1 Introduction

In this chapter we present some general background to the project, with reference to established methodologies and published literature.

The term *performability* is that given to the composite measure of performance and dependability. We take dependability to be a global term encompassing reliability and availability. The reasons for considering such a composite measure will be discussed later in the chapter.

Historically, dependability and performance modelling developed as separate fields until the increasing complexity of interacting systems dictated the need for combined measures. For this reason initially we shall consider the requirement for and meaning of performability, performance and dependability measures. A formal definition will be given of performability and from there some of the known approaches to performability modelling will be presented, along with a unified framework of measures. We shall then consider the evolution of research regarding this topic and mention some of the

tools that have been developed to define and analyse systems. Finally we examine some of the recognised problems that the performability modeller must be aware of.

2.2 Performability

The continuing growth of the sophistication of computer and communications systems has dictated the requirement for increased innovation in model construction/solution, tool development and the definition of analysis measures. This is especially so for networked systems comprised of a number of interacting components [12].

When evaluating a system the requirement is to relate what it is and does to what it is required to be and do. Two measures may be of interest for this evaluation, namely *performance* and *dependability*. Performance in this context, generally refers to how effectively or efficiently a system delivers a specified Quality of Service (QoS) provided it is delivered correctly. Dependability is taken as the reliance to be justifiably placed on the service it delivers. Dependability encompasses both *reliability* and *availability*. Reliability is the continuous delivery of proper service and availability the alternation between deliveries of proper and improper service.

Both performance and dependability have evolved as individual fields. If separate evaluations of performance and dependability are done to determine a deliverable QoS then appropriate constraints must be stated on how properties affecting performance interact with those effecting dependability. Results of each type of evaluation may be taken together to provide a complete assessment of overall QoS [13]. Generally however they are not easy to combine particularly if performance in the presence of faults is degradable, i.e. faults may reduce performance and thus QoS even though the system is still in the proper state. Analysis of degradable systems from a pure performance stance is therefore often optimistic as it ignores fault repair and gracefully degrading operation. Conversely pure reliability analysis tends to be overly pessimistic as no account of performance is taken [14]. Rather, a degradable system's probity should be viewed as a multi-valued variable reflecting the degree to which the system is operational. The need to accommodate this property using model based evaluation

methods was the raison d'etre for performability.

The general framework for model-based performability evaluation was first published in 1978 by Meyer [15]. He produced a more refined description in 1980 [16]. The framework was a development from his earlier notion on "computer based reliability" [17] and recognises the work done at that time by Borgerson and Freitas [18] who noted that degradable systems require special attention compared to non-degradable systems when developing measures and models to evaluate them.

The initial application area was the evaluation of ultra-reliable aircraft control computers being developed by the U.S. Space Agency (NASA). One aim of these systems was the ability to shed workload beginning with the least critical tasks if a loss of processing power due to faults occurred. This enabled the systems to operate at various degrees of service over a specified period of use.

In Meyer's earlier work [19] he viewed the unification of performance and dependability (or reliability as it was then referred to) as a measure of system effectiveness where its formulation depended on an intermediate association of 'worth' (defined as reward, benefit, utility) with each possible level of accomplishment.

As the concepts developed the desired amount of generality evolved and led to the conclusion that the performability-dependability aspects of effectiveness should be separate from worths that may be associated with their outcomes. The resulting refined concept could still however be employed at higher level worth-oriented evaluations of systems effectiveness. The term 'performability was adopted in 1980 [16] and can be simply stated to be a measure of a systems *ability to perform* in a designated environment.

In early work performability was taken to mean only distribution functions of accumulated reward. Since the calculation of distribution functions can be very complex it is often considered that the expected values of these distributions can be taken as performability measures. Using steady state values has also been debated. It is now common to consider any measure that takes both performance and dependability into

account as a performability measure and we shall take this same definition here.

2.3 Formal probability-theoretic definition of performability models

We use the term "*model*" to refer to a representation of a *total system.* Let S denote the *total system* in question, where S consists of an *object system* C (the communication or computing system being evaluated) and it's *environment* E (the workload and external faults etc.) thus

$$S = (C, E)$$

should be regarded as a probabilistic description of the total system that is sufficiently detailed to support a particular type of evaluation. Given an accepted set of interacting components, the distinction between C and E is dependant upon the subset of components, C, which is being investigated and its ability to perform. This comprises of C. The remaining components, lying outside the system boundary but whose interaction with C may affect the performability, form E.

The *performance* of S over a specified *utilization period* T is a random variable Y taking values in a set A; elements of A are the *accomplishment levels* (or performance outcomes) that might possible be obtained by S. T is the time period of use over which system performance is summarised (by the value of Y). T is an interval that may be discrete or continuous and either bounded or for systems that demonstrate meaningful steady state behaviour, unbounded.

Note that the interpretation of "performance" here is more general than in the context of traditional computing. It connotes any designated aspect of the total systems behaviour relative to which the object systems ability to perform is being measured. This permits choices of Y to be almost limitless, ranging from a binary variable that distinguishes whether or not a specified service is performed correctly throughout T up to a high level representation of service quality with a continuum of service levels. Meyer [14] states the generic meaning of performance within this context to be "what

a system accomplishes during its use". A systems ability to so perform, expressed by probabilities, is its performability.

Performability may be defined as follows. For a system S with performance Y taking values in accomplishment set A, the *performability* of S is the probability measure *Perf* (often denoted p_s) induced by Y where, for any measurable set B of accomplishment levels ($B \subseteq A$),

$$Perf(B) = P[Y \in B] = \text{the probability that } S \text{ performs at a level in } B.$$

This measure applies to any set B for which the event $Y = B$ has a probability although in practice these sets are typically intervals of accomplishment expressing performance requirements. Hence, for example, if $A = (-\infty, \infty)$ and $B = [a, \infty)$ then $Perf(B)$ is the probability that S performs at or above the level a.

These probabilities are determined via an underlying stochastic process X referred to as the *base model* of S. X is a time indexed set of random variables

$$X = \{X_t \mid t \in I\},$$

where the time set I must include the utilization period T associated with the performance variable Y. Hence X may be continuous-time or discrete-time, depending on the nature of the system. For any $t \in I$, the value of the random variable X_t is the state of the system S at time t given a state space Q. The state space Q may be considered as the product space $Q_C \times Q_E$ where Q_C and Q_E are the state spaces of the object system and environment respectively. This process, when restricted to the period T associated with Y conveys the dynamics of an object system's structure, internal state and environment during that period.

The base model must also, by definition, support a solution of performability in that, for any accomplishment set B of interest, $Perf(B)$ is indeed determinable, at least theoretically, from the probabilistic nature of X restricted to T. This requirement is ensured via a *capability function* which maps trajectories of X into corresponding values

of *Y*. A base model *X* together with a performance variable *Y* is a *performability model* of *S*.

When a performability model is solved analytically, the base model must be characterised explicitly in some suitable form, e.g. a state-transition-rate matrix in the case of a continuous time, time homogenous, finite state Markov process. If performability is estimated via simulation techniques then *X* refers to the behaviour of some simulation model *S*. Model-based performability evaluation thus involves two steps. Firstly performability model *construction* which consists of specifying the performance variable *Y*, relative to which *Perf* is defined and determination of a base model *X* that supports its solution.

2.4 Modelling techniques

We distinguish three methods for system performance, dependability and performability evaluation : measurement based, model based and hybrid methods [20].

Measurement based evaluation (also called empirical evaluation) requires access to a measurable system. This is often not possible, especially in development applications. Also obtaining measurements is often a complicated and expensive process and may be impractical for dependability events which often require extremely long measurement sessions.

Model based evaluation is an alternative. Models can be as simple or complex as the situation dictates. Once a model has been constructed it must be solved. This can be done using simulation or analytical techniques. Analytical techniques can be full symbolic, semi-symbolic or numerical. A distinction that can be made is whether the model solution requires the entire state space to be generated or not. The most common example of the former is the solution of a large but finite Markov model [21]. An example of the latter is the use of fault-trees for reliability analysis [22].

Many useful practical evaluations use a suitable combination of different modelling

approaches with measurements, e.g. fault injection simulation with the faults being measured values from an operational system. An example of this is given in Hsueh et al [23]. They use real error and resource usage data from a multiprocessor system in a semi-Markov process model with reward functions based on the service and error rates of each state. The model is solved to estimate system performability and depict the cost of different types of errors.

The state behaviour of a model is identified with X. The performability model *solution* is the procedure that yields values $Perf(B)$ for accomplishment levels in set B. B is chosen to contain levels that are of interest to the modeller. In general, knowledge of the probability distribution function (PDF) of Y suffices to determine such values and hence the performability model can be regarded as fully solved once the PDF of Y has been determined. Typically solutions of the PDF must be determined via numerical or simulation techniques though closed-form solutions are sometimes possible. In some applications it may be the case where only certain of the application sets have useful interpretations and hence a full solution will be unnecessary.

Numerous techniques exist for modelling systems. The factors which determine the technique chosen are the balance between ease of modelling and accuracy, the measures of system behaviour that are to be obtained and the properties of the system which are to be modelled. Below we discuss 4 main approaches. In section 2.4.1 *non-state space models* where explicit knowledge and numeration of the state space of the model is not required for evaluation are considered. In section 2.4.2 we discuss Markov chain type models and in section 2.4.3 Stochastic Petri Net (SPN) based models. In section 2.4.4 we introduce hierarchial and approximate modelling approaches.

2.4.1 Non-state space

Non-state space methods are attractive in that state space tends to increase exponentially with problem size. Four of the better know non-state space methods are *fault trees* (FTs), *task graphs* (TGs), *product form queuing networks* (PFQNs) and *matrix geometric methods* (MGMs).

Fault tree methods and reliability block diagrams give very accurate representations of systems and efficient solution algorithms exist. These techniques are mostly used for dependability and safety analysis. With FTs a tree structure with logic gates is used to express how systems fail. The leaves of a tree express component failures. System failure is expresses as a logical function of the failure of components and subsystems and this provides a combinatorial means of solving measures of interest. Subsystems and components must have stochastically independent failure behaviour and hence the limitation of these techniques is the complex compensation techniques required to model interaction between the constituent components of a system [24].

TGs [25] are a technique often applied for performance modelling of concurrent systems. These however make the assumption that resources within a system are infinite and for many applications this is not acceptable.

A method which does allow for finite resource contention is PFQNs [26]. In a PFQN the number of resources (queues and servers) as well as the way in which customers use these resources are specified. The active elements are the queues which may serve the customers in any of the recognised scheduling disciplines. Routing chains govern how customers move through a network and these customers may be grouped into classes. At each queue customers belonging to specific classes request a general differential service time distribution. After service the customers proceeds to the next queue along its routing chain. A vector giving the number of customers of each class at each queue specifies the state of the PFQN. The arrival of a new job or the completed service of a customer at a queue causes a change of state. Techniques that exploit the model structure and are much less memory intensive than solving the model at state space level can be employed to analyse the model. PFQNs are often applied in situations where there are finite resources but may not be extended in cases where concurrency or synchronisation is required as in these instances the product form is violated.

MGMs exploit the repetitive underlying Markov chain of a queuing model. The generating matrix for many queuing models often has a number of so called boundary columns and from some point all other columns are the same save that they shift downwards. Due to this special structure solutions can be obtained by solving a number

of linear equations [20]. By contrast the original Markov model would have involved the solution of an infinite system of linear equations. MGMs cannot be extended to cases where concurrency and synchronisation is required.

2.4.2 Markov reward models

A more powerful technique, free from any such limitations are Markov Models [27]. They are extensively used with equal effectiveness in modelling concurrency, synchronisation, resource contention, system dependencies, fault tolerance, system re-configuration etc. A Markov *chain* has a discrete state space and is a stochastic process whose past has no influence on its future if the present state is specified. We use the term stochastic *process* to indicate a continuous time parameter as opposed to *sequence* which would indicate a discrete time parameter. Formally $X(t)$ is a Markov chain if

$$Pr\ \{\ X(t_n) = j \mid X(t_{n-1}) = i_{n-1}\ \}$$

A Markov chain is memoryless. This implies that the amount of time spent in its current state, known as the sojourn time, is irrelevant to its probability of being in any other state. A semi-Markov chain is similar in most respects to a Markov chain except that its sojourn time does have an effect on the state transition probabilities and thus it does not posses the memoryless property.

An extension to this is the Markov Reward Model (MRM) which can be used to model degradable performance. In a MRM, a real variable termed the reward rate is associated with each state of the underlying state-space. The reward rate is an indication of the useful work of interest done by the system while it exists in any given state. It is also possible to associate reward impulses with state transitions. Hence it facilitates the calculation of performability measures such as total work done within a finite time interval, this being equivalent to the accumulated reward within that interval.

Formally, an MRM consists of an underlying continuous time Markov chain (CTMC), $X = \{\ X(t),\ t \geq 0\}$ with a finite state space S, and a *reward function* r where $r : S \rightarrow$

$\Re$. X is completely described by its generator matrix Q and the initial probability vector $\pi(0)$. For each state $i \in S$, $r(i)$, usually written as r_i, represents the reward obtained per unit time spent by X in that state, hence we can state a performance variable $Y(t)$ as the total reward accumulated over time t as

$$Y(t) = \int_0^t r(X_s)\,ds$$

A solution of performability F is the PDF of $Y(t)$ ie for any accomplishment level y the probability

$$F_{Y(t)}(y) = P[Y(t) \leq y]$$

This type of rate based model is given in [28] for a special class of degradable microprocessor model. Here the base model is a Markov process and a closed form solution of performability is presented.

CTMCs have equivalent modelling power. There are well known methods and software tools available for solving CTMCs but the solution methods are far more cumbersome. The primary disadvantage of the MRM technique is the large size of their state-space even for simple small systems. This frequently causes verbose specifications and ineffective solutions. Possible means of avoiding largeness will be discussed later.

A comprehensive treatise on Markov reward models for performability analysis is given in [29]. Krieger [28], describes a range of numerical solution methods which may be applied to these models.

2.4.2.1 Unified performability, performance and dependability framework

In this section we present a unified framework for developing performability, performance and dependability models in terms of MRMs. Although MRMs are the underlying medium, the measures derived may be applied to other techniques e.g. stochastic reward nets and discrete event simulations.

Definitions

If the analysis types applicable to stochastic processes are defined, it is then possible to derive performability measures that can be used to determine the behavioural properties of a system [3]. Where the performability measures incorporate reward rates, these reward rates can be reduced to give binary rewards between states and thus give dependability measures.

There are 4 categories of analysis applicable to stochastic processes :

1. Transient Analysis
2. Steady State Analysis
3. Cumulative Transient Analysis
4. Sensitivity Analysis

Let $\{\Theta(t), t \geq 0\}$ be a continuous-time, finite state, homogenous Markov chain (CTMC) with state space denoted by S and a constant reward r_i assigned to each state i. With the reward rate specifications the CTMC can be specified as an MRM. If the MRM spends τ_i time units in state i then the accumulated reward is $r_i\tau_i$. It is also possible to associate rewards with the transitions of the CTMC. The reader is referred to [25] for a basic coverage of MRMs.

Let P(t) be the state probability vector of the model , where Pi(t) is the instantaneous probability that the MRM is in state i at time t. Let $P(0)$ be the initial probability vector and Q be the generator matrix. Transient analysis of the models behaviour is dependant

upon the existence of $P(0)$ and is given by the Kolmogorov differential equation:

$$\frac{dP(t)}{dt} = P(t)Q \qquad (1)$$

Steady state analysis is conducted by determining π, the steady state probability vector:

$$\pi = \lim_{t \to \infty} P(t)$$

given the limit exists. Note that this is independent of $P(0)$ and is obtained by setting the l.h.s. of equation (1) to zero :

$$\pi Q = 0, \qquad \sum_{i \in S} \pi_i = 1 \qquad (2)$$

Here π_i is the steady state probability of the MRM being in state i.

Cumulative transient analysis involves computing the total time spent in state i during the time interval [0, t). Let $L_i(t)$ denote this value. In vector terms, integrating equation (1):

$$L(t) = \int_0^t P(x)dx$$

This value can be determined by solving:

$$\frac{dL(t)}{dt} = L(t)Q + P(0) \qquad (3)$$

When the MRM has absorbing states, the state space S can be partitioned into the two subsets : S_A (absorbing states) and S_T (transient states). The sub-matrix Q_T of Q can be defined, corresponding to the non-absorbent states. The mean time spent by the MRM in state i is given by

$$\tau_i = \int_0^{\infty} P_i(x)dx$$

which can be computed by integrating equation (1) from 0 to ∞ :

$$\tau Q_T + P_T(0) = 0 \qquad (4)$$

For such a Markov chain the mean time to absorption can be calculated by :

$$MTTA = \sum_{i \epsilon S_T} \tau_i \qquad (5)$$

If it is assumed that all the entries in Q are functions of some parameter vector θ, sensitivity analysis involves computing the variation in the state probability vector with respect to the model parameters i.e.

$$\frac{dP(t)}{d\theta} \qquad (6)$$

Using the above formulae it is possible to define performability, dependability and performance measures for the MRM.

Performability measures

Let the instantaneous reward rate of a MRM be $\Upsilon(t) = r_{\Theta(x)}$. The reward accumulated in the time interval [0, t) is given by:

$$\phi(t) = \int_0^t \Upsilon(x)dx = \int_0^t r_{\theta(x)}dx \qquad (7)$$

Figure 2.1 shows an MRM with possible rewards and corresponding values of X(t) (state

of Markov chain at time t), $\Upsilon(t)$ (the reward rate at time t) and $\varphi(t)$ (the accumulated reward at time t).

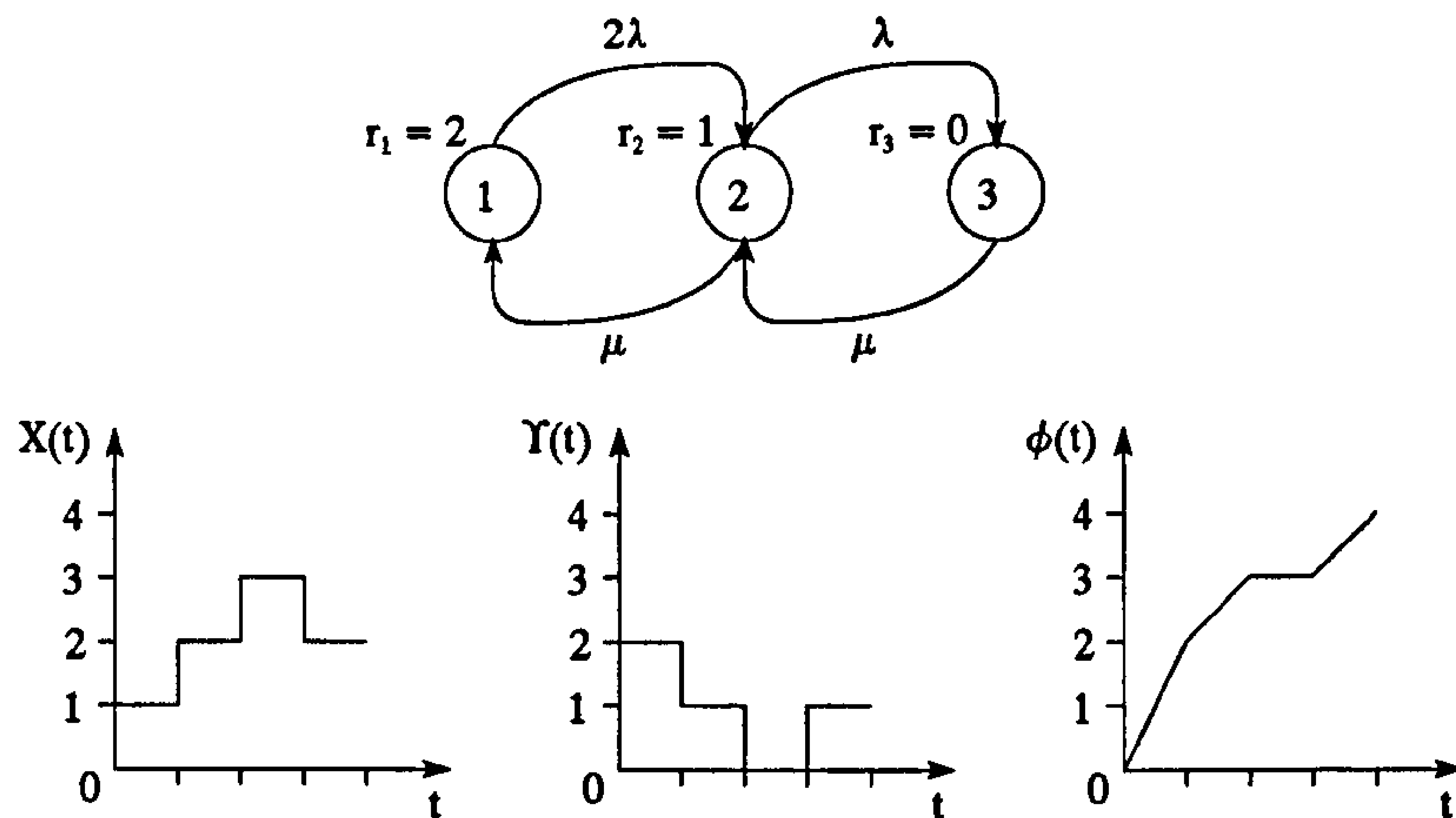


Figure 2.1 Example 3 state MRM with graphs of $X(t), \Upsilon(t)$ and $\varphi(t)$.

The instantaneous availability of the system is given by the expected instantaneous reward rate at time t, computed by:

$$E\,[\Upsilon(t)] = \sum_{i \in S} r_i P_i(x) \qquad (8)$$

In [7], Beaudry calls this value *computation availability*. The expected availability when the system has reached steady state is given by $\lim_{t \to \infty} E\,[\Upsilon(t)]$:

$$E\,[\Upsilon_{ss}] = \sum_{i \in S} r_i \pi_i \qquad (9)$$

The expected accumulated reward in the interval [0, t) denotes the total time the system is available in this interval and is given by:

$$E\,[\phi(t)] = \sum_{i \in S} r_i L_i(t) \qquad (10)$$

If the MRM has absorbing states it is often desirable to determine the mean time to failure (MTTF). With the binary reward assignment the MTTF is the expected accumulated reward until absorption, given by:

$$E[\phi(\infty)] = \sum_{i \in S} r_i \tau_i \qquad (11)$$

where $\tau_i = \lim_{t \to \infty} L_i(t)$. A measure that is commonly required is the distribution of accumulated reward, Y(t). This can be computed as

$$P[\Upsilon(t) \leq x] = \sum_{r_i \leq x, i \in S} P_i(t) \qquad (12)$$

For example, the distribution of time to complete a task that requires r time units can be computed by:

$$P[\chi(r) \leq t] = 1 - P[\phi(t) < r] \qquad (13)$$

where $\chi(r)$ is a random variable denoting the time to accumulate reward r.

Dependability measures

In dependability modelling a reward rate of 1 is assigned to all the working (or up) states and a rate of 0 to all the fault (or down) states. The instantaneous availability of a system is then $E[\Upsilon(t)]$ and the steady state availability is $E[\Upsilon_{SS}]$. The cumulative operational time of the system in the interval $[0,t)$ is $E[\phi(t)]$. Interval availability is the proportion that a system is available in a given interval t and is given by $E[\phi(t)]/t$.

Measures relating to the first system failure are also of interest. To determine these, all states must be absorbing, ie all outgoing arcs are removed. Reliability is then given by $E[\Upsilon(t)]$. A systems lifetime, analogous to the cumulative operational time [30], of the system interval $[0,t)$ is $E[\phi(t)]$ and mean time to system failure MTTF is $E[\phi(t\infty)]$. Note that steady state measures can only be computed when none of the systems states are absorbing. Conversely, reliability measures such as MTTF can only be determined if all the systems fault states are absorbing states.

Performance measures

Performance measures are determined by identifying some suitable reward assignment. For example the queue length at a particular part of the system. Then $E[\Upsilon_{SS}]$ and $E[\Upsilon(t)]$ will give the average steady state and average transient queue lengths, respectively.

In a performability model, the reward assignment will typically be determined from a performance model, which is computed for varying states of a dependability model. The performance probability measures can then be computed incorporating the effect of dependability.

The reader is referred to [31] for a comprehensive discussion on the hierarchy of techniques applicable to dependability modelling.

2.4.3 Stochastic Petri nets

Stochastic Petri Nets (SPN) have been developed by Marsan et al [32] and Meyer et al [33] and are an extension to Petri Nets (PN) [34] and differ in that their transitions have exponentially distributed firing times as opposed to being untimed and immediate. They provide a concise graphical means of high-level representation for modelling a system's behaviour and have been traditionally used in the modelling of concurrent systems [1].

With a SPN a set of places, P (depicted as circles), a set of transitions, T (depicted as bars) and a set of arcs, A (depicted as arrows) between circles and arcs and vice-versa : $A \subseteq (P \times T) \cup (T \times P)$. Each place can contain zero or more tokens (depicted as dots within places). The distribution of dots is termed the *marking* and is analogous to the state of an MRM. Arcs from places to transitions are termed *input arcs* and arcs from transitions to places *output arcs*. A transition may fire when it is *enabled*, that is when there is a token in each of its input places. Upon a transition firing, one token is removed from each of its input places and one token is put in each of its output places. This will result in a new marking, ie state. The firing of transitions takes an exponentially distributed firing time.

The time variant behaviour of an SPN is given by its reachability graph and this has been shown to be isomorphic to a CTMC [35]. Their evaluation is therefore conducted by generating and solving the underlying CTMC. Software tools exist which facilitate this, using as input an SPN specification [36].

An extension to the SPN which is applicable to dependability and performability modelling is the Stochastic Reward Net (SRN). Analogous to the MRM, in an SRN reward rates may be associated with the place markings and impulse rewards with the firing of transitions. The SRN is therefore a high level specification of the MRM.

It should be noted that even though SPNs provide an exact and efficient specification technique, their modelling power is the same as that of Markov models.

In dependability modelling a reward rate of 1 is assigned to all the working (or up) states and a rate of 0 to all the fault (or down) states. When describing the system by the use of SRNs the corresponding values are assigned to place markings which denote the working and failure states.

Catania et al [37] propose a generic framework using GSPNs that can be applied to all gracefully degrading systems to obtain performability measures. The procedure is based in the definition of three fundamental models that all system components can be represented by. The approach is certainly flexible and easy to apply but it's generic nature dictates that for many applications there may not be scope for the required amount of detail.

2.4.3.1 High-level Petri nets

There have been a number of extensions made to the basic SPN model described above resulting in a variety of types of *high-level* Petri nets. Below we introduce some of the more significant advances.

Coloured Petri nets [4] (CPN) where each token has an attached data value known as

the *token colour.* This provides the ability to produce a significantly more compact representation. The use of coloured tokens is flexible. Using a single colour is equivalent to an SPN. A number of colours proportionately reduces the size of the net and more information is given in a textual manner on the net. These textual descriptions are termed *net inscriptions.* A descriptive language *coloured Petri net - modelling language* (CPN-ML) can be used for the net inscriptions.

Generalised coloured stochastic Petri nets [2] (GCSPN) are more flexible than SPNs as they allow for immediate deterministic transition timings as well as stochastic timings. Interval timed coloured Petri nets [38, 39] (ITCPN) provide timing intervals for transitions. These are stochastic timings with a specified minimum and maximum firing time. Object oriented Petri nets [5] are an extension to GCSPNs where tokens are grouped into types (equivalent to sets of colours) relevant to the application. This enables data abstraction and inheritance.

Channels for synchronous communication have been suggested to connect transitions. These allow transitions to communicate via complex values and have been applied to modelling synchronous communication systems [40].

Three extensions that have a significant impact on the descriptive power of Petri nets are place capacities, test arcs and inhibitor arcs [41]. Place capacities restrict the colour and number of tokens that can exist in places to a specified value. Several test arcs are allowed to access the same token in a place but not in the same step as ordinary arcs. Test arcs cannot change the marking of a place and aid the modelling of concurrency. Inhibitor arcs can be considered to be the opposite of ordinary arcs. They are always input arcs to a transition and prevent the transition from firing if a corresponding token exists in the input place.

The Devnet [42] introduced by Evans is an adaptation of SPNs which is used as a graphical description of discrete event simulations.

A recognised limitation of Petri nets is their very small number of primitives which means models can quickly become complex. An extension which has been developed

by Sanders and Meyer [33, 43] is the *stochastic activity net* (SAN). SANs are similar to SPNs having *places*, *activities* (analogous to transitions) and *gates*. The gates form the fundamental difference between SANs and SPNs. They can be of two types, *input* gates which have inputs from several places and an output to one activity and *output* gates which have an input from one activity and outputs to several places. Gate functions prescribe how gates are enabled and the passing of tokens. The gates give SANs a greater flexibility and allow for increased conciseness over SPNs. In [44] a SAN performability model of a multiprocessor system is presented. The performance and reliability parts of the model are divided into sub-nets facilitating a hierarchial approach. This avoids the *stiff* problem caused by the large order of difference in timings between the two aspects. Sanders and Meyer developed METASAN [45] which allows for model specification using SANs. Solution is facilitated by either simulation or analytical techniques [56]. An input descriptive language similar to CPN-ML describes the SAN by specifying all places, activities, gates, the links between them and the gate functions.

2.4.3.2 Modelling and analysis with stochastic reward nets

Modelling and analysis is facilitated using SRNs by a 3 stage process.

i) The system is described with a SPN. The rates of timed transitions must be specified along with probabilities for immediate transitions. Priorities and weights are used to solve conflict between simultaneously enabled transitions. Some of the many extensions to Petri Nets e.g. inhibitor arcs, transition guards, marking dependant arc cardinalities etc may be required to produce a concise description.

ii) Generation of the underlying MRM. This requires the construction of the reachability graph. One of two techniques may be applied. Either it may be transformed into the equivalent MRM by elimination of all the vanishing markings [36] or the vanishing markings may be preserved and the stochastic process can be converted into a Discrete Time Markov Chain (DTMC).

iii) The MRM is solved for the system characteristics which are of interest.

Performability can be evaluated by replacing the binary reward values in the MRM with different reward rates appropriate to the individual states of the system. To illustrate this, consider a multiprocessor system which has N processors. Each processor is susceptible to failure. These failures occur with a rate λ. A single resource facilitates repair with a rate μ. The CTMC of this system is shown in figure 2.1. This is a finite birth-death process and can be simply specified by the two place SPN of figure 2.2.

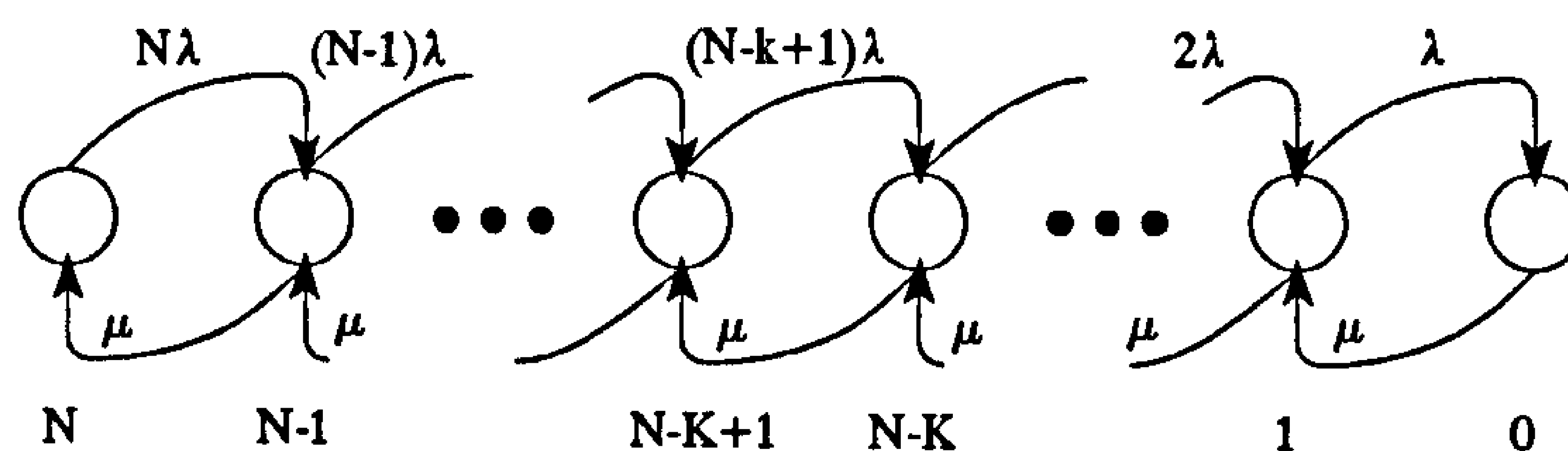


Figure 2.1 CTMC of multiprocessor system

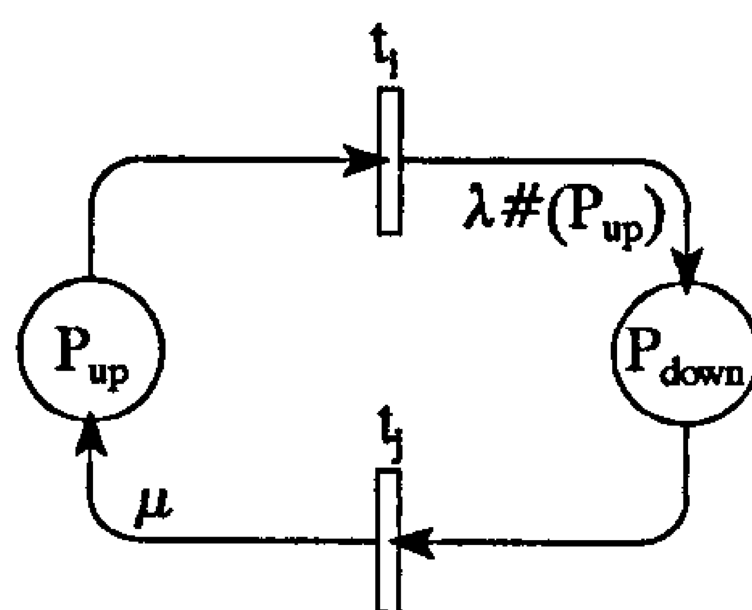


Figure 2.2 SPN of multiprocessor system

If each processor possesses a processing rate of σ the reward rate assigned to each state is different and is equal to $k\sigma$, given that $0 \leq k \leq N$. The SPN can be specified as a SRN with rewards associated to each place. The place marking would be $\sigma\#(p_{up})$, given that $\#(p_{up})$ is the number of tokens in the place p_{up}, each token representing a processor.

2.4.4 Hybrid and hierarchical Models

When two or more techniques are applied to the construction and solution of a single model the approach is termed *hybrid*. Often this takes the form of hierarchial modelling. Submodels may be specified and analysed using one methodology and the results incorporated into a higher level model. It should be noted however that not all hierarchial models are hybrid, for example the decomposition result in PFQN is a form of non-hybrid hierarchial modelling.

Malhotra and Trivedi [31] suggest a formal generic methodology for expressing hierarchy both within model specification and solution. The approach is flexible being suitable for both hybrid and non-hybrid specifications. A unified view is taken of all modelling techniques so that the hierarchial structure of the model is brought to the fore. An introduction to the theory of hierarchical SPNs is given in [46] and Buchholz describes a developed approach for CGSPNs in [2]. Although this approach is based in CGSPNs it is portable to other high-level Petri net types.

Balbo et al [47] combine queuing networks and GSPNs for the pure performance analysis of models with non-product form characteristics. The non-product form parts of the system are solved using GSPNs, the results of which are used in load dependant queues which are PFQNs. Szczerbicka [48] develops this approach using a decomposition approach where the system is split into both GSPN submodels and a special class of queuing models, termed BCMPs. The submodels are solved in isolation and replaced as marking dependant GSPN submodels in a high level GSPN model which can be solved to give performability measures.

The software tool, SHARPE [22] (described in section 2.5) allows many model types to be analysed using a variety of techniques. Results from submodel analysis can be incorporated into other models. This can also be done in a cyclic way with fixed point iteration techniques being applied to solve the whole model. A number of base model types are provided and the number of models of each type at any level and the information passed between models is left to the users discretion. Benitez and Trivedi [49] propose a method of multiprocessor performability analysis where queuing networks

are used to define performance measures and solved to derive reward rates that are used within an overall Markov failure-repair model. Several combinations are solved using SHARPE and compared.

Haverkort proposes a dynamic queuing network concept [50] where queuing networks are used to describe the performance aspects and GSPNs the dependability aspects of fault tolerant computer systems. A combined model is not explicitly constructed rather an approximate behavioural decomposition solution is presented, as is common in performability modelling. In [51] Haverkort develops a model where the performance aspects are also modelled with GSPNs and two heuristic state space truncation techniques, that allow good approximations of steady state performability values, are introduced.

2.4.5 How should realistic systems be modelled ?

Many of the characteristics of real systems cannot be modelled using current analytic techniques. Many systems cannot be separated into smaller independent subsystems and their detailed state representation may be excessively large. Also, the events in real systems may not be "memoryless" and event times can differ by several orders of magnitude. These factors imply that simple analytical models such as product form queuing networks may not be powerful enough, and detailed analytic models such as Markov chains, may be unmanageably large and stiff. This has lead to a dependence upon discrete event simulation in many instances. Discrete event simulation is often improperly used and complex models are applied to situations when simpler techniques would suffice. In the authors opinion correct implementations of discrete event simulation is often the most applicable method when accurate predictions of the performance, dependability and performability of systems comprising interacting components is required.

2.5 The evolution of performability

Initial work in the mid 1970s investigated a variety of alternative formulations for combined performance-dependability measures. This was motivated by different systems and application considerations. A major contribution was Beaudrey's treatment of "performance-related reliability" [52]. She associated a fixed computation rate with each structure state and thus constant fault arrival rates (in a Markov reliability model) are translated into "faults per unit of computation". As an example this method would translate a reliability measure such as "mean time to failure" to the performance related measure "mean computation to failure". Advantageously, techniques for evaluating the reliability measure also apply to the performance related measure since the translated model is also Markovian.

Meyer was one of the main contributors to the field and first proposed the term *performability* in 1978 [15]. His initial contributions are described in section 2.2.

Most of the work however came from interests in fault-tolerant computing. A number of contributions were made ranging from general evaluation methods to specific applications [53, 28].

The initial performability work focused on evaluation with respect to discrete valued performance. This was the focus of Meyer's paper [16]. In this case, for any level of accomplishment $a \in A$, the ability to perform exactly at that level is measurable. To account for variations in user demands during a bounded period T, the construction of the base model X could employ the notion of a *phased model* [28] where T is split into a finite number of consecutive time periods. For each period the systems intra-period behaviour is represented by a continuous time, finite state Markov process. The driving application for this work was the performability evaluation of fault-tolerant multiprocessors for aircraft control [54].

In the early to mid 1980s there was a concentration on the development of model based performability evaluation [55, 53]. The methods developed then are the basis of current techniques. One of the most influential advances was the introduction in the early 1980s

[28] of solution methods based on *reward models* as discussed in section 2.4.2.

It was also in the early 1980s that the development of stochastic Petri nets became an area of considerable interest to modellers. Their use in performance and dependability evaluation began. Motivated by the consideration of further features, such as extended timing and coloured tokens, they became suitable for performability modelling. Stochastic net models, particularly SANs and SRNs (section 2.4.3.2), are now the normal means used for automated performability evaluation. Prodromides and Sanders [57] show the ease of specification using such a graphical technique by evaluate two types of CSMA protocols, defining performance and dependability measures using SANs and producing results through simulation with METASAN.

To this point the only application area had been computer systems. As interest grew performability measures were applied to other fields such as satellite systems [58] and communications systems [59] which are now a major area.

Since the mid 1980s there has been increased interest in performability model construction and solution techniques, model based evaluation and the applying of these techniques to the areas of computer and communication systems.

Many of the solution methods focus on performability models incorporating some type of Markov reward model with accumulated reward, $Y(t)$, as the performance variable. Closed form solutions of performability, $F_{Y(t)}$, have been developed for acyclic, non-recoverable Markov reward models [60]. Using transform techniques e.g. Laplace transform, to derive solutions is a popular technique. If only the expected value $E[Y(t)]$ is required, solutions tend to be less complex as the expectation is normally of a linear value.

When systems incorporating some form of repair are considered the underlying process model is no longer acyclic. In these models, solutions of the PDF of $F_{Y(t)}$ are more complex [61]. Solutions appropriate to rate based Markov reward models have been based on Laplace transforms [53] that involve the transformation of both the time variable t and the accomplishment level y. Laguerre transforms have also been used

[62]. There have also been solution techniques developed for steady state performability measures. These are typically based on queuing network models of systems [63] for which repairs can be continually repeated giving meaningful steady state behaviour.

Application areas

Performability evaluation has proven useful when applied to a variety of aspects of communication systems. Van Dijk presents performability bounds for communications networks using a message throughput as a performability measures [64]. Jones and Malec [65] look at overall system performability and imply that it is the control software that has a major impact though no actual results are given. Yang and Kubat [66] propose a fast algorithm where the performability of a network is taken as the average performance over the most probable network states. This algorithm is fast and efficient but the approximation is made that states with a low probability of entry are never entered. Koren and Koren [67] have applied performability measures to gracefully degrading multiprocessor networks identifying a number of performance measures. These networks are very similar to the Banyan class of network which is a popular interconnection network used in ATM switches, discussed in chapter 7. Bhattacharya et al [68] model such a Banyan network with the performability measure relating to the likelihood of correct routing. They make the popular assumption that all input traffic has a uniform distribution of output addresses. This would not always be a valid assumption to make in the case of ATM networks as traffic is often bursty.

2.6 Tools

An onus is often put on system designers to develop the best possible product in the shortest possible time with minimum use of resources. For these reasons, tools that are quick to learn, easy to use and allow models to be constructed quickly will be highly sought after. Tools that are to be of practical use should have simple, self explanatory graphical user interfaces. Also the capability to define and reuse submodels that can be combined to produce large models and provide the ability of multiple instances is essential.

A variety of performability modelling and evaluation tools have been developed. A brief description is given of a number of them below. Most of the tools use either a Petri net or Markov process formalism for constructing base models. A comprehensive guide to Petri net tools is given in [69].

LOOPN [70] is a language and simulator for specifying systems in terms of coloured timed Petri nets. It utilises an object oriented approach and includes many of the features of extended Petri nets. The use of objects facilitates the use of existing Petri net models. *CPN simulator* [4] is another tool which uses CPNs to construct a base model. It enables graphical input in the form of CPNs with net inscriptions. The split between detail contained in the net and in the inscriptions is determined by the user. It does not support object oriented features.

DyQNtool [71] uses extended GSPNs to construct a base model. Reward rates are enabled through the use of PFQNs. This tool is one of the most advanced with respect to automating reward model construction using a separate window permitting designation of the source and type of performance values obtained via the queuing model analysis.

DSPNexpress [72] was initially developed as a performance and dependability analysis tool using DSPNs to generate base models. It has been well proven in producing steady state solutions for some quite complex systems. However it suffers from the restriction that no more that one deterministic transition in any marking may be enabled. This precludes its use in modelling many systems. The use of complimentary variables for analysis has been proposed to circumvent this, though it adds to the complexity.

METASAN [28] employs SANs to construct base models and has separate facilities for describing (i) the total system model and (ii) the performance variables used along with the performability solution required. (i) is facilitated by the use of an input language called *Sanscript.* (ii) includes both transient and steady state variables solved by either analysis (for Markov base models) or simulation otherwise.

HARP [73] uses a behavioural decomposition approach to model the reliability and performability of fault tolerant systems. Analysis is done via fault trees. It has been

used for quite large systems (> 24500 states). *SHARPE* [22] facilitates a number of approaches (e.g. reliability block diagrams, PFQNs, MRMs) and allows true hierarchial and combinatorial modelling. Using decomposition and aggregation (section 2.7.2) it can model large systems with Markov and semi-Markov submodels.

METFAC [74] utilises Markov base models which are generated by a production rule system. It has been used to model the performance, dependability and performability of computer systems giving steady state and transient as well as cumulative measures.

Many other simulation tools and languages exists, however we have limited our consideration to those that have been directly applied to performability modelling.

2.7 Problems encountered in modelling systems

In this section we consider some of the recurring problems in performance, dependability and performability modelling. They are *largeness* and *stiffness* of the model and the need for transition rates between model states to be *non-exponential.* Some methods which can overcome these problems to an extent are described.

2.7.1 Largeness

A model can be constructed that incorporates the performance level of a system (e.g. throughput, service times, cell loss) and the structural variations (due to failure, repair, reconfiguration etc) but due to the large number of states required the model may reach a prohibitive size very quickly. This is the *largeness* problem and is a main obstruction to the monolithic approach to modelling degrading systems. This poses problems in both model specification and analysis. Largeness must be either *tolerated* or *avoided.*

It is important to distinguish between *descriptive* and *computational* largeness. Often largeness is taken as a composite term covering both but in truth they are quite different. Whereas computing architecture continues to become increasingly more powerful, we

can safely predict that human capability to intellectually grasp the operation of highly interconnected systems will remain relatively limited. We are therefore primarily concerned with the descriptive largeness as presented to the modeller.

Using an GSPN framework to describe a model, whereby all that is specified is the GSPN and the underlying CTMC/MRM is automatically generated is a method of complexity hiding. The modeller need only be concerned with a degree of the entire complexity. When the state space is too large to be either efficiently stored or solved largeness must be avoided. This may be achieved by using approximation techniques such as *truncation*, *lumping*, *decomposition* and *fluid models*.

With *truncation* a number of similar states or those with a low probability of entry are combined [49]. Formally, given a reachability graph (S, A), (where S represents the set of *states* and A the set of connecting *arcs*), the state truncation results in a truncated reachability graph (S', A'). If $(S', A') \subset (S, A)$, the method is called *strict truncation*. Alternatively (S', A') might be a sub-graph of (S, A) augmented with one or more states and arcs, known as the *aggregation-truncation*. Constantinescu [75] uses this approach to model a fault-tolerant microcomputer with separate Markov models being used for fault handling and fault occurrence events.

In *lumping* the system is decomposed into a number of constituent subsystems with smaller state spaces. These are analysed separately and then recomposed to form the lumped model [36]. Lumping results in a state space reduction which is significant when the number of subsystems is large and their constituent number of states is small. If the subsystems have interactions then the application of lumping requires considerable care.

With *decomposition,* when there is a large difference between the rates of performance related events with respect to dependability related events which are rare, it is acceptable to assume that the system maintains a *pseudo* steady state with respect to performance related events in between occurrences of failure related events. The performance measures of the system for each of these pseudo steady states can then be calculated and the overall system characterised by weighting each of them by the structure state

probabilities.

In general, rather than solving a monolithic model, two submodels are solved independently, one to represent performance (the *reward model*) and one to capture the structural variations (the *structure state model*). The decomposition technique leads to a natural hierarchy of models. The structure state model is the higher level reliability/availability model representing the failure/repair/reconfiguration processes of the systems components. Each state in the structure state model will have an associated reward model which is a performance model for the system with the given stationary structural state. The performability measure is obtained by combining the performance measures associated with each structural state with the probabilities obtained from the structure state model. This is a common approach taken by Meyer [24] and Beaudry [52]. Most of the proposed approaches are based on the common theory of MRMs, introduced in section 2.4.2.

Fluid models with respect to GSPNs make use of the idea that as the number of tokens in a place becomes large and the underlying CTMC grows, it may be possible to approximate the number of tokens as a non-negative real number. From this it is then possible to write the differential equations for the dynamic behaviour of the model and in some cases, determine solutions. For this purpose Trivedi and Kulkarni [76] have proposed fluid SPNs.

2.7.2 Stiffness

Combined modelling of performance levels and structural changes can also cause *stiffness*, which is a direct result of extreme disparity between the occurrence rates of performance related events and failure/repair events. Stiffness can cause considerable problems during the analysis of a model that adversely affect its stability, accuracy and efficiency, even if the model is not large. This is especially true in monolithic models where performance related transitions can occur far more frequently (e.g. $\sim 10^9$ times more) than dependability related events. As with largeness, stiffness may be overcome by either tolerance or avoidance.

Tolerance is achieved by implementing special mathematical techniques for the solution of the system of differential equations [77, 78]. Avoidance is linked to that of largeness avoidance in that the same approximation techniques may be applied. Hierarchical modelling using aggregation such as that proposed by Bobbio and Trivedi [79] is an applicable method.

2.7.3 Non exponential behaviour

With some modelling techniques such as SPNs and some types of PFQNs, it is assumed that all transition rates are exponentially distributed. In many applications this is not acceptable as non-exponential rates may occur. Examples of this may be deterministic times in communications protocols and Weibull distributions for message acknowledgements.

With CTMCs one solution is to use a range of states each with exponential holding times whose overall transition rate approximates the non-exponential firing rate [80]. This approach is conceptually simple but it increases the state space and there may be a significant problem approximating the desired rate. Many stochastic extensions have been suggested for SPNs which facilitate non-exponential rates. As well as GSPNs discussed in section 2.4.3.1 these include DSPNs (Deterministic SPNs) deterministic and exponential rates and ESPNs (Extended SPNs) which allow generally distributed rates. When a simulation approach is taken, non-exponential timing is not a problem.

2.8 Conclusions

In this chapter we have considered the field of performability modelling. An introduction has been given to the topic and performability has been formally defined. We have discussed the evolution of performability from the first requirement that lead to its development up to current issues. Four modelling techniques were examined, these being non-state space, Markov, stochastic Petri nets and hierarchial and hybrid.

When considering the development of performability modelling tools it was noted that most utilised Markov chains and/or Petri nets. We have presented a unified framework for performability, performance and dependability measures in terms of Markov chains. Problems encountered in the modelling of systems have been discussed and the implication of these considered.

We have seen that when investigating modelling techniques it is important to consider the implications of the method on the descriptive complexity of a problem. It is the human interface to an approach that will often dictate its worth. In chapter 3 we present a novel formalism for the description and simulation of complex systems that addresses this issue. This new approach builds on many of the ideas presented in this chapter.

Chapter 3

The ICE Language

3.1 Introduction

In this chapter we describe the current implementation of the ICE language.

What became very clear during the development of the language was that its semantics are paramount to its understanding. The ethos adopted has been to make the syntax as intuitive as possible thus making the translation of real problems into language models as simple as possible. Although the nature of the syntax is declarative it is supported by a computational model which must be understood before the language can be used to solve real problems accurately. Additional modifications to the syntax have been adopted to give the language greater uniformity and to clarify the computational model.

The development of the compiler and simulator gave a clearer view of the potential uses of the language. Further examination of possible application areas has since shown a number of shortcomings in the original syntax. A main objective has been to increase the modelling power of the language but without significantly increasing its complexity.

For the sake of completeness, continuity of argument and preserving the flow of the text

a short overview of the language is given in section 3.2. Readers already familiar with the language may proceed to section 3.3. In the remainder of the chapter we limit our discussion to a few of the language's more interesting features and semantic considerations are discussed to illustrate the reasoning behind the developments. A complete description of the entire syntax and all the language's constructs is given in appendix A. This chapter should therefore be read in conjunction with appendix A.

3.2 Overview

The language has a declarative style that is based upon describing systems in terms of their constituent interacting discrete state components.

Each COMPONENT in a system has a finite number of operational states. The component moves between the various states in this STATE_SET according to its predefined BEHAVIOUR.

COUNTERS are component variables which are primarily used to counteract the problem of state explosion. For example, if we wished to model a buffer with 100 spaces, we could do so by using 101 states, i.e. 1 state for the empty condition and 100 for each of the levels of occupancy. Alternatively, there could be 1 state to represent the buffer and a counter which may take any value i where $\{ 0 \leq i \leq 100 \mid i \in N \}$. This clearly allows the state complexity of models to be greatly reduced.

The transitions between states may be governed by :

- Time delays, both deterministic and stochastic.
- A boolean function of one or more component states, known as a SYSTEM.
- The event of a transition between states of another component.
- The value or change in value of a COUNTER associated with this or another component.

Components may also have associated with them a variable AGE which can be used to manipulate their behaviour.

To fully define a component , three statements are required :

- STATE_SET, which lists the finite set of states a component can exist in, any counters belonging to the component and any modifications to be made to these counters when the component enters the different states.
- BEHAVIOUR, which defines all possible transitions that can be made between states.
- COMPONENT, which defines a component with a specified STATE_SET and BEHAVIOUR. It also gives the initial state of the component and optionally an initial age and initial counter values.

As well as components we can also describe passive resources which may be allocated to components. Resources may be consumable or non-consumable and are specified as STOCK and RESOURCE respectively. The WAIT_FOR statement allows the explicit manipulation of resource levels during simulation.

The language syntax is free-format in the sense that blank spaces (spaces, tabs, new lines etc) are ignored. The order of the statements is unimportant, except in the instance where this would cause a semantic conflict. This point is expanded in section A1.5.

3.3 Language details

In the examples of syntax given the following conventions are used :

Keywords are shown in UPPERCASE.
User defined names are shown in *italics*.
Optional syntax is shown in [square brackets].

The complete language syntax is described in appendix A.

3.3.1 Counters

One problem that became immediately apparent when considering modelling with the language was that of state explosion. In the original language each component in a system could be thought of as a finite state machine with inputs. A separate state was required to represent every possible condition of the component. This is feasible when conditions are distinct. However conditions are often very similar and using a large number of states to describe similar conditions was considered to be an inefficient way of modelling.

To illustrate this we consider the modelling of a buffer which has *N* places. Three distinct conditions may be *empty*, *occupied* and *full*. We might however wish to know each level of occupancy and this would result in a model with $N + 1$ states as shown in figure 3.1.

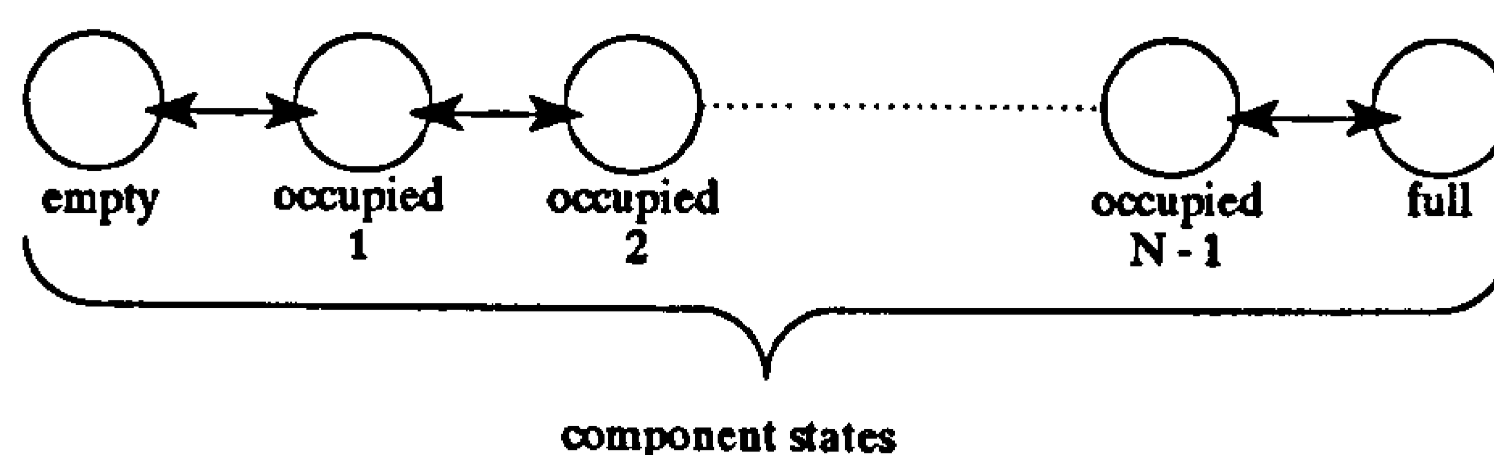


Figure 3.1 State diagram of N-place buffer

Models of this nature would obviously soon become very large. Consider say a simple multiplexer which has 8 inputs and 1 output all with a 100 place buffer. It was for this reason it was decided to add an extra descriptor to a component so that similar conditions could be grouped into a single state, whilst still retaining a mechanism to distinguish between them. In the buffer example this could reduce the number of states required to 3 i.e. one for each distinct condition with the extra descriptor being used to distinguish between the levels of *occupied* or indeed to just one state which covers all conditions. These simplifications are shown in figure 3.2.

Figure 3.2 State diagram of N-place buffer using an extra descriptor

3.3.1.1 State integer attributes

The first possibility investigated was to utilise the component state attributes. These are boolean variables which are solely used to group together selected states. Consider as an example the state set

```
STATE_SET link
{
        O       busy;
        O       idle;
        F       broken;
        F       erroneous;
}
```

The uppercase letters preceding the states are the state attributes. The attribute '*O*' groups together the operational states *busy* and *idle* and the attribute '*F*' groups together the failed states *broken* and *erroneous*. An attribute is considered to be '*true*' for all states that posseses the attribute and '*false*' for all other states.

An extra descriptor could be introduced by allowing integer state attributes which may take on any integer value rather than a boolean value. This seemed like a simple way to augment the description of a state. However upon reflection some disadvantages were identified.

Firstly, the existing attributes exist solely to link states together. Integer attributes would be used to enhance the description of a state thus having a different meaning altogether. This was considered unacceptable as it would remove the simplicity of the attribute concept by giving it two conflicting meanings. Secondly, attributes are only available when a component is in a state that possesses that attribute. This was envisaged to be highly restrictive as the integer value may be required to be known when the component is in any state. For these reasons it was decided to discard the idea of integer state attributes and seek an alternative which would avoid conflict with existing constructs and be accessible when a component is in any state.

3.3.1.2 Resources as state descriptors

Resources are passive entities which may be allocated to components. They are created either by use of the RESOURCE statement in a system model or during simulation time by the WAIT_FOR statement, as described in appendix A.5. A dynamic count of the number of resources available is kept and this is decremented when an ON_RESOURCE transition is activated. The number of resources specified is decremented from the count and they are considered to be allocated to the state the component has moved into. When the component moves out of this state the resources are freed and the count is incremented accordingly.

The inherent counting which takes place as resources are manipulated suggests that there may be a way to utilise resources as state descriptors. We will use the previously described buffer example to illustrate this concept. The buffer may be described by use of a single state component *buffer* and the resource *space* of which 100 entities are initially created as shown in figure 3.3.

Initially there will be 100 *space* resources none of which are allocated to state *normal* of *buffer*. Whenever a *get_space* transition occurs the number of 'free' *space* resources will be decremented by one and this resource will be allocated to state *normal*. The complementary operation will occur when the *drop_space* transition occurs. The level of occupancy of the buffer is now represented by the number of resources allocated to the component. This approach seemed quite promising initially but upon reflection a number of problems were identified.

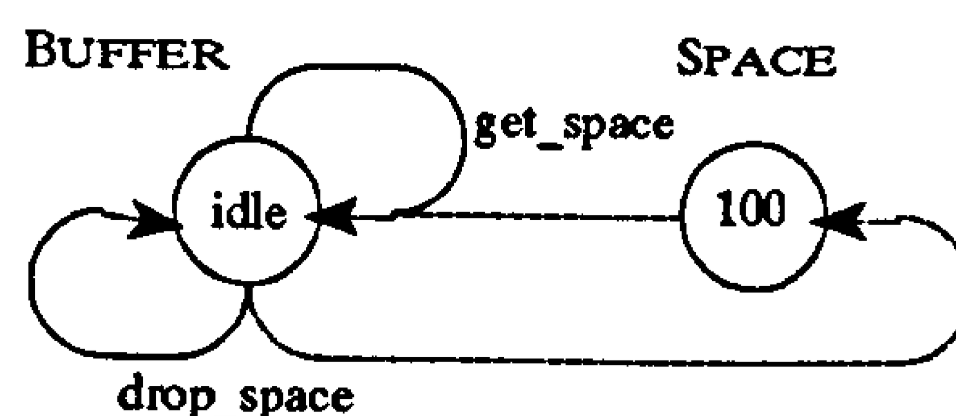


Figure 3.3 Buffer using Resources

The transitions *get_space* and *drop_space* are irregular in that they do not change the component state but only affect the resource allocation. This was thought to be

inappropriate as this would introduce transitions with no state change, conflicting with the intuitive understanding of a transition. This confuses the semantics and as was stated earlier the aim is to keep the semantics as intuitive as possible. In the traditional sense of resources we normally wish to know the number of free resources and this is immediately available. In this instance however we would wish to know the number of resources allocated which would involve determining the initial number of resources and calculating those allocated by subtracting the number which are free. Currently we only inspect resource levels when using ON_RESOURCE pre-conditions and when these are enabled and transitions occur resource levels are changed. We would now have to create new pre-conditions which could inspect initial resource levels and current resource levels but have no ability to change these levels. This of course could be done but it would change the concept of resources which would confuse their use.

A further problem is that component descriptions would no longer be self contained but have a level of abstraction. To fully obtain a components condition we would have to consider its state and the condition of an external resource. It was decided that the best solution would be to provide some means within a component statement to provide extra descriptive power but which would not be confined to any single state as with integer state attributes. To this end the concept of component integer attributes was developed. These have been named counters.

3.3.1.3 Component integer attributes

From the reasoning expounded above it was decided to use component integer attributes as the means by which to increase the descriptive power of components. Component integer attributes have been called COUNTERS. When considering the integration of counters into the language several questions had to be considered

1. Where to declare them and the syntax to use ?
2. Where and how to initialise them ?
3. Where to modify them and the method to use ?
4. How to use them as transition pre-conditions ?

3.3.1.3.1 Counter declaration

Since counters are component attributes the options for declaring them was either in the COMPONENT statement or in the STATE_SET statement. Since it is highly likely that components which share the same STATE_SET would also require the same counters it was decided to add them there. The modified STATE_SET takes the following form

```
STATE_SET name {
        COUNTERS : [ counter_list ] ;
        STATES {
                state_list ;
                ...
        }
}
```

The *counter_list* is an optional list of user defined counter names separated by commas. A full description of the statement is given in section A.2.

3.3.1.3.2 Counter initialisation

All counters associated with a component's state set must be given an initial value just as it is mandatory to define an initial state for each component. This value will be component dependant and not fixed for a given state set and therefore it was decided to initialise the counters in the COMPONENT statement. For continuity this is done in the same line as the setting of the initial state modifying this line to be

```
...
[INIT_STATE :]      state_name [ ( counter_init_list ) ] ;
...
```

The *counter_init_list* is optional, any counters not initialised default to 0. It will consist of one or more counter names which are assigned integer values separated by commas, for example

```
( counter_a = 3, counter_b = 7 )
```

3.3.1.3.3 Counter modification

When considering the possible uses of counters the modifiers identified as being neccesary were incrementing, decrementing, addition, subtraction, multiplication and division. It was also perceived that in certain scenarios modulo arithmetic may be necessary.

Incrementing and decrementing are of course specific cases of addition and subtraction therefore leaving the four standard arithmetical operations to be implemented. This could be done in two ways, either by

i)	using a function type operation, e.g.	ADD(*counter_a* , 7) or
ii)	an arithmetical statement, e.g.	*counter_a* + 7

Functions could be built into the language or defined by the user at the time of modelling. Inbuilt functions would provide a powerful set of tools for the user but unless they were restricted to basic single operations this could lead to quite an increase in syntax. It would be possible to have a mixture of inbuilt and user defined functions. Arithmetical statements have the disadvantage of having to be repeated if required in different places but they are more intuitive and provide the modeller with a large degree of flexibility.

A major influence in deciding what form to use was the consideration of the impact on the existing syntax. Arithmetical statements could be easily supported as they are already used within SYSTEM statements. However, use of one should not exclude the other and it was decided to utilise both.

The syntax for updating a counter is

counter_name = expression

where *expression* is any arithmetic statement supported by the expressions used in SYSTEM statements, eg

counter_a = counter_a + 7 ;
counter_b = counter_c / 12 ;

The single addition to this is a function type operation for modulo arithmetic, MOD*n*(*expression*), where *expression* is consistent with the above.

Once the syntax of counter modification was identified it was neccesary to consider where the operation would occur. Counters are global to a component and their manipulation is state dependant. This follows the ethos of the language in following a Moore model where actions are a function of current state rather than the Mealy model where actions are a function of transitions. This gives two possibilities for methods of updating

i) The update could be linked to a given state in the state set, as shown in the partial statement

```
STATE_SET comp_a {
{
        COUNTERS :        counter_a ;
        STATES {
        ...
                state_y : ( counter_a = counter_a + 7 ) ;
        ...
        }
}
```

ii) The update could incorporated into a state transition within a behaviour statement, as shown in the partial statement

```
ON_EVENT  comp_b.state_b
{
        state_x -> state_y : ( counter_a = counter_a + 7 ) ;
}
```

The original concept of counters was to add extra descriptive power to states. Updating a counter after a transition into the given state was felt to detract from this. It would mean that there would not be a consistent link between counters and states for the updating would become dependant upon how a component entered the state. The other disadvantage of this is that it would add more complexity to the already detailed behaviour statement. The one disadvantage of adding the update to the given state definition is that in some cases we may want an update to be dependant to the way a state is entered. In these situations though, we can use an additional state and preserve the state counter identity and for these reasons it was decided to link counter modification to the state set.

3.3.1.3.4 Counters as transition pre-conditions

Transitions can be enabled by pre-conditions which depend on the state of other components. Since counters are an extension of the state description it is desirable to be able to use them in pre-conditions also. It was decided to expand the expression fields of the ON_EVENT and IF pre-condition statements to include boolean counter expressions, eg

```
ON_EVENT ( counter_a > 9 )
{
        state_x -> state_y ;
}
IF ( ALL ( comp2.state_a, counter_a == 2, counter_b < 3) )
{
        state_x ->state_z ;
}
```

Since the boolean expressions used are the same as those that may be used in SYSTEM statements, counters by default may also be used in SYSTEM statements. Note that the equality symbol used is '=='. This is due to a single '=' signifying assignment and '==' was considered the next most intuitive syntax to denote equality.

3.3.2 The syntax of transitions

The originally proposed way of specifying transitions was

```
PROB( 0.3 )  state_a -> state_b    12 ;
PROB( 0.7 )  state_a -> state_c    18 ;
```

Probabilities must sum to 1 and they default to 1. Times are given at the end of a statement and default to 0, i.e. and immediate transition.

This syntax was satisfactory when first devised. When we start considering the language more as a tool for modelling real time systems, the timing of events becomes of greater interest. The simulator developed conducts discrete event simulation and the relative timing of events must be considered in all instances. To reflect this change in emphasis it was decided to restructure the ordering of transitions to be

```
12    state_a -> state_b    PROB( 0.3 ) ;
18    state_a -> state_c    PROB( 0.7 ) ;
```

Although this is a minor change it does preserve syntax consistency and is more intuitive.

3.3.3 Timing with transition pre-conditions

A fundamental feature of the language is the interaction of different components. This interaction is controlled by transition pre-conditions which enable the transitions depending upon the condition of other components. A review of these pre-conditions has lead to some major modifications.

A full explanation of all pre-conditions is given in section A.5, the three we are primarily interested in are ON_EVENT, IF and IF_ON. We will take a brief look at how these may be used by considering some examples

i)
```
exp(70)  first -> alt_1 ;
ON_EVENT other.fail {
        first -> alt_2 ;
}
```

If the component described by the transitions enters the state *first* then normally it would move to state *alt_1* after a random time, *T*, determined by the exp(70) function. However if after any time, *t*, during the components existence in state *first* the component *other* moves into state *fail* then the component will be forced immediately into state *alt_2*.

ii)
```
exp(70) first -> alt_1 ;
IF other.fail {
        first -> alt_2 ;
}
```

If the component *other* is in state *fail* when the compongnt enters state *first* then it will immediately be forced in to state *alt_2,* otherwise it would move to state *alt_1* after a random time, *T*, determined by the exp(70) function. If the component *other* moved into state *fail* at any time *after* the component had entered state *first* it would have no effect.

iii)
```
exp(70) first -> alt_1 ;
IF_ON other.fail {
        first -> alt_2 ;
}
```

If the component *other* is in state *fail* when the component enters state *first* then it will immediately be forced in to state *alt_2*, otherwise it would move to state *alt_1* after a random time, *T*, determined by the exp(70) function. If the component *other* moved into state *fail* at any time, *t*, after the component had entered state *first* then at time *t* it would be forced immediately into state *alt_2* and the transition to *alt_1* would be disabled.

Upon reflection ON_EVENT and IF_ON were thought to provide very useful means by which to model component interaction and they are both quite intuitive. IF on the other hand was thought to be quite confusing. IF_ON is a composite operation, whereas IF is a primitive and more favoured by purists.

As was shown in the example the IF expression is only checked upon entering a state and thereafter it is not considered. This conflicts with the intuitive feel for 'IF' and was thought to be misleading. Unless a modeller has an exact grasp of the sequencing of events the IF pre-condition could be misused and produce confusing results. From this reasoning it was decided to drop the IF precondition as it stood and to rename the IF_ON precondition IF. The result of this is no noticeable loss in modelling power and a 'safer' set of commands.

As shown in the examples there can be no timing associated with transitions which are enabled by a pre-condition. The original reasoning behind this is that all transitions of such a nature should be forced and thereby have the effect of an immediate interrupt on current processing. This was thought to be too restrictive. In many instances we will not want forced transitions to be immediate but have some time associated with them. In the original language this was done by adding *dummy* states. A pre-condition would enable a transition which would force the component into a dummy state and a second transition would move the component from this dummy state into the final state in the time desired. This is adding extra state complexity to the model which is never desirable. If timing were incorporated into the precondition dependant transitions then these would not be required.

Another problem which is not immediately apparent is that of the effect that different

preconditions have on each other. Consider the example of modelling a car. We may wish to model the scenario where on the event of the petrol light coming on we will wish to stop for fuel in 5 time units and on the event of the oil light coming on we will wish to stop for oil in 3 time units. To model this in the original language we could use the following

```
ON_EVENT car.fuel_light {
        running -> need_fuel ;
}
5       need_fuel -> getting_fuel ;
ON_EVENT car.oil_light {
        running -> need_oil ;
}
3       need_oil -> getting_oil ;
```

Consider the following possibility. At some time t_1 the fuel light comes on. This will cause the component to immediately go into state *need_fuel* and in 5 time units go into the state *getting_fuel*. Say then at time $t_2 = t_1 + 1$ the oil light comes on. Since the component is no longer in state *running* this will have no effect and although in reality the car will need oil before it needs fuel, this will not be modelled. The only way to effectively solve this problem is to split the component into two individual components, one which models fuel required and another which models oil required and then the pre-conditions will not interrupt one another. This however is not very satisfactory when we are modelling a large system as it will result in a great number of components, all closely related but with their own behaviours. The modeller would have to break down the functionality and at the same time ensure all interaction is catered for. This is a philosophy of the language but any reduction in complexity is desirable.

These two problems are very significant and can cause the modeller significant problems. Models of real problems could grow to be quite complex and loose their simple connection to the real problem. Investigation showed that adding timing to transitions which are enabled by pre-conditions could solve both problems. This would appear like a simple step to take but it has some significant and interesting consequences.

We will consider again the previous car scenario now modelled by adding timing

```
ON_EVENT car.fuel_light {
        5 running -> getting_fuel ;
}
ON_EVENT car.oil_light {
        3 running -> getting_oil ;
}
```

We have now got rid of the dummy states and immediately reduced the state complexity. The question then raised is when in time are the transitions enabled ? If the fuel light comes on do we immediately suspend any further interaction with the component until in 5 time units it moves into state *getting_fuel* thereby maintaining the forced nature of the ON_EVENT or will the component move to *getting_fuel* in 5 time units with the possibility that another transition may affect it before then and hence an ON_EVENT is no longer a forced event ?

By choosing the second alternative the problem of pre-condition interaction can be avoided. Consider applying this timing principle to the above scenario. If at time $t_1 = 0$, the fuel light comes on we can say that at time, $T_1 = t_1 + 5$, the component will move into state *getting_fuel*. If at time, $t_2 = 1$, the oil light comes on we can say that the component will move into state *getting_oil* at time, $T_2 = t_2 + 3 = 4$. Since T_1 is less than T_2 the transition to *getting_fuel* will be cancelled and instead the transition to *getting_oil* will occur. Notice also, that if t_2 occurred at time 3 then $T_2 = 6$ so that now T_2 is less than T_1 and hence the transition to *getting_fuel* will occur before the transition to *getting_oil*. The transitions enabled by the pre-conditions can now interact effectively. They can interrupt one another but they do so whilst maintaining correct timing integrity. We are therefore now able to model scenarios like the one above by just using the one component. This facilitates a significant saving in the complexity and ease of designing models to represent real situations.

It is however now the case that transitions enabled by pre-conditions are not forced and will not necessarily interrupt current processing. This significant change must be understood but it is not a problem. If we still wish to provide forced events we can simply do so by using pre-conditions with transitions that have timing equal to 0 and are therefore immediate.

3.3.4 Hierarchical editing of models

When the language was first developed the only systems modelled were quite small test applications. When considering the modelling of real systems two things became apparent. Firstly, models could grow to a very significant size producing quite a large language description and secondly, there may be duplication of code within the same model. To tackle these problems it was decided to implement some sort of hierarchial editing.

In the C programming language hierarchial editing is enabled by the use of '*include*' files. These are files which contain listings of C code. Any file may incorporate these include files by using the command

#include<*file_name*>

This allows the programmer to build up a programme from a group of smaller programs, keeping all files to a manageable size and providing a means of reusing sections of code. The code is not immediately compiled but is first run through a *pre-compiler* which when it sees any include commands expands them by copying in a new source file, the file specified within the command. This is transparent to the user and will not change the original source file in any way.

Such a mechanism was deemed to by ideal for our purposes and it was decided to incorporate a similar command into the language. The syntax chosen is

#<file_name>

This command is a simple modification to the language and has no effect on the processing. It does however have very significant implications for the simulator for it has meant the writing of a pre-processor. This is described in chapter 4.

3.4 Conclusions

The complete syntax of the ICE language is given in appendix A. This chapter has presented some of the significant features and the philosophy behind them.

The objective during the development of ICE has been to achieve a compact and intuitive syntax that provides a powerful descriptive space. A vital concern is the user interface to the language. When a modeller takes a state space view of a system it is desirable that the language can describe the system so visualised with a minimum level of abstraction.

The development of COUNTERs which add descriptive power without a corresponding increase in the underlying state space have been discussed. The additional syntax required is minimal in comparison to the modelling capability they contribute. The language possesses an inherent simplicity that makes it favourable as a generic modelling tool. Any changes to the syntax, no matter how apparently simple they may appear, must be considered on a global context. It is paramount that the language's intuitive approach and favourable human interface is maintained.

Consideration of the timing and priority of conditional transitions demonstrated the complex semantics that are behind the language. Seemingly simple constructs can model a range of different types of component interaction and it is therefore essential that the modeller has a firm understanding of these semantics. The computational models of chapter 6 provide a rigorous definition.

Chapter 4

Implementation of the ICE Simulator

4.1 Overview

This chapter describes the simulator that facilitates the compilation and simulation of ICE files.

In this chapter we give a brief introduction to simulation styles and languages and show how these relate to ICE. We then go on to consider in some detail the development of I_SIM, which is an event-scheduling discrete event simulator with an ICE interface.

Files containing ICE code may be edited hierarchically. These files are integrated using a pre-compiler. The code is then parsed to produced data structures. These data structures are compiled into simulation data objects and checked for any syntactical errors. The C++ simulation objects interact under the control of the simulation algorithm. A timed log of the object interaction is kept throughout simulation and may then be interpreted to produce a text listing of all simulation events. The post processor analyses these events to produce statistical information specified by the user on a spreadsheet.

Section 4.2 gives some background to simulation and simulation languages, specifically

discrete event simulation. Section 4.3 considers the operation of the pre-compiler, parser and compiler in converting the ICE code into simulation data objects. Section 4.4 investigates the flow of control and the functions used by the simulation algorithm. Section 4.5 explains the development of the post processors which produce event listings and statistically analyse the simulation.

Figure 4.1 shows the distinct stages of the simulator and the data flow between each stage.

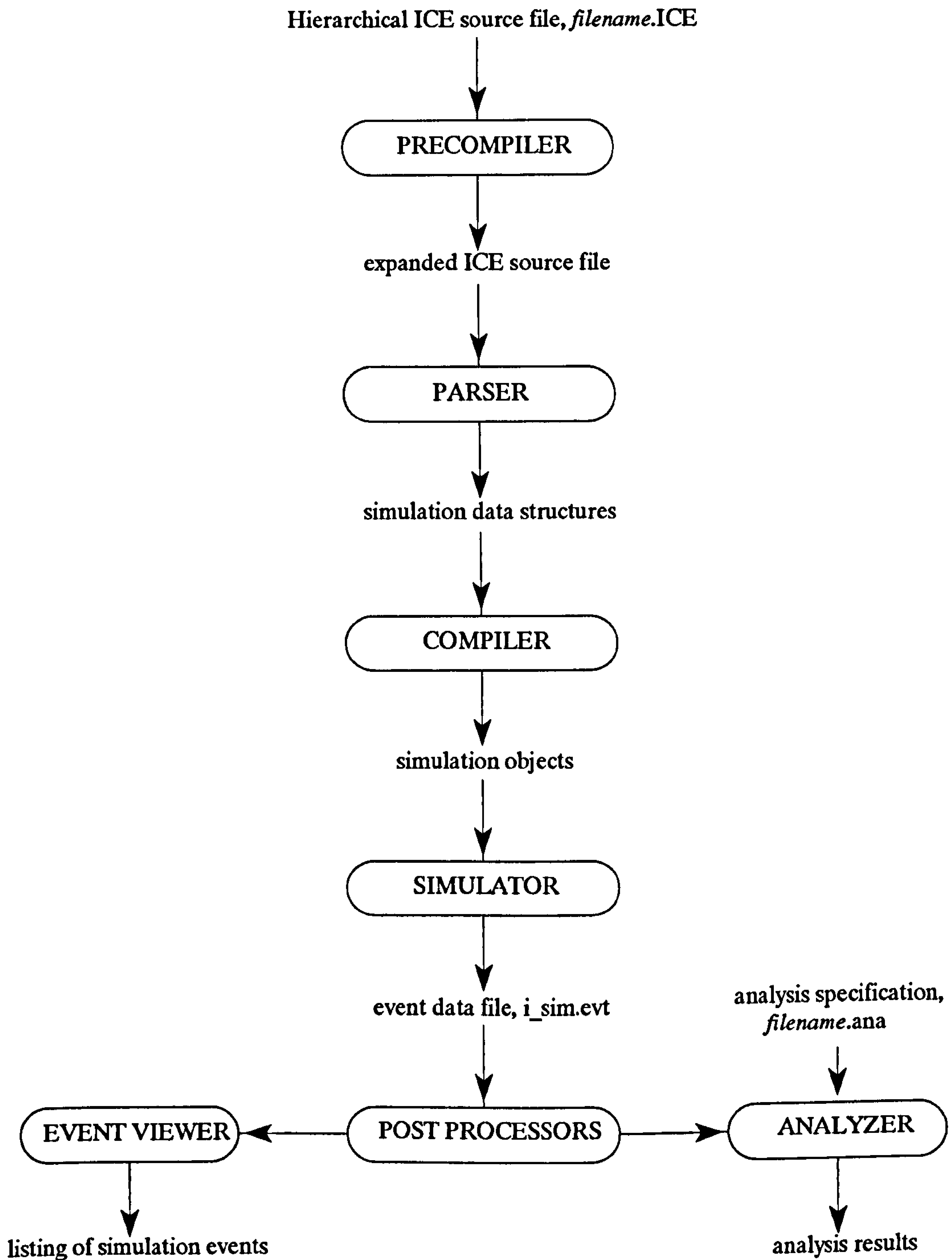


Figure 4.1 Overview of the Software

To provide a clear understanding of the operation of the software an example is given. A sample ICE program is listed in Appendix B. This is considered in parallel with the explanation of the software. We shall effectively consider the results of parsing, compiling, simulating and analysing this program. All data structures, objects and files created for this specific example are also given in Appendix B and will be referred to throughout the discussions.

4.1.1 I_SIM directory structure

A tree diagram of the I_SIM directory structure is shown in figure 4.2 along with a description of the type of files each directory contains.

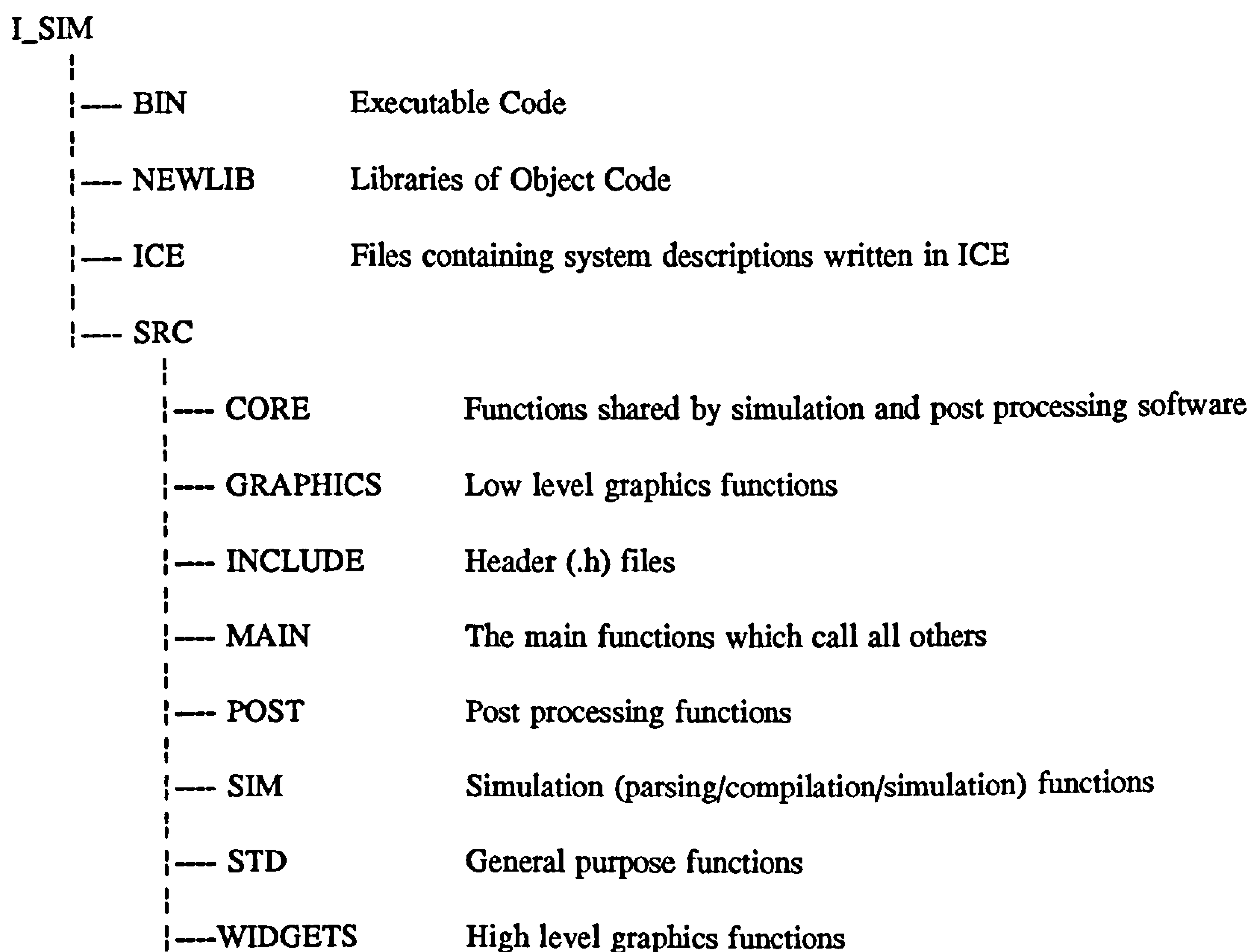


Figure 4.2 I_SIM Directory Structure

The total size of the combined source, object and executable code is about 4Mbytes. I_SIM is comprised of four executable files. *I_SIM* is the main graphical environment which provides the user interface. *ICE* runs the simulation software i.e. the pre-compilation, parsing, compilation and simulation of ICE code. *VIZ* runs the textual post

processor which gives a chronological listing of all events which occurred during simulation. *TPP* runs the statistical post processor. *ICE, VIZ* and *TPP* are all called from within *I_SIM* transparent to the user.

4.2 Background

In this section we give a brief overview of discrete event simulation.

4.2.1 Simulation styles and languages

There exist specific simulation languages. These are normally programming languages which have special constructs applicable to simulation. Often commercial languages are based upon the popular programming languages C, C++, FORTRAN and Pascal. These languages can be classified into three categories [81]

① Monte Carlo

② Discrete Event (asynchronous)

③ Quasi Continuous (synchronous)

Monte Carlo simulation may be defined as 'a scheme employing random numbers that is used for solving certain stochastic and deterministic problems where the passage of time plays no role' [82]. Monte Carlo simulators have features that allow random events to be generated internally and they are often used in quantum physics modelling.

Discrete event simulation is characterised by the passage of blocks of time during which nothing happens and is punctuated by events that change the state of the system. Emphasis is placed on these events which show the interaction of the modelled components. It is assumed that all important features of the system's behaviour may be modelled by these events. The blocks of time vary in length dependant upon the occurrence of events and therefore since the updating of the clock is not regular this type of simulation is also known as asynchronous. It is often used in the modelling of digital communications systems and in operations research.

Quasi Continuous simulation is concerned with modelling a set of equations that represent a system over time. The system may consist of algebraic, differential or difference equations whose solution continually vary with time. The simulation is 'quasi' continuous as the system time is updated by some fixed time interval Δt with the system parameters being re-evaluate within each time step. The length of Δt is a compromise between accuracy (sufficiently close to continuity) and computational overheads. It is suited to applications such as biological systems and computer aided design.

Owing to the nature of ICE we are interested in discrete event simulation. This can be further subdivided into three classes

① **Activity scanning** models a system by a set of activities which all have start and finish conditions. The simulator scans all the activities starting conditions to determine if they may be operated. The system clock is then advanced to the shortest of these activities finishing times. The consequences of the finishing conditions of all activities that will finish are then implemented.

② **Process orientation** models a system by the flow of constituent processes. These processes can communicate and interact with one another during simulation. They may also utilise various defined resources.

③ **Event scheduling** models system behaviour by a set of events which are stored time sequentially in an event list. The simulator advances the system clock to the first event in the list, determines any conditional events which are activated by the execution of this event and places them in the appropriate place in the event list. The process is then repeated by advancing the clock to the next event in the list.

These categories do not provide a strict framework but may be combined to provide the most suitable simulator for a given language.

4.2.1.1 Examples of specific simulation languages

Here we briefly consider a few modern simulation languages to give an idea of the variety available.

General Purpose Simulation System (GPSS) [83] is a discrete process interaction language.

It is most often used to model systems that consist of customer entities which compete for limited resources.

Simulation Language for Alternative Modelling (SLAM) [84] is a FORTRAN based language that was the first to allow modellers to approach a system with a discrete, continuous or combined view. It is powerful due to it's flexibility and FORTRAN subroutines may be called from within the language and run during simulation.

Simscript [85] is a FORTRAN based simulator that is not dependant upon a FORTRAN compiler as it translates code into assembly language. It has constructs which support a discrete event view with event scheduling and process orientation

SIMAN [86] like SLAM allows discrete event process oriented, event scheduled and continuous components integrated into a single system model. A SIMAN model typically consists of model code and a series of statements which are a framework for describing experiments.

Other languages include DYNAMO which is FORTRAN based and facilitates continuous simulation and SIMULA which is a PL1 based discrete event language. For a fuller description of these and a comparison between simulation languages the user is referred to Kreutzer [87].

All the languages mentioned are general purpose and suitable for the modelling of a wide variety of systems. The nature of ICE however is quite specific and prescriptive of the type of simulation required. The finite discrete state modelling of components suggests a discrete event style and the interactive nature implies an event scheduled approach. Given this we shall now consider the features of such a simulator.

4.2.2 Features of a discrete event simulator

Several generic features are important for all discrete event simulators. These have been extensively investigated and comprehensive lists of requirements exist e.g. [88]. The main features we need to consider are

① A system clock for advancing simulation time to order the events. In the instance of an event scheduled simulator this should be a global variable, advanced to the next event once all the events of the current time have been activated and their consequences implemented.

② General frameworks for model creation and editing. This is obviously the role of the simulation language. A compiler is required to convert models described by the language into suitable representation which may then be manipulated by the simulator. The I_SIM precompiler and simulator is detailed in section 4.2.

③ A method to schedule the occurrence of events. This requires two components, the scheduling algorithm and the event list. The scheduling algorithm is the main controlling function during simulation and dictates the simulators behaviour. The event list is the time ordered list of events, normally a doubly linked list for ease of insertion and deletion of events during simulation. The I_SIM scheduler is described in section 4.3.

④ Tools to aid in the collection, analysis and reporting of the behaviour of the modelled system during simulation. I_SIM has two analysis tools described in section 4.4.

⑤ A random number generator which must produce a uniform and statistically independent series of random numbers over a variable length of runs. These should include tools for mapping the numbers into applicable distributions. The generation of random numbers is a large field in itself and one that is considered in some detail in chapter 5.

4.2.3 The development environment

I_SIM was developed on a 486DX2 PC running a DOS environment. It is written in C++ which supports object oriented programming (OOP).

This style of programming allows the programmer to model components found in the problem domain as a set of abstract data types or *objects* which interact by parameter passing. Objects may be created that inherit properties from other objects thus allowing proven code to be reused for different applications. An excellent overview of OOP is given

in [89].

OOP is very suited to discrete event simulation. Its emphasis on communicating objects matches well a modellers view of a system as a set of interacting components. The first OOP language Simula67 [90] was written especially for simulation applications. From the ideas first proposed in Simula67, the two currently popular OOP languages Smalltalk [91] and C++ [92] were developed. Smalltalk was entirely based on the OOP concept whereas C++, was developed by Stroustrup [89], as a superclass of the language C. Due to the already established popularity of C, C++ is emerging as the dominant OOP language.

I_SIM utilises many of the benefits of C++, especially inheritance and data hiding. Simulation elements are represented as objects which simplifies the design and understanding. Similar objects are constructed by inheriting common features of a base class. Data hiding allows data within objects to be categorised so that it may only be altered by functions which are given specific access.

Development of I_SIM was both helped and hindered by the desire to re-use much of the code of the earlier Tecsim simulator. This saved rewriting many proven routines but it also restricted some of the data structuring.

4.3 Compilation of the language

The file *ICE* (MAIN) contains the main function which calls the compilation functions.

4.3.1 The precompiler

The precompiler was designed to facilitate hierarchical editing which allows files of ICE code to be kept to an easily manageable size and also makes it easier to re-use sections of code. The flow of control of *file_merge* (MAIN/ICE) the main function of the precompiler is best illustrated by considering the flowchart of figure 4.3.

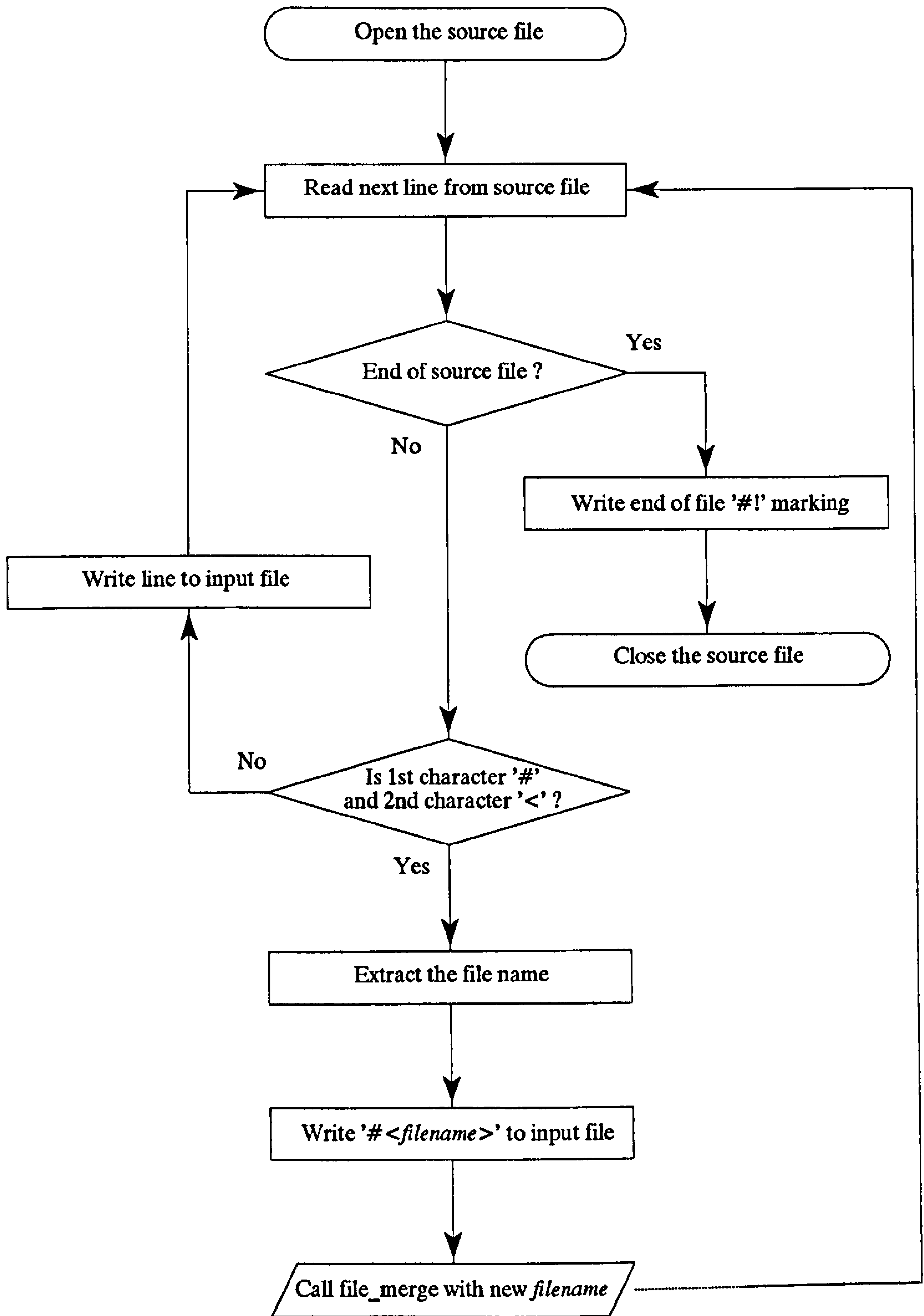


Figure 4.3 Flow of control of *file_merge*

As an example, given the three files, *file_1*, *file_2* and *file_3* with contents

FILE_1	FILE_2	FILE_3
LINE 1	LINE 4	LINE 6
LINE 2	#<FILE_3>	LINE 7
#<FILE_2>	LINE 5	
LINE 3		
END		

If *file_1* was passed to the pre-compiler the resulting input file would be *file_1.new* with contents

FILE_1.NEW

```
#<FILE_1>
LINE 1
LINE 2
#<FILE_2>
LINE 4
#<FILE_3>
LINE 6
LINE 7
#!
LINE 5
#!
LINE 3
END
#!
```

At first this file would seem rather verbose. Why write the individual files names and use end of file characters in the new file when what we are aiming to achieve is an expanded file containing all the ICE code ? The reason has nothing to do with the meaning of the code but is solely used for error reporting. When the new ICE input file is passed to the parser, all the lines beginning with a '#' will be ignored and thus have no effect on the processing. However if there is a syntactical or logical error in the code we want to be able to locate which file and on which line in this file the error occurred.

4.3.1.1 Error reporting

Prior to the introduction of the pre-compiler, error reporting was a simple task. For every line in the ICE file which was converted into a data structure for compilation, the line number was stored. If any errors were detected during compilation into simulation objects, they were reported with this line number being given and of course there was only the one

source file. The line number that is now stored is that in the resulting *.new* source file. This is meaningless to the user as this file is created transparently. We are now required to translate this line number into the line number in the appropriate source file. This is done by creating a stack of file elements. Each element takes the form of a structure shown in table 4.1.

char	sfname[13];	// the name of the source file
int	line;	// count of number of line read from file
struct stack	*next;	// points to next stack element

Table 4.1 Structure of error reporting stack elements

When an error is detected a stack element is created for the first source file. The lines of the *.new* file are read sequentially and the *line* field of the stack element is incremented for each line read. When a '#<file_name>' line is encountered a new element is pushed onto the top of the stack for this new file *file_name.* Now as the lines are sequentially read the *line* field of the new stack element is incremented. This process is repeated each time a '#<file_line>' is encountered. Whenever a '#!' line is encountered the top element is popped off the stack and the reading of lines continues using the element which is again on top of the stack. When the total number of lines read is equal to the line number of the error then the top stack element will contain the name of the original source file and line number.

4.3.2 The parser

This section describes how the ICE source code is converted into data structures which are entered into a symbol table. This parsing phase uses two tools, one a hand written lexical analyzer based on the Unix tool Lex and secondly a Dos version of the Unix compiler writing utility Yacc. It is beyond the scope of this text to give a detailed description of Lex and Yacc and the reader is therefore referred to the excellent text by Levine *et al* [88].

Both Lex and Yacc are program generators which take a high level lexical or syntax description and generate C[92] or C++[89] programs. The lexical analyzer function *yylex* (src\lex.cpp) recognises character patterns in an input file and converts them into a stream of *tokens.* The set of character patterns is written to suit the application and is known as

the *lex specification*. Tokens may take integer values or represent language features such as integers, ICE keywords or proper names. Associated with each token are variables which may represent the actual value of an integer or proper name.

The I_SIM compiler identifies the declarations, expressions, statements and blocks in an ICE program. This task is known as *parsing* and the list of rules that define the relationships that the program understands is a *grammar*. Yacc (Yet Another Compiler Compiler) generates a parser which generates a C++ function *yyparse* (sim\ytab.cpp). The grammar is a declarative style syntax description describing the language in terms of the tokens defined in *yylex*. Linked with each element of the grammar are possible *actions*, which are portions of C code that define and build the data structures used to hold all required information about the ICE code.

The parser also provides syntax error detection by noting mismatches between the tokens produced by *yylex* and the given grammar. The error reporting is handled as described in section 4.3.1.1. Syntax errors produce an appropriate error message and the parser will then attempt to recover so that further syntax errors are noted.

The data structures produced take the form of linked lists of C structures, one list for each type of ICE statement. The main structure is the symbol table entry (symtabentry). There are symbol table entries for each STATE_SET, BEHAVIOUR, COMPONENT, SYSTEM, STOCK and WAIT_FOR statement. Further types of structures are used for the component parts of these statements such as transitions, state expressions, lists of states, lists of resources etc and are linked via pointers to the symbol table entries to produce a complete description of the ICE code. These collective data structures, all accessed by the single array *symlist* of pointers to each linked list, provide the link to the compilation phase.

An example of an ICE program and a description of the system it models is given in Appendix B section B.1. Section B.2 goes on to show the data structures produced when this program is parsed.

4.3.3 The compiler

In ICE the order of statement declarations is unimportant. This means that items may not be defined until after they are used and thus ICE code cannot be fully checked for errors during the parsing phase, e.g. a STATE_SET may be defined after a BEHAVIOUR statement and hence during parsing it is not possible to check the state names given in the transitions. This allows code to be entered in whatever order the modeller wished, it is however recommended that a structured design procedure is adopted. The majority of error checking is done in the compilation phase. The related information in the symbol table entry linked lists is cross checked and converted into a set of self-consistent C++ simulation objects.

4.3.3.1 Simulation objects

There are C++ simulation objects for all active parts of the ICE description. By active we mean those parts such as component states, counter values and resource levels etc which may change state or value during simulation. All objects are derived classes from the base class *Object*. Table 4.2 lists the different objects and gives a summary of their functions.

Base Class	Derived Class	Function
Object	Component	Component description including pointers to behaviour and state information.
	Sexp	State and Counter Expressions.
	Resource	Resource name and current quantity.
	Waitfor	Interrupt which causes change to a component or resource description.

Table 4.2 Simulation object classes

4.3.3.2 General compiler operation

At the start of this phase an array *object* of pointers to simulation objects is created with enough elements for the number of objects required. A simulation object is created for

each *symtabenrty* (discussed in section 4.3.2) created during parsing, except for STATE_SETs and BEHAVIOUR statements which are handled differently, as will be discussed. The order of conversion is as follows.

Function *state_set_check* (sim\tables.cpp) checks that each state and counter name is unique, and gives each a corresponding number. These numbers are entered into the state set *symtabentry* and the names are entered in a hash table for quick lookup in checking further names.

Function *resource_check* (sim\tables.cpp) creates a separate resource object for each resource and stock structure.

Objects are created for each component and state expression or system statement, then function *component_check* (sim\compile.cpp) checks and converts the component *symtabentry* into an object. An array is created to hold all the components counters and their initial values are stored. All counter function expressions for these counters are added to the component. A statespace object is created for each distinct behaviour and state_set *symtabentry*. The initial state of the component is set.

System *symtabentries* and the anonymous system *symtabentries* created for IF, ON_EVENT and counter expressions are checked. Anonymous denotes that the expression is not an explicit component but a conditional clause which may be related to any component. The format of the expressions are checked for validity. A permanent consequence graph link is made from the state expression to the parent object. Section 4.3.3.4 describes the function of consequence graphs. The expression is evaluated and the initial value entered into the object.

Waitfor *symtabentries* are checked and converted into objects.

Thus in summary, during the compilation phase, the component, resource and system *symtabentries* are converted into simulation objects, the state_set and behaviour *symtabentries* are converted into StateSpace objects and system objects are linked to their parent objects by consequence graphs.

In appendix B, section B.3 all the objects created during the compilation of the

symtabentries listed in section B.2 are given.

4.3.3.3 The StateSpace

Information about which state a component can exist in and the transitions between states is stored in a *StateSpace* object.

A StateSpace object is created for each behaviour symtabentry. It contains an array of pointers to State objects. One State object is created for each state in the STATE_SET. Each transition in the BEHAVIOUR statement is converted into a transition node by the parser (refer section B.2.1). Transition nodes store the names of the states the transition is from and to. The State objects for each state reference each transition node who's *from_state* is this state. Also during parsing, any counter expressions associated with a particular state are referenced by the statelist structure created for the state. This reference is copied into the corresponding State object during compilation.

The StateSpace objects created for the behaviour and state_set symtabentries of section B.2.1 are shown in section B.3.

4.3.3.4 Consequence graphs

A consequence graph is the method by which objects that interact during simulation are linked together. Consider the conditional statement

```
IF   comp1.active {
       4    comp2.first -> comp2.second ;
}
```

At compile time a permanent link is set up between the component *comp1* and the state *active*. The *comp1* end of the link is known as the parent and the *active* end the child. During simulation, when *comp1* is not in state *active* this is the only link. If *comp1* enters state *active* a further *dynamic* link is established between the state *active* and the component *comp2*. In this link, *active* is the parent and *comp2* the child. When component

comp1 moves out of state *active* this link is removed. This *dynamic* link will enable the transition from state *first* to state *second* if *comp2* enters state *first.* These consequence graph links provide the current state of component interaction during simulation.

The simulation objects all have the class *Object* as their base class. This is a composite object as it has another class as one of its members. This class forms part of *Objects* private data and is an instance of the class *CqGraph.* CqGraph is used in its basic form and it also forms the base class of three other types of consequence graph. These are listed along with their prospective uses in table 4.3.

Base Class	Derived Class	Use
CqGraph	OnNode	Links objects related by ON_EVENT expressions.
	ResrsNode	Used to link component objects to resource objects when resources are required for a change of state.
	ForceNode	Links an object to another object which is forcing it to change state.

Table 4.3 Consequence Graph types

The CqGraph node has four pointers. These are used to point to the previous and next nodes in its parent link and to the previous and next nodes in its children's links. It also contains the object numbers of its parent and child thus allowing any object to be doubly linked to any other object via these nodes.

Consequence graphs may be used to connect objects for the various purposes listed in table 4.3. The way in which they work is the same in each instance.

Their operation is best understood by the use of an example. A detailed example using of the consequence graphs created for one of the conditional statements of the program in section B.1 is given in section B.3.1.

4.4 Simulation phase

The simulation software uses as its input the set of C++ objects that have been built during the parsing and compiling stages and models the interaction of these objects under the control of a simulation algorithm. The overall flow of control of the simulator is shown in figure 4.4.

Before commencing the simulation the stop time (which must be given in the ICE code) is recorded and the current time of simulation is set to 0, to indicate the start of a new simulation.

A new Event Manager is created. This contains all the control information for the calender queue of events. It has an array of pointers which form a linked list of the fundamental simulation object type, Event. An Event object is created for every scheduled event (state change) and is deleted once this event has occurred or has been surpassed by the occurrence of another event. This linked list forms a calender queue of simulation events.

The simulation objects are initialised in two stages, the initialisation being dependant upon the type of object. All objects contain a field *state* which holds the state of an object during simulation. All these fields are set to the initial state. For components this will be the initial state the component resides in and for state expressions it will be the initial condition of the corresponding SYSTEM statement i.e. true or false.

For components, any attributes of the current state are entered into the component object. Any events which are a consequence of this component being in the current state are processed and if any IF state expressions for transitions from the current state are true the component changes state. The next timed event is determined and a new Event object is created for it and entered into the calendar queue. The state that this component will move into upon the occurrence of this event is entered into the component object as the *next_state*. If there are any ON_EVENT or ON_RESOURCE transitions from the current state then an *OnNode* consequence graph is created to link this component as a child of the anonymous system expression object of the ON_EVENT or ON_RESOURCE statement. If the resources listed in an ON_RESOURCE statement are free then the resource transition is activated.

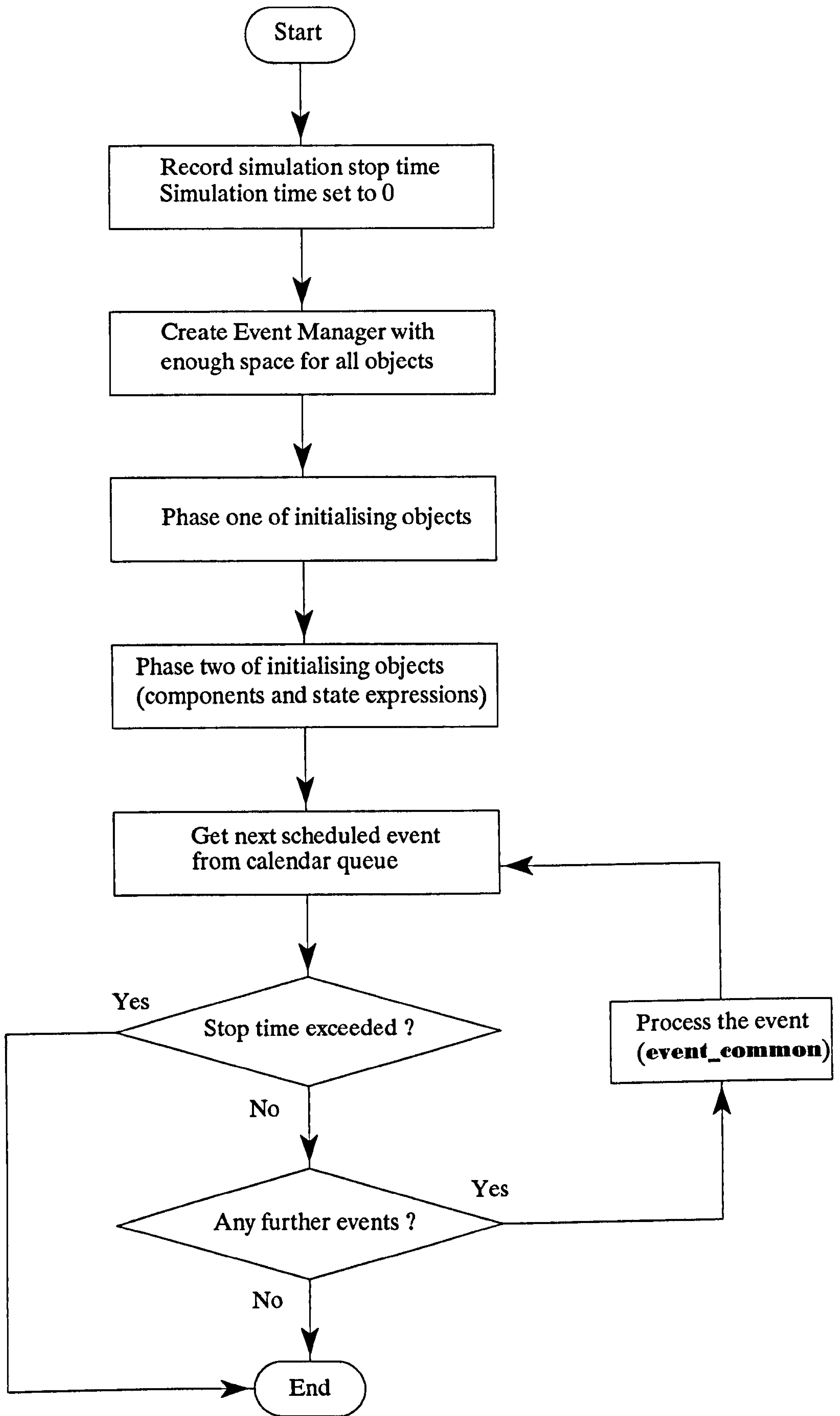


Figure 4.4 Flow of control of simulation phase

Once all objects have been initialised and any state changes that this instigates processed, the first scheduled event is retrieved from the calendar queue. Providing the occurrence time of this event is not greater then the stop time it is processed and consequences of this event implemented. The next scheduled event is then retrieved and so the process continues until the stop time is reached.

4.4.1 The event processing cycle

Scheduled events relate to component state changes. The function Component::event _common() forms the heart of the simulator as it coordinates the processing of every event i.e. every state change. It calls other functions, come of which effect the processing of events are listed in table 4.4.

Name	Description
event_common()	Main routine that processes the event calendar queue.
applyfns()	Modifies counter values by applying counter functions linked to the current state.
triggerChildren()	Activates any child consequence graph nodes that are enabled in the current state. These are links to IF, ON_EVENT and ON_RESOURCE statements in other components.
nextIFstate()	Selects the next enabled IF transition from the current state.
nextTstate()	Selects the next unconditional timed transition from the current state.
AddCnodes()	Creates dynamic consequence graphs and links for any ON_EVENT, ON_RESOURCE or SYSTEM statements that are enabled in the current state.

Table 4.4 Key simulation functions

event_common()

The flow chart of figure 4.5 shows the flow of control of event_common().

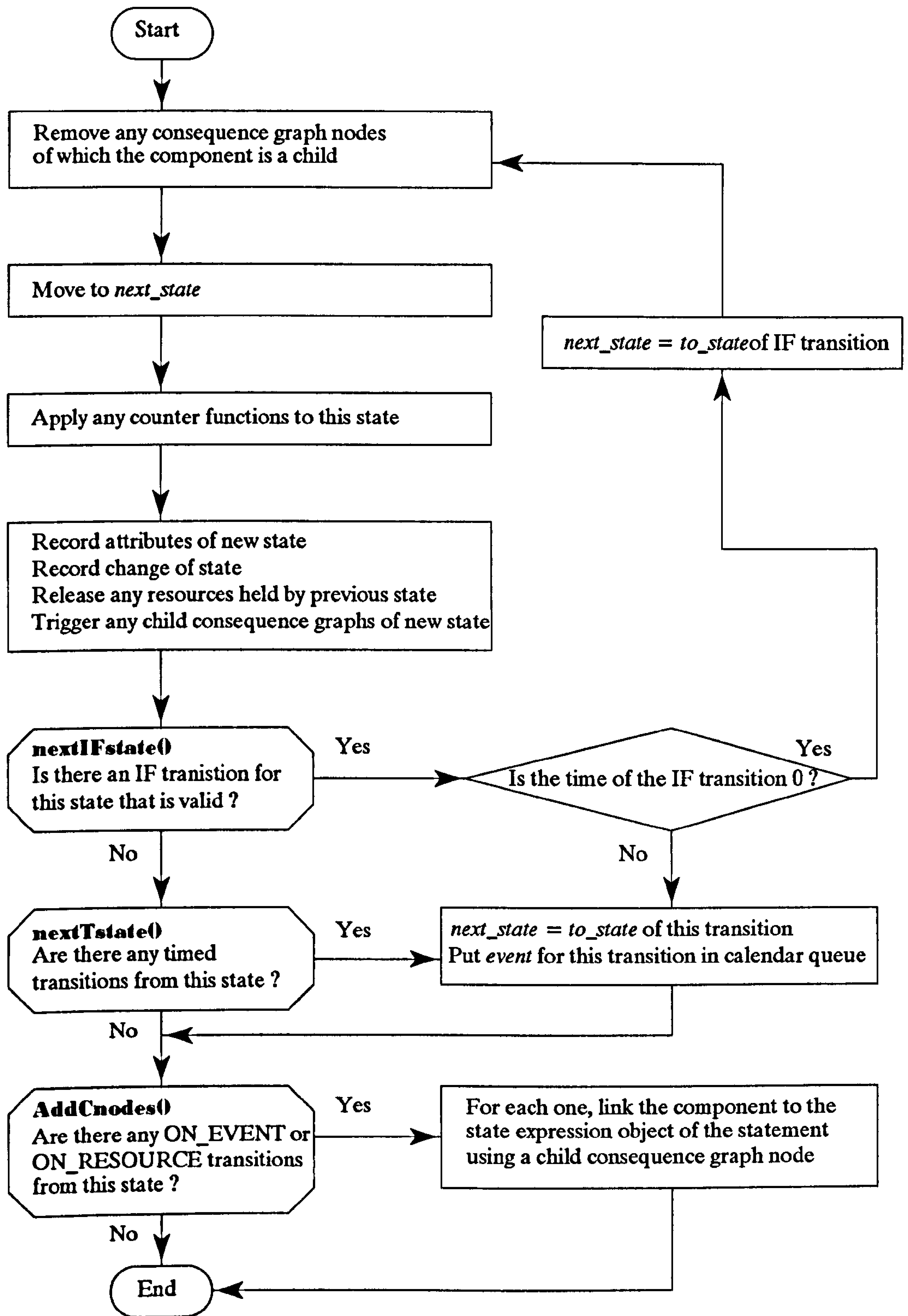


Figure 4.5 Flow of control of event_common()

The first action is to remove any consequence graph modes of which this object is a child. These links were created when the component moved into the current state and link the component to any ON_EVENT or ON_RESOURCE statements which could move the component out of this state. As the component moves into the next state they no longer

can effect the components operation, unless they apply to the new state as well, in which case they will be added again at the end of the processing for this event.

The component changes state into the next state and adopts the attributes of this state. If there are any functions attached to this state that manipulate any counters then the function *applyfns()* calculates the new counter values and enters them into the corresponding components counter list.

The new state and any change in counter values are recorded in the event date file. This file is described in section 4.4.2.

Any resources the last state held are released and any resources required by the new state are held. The free resource levels updated accordingly. The changes in resource levels are recorded in the event date file. When resources are released any other ON_RESOURCE statements now enabled are activated.

If the component has any consequence graph links to children, e.g. for system statements that are enabled when the component is in the new sate they are activated by the function *triggerChildren( )* and the events generated processed.

Function *nextIFstate()* determines if there are any IF transitions from the new state which are enabled then the timing of the transition is determined. If the transition is immediate then the new event is generated and processed. If it has a time delay the next state is set and this new event is entered in the calendar queue.

If there are no timed IF transitions then function *nextTstate* determines whether there are any timed transitions from this state. In the occurrence of there being more than one, the one with the shortest transition time is selected. The *to_state* of this transition is set as the next state and a new event generated and entered in the calendar queue.

Function AddCnodes() determines if there are any ON_EVENT or ON_RESOURCE transitions from the new state a child consequence graph node is created for each to link the component to the related state expression.

Applyfns()

This function ensures that all counter values are updated as required. Counter values are calculated by functions which are associated with component states. Applyfns() applies on functions which are related to the current state and updates the associated counters.

triggerChildren()

This function checks any consequence graph nodes of this component. These nodes will be for SYSTEM or ON_EVENT statements which are dependant upon this component. If any of these statements are enabled by the component moving into this state an event is created for the associated transition and placed in the calendar event queue.

nextIFstate()

This function reads through all the possible transitions from the current state and selects only those which are conditionally governed by an IF statement. If a transition is found which has a zero transition time i.e. an immediate forced transition, the one that is listed first in the source code is selected.

If there are no immediate IF transitions, the timed IF transitions are considered. The random number generator is used in determining the transition time when the timing is given in terms of a time distribution. The transition with the shortest time is selected. Again if there is more than one transition with the same timing then that listed first is selected.

It is possible for more than one transition to be governed by the source IF statement. In this case each transition will have a probability of occurrence. nextIFstate() uses the random number generator to give a random selection of one of the possible transitions.

nextTstate()

This function selects all non conditional timed transitions from the current state. If the transitions are probabilistic then the random number generator is called to randomly select

one. In the instance where there is more than one timed transition the one with the shortest time is selected. The random number generator is called to determine times for transitions with time distributions. IF more than one transition has the same time, the transition listed first in the source code is selected.

AddCnodes()

The function AddCnodes checks if there are any ON_EVENT or ON_RESOURCE statements which can enable transitions from the current state. For each conditional transition a dynamic control consequence graph node is created to link the component object to the ON_EVENT or ON_RESOURCE statement state expression.

If the ON_EVENT or ON_RESOURCE enables a number of transitions from the current state, each transition will have an associated probability. AddCnodes() calls the random number generator to select which transition will be used.

If enough free resources exist to enable any ON_RESOURCE transitions an object is created for this transition and placed in the event calendar queue.

4.4.2 The event data file

The event data file, I_SIM.EVT, forms the interface between the simulator and the post processors.

The file is comprised of two main parts. The first section is a description of the simulation objects, written as they are created. This is used by the analyzer to construct analysis objects and as a means of validating component, state, counter and system names in the analysis specification. The second section is written during simulation and is a record of every simulation event. When each event occurs the current simulation time, component or system object number, state number and if appropriate counter number and value is written to the file.

The start of the event data file that is written for the simulation objects in section B.3 and a few sample lines of simulation is listed in section B.4.

4.5 Post processing

The event data file is the interface between the simulator and the post processors. There are two post processors; VIZ, the visual post processor which provides a textual listing of all simulation events and TPP the analyzer which performs statistical analysis on the simulation data.

4.5.1 VIZ the visual post processor

A fundamental requirement of the post processor is to be able to view all of the simulation events in the order in which they occurred. This is critical when analysing a models behaviour. All events are listed in chronological order in the event data file. To provide a listing of these events the relevant data must be retrieved, interpreted and displayed.

The initial facility sequentially read the event data file, interpreted all transition data and created a textual file of events. Experimentation with the software found this method to be impractical for a few reasons. The text file may be very large which is a waste of disc space when often we just wish to view part of the simulation. Secondly, to be viewed from within the software it was necessary to incorporate the use of an editor. This is wasteful of resources as we only require to view the file and not to modify it. For large files that cannot be handled in their whole by an editor it was necessary to swap in and out sections of the file as required. This is untidy and annoying to the user.

The solution that was decided upon was to interpret and display one screen of event data at a time as required. This alleviates the need to create a text file of the entire simulation which saves space and means an editor is no longer required thus solving all the aforementioned problems.

Operation of the event viewer

To provide access to the events in the event data file, the file is sequentially read once. Whenever a transition or counter modification is detected an index is constructed to the corresponding line of the file as shown in figure 4.6. A binary file of the complete set of

indices is written to disc. It is never known how many indices may be required so they are created dynamically and added to a doubly linked list. The first list element is an index to the first event in the event data file and the indices are chronologically ordered from there. The list is doubly linked so that it may be read in either direction.

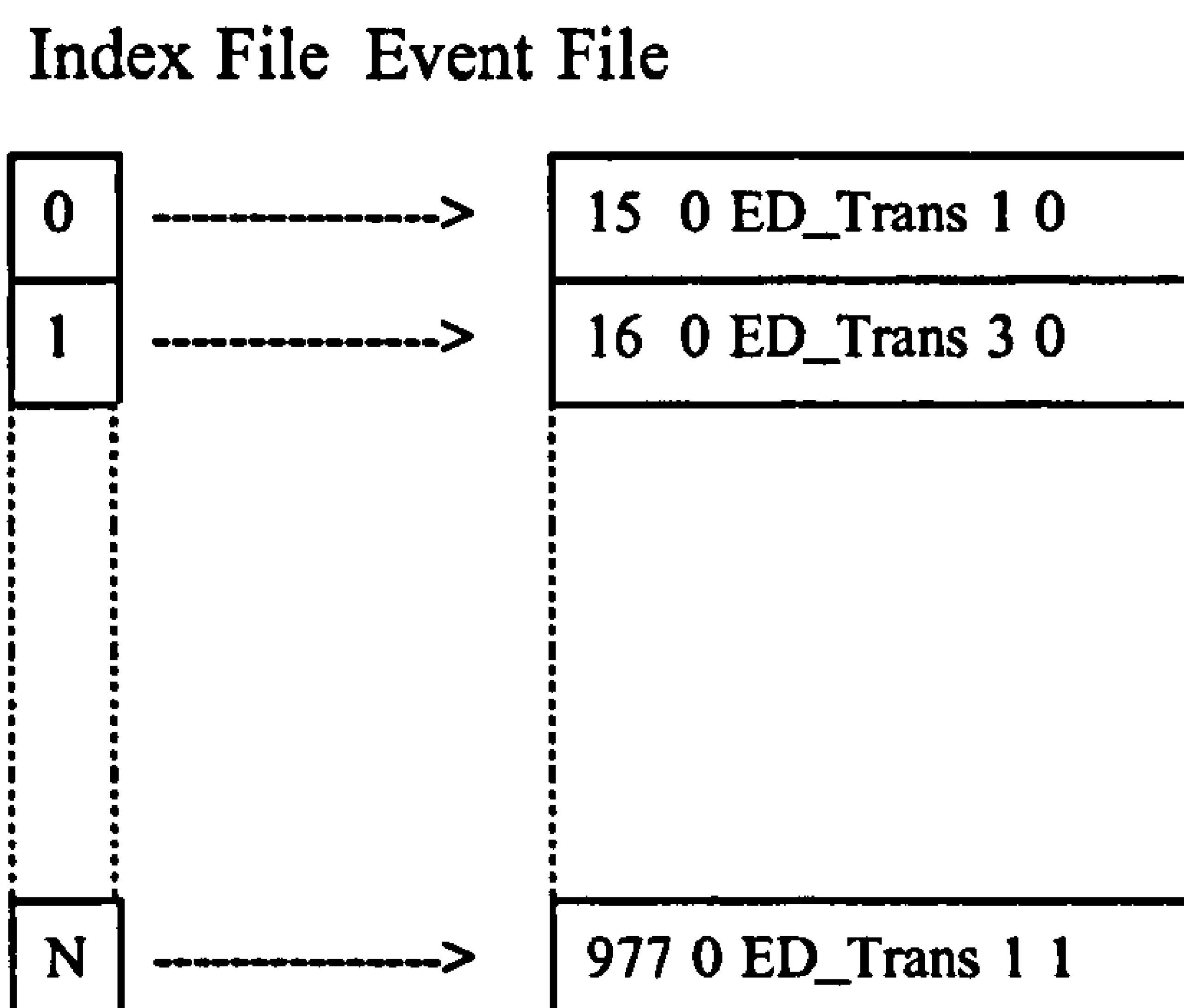


Figure 4.6 Index for accessing the transitions in the event data file

The events are displayed in a screen window. The window has a fixed number of rows, *n*. Only enough events are interpreted at a time to fill one window. When the window is first opened a pointer is set to the start of the index file. The first *n* indices in the index file are used to read the corresponding *n* lines of the event data file. The pointer is now placed at *n-1* index in the index file. The events listed in these lines are interpreted and displayed in the window as shown in figure 4.7.

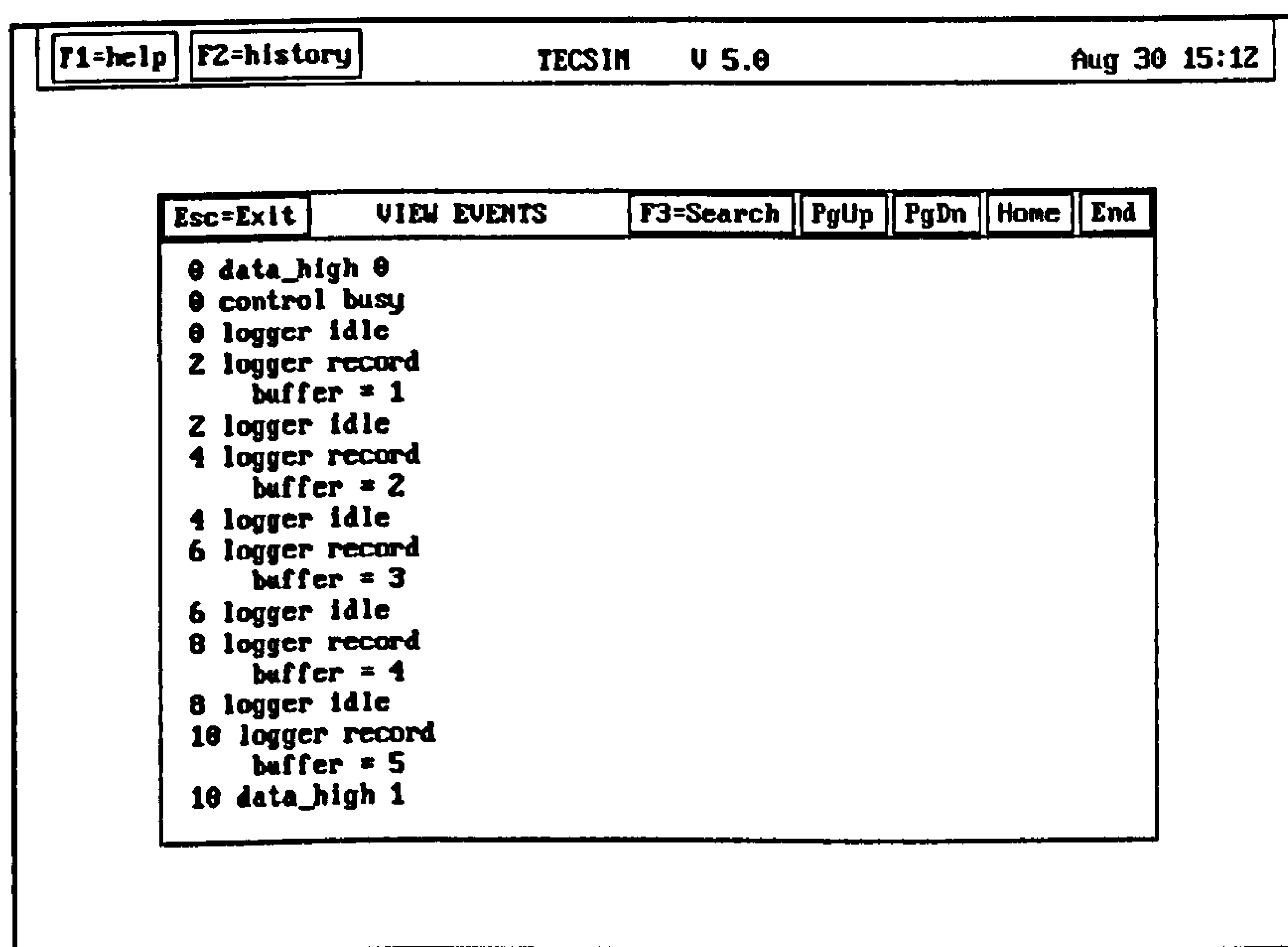


Figure 4.7 The event viewer screen

Commands used in the event viewer

As figure 4.7 shows, a number of user commands may be utilised in the event viewer.

Search enables the user to enter a string of search conditions. The index pointer is reset to the start of the index file. The pointer is incremented through the indices and each reference in the event data file is examined to see if it matches the search conditions. When an event is found the search stops. That event which matched the search conditions at the top of the screen and the other n-1 lines are the events which chronologically follow.

PgUp reads the previous n indices from where the index pointer is currently located and displays the corresponding n events.

PgDn reads the next n indices from where the index pointer is currently located and displays the corresponding n events.

Home moves the index pointer to the top of the index file and reads the first n indices. These are used to locate the corresponding first n events which are then converted and displayed.

End moves the index pointer to the last -n index i.e. the index file. The last n indices may then be read and used to locate the corresponding last n events which are then interpreted and displayed.

The above listed functions provide an easy and complete method of viewing all or selected sections of events.

4.5.2 TPP the statistical analyzer

When we model systems using the language and simulator we will often wish to perform analysis upon the resulting list of events to gain greater understanding of the system. To this end the statistical analyzer was developed.

When considering the development of such a tool two requirements were identified:

i) A means by which the user can select the required statistics

ii) Given this input, a method of interpreting the event data file to provide the required statistics.

The analysis editor

The initial idea for allowing the user to select required statistics was by the use of an analysis language. This language would describe which statistics were required for which component states, counters and system expressions. This method was cumbersome and presented challenges to the user. A parser was written to interpret the language and convert it into useful data structures. This was reasonably efficient but the main problem was that the user had to learn an extra language. A prime object has been to keep the software as simple to operate as possible and it was felt that this extra analysis language was an unnecessary complexity. It was therefore decided to design and implement a simple graphical interface resembling a spreadsheet to allow the analysis data to be entered.

To give a complete understanding of this tool we shall consider both the users perspective of entering requirements and viewing results and also how this 'specification' is stored for input into the analyzer.

The users perspective

The analysis editor that the user is provided with is shown in figure 4.8. The screen has three sections, a key prompt at the top, a 7 x 12 editing grid and an editing window at the bottom. The editing grid provides a suitably formatted layout for the requirements to be entered and displayed. Each grid location is known as a slot. The editing window is used to edit text to be input to the individual slots.

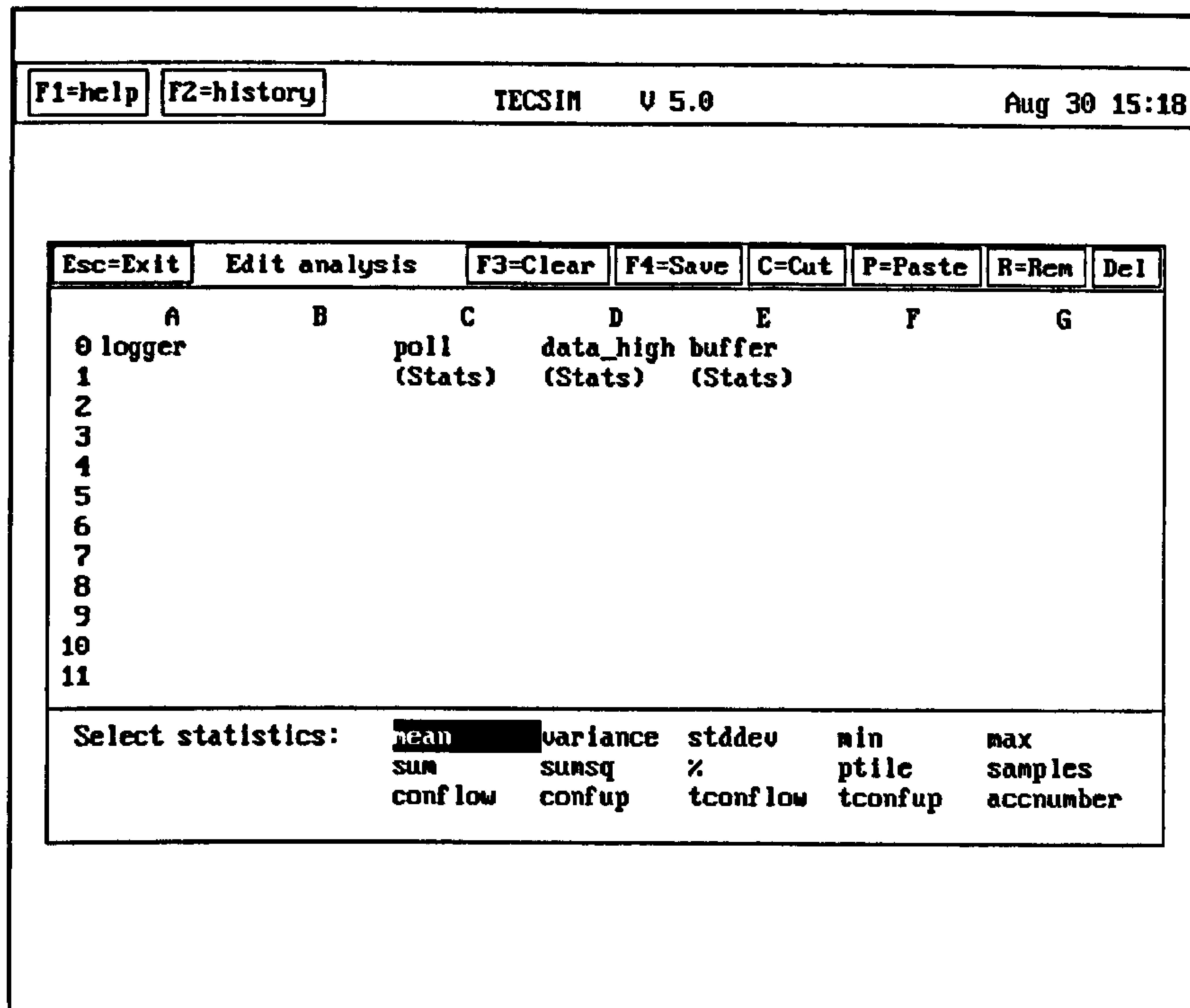


Figure 4.8 The Analysis Editor Screen

The key prompts are to remind the user of the various editing commands. These functions are as follows.

F3 = clear Clears the entire editing grid

F4 = save Saves the grid contents to a filename and analysis file. The user will be prompted for a filename and appropriate comment. This comment is displayed when the user selects an analysis file for editing as a reminder of the contents.

C = Cut Removes the contents of the slot the cursor is currently in and stores them in an editing buffer.

P = Paste Pastes the contents of the editing buffer into the slot the cursor is currently in.

D = Del Removes the contents of the slot the cursor is currently in but does not store them in the editing buffer.

The slot the cursor is currently in will have its colours inverted. The selection of the slot is altered by using the cursor control arrow keys.

When editing the grid any slot may have one of three content types. These are:-

empty : When nothing is in a slot it is denoted empty and has no effect on the analyzer.

text : This is user entered text. It serves only as a comment of what statistical information is displayed and has no effect on the analyzer.

stats : These are user defined statistics. When a statistical specification is entered in a slot, the slot will just display the text "[stats]" during editing. On displaying results the statistical information is displayed in this slot. The user should use a neighbouring slot for text as a reminder of what is displayed.

To insert text in a chosen slot the user presses *<enter>* and to insert stats *<ins>*. All editing of text and stats is done in the editing window not on the grid.

When statistics are required a menu of those available is displayed. Statistical information is available on component states, system statement states and counters.

The statistics available and their meaning for component states and system statement states is identical but varies for counters. Details of the statistics are given in tables 4.5 and 4.6.

Stats	Meaning
mean	The mean length of time the state is occupied.
variance	The variance from the mean value.
stddev	The standard deviation from the mean value.
min	The minimum length of time the state is occupied.
max	The maximum length of time the state is occupied.
sum	The total time spent in the state during the simulation run.
sumSq	The sum of the squares of the individual times spent in the state.
%	The percentage of the total simulation time spent in the state.
ptile	The percentile value of the state.
samples	The number of individual times the state was entered.
conflow	The lower confidence interval for a normal distribution.
confup	The upper confidence interval for a normal distribution.
tconflow	The lower confidence interval for a t type distribution.
tconfup	The upper confidence interval for a t type distribution.
accnumber	Prompts the analyzer to create a file which contains every sample recorded for this state. Each sample is the individual time spent in the state. This file may be used for external analysis and graphing of results.

Table 4.5 Statistics for component states and system statement states.

Statistics	Meaning
mean	The mean counter value during simulation.
variance	The variance form the mean.
stddev	The standard deviation from the mean value.
min	The minimum counter value.
max	The maximum counter value.
sum	The sum of the product of each counter value and the duration for which it held that value.
sumSq	the sum of the product of the square of each counter value and the duration for which it held that value.
samples	The number of times the counter changed value.
accnumber	Prompts the analyzer to create a file which contains every sample recorded for the counter. A sample is the counter value and the duration for which the counter held this value. This file may be used by externally for further analysis and the graphing of results.

Table 4.6 Statistics for counters

It is possible to determine statistics on statistics when more than one run of a simulation is done. For example we may run a simulation ten times and be interested in the percentage value of a certain state occupancy over the complete number of runs. In this instance we would denote the *StatsOn* to be *percentage*, which would denote the calculation of percentage of total simulation spent in the given state during each individual run, and the *Stats* to be *mean* which would calculate the mean value of the ten percentage values. This is an important feature as in modelling a system it is normal to run a large number of simulation runs to gain a truer understanding of the systems behaviour.

Processing of the analysis editor input

A data structure of the type *slot* is created for every slot on the editing grid. For the grid shown in figure 4.8 the structures created would be as shown in figure 4.9. Note that for clarity only the structures created for slots 0A..1G are shown, the other slots being 'EMPTY' slots. Note also that only the fields within the structures used for this specific example are shown.

struct slot	OA	OB	OC	OD	OE	OF	OG
t	TEXT	EMPTY	TEXT	TEXT	TEXT	EMPTY	EMPTY
U{*text	logger	-	poll	datahigh	buffer	-	-
*next}	-	-	-	-	-	-	-

struct slot	1A	1B	1C	1D	1E	1F	1G
t	EMPTY	EMPTY	STATS	STATS	STATS	EMPTY	EMPTY
U{*text	-	-	-	-	-	-	-
*next}	-	-	STA	STB	STC	-	-

struct Stats	STA	STB	STC
*object	control	data_high	logger
*state	poll	T	buffer
type	ANmean	ANpercentage	ANmean
stats_on	ANpercentage	-1	-1

Figure 4.9 Data structures produced from the Analysis Editor

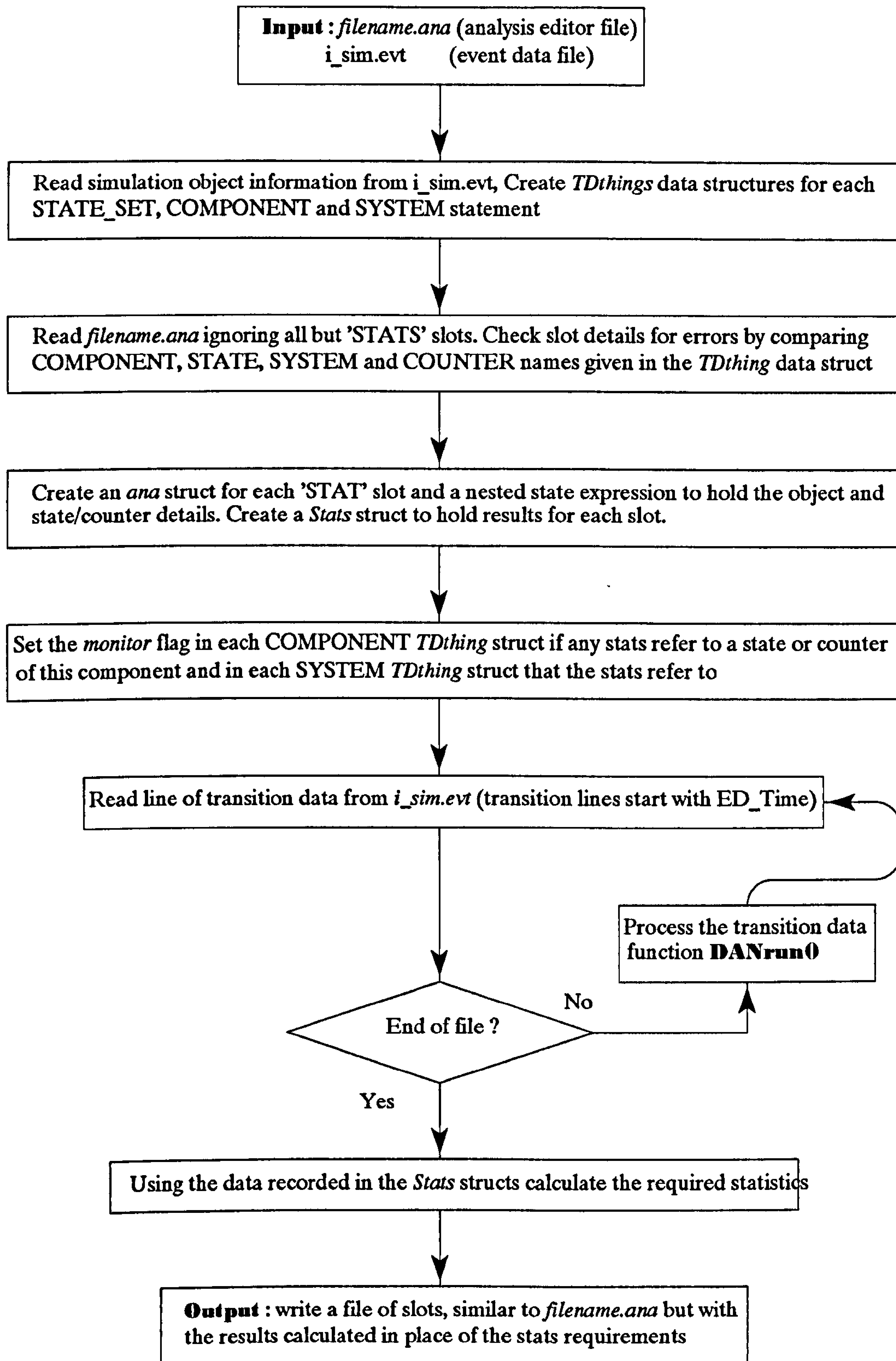


Figure 4.10 Flow of control of the Statistical Analyzer

When the grid is saved every slot is written sequentially to the *filename.ana* analysis file. The file is used as the second input to the analyzer, the first being the event data file.

The statistical analyzer

The statistical analyzer takes as input the analysis file produced by the analysis editor and the event data file produce by the simulator. The analysis file is effectively the specification of the statistics required by the user and the event data file is the raw data of the simulation upon which the analysis is performed. The flow of control of the analyzer is shown in figure 4.10.

The processing of the analyzer is best understood by referring to an example. Given the event data file in section B.4 the data structures created by the analyzer are as shown in section B.5.1. Given the *stats* structs created by the analysis editor in the example above (figure 4.9) the corresponding data structures produced by the analyzer are as shown in section B.5.2.

DANrun()

This function is the heart of the analysis software. It reads each line of the event data file after the EDstart line and updates the data held in the *Stats* structures accordingly.

For each line that is read all the *ana* structs are considered. If the line refers to the same component or system statement as the *ana Statexp* reference then it checks to see if the state or counter value has changed since the previous recording.

In the case of component.state or system statements *ana* structs, eg *ana1* and *ana2* in our example, then we record the time given in the I_SIM.EVT line minus the time of the previous recording, but only when the item has just exited the state of interest. This in effect gives a record of the individual duration times spent in the state of interest. For counter *ana* structs, i.e. *ana3* in our example, then every time the counter changes value the previous counter value and the duration for which the counter held this value is recorded.

4.6 Conclusions

In this chapter we have considered the implementation of I_SIM the software language simulator.

Due to the nature of the language the simulation algorithm that has been developed is very complex. This meant that extensive system testing was required which has proven the software to be reasonably robust.

Implementation of the simulator proceeded concurrently with the development of the language. Consideration of the various aspects of the simulation algorithm has lead to reflection upon and hence greater understanding of the language. By using a variety of simulation examples many ideas were formulated about the nature of the languages semantics. This caused considerable debate about the merits of alternative semantics and greatly helped to refine the language to its current form. At the time of writing, this was thought to be optimal for the proposed applications though as different uses are considered it may be necessary to re-evaluate some aspects of the semantics. The refined knowledge of the semantics in turn helped in the development of the software.

The nature of the software development has been iterative in that it has undergone a number of cycles of testing, debugging and modifying, both to correct errors and implement changes in semantics. This approach is typical when prototyping a system but is not optimal for producing a final product. With hindsight there are some parts of the software which could be written in a more efficient manner. The observed robustness and satisfactory simulation times indicates that this is not critical but may be worth considering in future developments.

CHAPTER 5

Random Number Generation

5.1 Overview

Random numbers are used to provide the required stochastic behaviour within software simulation. Due to the increasing variety of contexts within which random numbers are used, such as for the generation of test data for algorithm checking and the simulation of games of chance, extensive study has been conducted into techniques for their generation.

The ICE language facilitates stochastic and probabilistic modelling. Random numbers are required to determine transition times from stochastic distributions and to select between probabilistic transitions. It is therefore essential that the numbers used appear truly random for the accurate simulation of models. This requirement is especially true when modelling such things as queues. Queues are particularly sensitive to any disparity in the timing of arrivals and departures and any small disruption in reading or writing rates can significantly effect mean queue lengths [26].

An obvious source of true randomness is white noise. Hardware simulation systems may periodically quantise the output of decaying pn junctions to produce streams of random numbers [93]. The obvious disadvantage of such techniques is the inability to reproduce

the obtained sequences. The natural solution to this was the development of deterministic algorithms to generate sequences which although entirely predictable behave as if they were truly random. The accepted criteria for measuring their randomness is the application of a suite of statistical tests. Many tests have been proposed and the consensus view [94] is that a combination of tests checking a variety of properties must be passed for a sequence to be considered random. A large number of algorithms have been developed. These vary not only in the techniques employed but also in the speed of generation, length of repeatable sequence and ease of software implementation.

The pseudo random number generators which produce maximum length binary sequences or m-sequences (i.e. a sequence where all numbers in a range 0 or 1 to m are generated before any are repeated) are of interest as they are known for their good randomness properties. Each number generated is normalised to give a number, n, where $0 \leq n \leq 1$. Any point in the sequence can be used as a starting point by stating an initial *seed*. Any run of the generator starting with the same initial seed will produce identical results. The seed is revised every time a number is generated and indicates the current point in the sequence.

In this chapter we are concerned with the production of sequences of random numbers within the I_SIM software.

I_SIM facilitates the simulation of systems which can contain a variable number of components. Each component may also have a variable number of transitions between its constituent states. What is required is the generation of streams of random numbers for every stochastic and probabilistic transition. The initial implementation used one generator to produce a single stream of random numbers and these numbers were distributed to the different transitions as required. The result is many streams of random numbers each of which is comprised of random samples of the original sequence. These sequences will not necessarily hold the same properties of randomness as the original sequence.

The I_SIM generator is discussed and statistical tests applied to the generated sequence. Original modelling with I_SIM did not suggest any loss of randomness with these subsequences however the occasional questionable result when Counters were implemented caused suspicion and lead to the decision to investigate the randomness of these subsequences. The same tests when applied to the subsequences gave some interesting

results.

To guarantee the integrity of the sub sequences a new method of number allocation is suggested and implemented. The statistical tests are repeated to ensure the randomness of the improved implementation.

Finally a novel generator is proposed and implemented. This new generator shows a number of desirable properties and is compared with the previous one.

5.2 The random number generator

Many deterministic methods of producing random numbers have been proposed. The majority fall into one of five categories,

① prime modulus multiplitive linear congruential generators or Lehmer generator

② mixed linear congruential method generators

③ additive random number generators

④ shift register random number generators

⑤ combined random number generators

A good recent comparative study of public domain pseudorandom number generators is given by Vattulainen et al [95].

5.2.1 The Lehmer random number generator

These algorithms were first proposed by D H Lehmer in 1951 [96] and have become one of the most accepted and widely used methods of generating pseudo random number sequences.

The Lehmer generator takes as its input a seed and performs a mathematical operation upon it to produce another number, statistically independent from the first. The mathematical

operation used is multiplication by an integer, a, modulus a large prime integer, m. The multiplier is an integer in the range [2,m-1]. The generating function is then

$x_{n+1} = a\, x_n \bmod m$; where x_n is the nth number in an integer sequence.

The values of a and m determine the statistical randomness of the generated sequence and if chosen correctly a pseudo random sequence comprised of integers in the range [1, m-1] will be produced. The divisor m must be prime (known as the Mersenne prime) to prevent the sequence from collapsing to 0 which would occur in the event of $a.x = m$. In this occurrence the subsequent seed would remain zero, terminating the sequence. This limits the seed to the range, $0 < \text{seed} < m$. The number produced is normalised by division by m to produce a number in the range, ($0 < \text{number} < 1.0$). Many excellent Lehmer generators exist and they are used widely in multiprocessor platforms [97].

The Lehmer generator is periodic as it produces a deterministic sequence of numbers and then repeats. Ideally this sequence should be as long as possible and this is achieved by selecting values for a and m that yield a full period multiplier. This is a generator that will not repeat until it has selected every value in the range 1 to m - 1 once. The necessary and sufficient condition for this is that a is relatively prime to m ie the greatest common divisor of a and m is 1. The disadvantage of this is that ideally we would wish to have a sequence with replacement, ie if a number is selected then there is an equal likelihood of it being selected again. With the Lehmer generator once a number has been selected it will not be selected again until all other numbers in the sequence have been selected. Hence the concept of equal probability for all numbers is violated. Another known but accepted possible problem is that the pseudorandom numbers may lie in a relatively small number of parallel hyperplanes. Hyperplanes are bands of values in which the generated numbers lie. For example, numbers generated in the range 0.3 to 0.4 may not be distributed across the entire range but limited to the field, or hyperplane, of 0.34 to 0.36 [95].

5.2.2 The Park and Miller generator

A vast amount of work has been done on Lehmer generators to determine good and bad values of a and m. The Lehmer generator originally chosen for use within I_SIM was that

proposed as an industry minimal standard by Park and Miller [98]. It was chosen for three reasons :

1. It is a full period generator.
2. It has passed a number of empirical tests [99].
3. The algorithm is easily implemented in software.

The values for a and m selected were

$$a = 7^5 = 16807$$
$$m = 2^{31} - 1 = 2147483647$$

This generator has since been updated by the original authors who now recommend the values

$$a = 48271$$
$$m = 2^{31} - 1 = 2147483647$$

stating that the original generator is suitable for most situations but the new generator is "*a little better*" [100]. Note here that both moduli are prime. Moduli that are a power of two are never used as Marsaglia has shown [101] that they may produce sequences that demonstrate bad randomness properties.

Initial simulations within I_SIM using this Lehmer generator seemed to produce satisfactory results. However it was occasionally noticed that results obtained were not as expected and this lead to some suspicion as to the quality of the generator. The Park and Miller generator had previously been shown to produce a sequence of numbers with good randomness properties but what was being used within I_SIM was a number of sequences each of which was comprised of numbers obtained by randomly sampling the original sequence. To our knowledge no tests had been conducted upon randomly generated subsequences of the Park and Miller generator. It was therefore considered necessary to test the statistical properties of the subsequences and if results were unfavourable devise an alternative method of random number allocation to stochastic transitions within I_SIM.

5.2.3 Statistical properties of random number sequences

There are two categories of statistical tests that may be performed on a random number generator. The first are theoretical tests which mathematically analyse the generator itself. The second are empirical tests which involve the analysis of the sequences of numbers produced by the generator.

Since theoretical tests are designed for the analysis of the sequence directly produce by the generator they are no use for the analysis of subsequences unless these subsequences are reproductions of the original sequence. Most empirical tests hold good for subsequences as well as for full periods and it was therefore decided to use these.

The empirical tests are based on the theoretical properties of an ideal random sequence. Pseudo random number sequences are predictable and therefore non ideal. We can consider a pseudo random sequence as approximating the ideal if it passes a variety of statistical tests which would be passed by a true non deterministic sequence.

5.3 random number properties

If B_i is an N bit random number uniformly distributed in the interval $(0, 2^{N-1}]$ then each number should be equally as likely to appear at any point in the sequence. Each number should be independent of all previous numbers in the sequence. This criteria would seem hard to satisfy as each number is strictly determined by the previous number. However if the numbers produced pass a series of statistical tests they are deemed acceptable. The number of runs of consecutively increasing or decreasing values should be directly related to the length of the run,

half the runs should be of length 1,
a fourth of length 2,
an eighth of length 3,
...

Also the autocorrelation function should be peaked at zero phase shift and near zero for all other values.

In addition to the above properties of randomness a good sequence should be m distributed. That is, the generator should be capable of producing m independent pseudorandom numbers which are uniformly distributed over the interval $(0, 2^{N-1}]$. Such a generator is said to be uniformly distributed in m dimensional space.

5.3.1 Statistical tests

Statistical test performed on a sequence of random numbers are designed to measure the closeness of selected properties to the theoretical properties of an "ideal" sequence. Due to their statistical nature it is not possible to state categorically whether a sequence has passed a given test, but rather we can state that a test has been passed with a given degree of confidence. If the degree of confidence chosen was 95% then we would expect the test to be failed one time in twenty on average. We may then categorise a bad random number generator as one that fails the tests on average more often then it should. It is common practice to perform multiple runs of tests to determine reliable results.

A variety of proven tests are available. Different tests are designed to measure the different desired properties of a random number sequence. When choosing tests to apply to a sequence we must select those that will give a comprehensive guide to the "randomness" of the sequence. The tests that were chosen analyse different properties of the sequence, many other tests could have been done but would have overlapped on the properties being examined. Below is given a brief description of the tests selected for application to the Park and Miller generator. These tests chosen were all equally applicable to subsequences as to an entire sequence. All of the tests except the serial correlation use the Chi-square test for analysis of results. This facilitates the ability to directly compare results between the test ie we are able to say that a sequence performs better on test A than on test B.

5.3.1.1 The Chi-square test

The Chi-square test gives a measure of the difference between the observed frequencies of a number of events and the theoretically predicted frequencies of events. It is used to

interpret the results of most of the tests used here.

The Chi-square test is best illustrated by an example. Consider a six sided unbiased die. The probability of rolling any number is 1/6. Therefore if the die is rolled *n* times we would expect each number to come up *n*/6 times. If what was observed differed from this value then it would be possible to say, with a degree of certainty, that the die is biased. The difference is expressed by the equation

$$V = \Sigma (Y_s - np_s)^2 / np_s \, ;$$

where : Y is the observed frequency

: np is the expected frequency

The computed value of V is compared to the Chi-square distribution table of percentage points. The row of the table is selected dependant upon the number of degrees of freedom required which is one less then the number of possible outcomes tested for ie 5 in the die example. The column of the table is selected dependant upon the desired confidence limit.

This procedure is important as it gives an indication of how close test results are to the results that would be expected, thus providing a performance measure.

The Equidistribution test

This test is based on the requirement that a random sequence of numbers should be equally distributed between the minimum and maximum numbers. We achieve this by multiplying each number by a constant number *d* and rounding the result to the nearest integer. The expected frequency of occurrence of each integer value is therefore m / d (where *m* is the length of the sequence).

The Serial test

This test is preformed for the same reason as the equidistribution test but it uses pairs of values. This measure gives the distribution of a sequence of numbers from 0 to *d* over d^2 possible values. The expected frequency of occurrence of each pair as $1/d^2$. The increased range of values dictates that the sequence to be analysed must be of sufficient length to validate the Chi-square test conducted on the integer pairs. The rule proposed by Knuth

[94] was used, which states that each category suggested should be expected to occur at least 5 times and hence the use of sequences of length $\geq 5d^2$ is required.

The Gap test

Gap tests check the distribution of lengths of gaps between numbers within a specified range. This is important as we can check for the clustering of values within a chosen range. It was decided to use the ranges 0 to 0.5 and 0.5 to 1 to test for runs above and below the mean respectively.

The Poker test

This test analyses the sequence of numbers five at a time, recording the occurrence of a number of patterns. This detects if patterns are repeated too frequently or significantly more than other patterns. The patterns that were selected are one, two, three, four and five numbers falling within the same division. The range of possible values 0 to 1 being split into ten divisions of equal length. This is a simplification of the classical poker test which has seven categories. The five categories are sufficient for our purposes and are significantly simpler to implement.

The Permutation test

This test is similar to the Poker test, splitting the sequence into groups of numbers of size t. The numbers in any group can then have t! possible relative orderings. The algorithm then counts the number of occurrences of each possible ordering. The expected frequency of occurrence of each ordering is n/t!. The value of t was chosen as 5 to be suitable for analysis.

The Run test

The run test analyses the distribution of runs for consecutive numbers up and down in the sequence. Subsequences of the original sequence in which all the numbers are either running up and down are examined. Each sequence analysed was offset from previous sequences by at least one number to ensure that adjacent runs were independent. This is a proviso of the Chi-square test that all events analysed be independent of each other.

Maximum of t test

This test splits the sequence into groups of length t and records the maximum value in each group hence checking for any bias in groups of values. These recorded results are effectively a new sequence of length m/t. This new subsequence can then be analysed by the equidistribution test. The value of t was again chosen as five for the previously given reasons.

The Serial Correlation test

This test generates a serial correlation coefficient which is a measure of how much any number within the sequence is dependant upon any of the previous numbers. This is a valuable test as it gives an indication as to the unpredictability of a number sequence.

5.3.2 The testing procedure

Large sequences of numbers were taken from generated sequences. The poker and permutation tests required numbers to be split into groups of five and the serial test required pairs therefore the amount of numbers used had to be divisible by ten. The amount of numbers tested at any one time must also be representative of typical I_SIM applications and for this reason tests were done on sequences of numbers from 500 to 5000.

All of the tests were implemented in software. These tests took as their input a file of random numbers. The relevant parts of the Chi-square tables were also implemented in software to facilitate immediate analysis. All results were considered to fall into one of three categories

i) Pass. The result is what would be expected to occur 90% of the time for a truly random sequence.

ii) Suspect. The result is one which would only be expected to occur 10% of the time.

iii) Reject. The result is one which would only be expected to occur 2% of the time.

Each test was repeated at least ten times using different subsequences. When results were inconclusive the tests were repeated in blocks of ten. When generating subsequences seeds were chosen such that different subsequences would have no overlap.

5.4 Testing the Park and Miller generator

The first step of testing was to apply the statistical tests to the Park and Miller Generator. Park and Miller's original spectral tests were global. Knuth [95] has shown that this is not a sufficient test and that local randomness must also be tested. What we are testing here are relatively short subsequences ie the local properties. The Lehmer generator originally proposed [98] was tested as well as the modified generator [94]. Each generator was tested with multiple runs incorporating 500 to 5000 samples.

A typical set of results for the Chi-square tests on the first generator are shown in table 5.1. The results shown are for 2000 samples. The failure rate here is 12% at the 10% confidence level and 2% at the 2% confidence level. These results are almost perfect. Any discrepancies in these tests were small enough to be incidental to the tests and did not indicate any statistical inadequacies. The results from the serial correlation tests applied to the same 10 runs are shown in table 5.2. There is no indication of significant levels of correlation.

#	EQ	SE	G1	G2	PK	PE	RD	RU	MX
1									
2			S						
3								R	
4		S				S			
5									
6			S				S	S	
7						S			S
8					R				
9							S		
10									

Table 5.1 Test results for Park and Miller generator with multiplier 16807

Run #	% failing at 5% level
1	5.20
2	3.35
3	4.90
4	3.80
5	5.20
6	5.80
7	4.10
8	4.30
9	4.75
10	4.20
Average	4.56

Table 5.2 Serial Correlation results for Park and Miller generator with multiplier 16807

The generator tested above was that which was first implemented in I_SIM. This was updated by Park and Miller who now suggest a multiplier of 48271. A typical set of results for the Chi-square test, again for 2000 samples per test is shown in table 5.3. The results were very similar to the previous ones. In this example it failed the same number of tests as the first generator at the 10% level, ie 12% but slightly more at the 2% level, 3% as opposed to 2%. These results are typical of all the tests done. The serial correlation tests also produced results very close to those of the first sequence.

#									
1		S					S	R	
2									
3									
4				S					
5							S		
6									
7	R			S					
8							S		
9									
10	S				S		R		

	EQ	SE	G1	G2	PK	PE	RD	RU	MX

Table 5.3 Test results for Park and Miller generator with multiplier 48271

Run #	% failing at 5% level
1	5.00
2	5.60
3	4.80
4	5.25
5	3.50
6	6.35
7	7.20
8	3.65
9	3.70
10	3.90
Average	4.895

Table 5.4 Serial Correlation results for Park and Miller generator with multiplier 48271

From the tests it may be concluded that there is neither generator is appreciably statistically better then the other. It can also be stated that with respect to the statistical tests used, the Park and Miller generator is virtually ideal and therefore suitable for use within I_SIM.

5.5 Random number allocation

Some systems may have a number of independent components which require random numbers. Such is the case with I_SIM. Deng et al [102] considered the similar case of the allocation of random numbers within multiprocessor systems. They identified four criteria that should be satisfied, three of which are equally applicable to the instance of multi-component systems. These are

i) Each subsequence should be indistinguishable from that produced by a

random sequence of standard uniformly distributed random variables.

ii) A subsequence generated for one process should be independent of the subsequence generated for another process.

iii) The random numbers subsequences should be identically reproducible on a subsequent execution of the same program.

Makino [103] considered randomness of a parallelized source with application to congruential and shift register generators used within parallel computers. He found that the randomness of the parallelized streams or subsequences ranged from being good to bad dependant upon the method by which the source is parallelized. This work concentrated on uniform means of parallelization, within I_SIM the subsequences will be generated in a random manner dependant entirely upon the individual model being simulated.

Random number allocation may be implemented sequentially, parallely or randomly. With sequential allocation each of n components will be allocated every nth number from the sequence. This is simple and requires no set up procedures however it has been recognised that the subsequences may not be statistically as good as the original [103]. If the components consume elements at variable rates then there may be space leakage, a problem which is only partially solved by the wasteful solution of providing buffering for each component. Space leakage is the loss of elements allocated to components which require numbers at lower rates than those dictating the speed of allocation. Buffering is a wasteful solution as it necessitates the use of extra resources to store the elements allocated to these components. For this reason sequential allocation was unsuitable for I_SIM.

In parallel allocation the generated sequence is split into n sequential sources so that each component has its own linear non overlapping subsequence of the original sequence. The statistical goodness of the subsequences should then be identical to that of the generator. The primary problem with this method is what Burton and Page [104] refer to as "awkward plumbing". Each component must have its own seed sufficiently far apart in the sequence to avoid overlap. The allocation of seeds must be set up initially and this task becomes complicated when the number of components is unknown.

Random allocation, allocates numbers from the generated sequence to components as they are required. This results in subsequences which are a non deterministic irregular sampling of the generated sequence. In one sense this is a refinement on sequential allocation as it

solves the problem of space leakage and saves on the use of component buffering. For these reasons, random allocation was the method chosen for I_SIM. However the statistical quality of the subsequences is non determinable by theoretical means.

Deng at el [102] proposed a novel method of allocation for multiplitive linear congruential generators that they have termed the systematic random leapfrog method. This involves systematically choosing different multipliers for each of the different random number generators allocated to each process. They have shown this method to be robust if a good modulus is used. It is however not ideal for our purpose as we wish to use the Park and Miller generator and therefore the multiplier must be constant. This is not an insurmountable problem, though the implementation of such a means of allocation would require the testing of a number of multipliers.

5.6 Testing the subsequences produced by the Park and Miller generator within I_SIM

We have seen that the random sequence produced by the Park and Miller generator is virtually ideal. In section 5.4 it was explained how random allocation is used to provide the stochastic transitions within a I_SIM simulation with randomly sampled subsequences of the generated sequence. As Makino [103] has shown we cannot assume that these subsequences will exhibit the same statistical properties of the generated sequence.

The subsequences will be unique to individual models therefore to provide a representative set of data for analysis it was necessary to collect data from a variety of 'typical' ICE programs. The remit for these programs was that they should include all of the stochastic and probabilistic features of the language that require random numbers and that they should generate a suitably large number of calls to the random number generator for each transition under analysis so that an acceptable amount of test data could be produced.

5.6.1 Data logging modification to I_SIM

A number of modifications were required by the I_SIM software to produce the data for

analysis. All of the modifications were functionally transparent to the user. The data logging software creates a file which contains all data required for analysis. Every time the random number generator is called a string of data is written to the file including the transitions to and from states, whether the call was for a probabilistic or stochastic decision, the seed to be used by the generator and the random number produced. This data allows the analysis software to pick out any individual subsequence of random numbers and create files containing the subsequence for any chosen transition.

5.6.2 Example results of I_SIM generated subsequences

Three ICE files were used to produce typical subsequences for analysis. The results obtained from each were all quite similar as would be expected. We present here one of the models and the results obtained.

The model has two components with the same state space. The state space for a component is shown in figure 5.1. All transitions are stochastic and two of them are probabilistic.

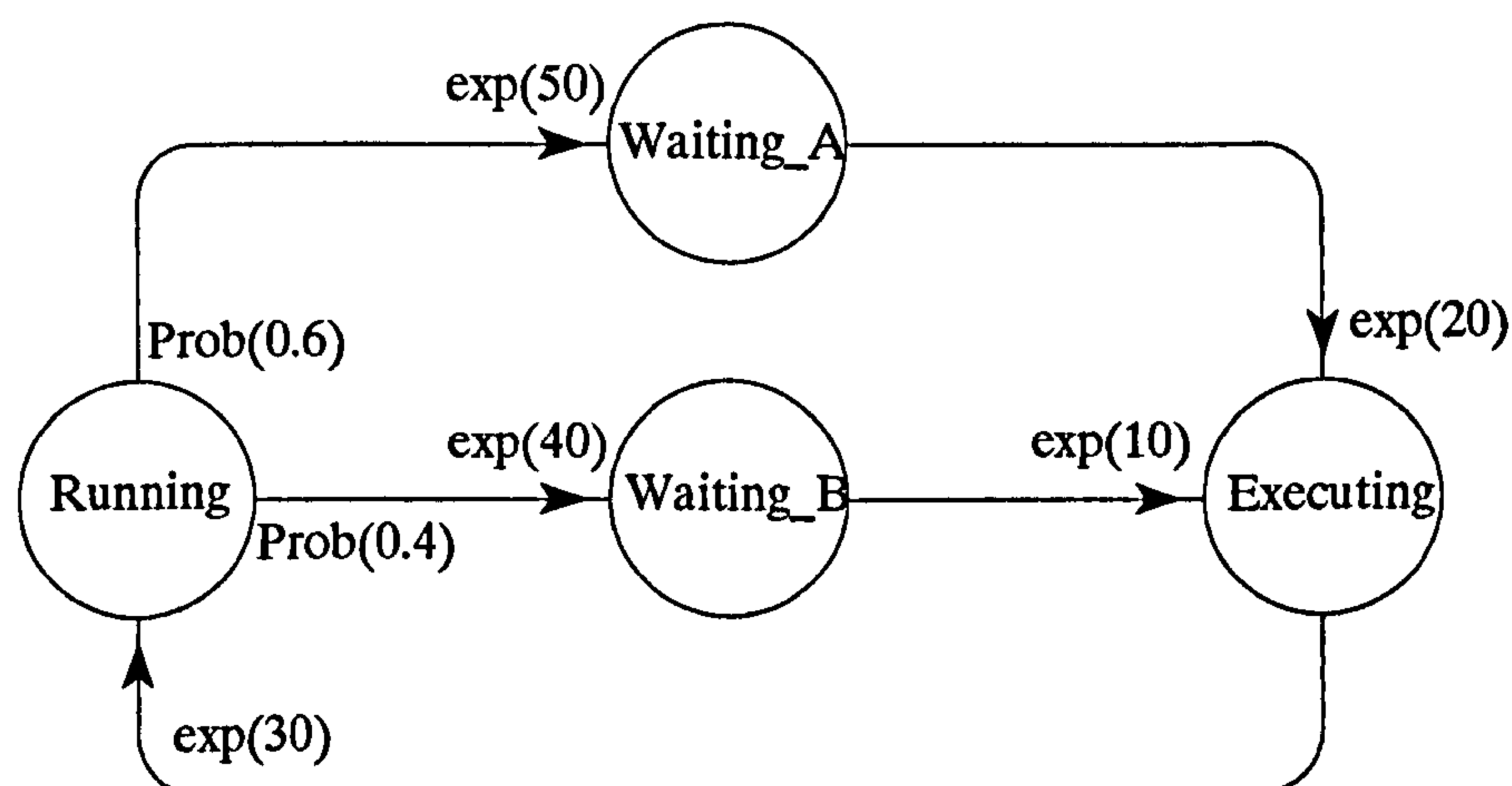
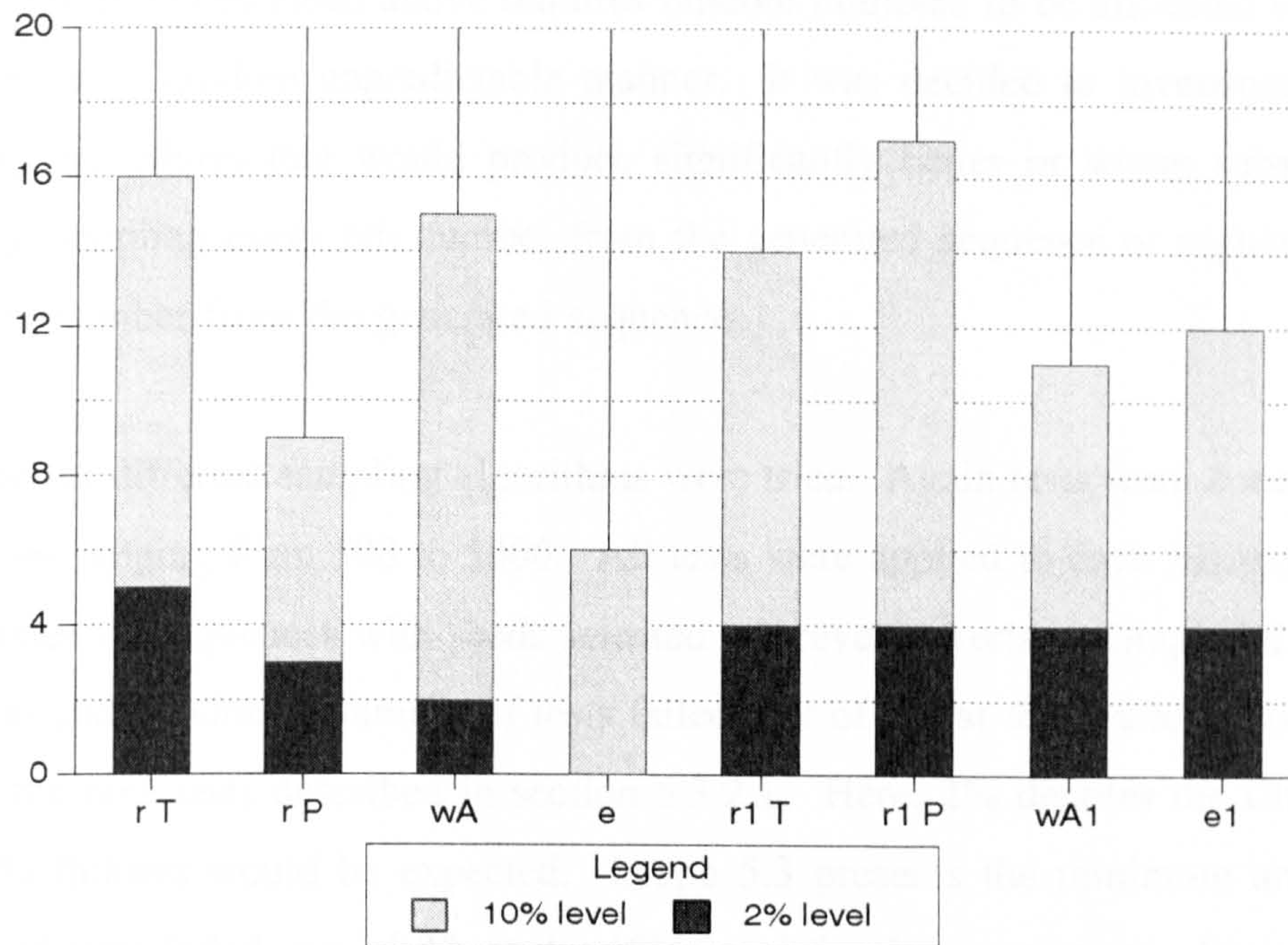


Figure 5.1 State space of model used for analysis of Subsequences

The subsequences applied to each transition were analysed. In the case of the transitions from the *running* state to the *waiting* states the subsequences used for the stochastic behaviour and for the probabilistic behaviour were both analysed separately. Graph 5.1 shows the number of tests failed at both the 10% and 2% levels for a selection of the

transitions. These results are typical of all the results obtained for this and the other test programs.



Graph 5.1 Tests failed at the 10% and 2% levels for selected transitions

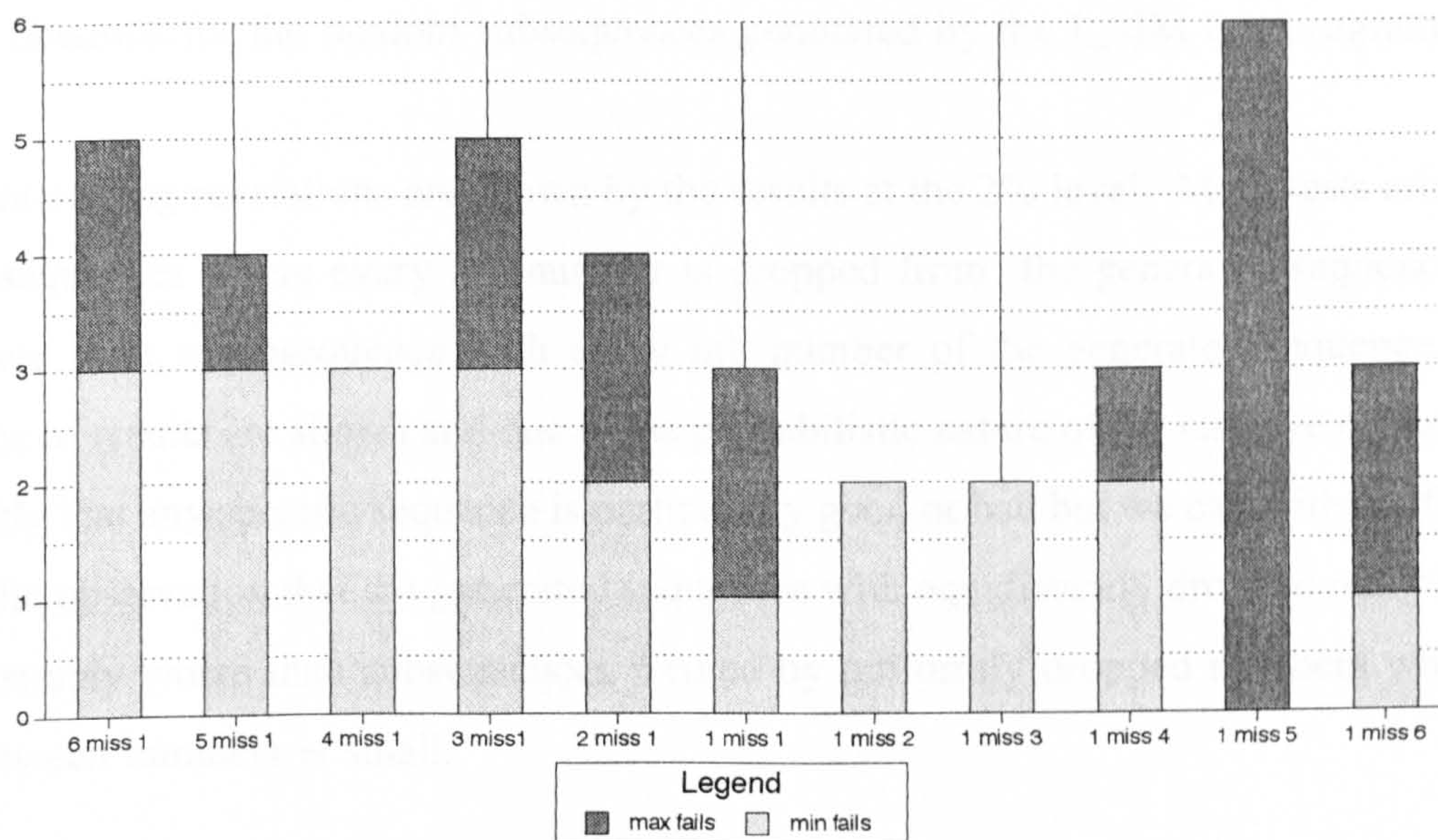
There is a considerable range of results. They range from less than 7% of tests failed at the 10% level and 0% of tests failed at the 2% level, which is better than would be expected for an 'ideal' sequence to 19 % of tests failed at the 10% level and 5.6% failed at the 2% level. By random chance we would expect to have some very good results and some poor results. What is significant is the proliferation of failures of 4 out of 90 at the 2% level. This is twice as many as would be expected and though not very bad is certainly a strong enough trend to be able to state that the subsequences on average are not as statistically good as the generated sequence. This evidence indicates that the initial method of random number allocation is not acceptable and must be improved if all transitions are to be provided with random numbers as statistically good as the generated sequence.

An interesting point to note is that all tests were failed with equal likelihood. The subsequences were statistically poorer in general and not with relation to any specific attribute.

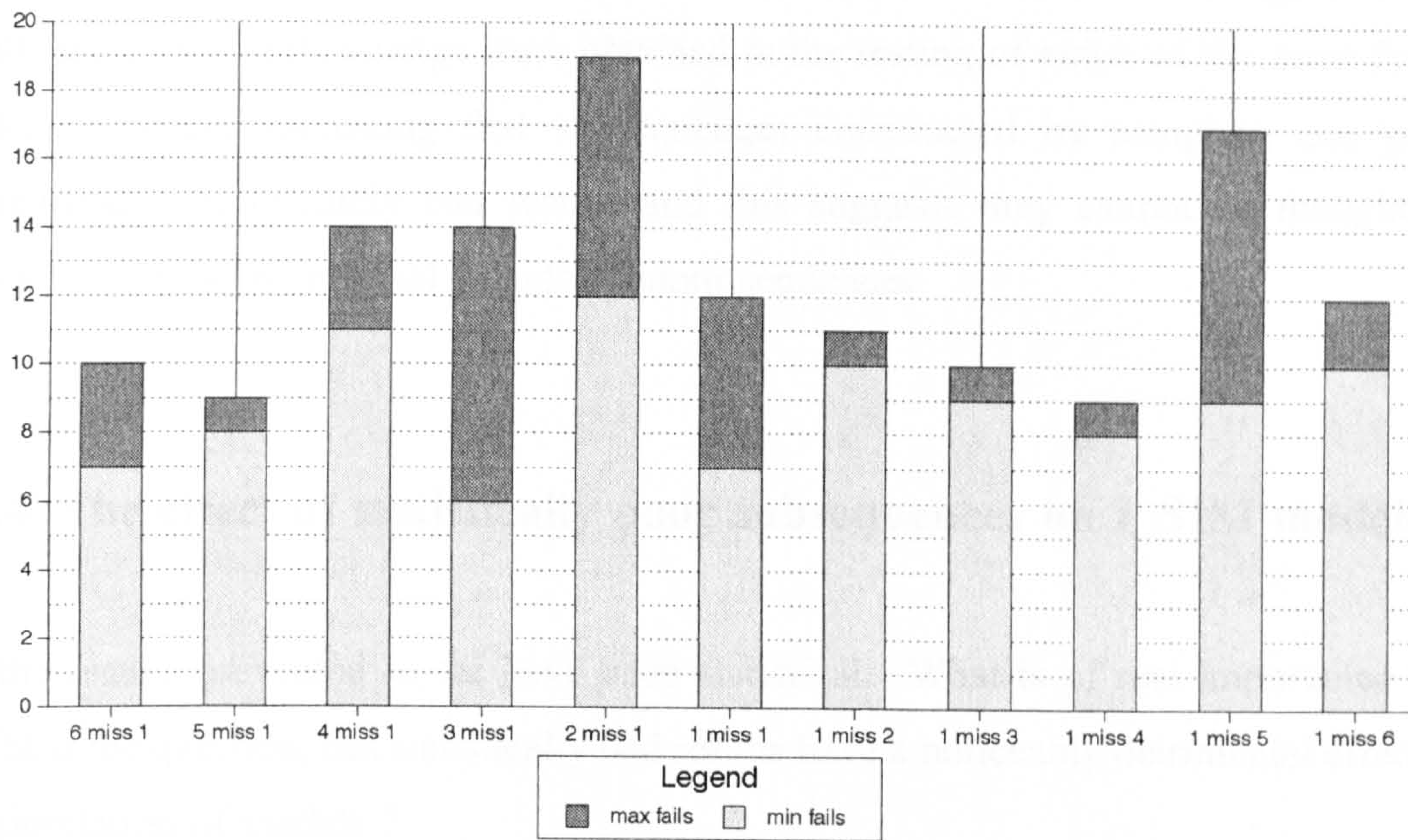
5.6.3 Detection of any predictably statistically poor subsequences

The test program described above requires random numbers to be allocated to the various transitions in a random unpredictable manner. It was decided to investigate alternative sampling procedures that would produce significantly better or worse subsequences eg regularly sampling every nth number from the generated sequence or regularly dropping every nth number from the generated sequence.

A number of different sampling algorithms were tried. Again tests were done on numbers of samples ranging from 500 to 5000. All tests were applied to each pattern a minimum of ten times on sequences with seeds selected to prevent overlap. Graph 5.2 presents the minimum and maximum number of tests failed, out of 90, at the 2% level for ten runs of each of the nine tests described in section 5.3.2.1. Here, 2% denotes the Chi-square test when 2% failures would be expected. Graph 5.3 presents the minimum and maximum number of tests failed, out of 90, at the 10% level for the same range of tests.



Graph 5.2 Minimum and maximum number of tests failed at the 2% level



G

raph 5.3 Minimum and maximum number of tests failed at the 10% level

Considering the results at the 10% level we cannot detect any particular trends. The values range from less than 7% to 21 % ie from very good to unacceptable. This confirms the results obtained for the random subsequences generated by the I_SIM test programs.

More interesting revelations are shown by the results at the 2% level. More tests are failed in the sequences where every nth number is dropped from the generated sequence than when we form a subsequence with every nth number of the generated sequence. The variance of results are shown and due to the probabilistic nature of the tests we cannot state explicitly that any specific sequence is particularly good or bad but we can with confidence make the observation that the generated sequences with equidistantly dropped numbers are demonstrably worse than subsequences formed by uniformly dropped numbers when the gap between numbers is small.

It should also be noted that the results can vary significantly for the same pattern of samplings done on different parts of the same generated sequence. Consider the instance when a subsequence is created by sampling every 6th number in the generated sequence ie '1 miss 5' on the graphs. The tests failed at the 2% level range from 0% to 6.67%. This is a large margin at the 2 % level ie from better than expected to over three times as worse

than expected. A number of extra tests were done on this sampling pattern and no other tests were found that were worse than 2.22%. From this we can see that the 6.67% was a rare chance result. Rare results are to be expected with statistical testing, however no results with such great a range were obtained in the testing of either of the main Park and Miller sequences indicating that subsequences constructed by sampling can produce unpredictable significantly bad results and this suggests they cannot be thought of as representative of the original pseudo-random sequences.

5.6.4 The effect of statistically poor subsequences on I_SIM models

All the results prevented so far have been statistical. What is of real importance within I_SIM is the question, can statistically bad results have a noticeably detrimental effect upon the simulation of models ?

To investigate this a number of ICE programs were written which incorporated a mixture of stochastic and probabilistic transitions. These programs were kept simple so that the relative order in which transitions would call the random number generator could be predicted. This allows transitions to be fed with subsequences that are known to be statistically bad. Statistically bad subsequences were chosen from tables of results and the related starting seed used within the simulation.

We shall consider here one of the models used and the range of results obtained. This model is of two communication network sources. It comprises two channels which feed fifo queues in two network terminating units (NTUs). Each NTU is connected to the queue of a switching element within a switch. The two sources are theoretically independent though we shall introduce statistical independence via the traffic loads. A block diagram of the model is shown in figure 5.2. The state space diagram for one source is shown in figure 5.3.

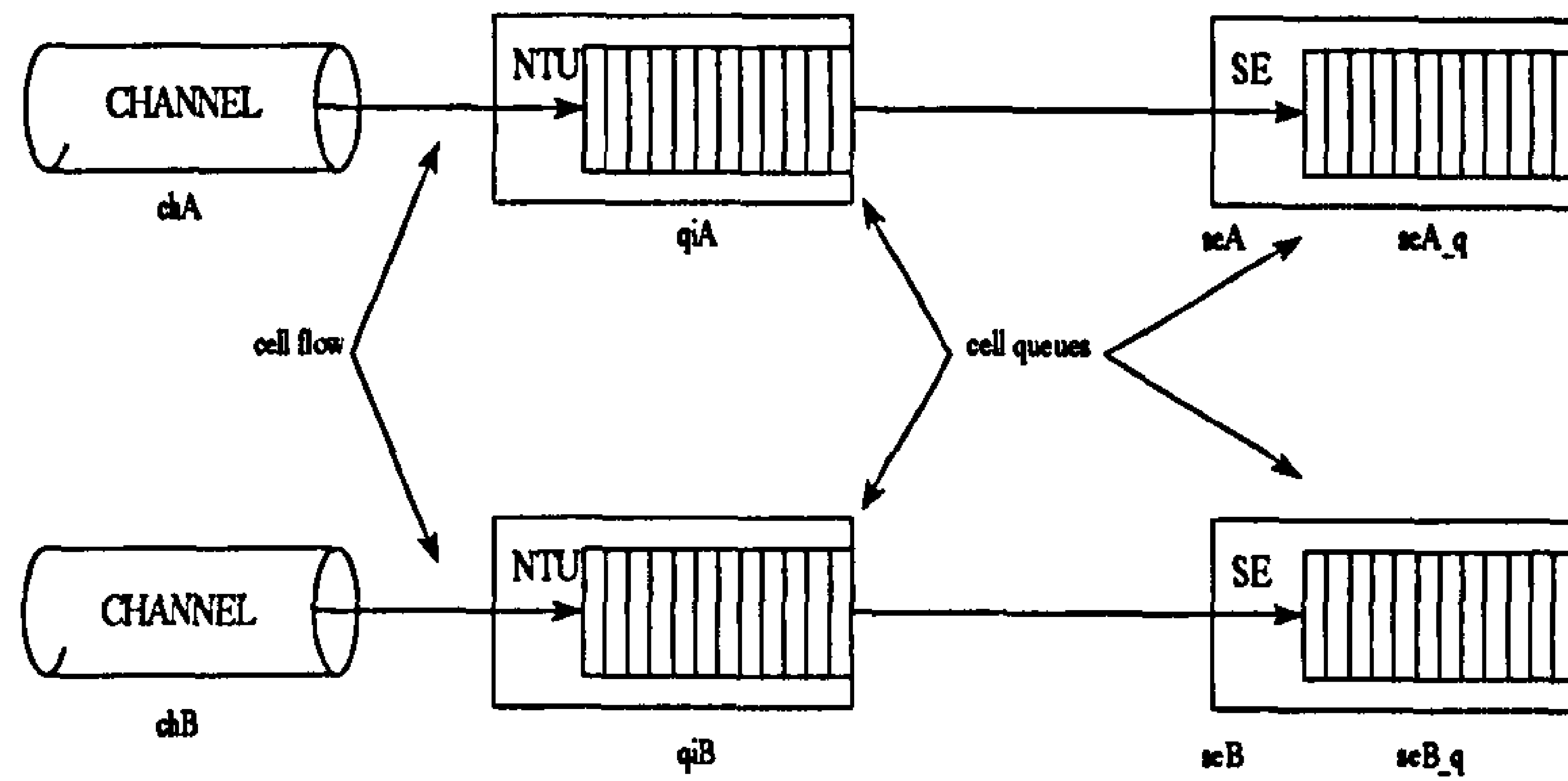


Figure 5.2 Model of communication sources

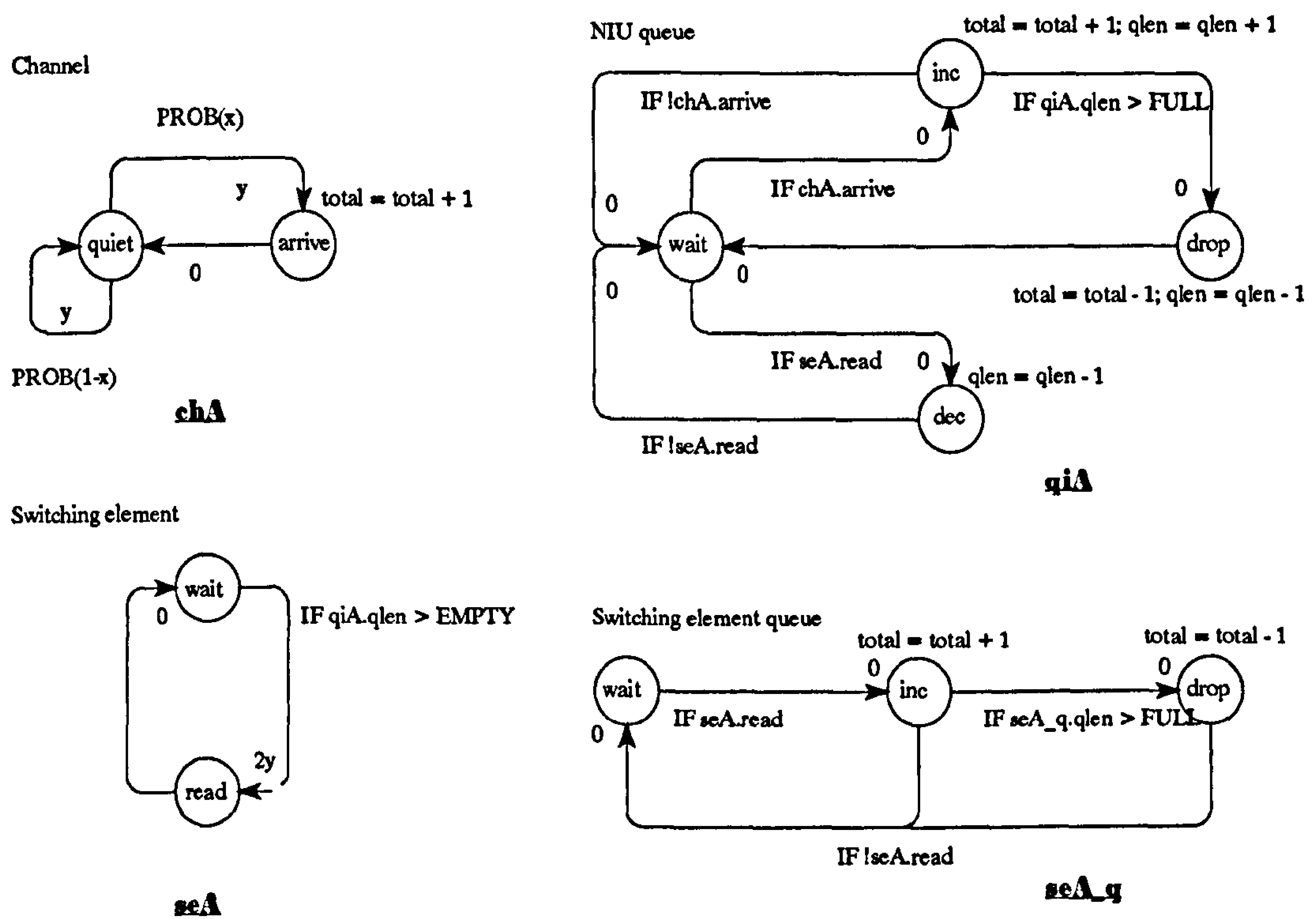


Figure 5.3 State space diagram of a model of communication sources

We shall briefly consider the functionality of the model by reference to the state space diagrams for each of the four components in turn.

The channel component generates traffic at any time instant with probability x. By setting

the values of x in the two different sources we can achieve the desired pattern for generating the subsequences of random numbers. For example if we set x to be 1 in channel *chA* and 5 in channel *chB* then the subsequences used by *chA* will be equivalent to missing every 6th number from the generated sequence ie 5 miss 1. Conversely this will set the subsequence used by *chB* to be every 6th number from the generated sequence ie 1 miss 5. This probabilistic method of determining the load creates the statistical interdependence between the theoretically independent models. By choosing the starting seed in the ICE program we can recreate any of the subsequences statistically analysed previously.

The NIU queuing component *qiA* models the queue of traffic generated by *chA*. Whenever traffic arrives the queue is incremented. If it grows to be over a given length, *FULL*, it is decremented to remain at this maximum value, ie this simulates queue overflow. The queue is decremented whenever the switching element *seA* reads from it. The rate at which *seA* reads is $2y$ ie twice the rate at which traffic arrives. Using this ratio we can set the load so that traffic in the queues should either build up, stay stable or decrease. If we set the load to be 0.5 the level of the NIU queues should stay reasonably steady. When *seA* reads traffic from *qiA* the queue *seA_q* is incremented. The counters *total* associated with *chA, qiA* and *seA_q* show the total amount of traffic that has arrived at the respective components. The counter *qlen* associated with *qiA* keeps a running check of the amount of traffic in the queue.

The values of the traffic load in both channels were set to different ratios to recreate some of the traffic patterns previously analysed. As an indicator of the effect the subsequences have on the model's behaviour we consider the mean length of the queues *qiA.qlen* and *qiB.qlen*. Since the ratio between the feeding and reading of each queue is the same the mean values should be similar.

This model was coded, see appendix C, and simulated. A number of simulations were observed with the mean values of corresponding queues not usually deviating by greater than 1 when runs lasting at least 1000 traffic arrivals were done.

To test if statistically bad subsequences would effect the models behaviour, the subsequences that gave particularly bad results with the sampling pattern 5 miss 1 were chosen. The starting seeds were selected to run simulations that would recreate the

subsequences previously recorded. The results of the mean values for the lengths of *qiA* and *qiB* are shown in table 5.5.

qiA	qiB
2.105	1.925
2.875	3.820
10.27	4.25
15.407	3.753
2.5	1.575

Table 5.5 Mean queue lengths from simulations using statistically bad subsequences.

Some of the results are reasonable and are not affected by the statistically bad subsequences. However there are two results which are significantly bad. To investigate this further the result with the biggest difference between queue lengths, *qiA* = 15.407 and *qiB* = 3.753, was considered. By taking the generated subsequence, splitting it into ten equisized sequences and taking the seed at the start of each of these sequences we could run ten individual simulations each of one tenth of the length of the original and thus when concatenated they are directly equivalent to the original. Table 5.6 shows the mean lengths of the queues for each of the individual simulations.

#	qiA	qiB
1	1.66	1.6
2	2.92	1.45
3	2.29	1.65
4	4.17	3.55
5	4.05	0.45
6	3.24	1.05
7	7.35	2.95
8	5.14	1.45
9	2.47	0.85
10	1.66	2.3

Table 5.6 Mean queue lengths from one simulation broken down into ten equisized constituent parts

It can be seen that the mean value of *qiA* increases by over 4 from the 6th to the 7th run. Significantly the highest mean is 7.35 for the 7th run which is less then half the mean of 15.407 for the complete sequence. When the file of random numbers forming this subsequence was examined there were two extended runs of values greater than 0.5. This would have caused channel *chA* to produce traffic feeding *qiA* at a greater rate it than it was been read by *seA* thereby causing the queues length to increase. This was an interesting result as the statistical test for runs up was failed badly by this subsequence. Since *qiA* is being read at a steady rate any runs up in the traffic can take an undeterminable time to clear from the queue dependant upon the rate of arrivals. When, as in the case being examined, runs up in traffic are interspersed with steady traffic without any runs down then the queues mean values can be falsely high.

This example has shown how statistically bad subsequences can produce actual bad simulation results. The inherent nature of queues effectively illustrated how the poor subsequence of random numbers directly altered the results from the simulation. Counters and queues within ICE models can be significantly effected by statistically bad subsequences. It was also observed that not all statistically bad subsequences will effect the behaviour of the model. Some statistically bad subsequences will have no noticeable effect on the models. It will also often be the case that the effect will be entirely dependant upon the model itself. The nature of the transition interactions within a model and their dependency upon the subsequences can vary substantially and a statistically bad subsequence may significantly effect one model and have no apparent effect upon another.

5.7 A revised scheme for random number allocation within I_SIM

It was shown in section 5.6 that the random method of allocating random numbers to transitions from the generated random number sequence produced subsequences which could be statistically poor. These subsequences could have a detrimental influence on the behaviour of simulations producing unexpected and sometime erroneous results. For this reason it was decided to revise the method of random number allocation.

In the random method of random number allocation there is only one generated sequence being consumed by a number of components. The logical progression is to have an independent random number sequence for each consuming component. This may be

achieved by having one random number generator and using the parallel allocation method described in section 5.4.

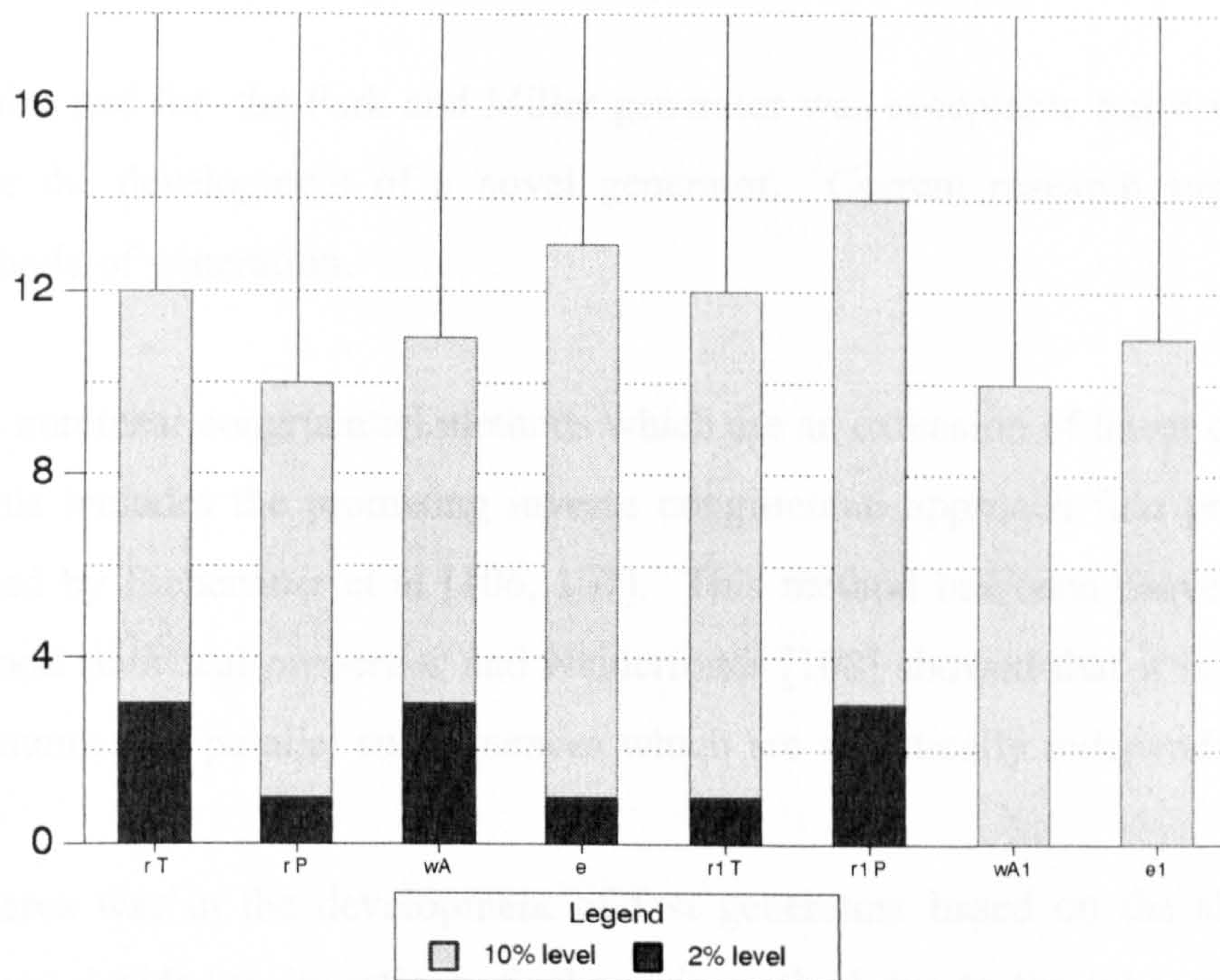
Within I_SIM parallel allocation was implemented by giving each probabilistic and stochastic transition its own seed. This is the equivalent of splitting the sequence into a number of disjoint subsequences, each of which should act like a virtual generator [105]. When any transition requires a random number to generate a probability or time it calls the random number generator with its own seed. The generator produces a random number using this seed and in the process will produce the sequentially following seed. This new seed is returned to the calling transition. This process allows each transition to generate its own independent sequence as required.

The main problem with random number allocation is determining the initial seeds. Burton and Page discuss several methods in [104]. The primary concern is that each seed should be sufficiently spaced within the overall sequence so that no part of the same linear sequence will be used by more than one component within the same simulation run. If this did occur then there would be high levels of correlation between the sequences being allocated to supposedly independent components.

The method of determining the initial seeds was that which would give the greatest relief from cross correlation. It was predicted that the highest conceivable amount of transitions requiring random number sequences would be 500 (this may be simply changed if future application areas require it). It was then predicted that the maximum amount of random numbers required by any transition would be no greater then 1000000. Using these figures the generator was run for 500 x 1000000 iterations and every 500th seed generated recorded. A table was created with these seeds and they are used as the initial seeds within I_SIM. Within a simulation these seeds are allocated to transitions as required. It is assured that the maximum spread of seeds is used. For example, if a model requires 10 seeds every 50th seeds in the table will be allocated to a transition.

It is desirable to be able to repeat simulations and to run simulations with different seeds. To facilitate this the SEED command is used within an ICE model. The value of SEED may be from 1 to 500 and dictates the starting point in the table from which seeds are allocated. This will allow 500 different simulation runs of the same model, which if deemed insufficient at some point may be increased by increasing the table size.

The statistical properties of the subsequences produced by this revised method of allocation should in theory be equal to that of the parent generated sequence. To test this the same programs were used as described in section 5.3.2. We shall consider here the model with the state space shown in figure 5.1. Graph 5.4 presents the test results for the same transitions as in graph 5.1, which shows the test results produced for the subsequences created using the previous method of random allocation.



Graph 5.4 Test results for 10% and 2% levels using subsequences produced by parallel allocation.

It can be seen that these results are comparable to those for the generated sequence (table 5.1). It is interesting to compare these results to the results for the equivalent tests in graph 5.1. It can be seen that there are slightly less failures at the 10% level and on average half the failures at the 2% level. For all the tests done on this improved method of allocation the average percentage of failures at the 10% level was 13% and at the 2% level 1.67%. These results are very close to what would be expected for an 'ideal' random number generator.

The revised method of parallel random number allocation has significantly improved the statistical goodness of the subsequences being used by the transitions within I_SIM simulations. The subsequences are now close enough to the theoretical ideal to be

considered ideal. Out of all the tests done there were no particularly bad results, the worst number of failures being 15.5 % at the 10% level and 3.3% at the 2% level. These values are still within acceptable limits and we can therefore be confident in the statistical goodness of all random number subsequences being used within I_SIM.

5.8 The development of a novel pseudorandom number generator

The results obtained for the Park and Miller generator was acceptable but it was decided to investigate the development of a novel generator. Current research suggested two possible methods of generation.

The first was nonlinear congruential methods which are an extension of linear congruential methods. This includes the promising inverse congruential approach first proposed and then developed by Eichenauer et al [106, 107]. This method has been proven to have a number of good statistical properties and Neiderreiter [108] showed that it is suitable for producing a number of parallel subsequences which are statistically independent.

The second area was in the development of fast generators based on the shift register method. It was decided to investigate further this method due to the inherent properties of long period and fast iteration of such generators. The long periods possible offer the possibility of the production of uncorrelated parallel subsequences.

5.8.1 Linear feedback shift register sequences

Using hardware shift registers to generate binary sequences has been extensively researched [109]. In this method selected bits of a shift register are added modulo 2 to the least significant bit to compute the next logical input level. This level is input to the register on the next clock pulse during which the contents of the register are shifted along one bit. The succession of states in the register is periodic with the period, P, being

$P \leq 2^n - 1$; where n is the number of bits in the register.

To produce a new number of bit length n there must be n iterations of the feedback operation. Each state is wholly determined by the previous state. Obviously the all zeros state is not allowed as this would result in logic zero being fed back and the register would remain in this state.

Let X be the sequence of 1's and 0's generated by the linear recursion relationship, where

$$X = \{ x_i \} ; \quad \text{where } i = 0, 1, 2, ...$$

The feedback operations, ie the definition of which bits are added modulo 2 and fed back, is given by the feedback equation

$$x_i = a_1 x_{i-1} + a_2 x_{i-2} + ... + a_n x_{i-n} \qquad (1)$$

where a_j ($j = 1, 2, ..., n$) takes the value 1 or 0. For the sequence to be of degree n, a_n must be 1. The integers a_j determine which stages of the register are fed back. Equation (1) shows that the sequence x_i depends only upon the preceding n-tuple (x_{i-1}, x_{i-1} ,..., x_{i-1}). Each new n-tuple has a unique successor determined only by the recursion formula. The maximum period P of x_i is $P = 2^n - 1$. When the period is maximum the sequence is called a maximal length linear recurring sequence or m-sequence.

For the sequence to be an m-sequence the polynomial

$$f(x) = 1 + a_1 x + a_1 x^2 + ... + x^n \qquad (2)$$

must be primitive over the Galois field of order 2 [109]. Zieler [110, 111] has compiled lists of primitive polynomials.

Equation 2 is known as the characteristic equation. For computational ease the characteristic equation is normally a primitive trinomial. When the period is prime, P is known as the Mersenne prime and n is termed the Mersenne exponent.

5.8.1.1 Decimation of m-sequences

Decimation is an important principle used in newer shift register pseudorandom number generators. Let x be an m-sequence. Let x(k) be a sequence generated by sampling every kth bit of x. This operation is termed decimation of order k [112].

If k and P are coprime, $(k, 2^n - 1) = 1$, the decimation is known as proper decimation and x(k) is also an m-sequence with the same period. If $k = 2^a$ $(a \in I)$, x(k) is a shifted version of x.

Let y be a binary m-sequence such that y = x(k). It is possible to determine the sequence x by decimation of y. The order of the decimation should be such that [113] for $k = 2^a$

$$m = 2^{n-a} \qquad (3)$$

Consider two m-sequences x1(k) and x2(k) generated by sampling every kth bit of x starting with the first bit for x1(k) and the second bit for x2(k). The sequences x1(k) and x2(k) represent the same m-sequence with a phase shift between them. From equation 3, if we assume x2(k) is delayed with respect to x1(k) by d bits, d can be shown to be

$$d = 2^{n-a}\,; \qquad \text{when } k = 2^a$$
$$d = 2^n/k\,; \qquad \text{when } k \neq 2^a$$

We can generalise this. Let $x_m(k)$ be the kth decimated sequence starting with the mth term of x. The phase shift between x1(k) and $x_m(k)$ is

$$d = m/k \ (\mathrm{mod}\ P) \qquad (4)$$

This result has been used extensively in the development of linear feedback shift register pseudorandom number generators.

Tomlinson et al [114] compared the m-type weight distributions of an m-sequence based on a primitive trinomial and its decimation by consideration of a third central moment which is a measure of skewness and is zero for symmetrical distribution. They have shown that the third central moment can be very much less for the decimated sequence than for

the trinomial sequence.

5.8.2 The generation of random numbers from linear feedback shift register sequences

Most of the techniques for producing random numbers from m-sequences rely on the fact that a collection of N independently derived random binary digits when assembled in an N-bit word may be interpreted as a number provided a suitable weighting is applied to each bit in the word [115]. In this way a number B_i in the range (0, 2^n - 1] may be created. Assuming each digit is generated by a source which produces 0's and 1's with equal probability the N-bit composite number will be random and uniformly distributed over its range. The difficulty lies in producing and N-bit number from n independent sources. Using N stages of an n stage shift register presents itself as a simple solution but it is undesirable as the generated numbers are cross correlated. It is therefore desirable to spread the pick off positions by shifting the N digits adequately relative to each other. Two techniques are proposed to generate these n shifted version of the m-sequence

i) Linear Combination of the n register stages of the feedback shift register.
ii) Decimation of the m-sequence

The former requires the calculation of the proper stages to be added modulo 2 to get the required shifts. The latter is simpler but slower due to the sampling. For n shifted versions of the m-sequence to be independent the shift d must be high enough to ensure there is no overlapping of the shifted subsequences. This calls for the use of high degree polynomials representing the feedback shift register. In this way the generated pseudorandom number sequences are uncorrelated to shifts up to d. Techniques were proposed which use N different m-sequences to generate N-bit pseudorandom numbers but they received little attention because of the correlation between the individual bits [116].

5.8.3 The Tausworthe generator

Tausworthe [117] proposed a method of pseudorandom number generation by decimation

of m-sequences produced by a feedback shift register. The advantages of this method is the theoretical guarantee of good properties of randomness. Tausworthe showed that the numbers produced had good mean, variance, autocorrelation and uniformity properties.

Tootill et al [118] observed however that these results applied to global characteristics and give no insight into local behaviour. Local behaviour in our application is critical as we will only use relatively small subsequences for any given application. They investigated the runs properties of k-decimated generators based on the primitive trinomials ($x^n + x^k + 1$) and their compliment ($x^n + x^{n-k} + 1$) and showed that sequences of k-bit numbers may have good runs and uniformity properties provided the following criteria are met

i) The greatest common divisor between (km) and P is 1 for m = [n/k].
ii) k must be less that n/2.
iii) k is neither too small nor too close to n/2.

Provided neither k nor n was too small a generator meeting the above requirements has predictably good runs properties. The main disadvantage of the Tausworthe generator is its slow speed due to the sampling required for decimation. It does however remain a widely used generator due to its theoretical randomness properties.

5.8.4 The Lewis Payne generator

Lewis and Payne [119] proposed a generalised feedback shift register generator (GFSR) algorithm capable of producing long sequences of pseudorandom numbers which may posseses m-space properties for any word size of machine. Their algorithm is commonly used in cryptography where it as known as the TLP algorithm [120]. Their algorithm relies upon shifted versions of the m-sequence and involves the introduction of delays between words.

The GFSR algorithm is again based on a primitive polynomial, generally a trinomial ($x^n + x^k + 1$). A recurrence relation is used to produce a pseudorandom m-sequence b_i. The algorithm is initialised by the selection of suitable initial words. If N-bit words are desired, the GFSR sequence is initialised by setting the elements of b_i into N columns with a suitable delay between adjacent columns.

This generator is fast and widely used. There exists efficient software implementations including Hamilton's [121]. Tests on this generator demonstrated reasonable confidence in the properly stochastic nature of the number generated though for this as in most shift register generators the length of period dictates that no existing computer could run close to full period tests without reaching obsolescence and therefore only relatively small subsequences have been tested.

Major disadvantages of this generator are the computation involved in its initialisation and the fact that there is no theoretical assurance of m-distributivity. Fushimi et al [122] established a sufficient condition for the GFSR sequence to be m-distributed and also proposed a relatively time efficient algorithm for testing the m-distributivity. Fushimi's theorem is an advance on the GFSR algorithm. Another remaining disadvantage is that since the relative delay between columns is suggested to be 100n, it cannot deliver more than 100n numbers without correlation problems.

5.8.5 The Split-up feedback shift register generator

Arvillas and Maritas [123] have shown that the m-sequence based on the primitive trinomial $(x^n + x^k + 1)$ can be generated a splitting up a feedback shift register composed of k-toggle and (n-k) shift elements. They state that for k being coprime to (2^n-1) the split up shift register can generate, in parallel, k m-sequences each of which is a kth decimation of the base trinomial sequence. These m-sequences are Tausworthe sequences with statistical independence ensured over a length equal to $[(2^n-1) / k] -1$.

This generator was first proposed as a hardware implementation of the Tausworthe sequence. Arvillas and Maritas suggested that for efficient software implementation using split-up feedback shift registers based on the trinomial $(x^n + x^k + 1)$ where $k = 2^i$; $i \in I$ [124]. Here the k outputs of the leading stages of the sub-registers are phase-shifted versions of the basic m-sequence based on the trinomial.

The main advantage of this technique is that it gives fast algorithms. However the statistical properties can vary greatly for sequences of the same length but having different characteristic polynomials.

5.8.6 The Barel generator

Barel [125] proposed a hardware Tausworthe pseudorandom number generator based on n-wise decimation of primitive trinomials ($x^n + x^k + 1$). His method is advantageous in that it is very fast and independent of the number of bits or the trinomial characteristics. Barel's algorithm is given below

i) Start with an initial seed, say B'_t

ii) Store B'_t in locations A and B

iii) Shift contents of B k positions to the right, replacing the contents of bits 1 to k by zeros.

iv) Add contents of A and B modulo 2

v) Store result in A and B

vi) Shift contents of B (n-k) positions to the left, replacing the contents of bits K+1..n by zeros

vii) Add contents of A and B modulo 2

viii) Store result B'_{t+1} in A and B

ix) Repeat from step (iii) to generate next number

Barel's hardware implementation of this algorithm is very fast as it replaces all n bits of the register in parallel. It is also very component efficient as it requires only two registers and two layers of exclusive OR gates but it is very powerful.

Barel used this generator to test three primitive trinomial generators giving 31 bit random numbers and obtained satisfactory results. However it is significant that the number of dimensions m achieved by these types of generators will never exceed one. From Tausworthe's theory the number of dimensions is equal to [n/q] where q is the order of decimation. Since n = q, then m = 1. Tootill et al [118] have shown that the number of dimensions influences the runs up and down properties and hence Barel's generator will not give good results in this respect.

5.8.7 A proposed software fast Tausworthe generator

Talib [7] proposed a hardware Tausworthe generator which was a development of Barel's but used k-wise decimation to give a higher number of dimensions. Here we develop Talib's proposals and suggest a fast software Tausworthe generator.

Let b_i be an m-sequence based on the primitive trinomial $(x^n + x^k + 1)$in which $(k,P) = 1$ and $k < n/2$ (Tootill's first and second criteria [118]). Let $S_0 = (b_{n-1} b_{n-2} \dots b_2 b_1 b_0)$ be the starting n-tuple of the sequence. By applying the base recurrence relationship to S_0 we get the following succession of sequences

$$S_1 = (b_0 \oplus b_k)\, b_{n-1} \dots b_2\, b_1$$

$$S_2 = (b_1 \oplus b_{k+1})\, (b_0 \oplus b_k) \dots b_3\, b_2$$

...

$$S_k = (b_{k-1} \oplus b_{2k-1}) \dots (b_0 \oplus b_k)\, b_{n-1} \dots b_3\, b_2$$

If A and B are integer variables then S_k can be generated from S_0 by the following algorithm

i) $A = B = S_0$
ii) $B = B \text{ DIV } 2^k$
iii) $A = A \oplus B$
iv) $A = A \text{ MUL } 2^{n-k}$
v) $A = A \oplus B$
vi) $S_k = B = A$

This process can then be repeated (replacing S_0 with S_k in the first step) to generate S_{2k}, $S_{3k}, \dots, S_{tk}$ for $t = 0,1,2,\dots$. S_k is then a Tausworthe sequence based on k-wise decimation.

This generator produces the same sequence as the split-up feedback shift register type but does not require any computations to determine the sub-register lengths.

Our algorithm is similar to Barel's but since we use k-wise decimation we have $m = n/k$ dimensions of uniformity. By choosing k to be not too small and $k < n/2$ (Tootill's third criteria) we can satisfy the requirements for good runs properties. Note that this generator produces a k bit random number. The value of k must be of an order such that the k-bit accuracy is suitable. If a greater bit degree of accuracy is required then Barel's generator

may be used as it gives n-bit random numbers on each iteration.

5.8.7.1 Software implementation

With the use of bitwise arithmetic and replacing the DIV and MUL operations with shift left and shift right respectively we can easily implement this algorithm in software. One obstacle is that we would wish to use trinomials with a high order of n to give good randomness properties. In most computers register and therefore software variable lengths are limited to 32 bits. To overcome this we must concatenate variables and write routines to model registers of the required length and this extra processing will have a detrimental effect on the speed of the generator. Accepting this, the algorithm lends itself to efficient software implementation and the use of shift rather than divide and multiply operations will give and enhancement in computational speed in comparison to Lehmer generators.

5.8.7.2 Testing

The algorithm was implemented in the C programming language and tested by analysing a number of generated sequences. The starting seeds for each sequence were always chosen such that the delay between different sequences was always greater than the length of subsequence being analysed. The same statistical tests were used as for the Park and Miller generator and again a range of subsequences were tested with lengths ranging from 500 to 5000 samples to be representative of typical I_SIM applications.

Talib [7] tested a number of characteristic polynomials for his proposed hardware generator. We used his recommended optimal trinomial ($x^{47} + x^{14} + 1$). We also implemented and tested the trinomial ($x^{31} + x^{11} + 1$) and its compliment ($x^{31} + x^{20} + 1$) as recommended by Miller et al [126] for linear feedback shift register generators.

Best results were obtained for the trinomial ($x^{47} + x^{14} + 1$). These results varied depending upon the initialisation seed. Generally results were acceptable at both the 10% and 2% levels, falling within the limits of an ideal generator. However, on occasions failures at the 10% level reached as high as 16.5% and at the 2%level, 7.5%. Clearly the subsequences that gave these results are unacceptable. Conversely we obtained some

exceptionally good results with failures falling as low as 1% at the 10% level and 0 at the 2% level. From these results it may be concluded that the generator gives generally acceptable results but the choice of initial seed for relatively shot runs is critical.

As was discussed earlier, what is required for I_SIM is an efficient generator with good local statistical properties over short subsequences and a table of 500 seeds to initialise such subsequences. Our new generator matches these requirements provided the table of seeds is suitably determined.

5.8.7.3 Timing of the generator

Theoretically a software implementation of the new generator should be faster then the Park and Miller Lehmer generator as it replaces the computationally intensive divide and multiply operations with shifting. Timing tests were conducted on both generators with each going through an equal number of iterations. The results are given in table 5.7.

Amount of Numbers Generated	Time Taken (seconds) New Generator	Park and Miller
10^6	5	21
10^7	49	209
10^8	489	2083

Table 5.7 Timing of the Generators

The results reflect the theoretical predictions with the new generator being 4.25 times faster than the implementation of the Park and Miller generator. Note these tests were run on a 100MHz computer to give an idea of scale relative to other machines. This is a significant result for if we were to produce a Lehmer generator with a longer period the speed of generation would increase. However even though we have decreased the speed we have increased the period by $2^{41} - 1/2^{31} - 1 = 2^{10} = 1024$ times.

For interest, the time taken to generate one row of the parallelization matrix, ie a single complete subsequence of length 2^{24} numbers was 80 seconds.

From these timing results and the statistical results discussed in section 5.8.3.2 it can be seen that our generator meets all of L'Ecuyer's [127] properties for a good generator. These are: Good statistical properties [128]; long period [129]; speed; low memory usage [130]; portability [131]; reproducibility [132] and splitting facilities (ease of parallelization).

5.9 The generation of seeds to produce parallelized random number sources

Some work has been done on the parallelization of random number sources. Durst [133] suggests the use of random seeds whereas Makino [103] has proposed a formalised means of producing a seed matrix whose rows are equipartitioned subsequences of the original sequence. L'Ecuyer and Côté support this approach in their random number package [105] and it is also suitable for our generator. It is not always applicable to linear congruential generators where bad long range correlations can occur between subsequences when the modulus and length of subsequence are both powers of two. This is why Durst prefers random seeds. By number theoretical argument Makino produced general conclusions which depend upon the method of parallelization and the period of the source sequence.

It was decided to follow Makino's conclusions to produce an optimal matrix of subsequences for our generator. This method satisfies Deng et al's criteria listed in section 5.4. Since each row of the matrix gives an independent subsequence , the first column of the matrix may be used as a table of possible initialisation seeds.

The matrix may be produced to give either a vertical or horizontal configuration, here we are considering the horizontal case. Properties of the original sequence will be reflected in each row. A row is continued on the following row so that the delay between subsequences is the row length, v. The two dimensional properties of such matrices have been studied [134] where the numbers may be allocated to components in the orthogonal direction. The statistical properties of these subsequences are not guaranteed. For the proposed application orthogonal allocation would not be used. Our only concern would be the allocation of numbers to an amount of components equal to the number of rows in a constant uniform manner. In this rare but possible instance the delay between orthogonal subsequences would be important. However this would only be a problem if the final row

of the matrix was continued with a re-run of the original sequence to produce extra rows. Since the period of our generator is very long (2^{41} - 1) there will be more than a sufficient amount of numbers to give enough rows of good length and this will not be a problem.

Dimensioning of the matrix

Let P be the period of the generated sequence and v the chosen length of a row. The number of rows, μ, will then be

$$\mu = \lceil P/v \rceil \qquad (5)$$

where $\lceil x \rceil$ is the least integer greater than or equal to x. The rows will be of equal length save for the final row which will be shorter than the others by k

$$k = \mu v - P \qquad (6)$$

terms. Note that for a shift register generator of period P = (2^n - 1), if v is chosen such that it is a power of 2 then k will always be equal to 1. From this it is apparent that all rows, including the final row will be subsequences of suitable length.

The value of v is equal to the delay, d, between subsequences and as such must be chosen to be suitably high. It was decided to set the value of v to be 2^{24}. Using equation 5 to calculate μ, the number of columns

$$\mu = \lceil P/v \rceil = \lceil 2^{41} - 1 / 2^{24} \rceil = 2^{23}$$

Note that this is the maximum number of subsequences and as we have seen some initial seeds produce unacceptable subsequences. To produce the required 500 initial seeds software was written to generate a number of rows of the parallelization matrix. The first column was noted and the values from this column were used as initialisation seeds for the generator. Subsequences were produced of suitable length for analysis. The statistical tests were run on each subsequence for sample lengths of 1000. A minimum of ten complete runs of tests were conducted for each subsequence to give properly indicative results. From these results a table of 500 initial seeds was selected from the first column of the matrix. Seeds were only selected if they produced subsequences with statistical properties

matching or better than the ideal ie $\leq$ 10% failures at the 10% level and $\leq$ 2% failures at the 2% level.

5.10 Conclusions

In this chapter we have reviewed the use or random number generation within I_SIM and proposed a fast software pseudo-random number generator.

The initial generator used was the popular Park and Miller generator. This has previously been shown to be statistically good over long runs, however within I_SIM a generator must be used that demonstrates good statistical properties over various run lengths. In the first implementation, one generator was used within I_SIM and numbers produced were allocated to stochastic and probabilistic transitions during the simulation as required. The result of this was the creation of sub-streams which are generated by randomly sampling the parent sequence. Investigation showed that these sub-streams were in some cases not as statistically good as the parent sequence.

What is of concern is the effect of the random sequences on an I_SIM simulation. Through actual ICE examples it was shown that sub-streams with less than ideal statistical properties could affect the simulation. However this is not always the case and it is impossible to predict. For these reasons a new method of allocation is suggested whereby each stochastic or probabilistic transition is allocated a unique seed for the parent sequence. All streams of numbers then allocated are space shifted versions of the parent sequence.

The Park and Miller generator is a linear congruential generator and as such there is a notable computational overhead occurred in the generation of numbers. Pseudo-random generators of the shift register type previously used within hardware reliability simulators were also considered. We have proposed a fast software implementation of a shift register generator based on one of these hardware generators. It has been shown to be of a similar statistical standard to the Park and Miller generator but over four times faster. For this reason this proposed generator has been selected for use within I_SIM.

Chapter 6

Computational models for ICE

6.1 Introduction

In this chapter we consider computational models for ICE. The purpose of this is twofold, firstly to produce a formalised definition of the language's semantics and secondly to compare ICE to a recognised modelling technique.

In chapter 3 it was shown how the language's simple syntax can hide some complex semantics. A modeller may become competent with ICE and begin to apply it within a few hours but for it's full power and flexibility to be utilised the underlying semantics must be grasped. With the ethos during development being to keep the syntax as intuitive as possible, consideration of this without the semantics would most likely give an incomplete rather than erroneous understanding.

It was decided to adopt Petri nets, described in section 2.4.3, for the above purposes as they are well documented and currently the most widely utilised technique for modelling real time concurrent systems and in particular performability modelling. By producing a detailed comparative study a credible recognition is obtained of ICE's relative power.

The chapter has two main sections. The first considers each ICE construct and builds up an equivalent CGSPN model. We then go on in the second to identify some recurrent problems in systems modelling and propose ICE solutions to these.

6.2 COMPONENT

The language is primarily concerned with interacting components, each of which will exist in one of a number of states at any given instance in time. We may model a component with a Coloured Petri Net (CPN). Each place will represent a state. A CPN representation of a simple two state component is shown in figure 6.1. An explicitly defined colour *COMP* with token *c* is used to show the current state of the component. The component is shown in state *idle* which is also the initial state. The arc expressions and transition markings would be determined by the component's BEHAVIOUR statement.

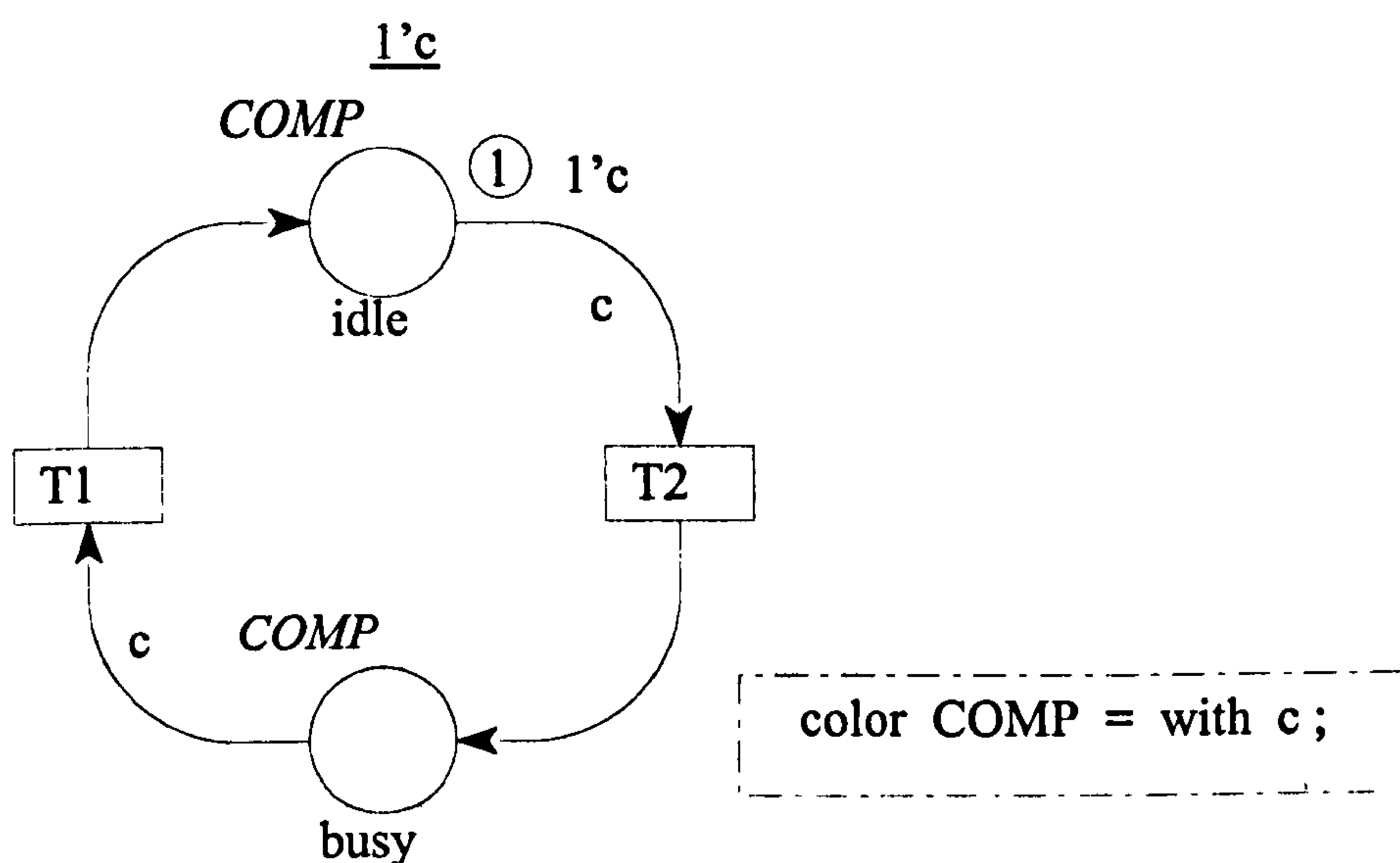


Figure 6.1 Component

6.3 CONSTANTS

Constants may be defined as distinct colours using CPN ML. Each constant would be a subset of the basic colour type int. An example of a constant and the equivalent colour is

```
CONSTANT = { VALUE = 4 }      <==>      colour VALUE = int with 4;
```

6.4 STATE_SETS and COUNTERS

A STATE_SET defines the finite set of states that a component may exist in. Each state may be represented by an individual place in a CPN. One token of a defined colour will be shared between all state places. The current state will be indicated at any time by the place which contains this token. An example of this was seen with the token *c* in figure 6.1. Counters may also be represented by places within the same CPN. This highlights a significant advantage of using CPNs rather than PNs to model the language. With PNs counters would have to be modelled as separate nets or sub-nets interacting with the component nets. The use of different coloured tokens in CPNs allows us to model using one net, the token colours distinguishing the functionality of the components and the counters.

Counter places will be connected to state places in a manner controlled by the counter definitions within a STATE_SET. The arc expressions will reflect the arithmetic statements of the counter modifiers. As an example, the statement

```
STATE_SET   comp_states {
  COUNTERS   count_a, count_b, count_c ;
  STATES {
    state_1  :  { count_c = count_a + 2 };
    state_2  :  { count_b = count_b - 1 };
    state_3  :  { count_c = count_c / 2 };
  }
}
```

may be modelled by the CPN shown in figure 6.2.

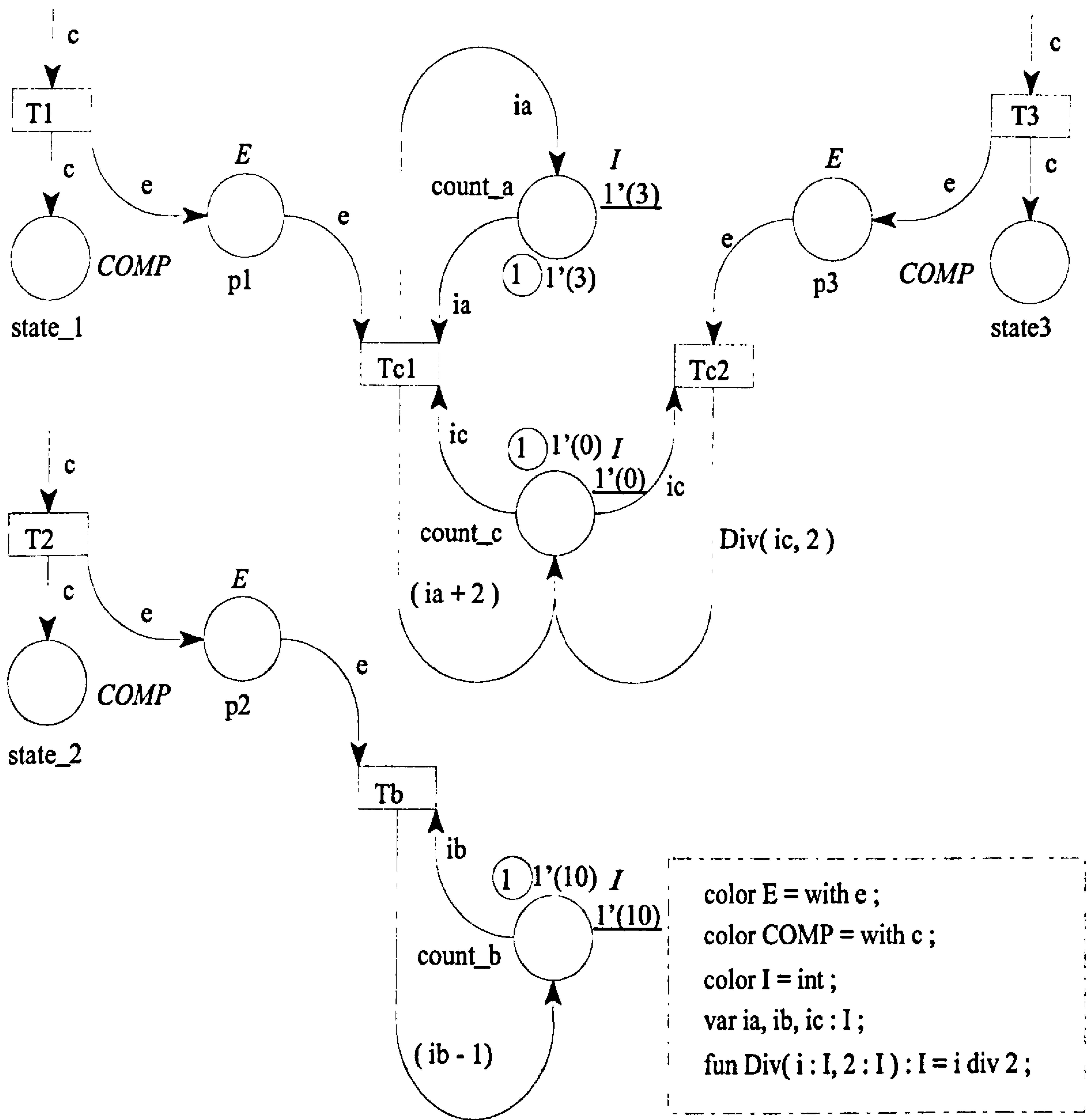


Figure 6.2 Counters

Component and counter places are marked by *C* and *I* colour tokens respectively. Counter values are given by the value of the integer tokens attached to the counter places. Updates to these values are denoted by the counter transition post-arc expressions which reflect the counter modifiers in the STATE_SET expression. The initial values of counters are component dependant and can be marked on the counter places. In this example the initial values of count_a, count_b and count_c are 3, 10 and 0 respectively.

6.5 SYSTEM

One way to represent a system statement by a CPN is to use a combination of integer tokens as counters and Boolean tokens to monitor the state of the constituent Boolean functions. Figure 6.3 shows the CPN which can be used to model the statement

SYSTEM sys.all = ALL { comp1.state1, comp2.counter = 2 } ;

A single *u* token of colour *Cond* in one of the places *sys_true* or *sys_false* reflects the condition that *sys_all* is either true or false. The place *list* with token *i* of colour *integer* keeps a count of the number of expressions within the system expression which are currently true. When *comp1* moves into *state1* a *c* token is put in place *state1* and is removed when *comp1* leaves *state1*. The entering and leaving transitions *T1* and *T1'* also put enable, *e*, tokens in places *c1_t* and *c1_f* respectively. These tokens will enable transitions *T1t* and *T1f* which, when fired, will update the *list* counter by modifying the value of the *list* place token *I*.

The *list* counter must also be updated to reflect the condition of the expression *comp2.counter = 2*. When any transition fires that will change the value of token *cnt* in place *counter* (used to model the value of *comp2.counter*) e.g. *T2* a token *cnt* of colour *int* and value equal to the modified *counter* token *ic* is put in place *cnt_mod*. This token will enable one of the two transitions *Tct* and *Tcf*. Which one is enabled is determined by the guard expressions. We will consider the case when *Tct* is enabled, the complementary case of *Tcf* being enabled is very similar. Transition *Tct* will fire and place an *e* token in place *cnt_t*. If there is also an *e* token in place *enable_t*, *Tc1* will fire putting an *e* token in place *cnt_1*. An *e* token in this place makes it possible to move a *u* token between places *false* and *true*. The place that the *u* token is currently in reflects the condition of the expression *comp2.counter = 2*. If the *u* token is currently in place *false* then along with the *e* token in place *cnt1* it will enable the transition *Tc_ft*. This transition will fire, removing the *u* token from *false* and placing one in *true*. It will also place an *e* token back in *enable_t*. If however the *u* token is already in *true* the counter expression must already be true. The inhibitor arc which disables transition *Tc3* when there is a *u* token in *false* would now, along with the *e* token in *cnt1*, enable *Tc3*, which would then fire and place an *e* token back in *enable_t*. Thus, in this second instance, the change in counter value has no effect on the value of the counter expression. When either transition *Tc_tf* or *Tc_ft* fires, signifying a change in the value of the counter expression, an *e* token is placed in either place *ct* or *cf* to

enable transitions *T2t* and *T2f* respectively. These transitions, once fired, will update the value of the *list* counter.

When any of the transition *T1t, T1f, T2t* or *T2f*, that change the value of the *list* counter fire, they place a token *si* of colour *integer* in place *sys_mod.* The value of *si* will be the same as that of token *li,* the *list* counter. This token will enable one of the transitions *Tst* or *Tsf.* Which one is enabled will be determined by the guard expressions which reflect the condition of the overall system statement. These transitions will place an *e* token in either *sys_t* or *sys_f.* Tokens in these places allow the overall system expression to be modified, if required, by placing a single *u* token in either *sys_true* or *sys_false.* The enabling mechanism is the same as that used in the part of the net which models the *comp2.counter = 2* expression.

The example given above was for a system statement of type ALL. We can also utilise system statements of type ANYn and EXACTLYn. A very simple modification of the CPN in figure 6.3 allows these different types of statements to be modified. All that need be altered are the guard expressions on transitions *Tst* and *Tsf.* Table 6.1 shows the guard expressions which would be used in each instance.

System Statement	Transition Guard Expressions	
	Tst	Tsf
ALL	I = 2	I <> 2
ANY1	I > 0	I = 0
EXACTLY1	I = 1	I <> 1

Table 6.1 Transition Guard Expressions for different System Statements

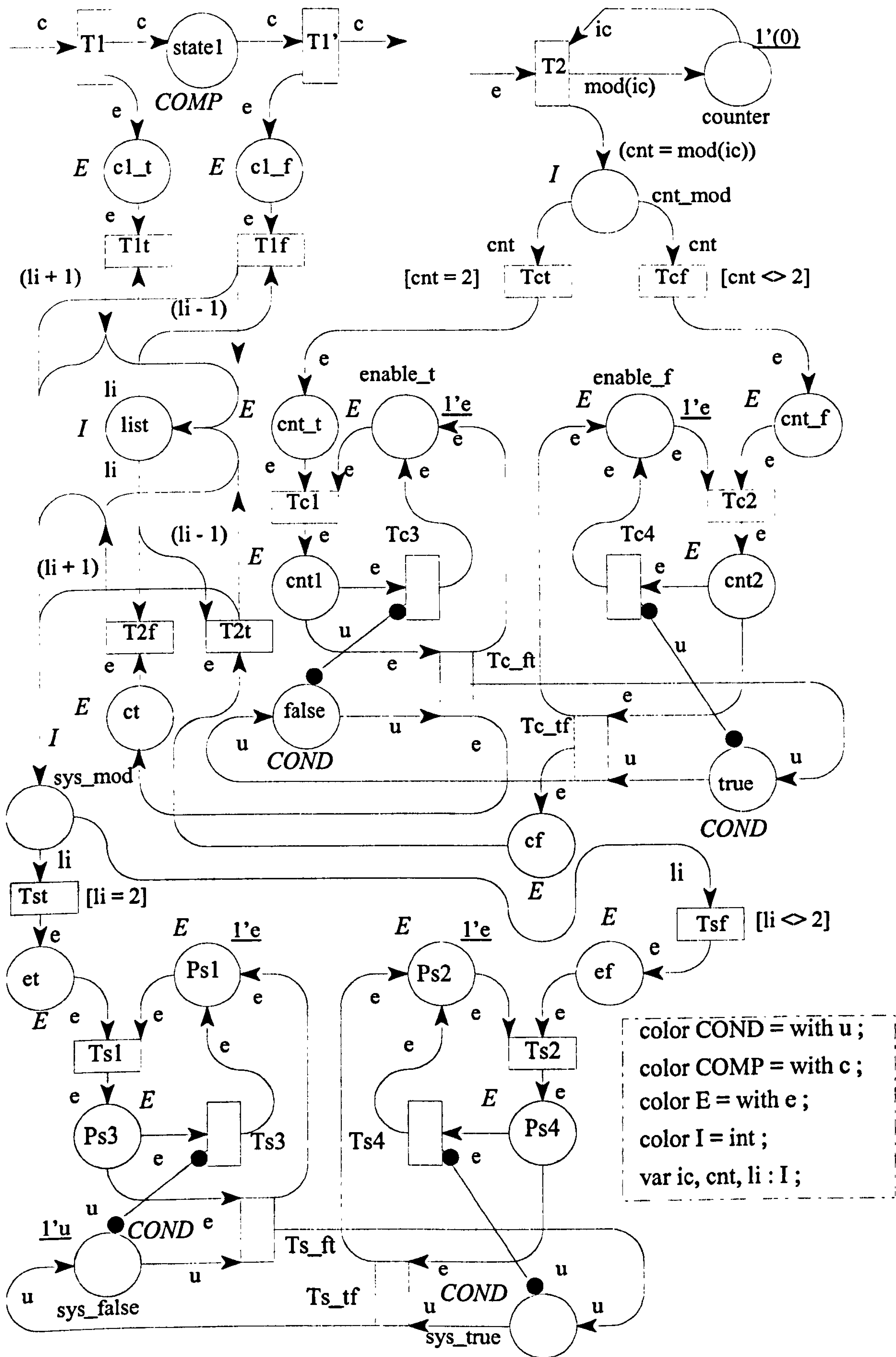


Figure 6.3 SYSTEM

6.6 RESOURCES

When representing RESOURCES and STOCK with CPNs we must consider both their definition and allocation. The resources defined in the statement

RESOURCE { res_a : 2 ; res_b : 4 } ;

may be represented by the CPN in figure 6.4.

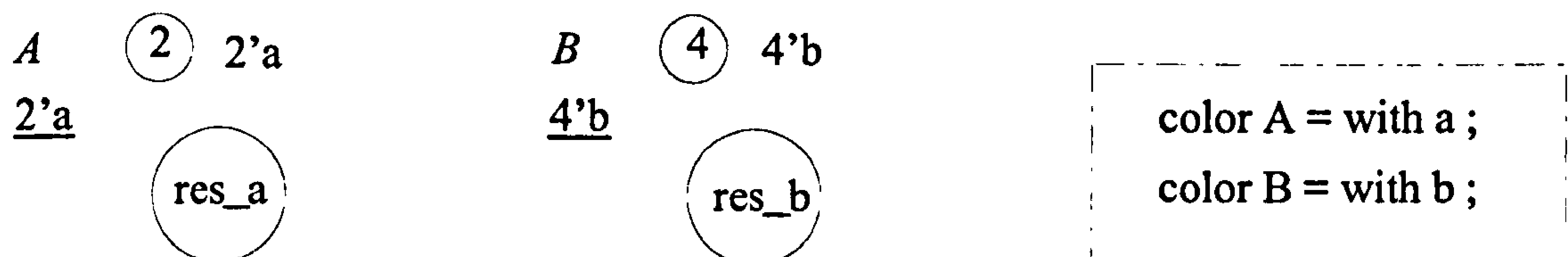


Figure 6.4 Definition of Resources

The places *res_a* and *res_b* hold the free resources. Token colours have been defined for each resource type. The initial marking will always reflect the number of resources given in the resource statement, whereas the current marking will be modified as the CPN is executed.

The allocation of resources may be represented as transition post-arcs. The non-consumable nature of resources can be reflected by the use of post-arc 'feedback loops'. Figure 6.5 shows a CPN representation of the statement

```
ON_RESOURCE res_a = 1, res_b = 2 {
  state1 -> state2 } ;
```

The transition *T1* is only enabled when the component is in state1 and the required number of both *res_a* and *res_b* resources are available (1 and 2 respectively). When *T1* fires it removes this number of resources from *res_a* and *res_b* and puts an equivalent number of each in place *state2*. This operation is detailed in the pre and post arc inscriptions of *T1*. When the component leaves *state2*, represented by the firing of transition *T2*, the resource tokens must be returned to the resource places. This is shown on the post-arc inscriptions of *T2*.

Stock may be modelled in a similar way. Stock however is consumable and therefore there would be no returning of stock to the free stock places.

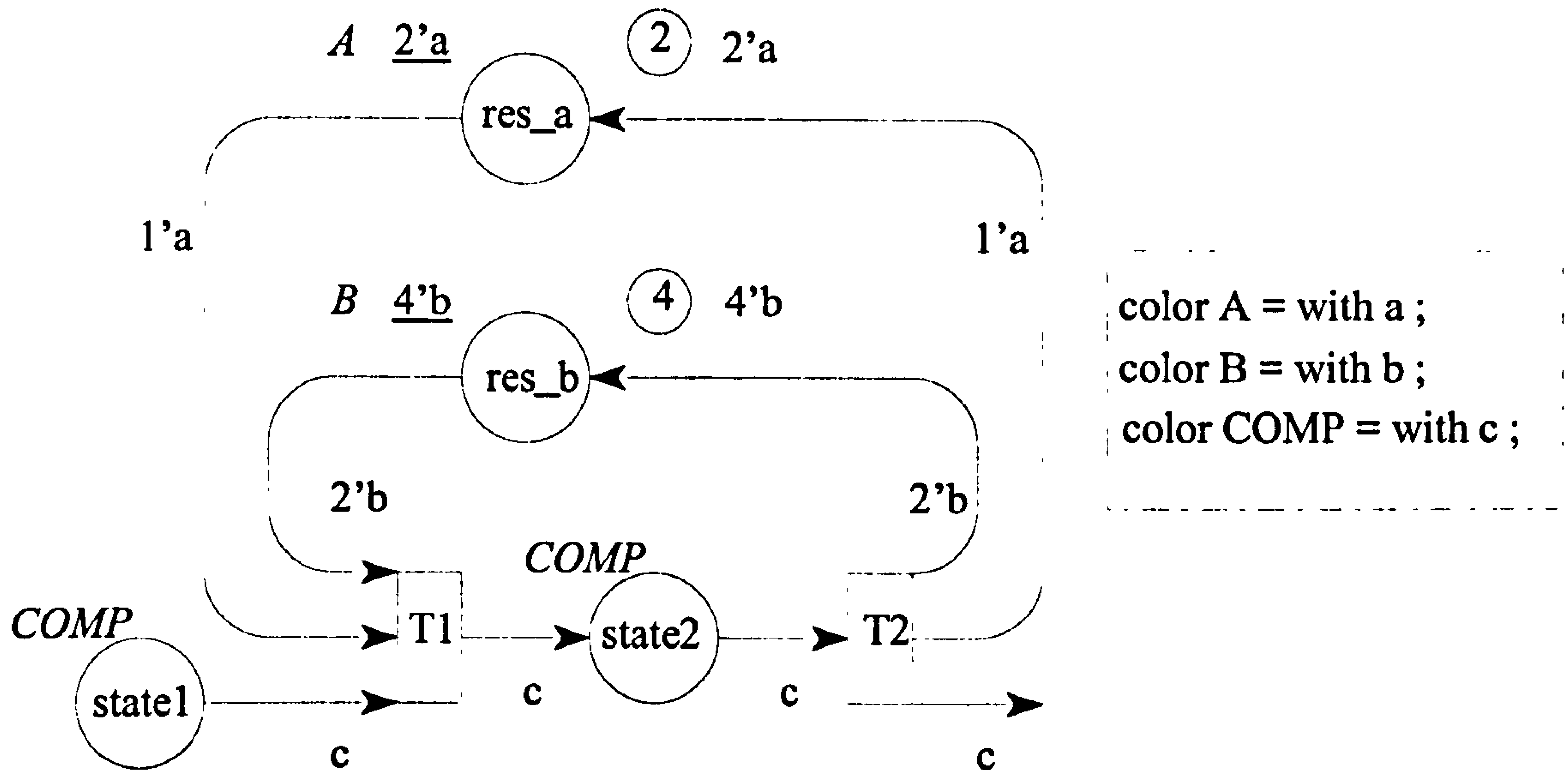


Figure 6.5 Resource Allocation

6.7 BEHAVIOUR

Behaviour statements can comprise of a number of different transition statements which may have varying preconditions. We will first consider the transitions and then go on to look at the preconditions.

6.7.1 Timed probabilistic transitions

Timed probabilistic transitions can simply be modelled by transitions in CPNs. For example, figure 6.6 shows a CPN which represents the statement

```
7       state1 -> state2  PROB(0.7) ;
exp(2)  state1 -> state3  PROB(0.3) ;
```

The transition inscriptions show the firing times and the probabilities are shown on the pre-arc inscriptions.

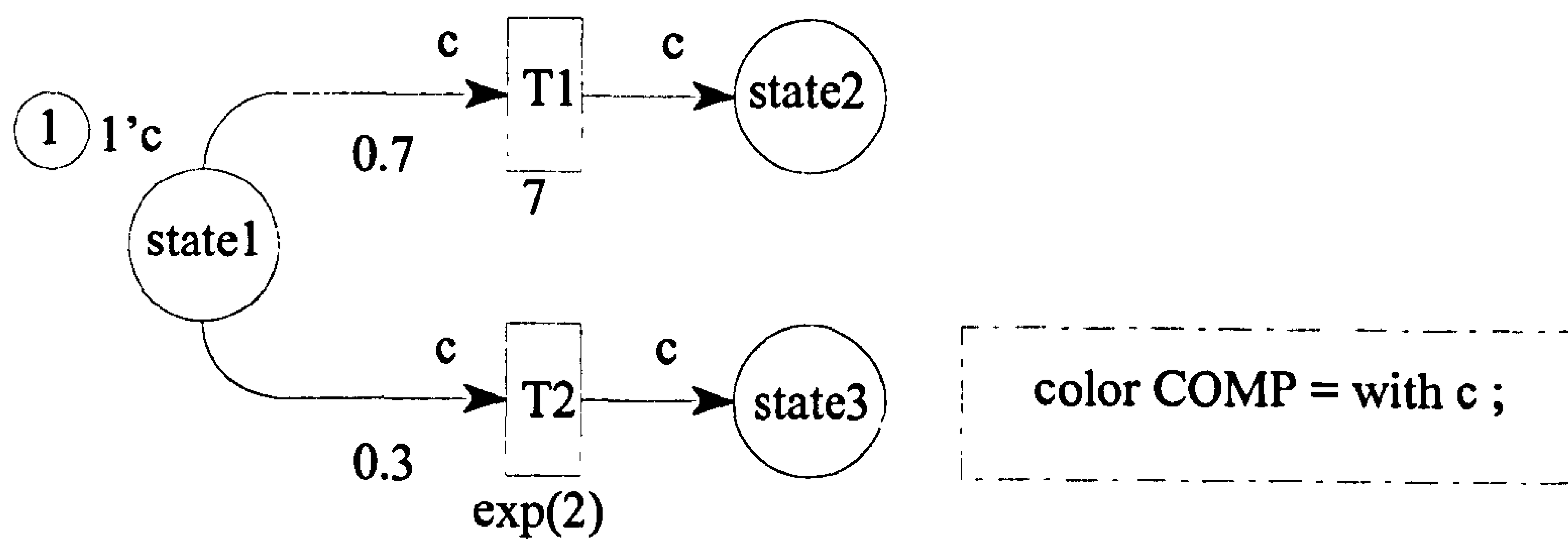


Figure 6.6 Timed Probabilistic Transitions

6.7.2 Transition firing policies

When modelling the different transitions within a behaviour statement the transition firing policy of the equivalent CPN becomes critical. Various policies have been suggested and [32] gives a detailed analysis.

The issue that is of most concern to us is when exactly a timed transition is fired. There are two broad possibilities

1. When a transition is enabled the firing delay is started. Once this time has elapsed the transition is fired, removing the enabling tokens from the input places, as listed in the pre-arc expressions and places the tokens listed in the post-arc expressions are immediately put in the output places. If during the time the firing delay is elapsing the enabling tokens are removed from the input places, the transition is no longer enabled and will not fire.

2. As soon as a transition is enabled it fires, removing the enabling tokens listed in the pre-arc expressions from the input places. Tokens are placed in the output places as listed in the post-arc expressions but do not become active until the period of the firing delay has elapsed.

The significance of the difference between these two possibilities with respect to model behaviour is best illustrated by a simple example.

Part of a behaviour statement

```
5  state1 -> state2 ;
ON_EVENT comp2.state1 {
  2 state1 -> state3 ; }
```

is modelled by the CPN in figure 6.7. Note that the control logic of the ON_EVENT precondition is not shown.

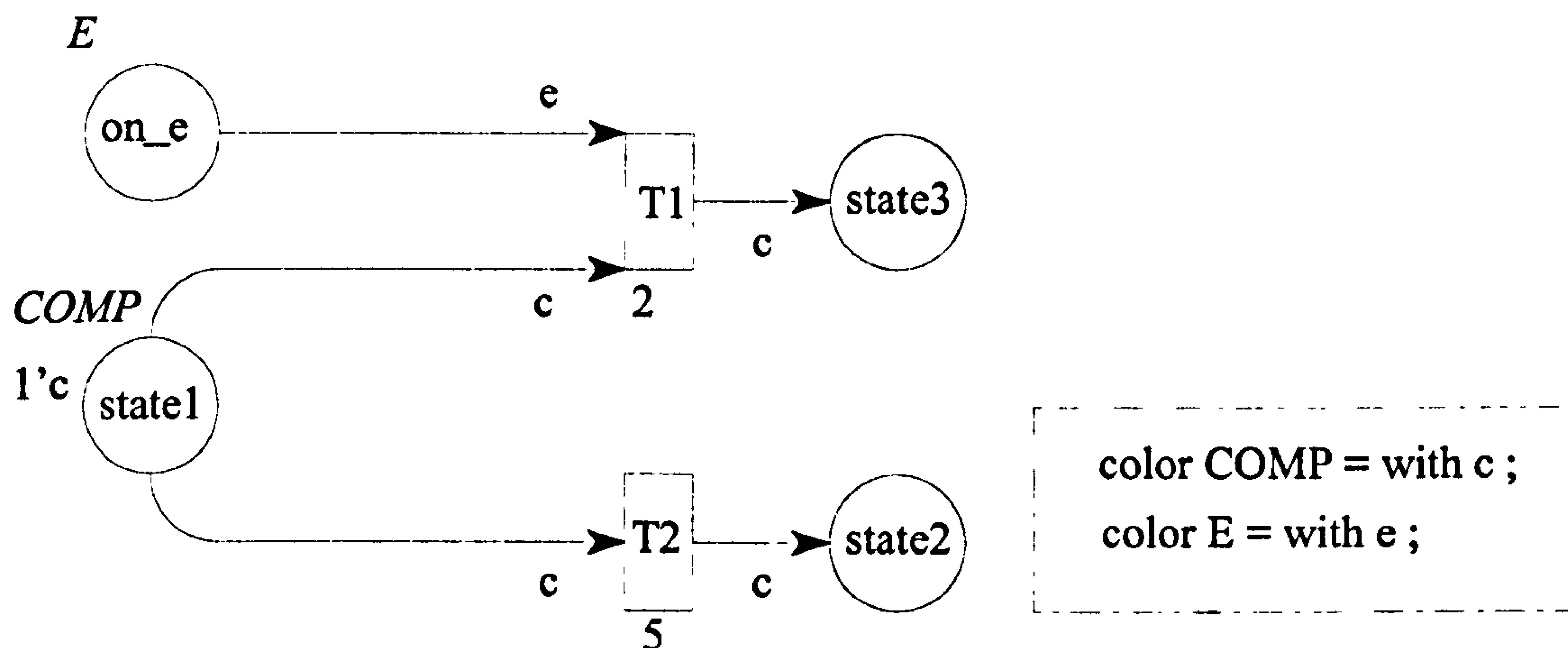


Figure 6.7 Transition Firing Policies

Using the first firing possibility the model would be correct, however using the second would cause an error. If at time $t_1 = 0$ units a *c* token is put in place *state1* but there is no *e* token in place *on_e*, transition *T2* will be enabled but transition *T1* will not and therefore *T2* will fire, removing the *c* token from *state1*. If then at time $t_2 = 1$ unit an *e* token is put in place *on_e*, *T1* will not fire since it is still disabled due to the *c* token having been removed from *state1*, thus the ON_EVENT will not be executed. Clearly this is incorrect as the ON_EVENT transition would be completed at time $t_3 = t_2 + 2 = 3$ units before the first transition which would finish at $t_4 = t_1 + 5 = 5$ units and should then take priority.

This example would seem to dictate that the first firing possibility be adopted, however it is the second that is now proving more popular in CPN modelling and simulation tools eg SymNet. It would then be desirable to implement a solution which utilises the second possibility. This may be done by using test arcs. Test arcs for CPNs have been formally defined in [41].

A test arc behaves in a similar manner to a normal arc but they are conservative and several test arcs may access the same enabling tokens concurrently. They may not however access enabling tokens at the same time as normal arcs. Test arcs cannot change the marking of a

place. Figure 6.8 shows a solution to the timing problem utilising the second firing possibility by using test arcs.

If a time $t_1 = 0$ units a *c* token is placed in *state1*, *T1* will be enabled and fire, but since the pre-arc is a test arc, the token will not be removed from *state1*. If then at time $t_2 = 1$ unit an *e* token is put in *on_e*, *T2*, will be enabled and fire. *T2s* firing time will elapse first at time $t_3 = t_2 + 2 = 3$ units and an *e* token will be put in place *P2*. This will enable *T6* which will fire, removing the *c* token from *state1* and placing a *c* token in *state3*. When *T1s* firing time elapses at $t_4 = t_1 + 5 = 5$ units it will place an *e* token in *P1*. Since the *c* token will already have been removed from *state1*, *T3* will be disabled and instead *T4* will fire to clear *P1* and place an *e* token back in *Pe1*. The inhibitor arc on transitions *T4* and *T5* will prevent any conflict. *T4* and *T5* have no effect on the state of the component.

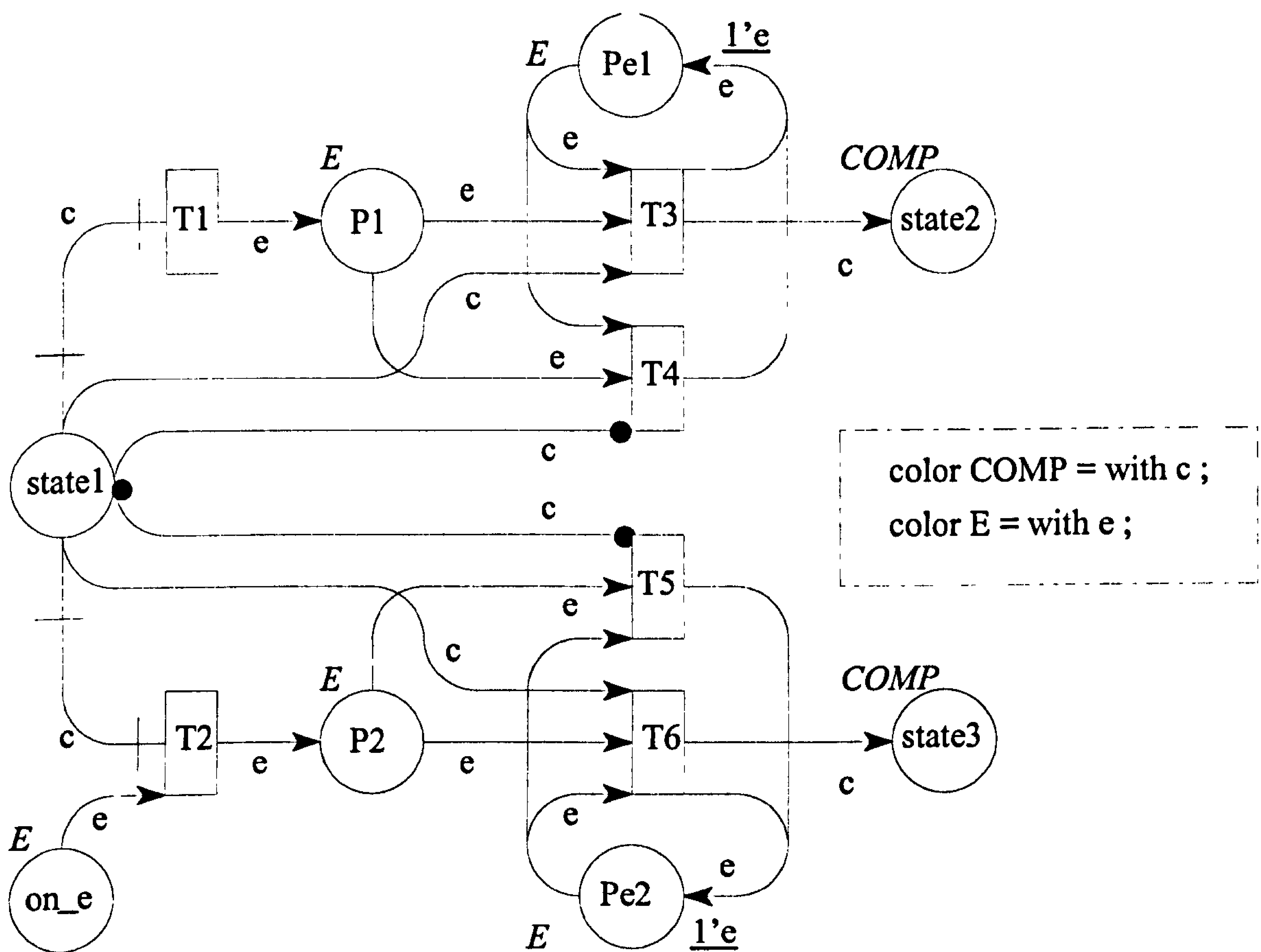


Figure 6.8 Timing problem using Test Arcs

6.7.3 ON_EVENT

It was shown in the previous section how it was possible to resolve firing conflict that may arise when there are multiple transitions from the same state. Figure 6.9 shows a CPN which models the interaction of components through the ON_EVENT transition

```
ON_EVENT comp1.statey {
  state1 -> state3 } ;
```

Components 1 and 2 communicate via places *P1, P2* and *P3*. When any transition fires that moves component 1 into *statey*, eg *T1*, an *e* token is put in place *P1*. This, along with the *e* token in place *P2* enables *T2* which will fire and place an *e* token in place *P3*. If component 2 is in *state1*, transition *T4*, the 'forced event', will be enabled and fire moving component 2 into *state3*. An *e* token will also be placed back in *P2*. Transition *T3* is required to place an *e* token back in *P2* regardless of the state of component 2. The inhibitor pre-arc of *T3* prevents conflict.

By combining the CPNs of figures 6.8 and 6.9 multiple possible transitions from the same state can be modelled. Places *on_e* in figure 6.8 and *P3* in figure 6.9 are the same place and transitions *T2* in figure 6.8 and *T4* in figure 6.9 are the same transition. The resulting net will model the statement

```
5 state1 -> state2 ;
ON_EVENT component1.y {
  2 state1 -> state3 } ;
```

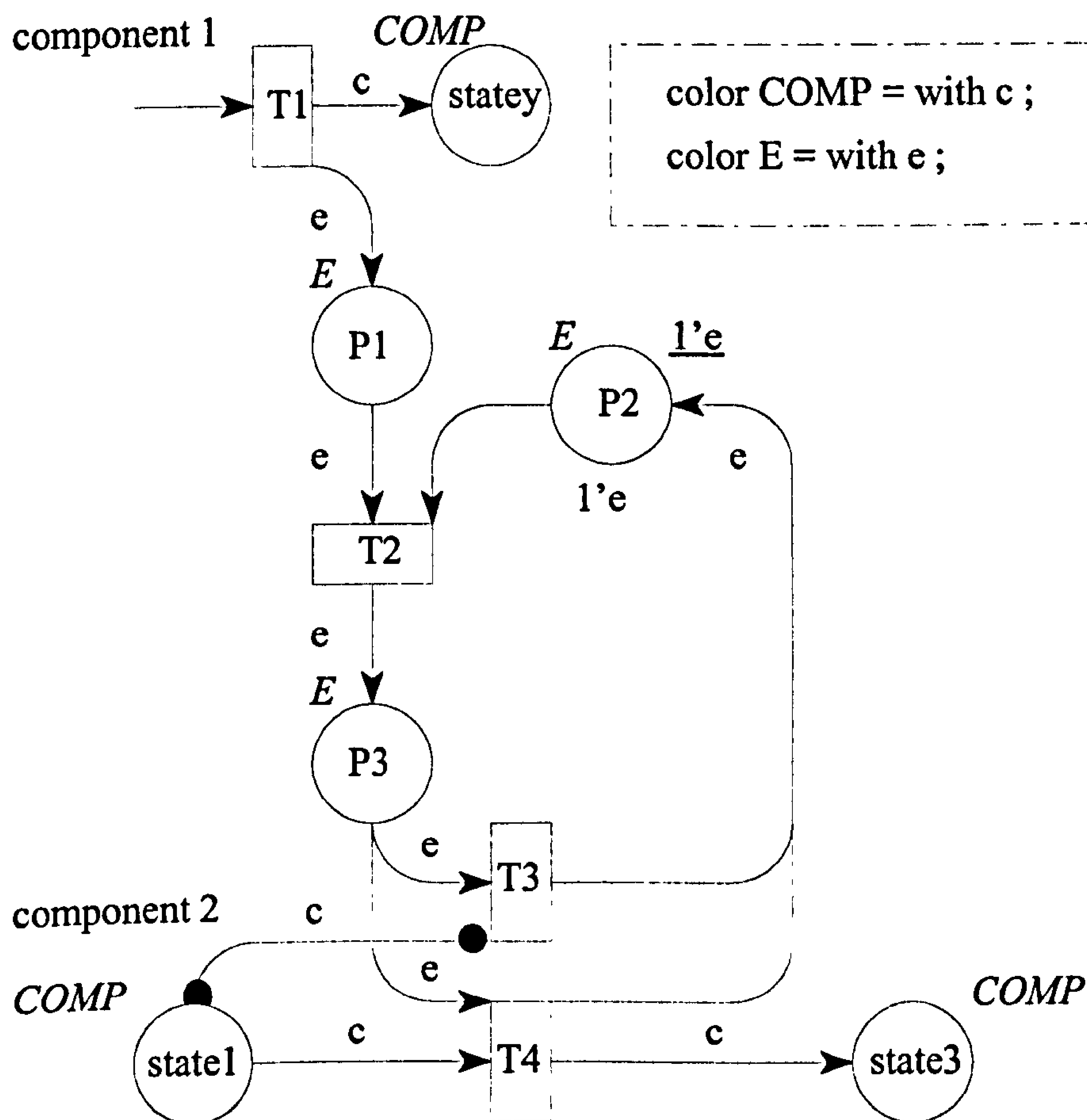


Figure 6.9 ON_EVENT

6.7.4 IF

A CPN representation of the statement using the precondition IF

```
IF comp1.statey {
  state1 -> state3 } ;
```

is shown in figure 6.10.

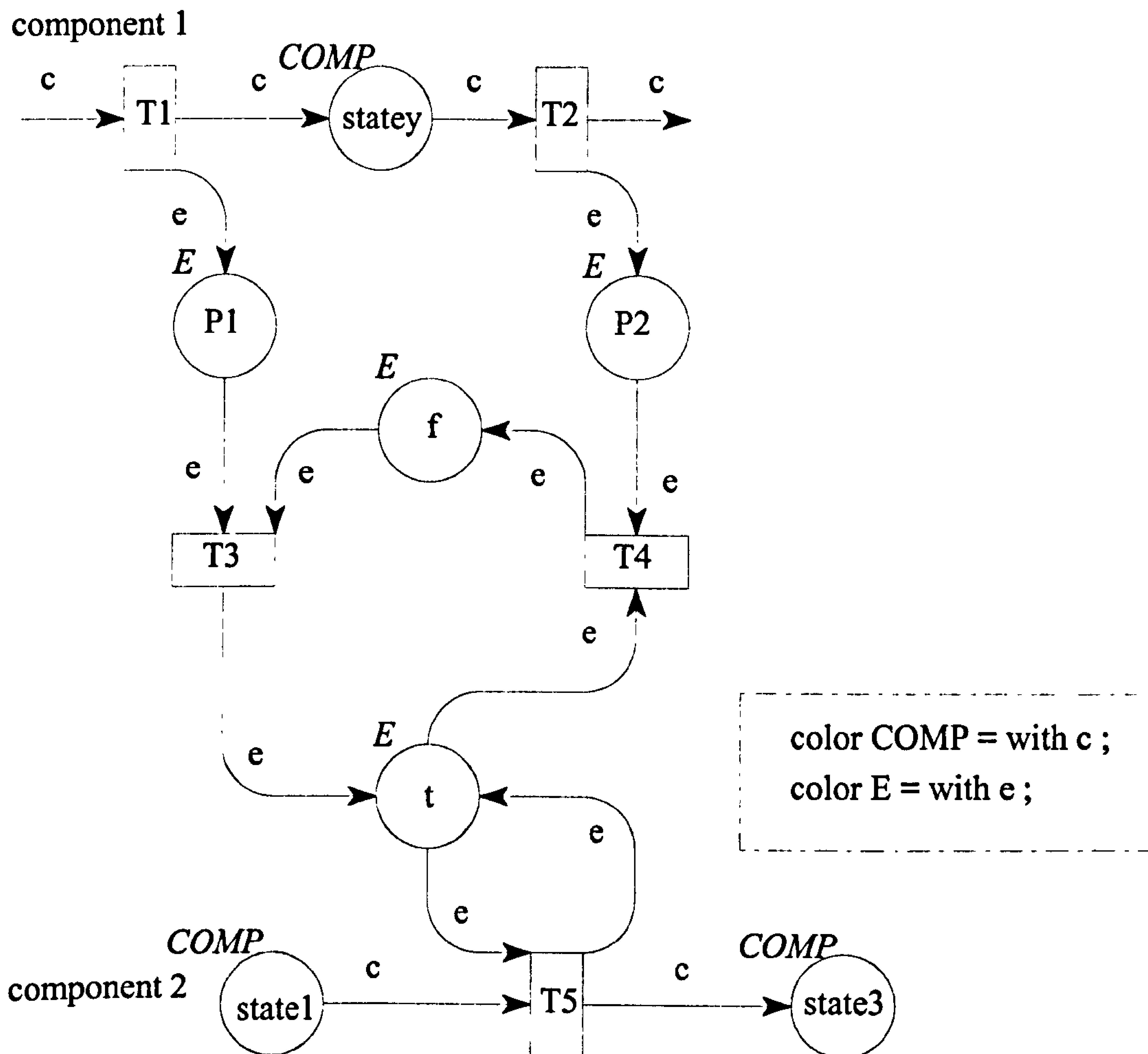


Figure 6.10 IF

The sub-net of places *t* and *f* is an indicator of the state of component 1. An *e* token in *t* represents 'comp1.statey' and an *e* token in *f* represents 'not comp1.statey'. Transition *T5* will be enabled if there is an *e* token in *t*. An *e* token is placed in *t* whenever a transition fires that places component 1 in *statey*, eg *T1*. When *T5* fires, if component 2 is in *state1* (represented by a *c* token in place *state1)* it will move to *state3*. Whenever a transition fires that takes component 1 out of *statey* an *e* token will be put in place *P2* enabling *T4* which, when it fires, will remove the *e* token from *t* and place an *e* token in *f*. This will disable *T5*. This represents very well the semantics of the IF precondition.

The CPN in figure 6.9 representing the ON_EVENT precondition was combined with the CPN of figure 6.8 to model multiple transitions from the same state. The CPN of figure 6.10 may be similarly manipulated to model multiple transitions.

6.8 WAIT_FOR

We will consider firstly the use of the WAIT_FOR statement to dynamically create components. Figure 6.11 shows an example of a CPN which models the creation of a component after some given time delay *t*.

WAIT_FOR t { COMPONENT... } ;

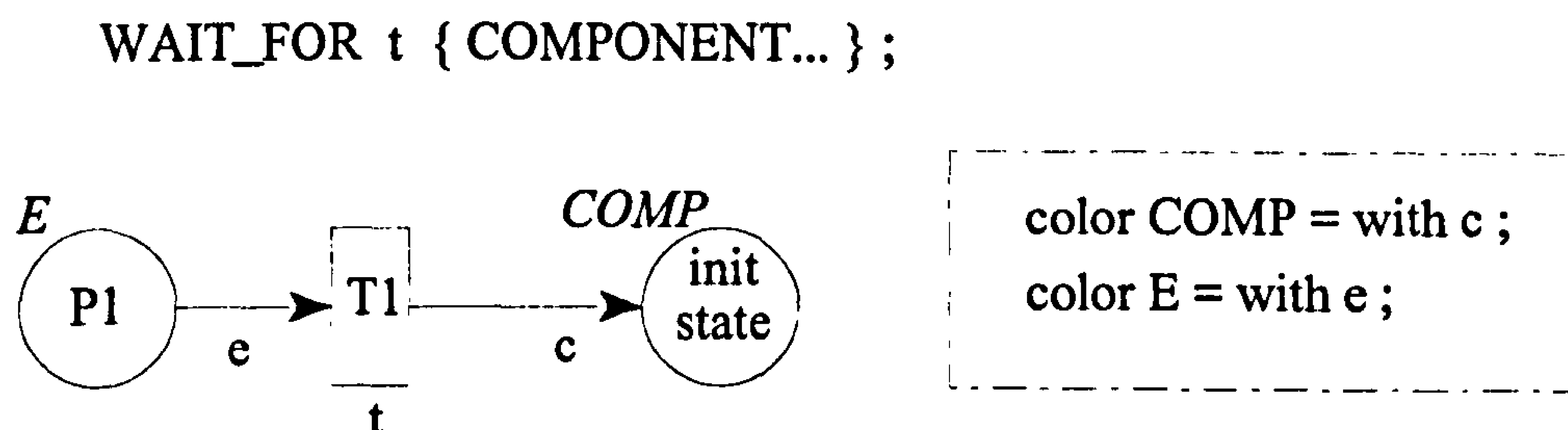


Figure 6.11 WAIT_FOR component

Transition *T1* is initially enabled but will only complete firing after time *t*. The result of this firing will be to place a component, *c*, token in place *init_state* and thereafter the component will behave as a normal component. Notice that no other details of the component are shown in figure 6.11.

The second use of the WAIT_FOR statement is for the explicit creation and removal of stock and resources during simulation time. Figure 6.12a gives an example of a CPN which models the creation of resources, *res1*, when component *A* moves into state *a*. Figure 6.12b gives an example of a CPN which models the removal of stock, *stk1*, after a time delay *t*.

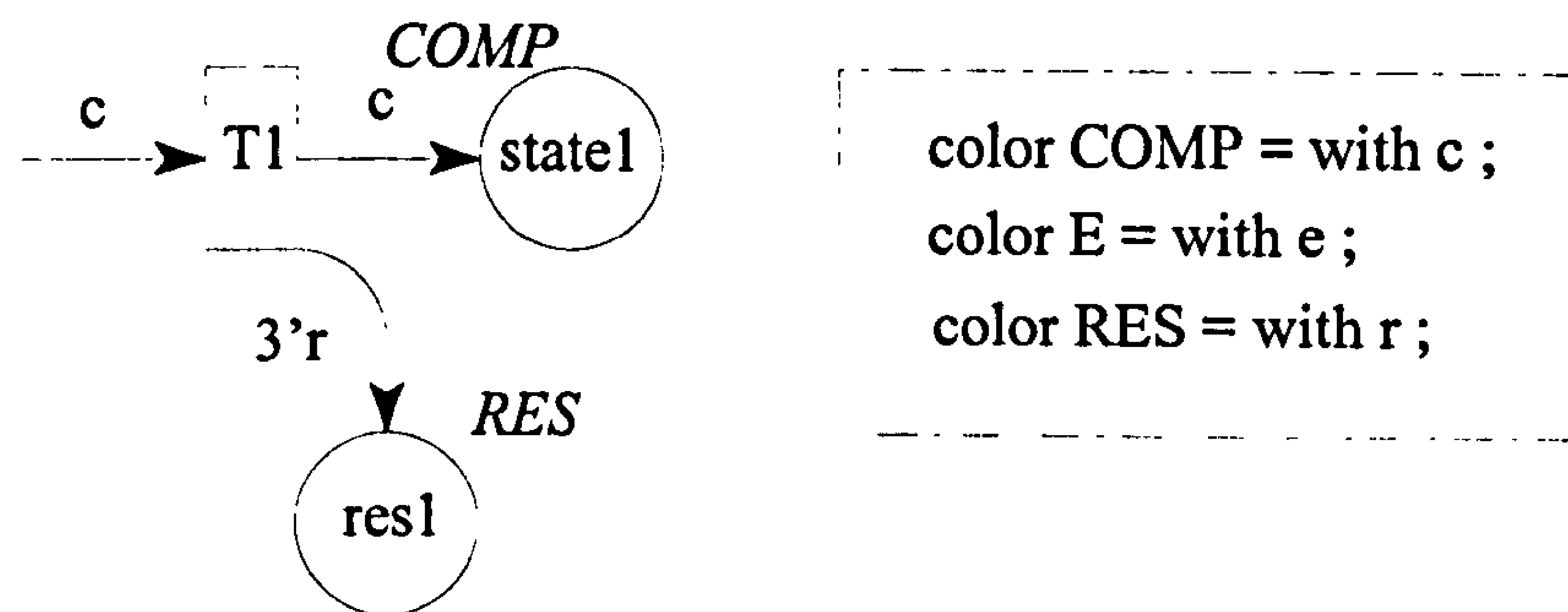


Figure 6.12a WAIT_FOR component.state

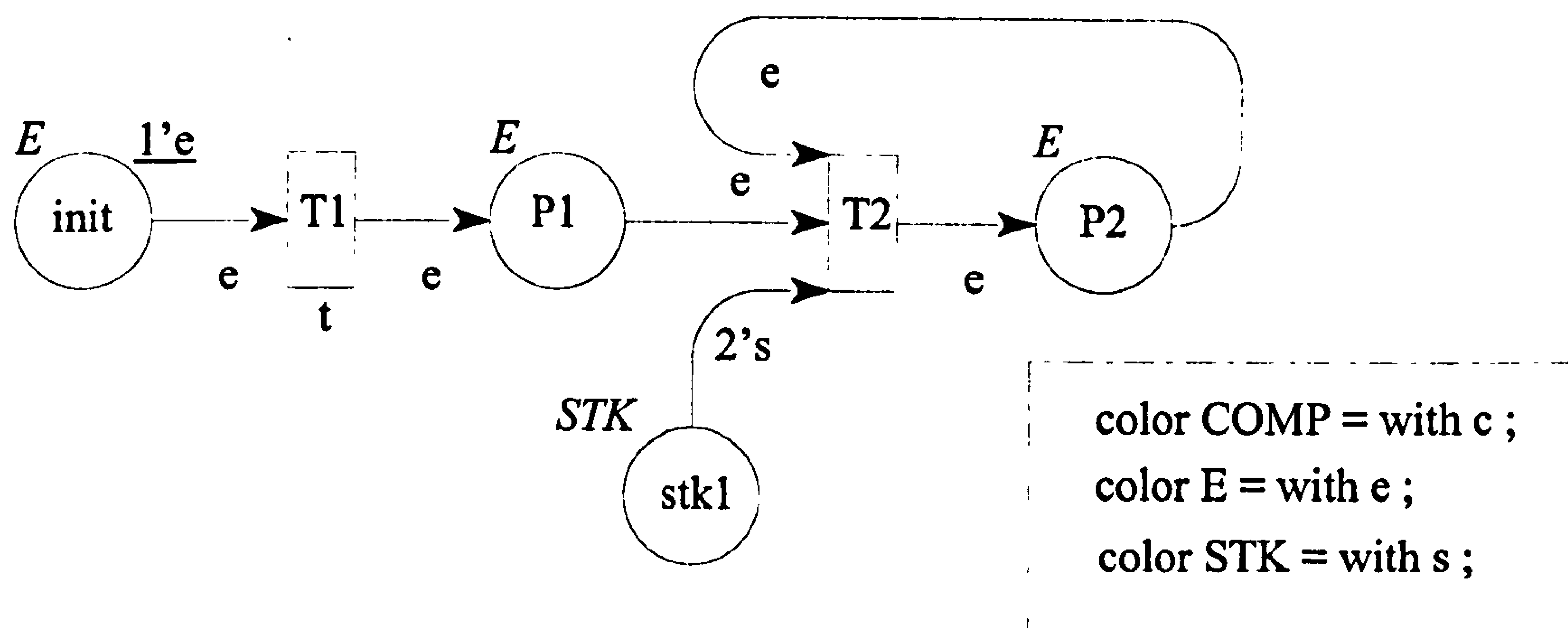


Figure 6.12b WAIT_FOR time

6.9 ATTRIBUTES

Attributes have been defined as simply identifying a set of states. A group of states which share a common property will be given the same attribute to denote this property thus making it easier to refer to the group of states. Behaviour preconditions may refer to component attributes as well as states. Defining CPNs which reflect the ON_EVENT and IF preconditions using attributes will simply involve extending the previous CPNs. Figure 6.13 shows a CPN which models the ON_EVENT

```
ON_EVENT comp6.a {
   state1 -> state3 } ;
```

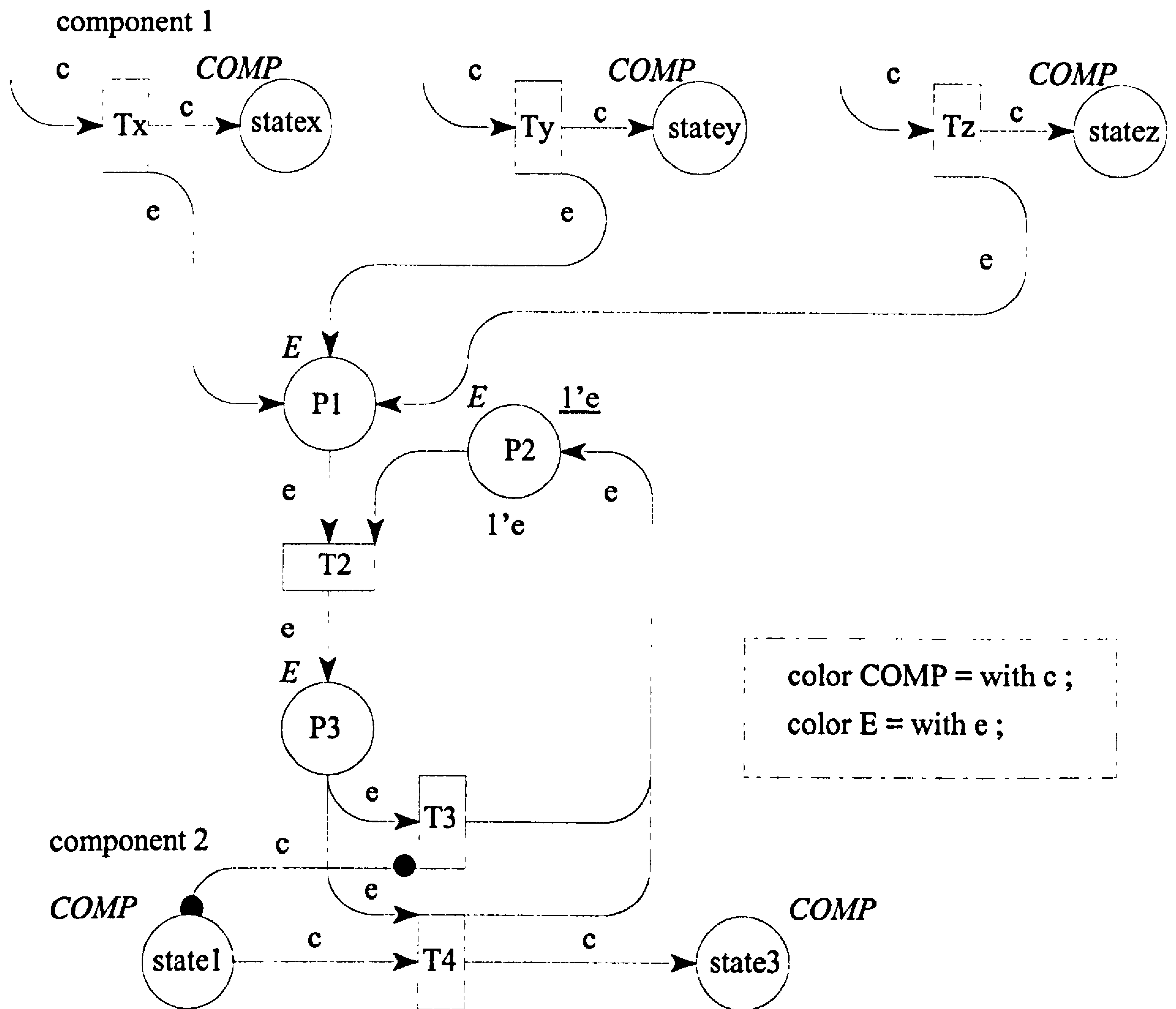


Figure 6.13 ON_EVENT with Attributes

Notice that this is an extension to the CPN of figure 6.9. In this instance, the states *x*, *y* and *z* of component 1 all possess the attribute *a*. The transition *T4* which moves component 2 from *state1* into *state3* will be enabled when component 1 moves into any of the states *x*, *y* or *z*. The enabling place *P1* is shared by all transitions which move the component into a state which possesses the *a* attribute. There will only ever be a maximum of one *e* token in *P1* for by the definition of a component, all states are mutually exclusive.

The modification to the CPN of figure 6.10 for the IF precondition with the same component attributes as above is shown in figure 6.14.

This CPN behaves in a similar manner to that in figure 6.10 although transition *T3* is now enabled by any transition which moves component 1 into a state which possesses the *a* attribute. Transition *T4* will be enabled by any transition which moves component 1 out of a state which possesses the *a* attribute.

Figure 6.14 IF with Attributes

The CPN of figure 6.12a which models a WAIT_FOR statement may also be easily modified to reflect the use of component attributes rather than a state. This would simply involve duplicating the post-arc (including inscriptions) from transition *T1* for any other transition which moves the component into a state which possesses the given attribute.

6.10 Behavioural modelling with generalised stochastic Petri nets (GSPN)and ICE.

In this section we compare the approach of ICE and GSPNs to some important modelling conditions.

For a model to correctly encompass the functionality of a system all relevant events must be specified as well as the pre-conditions that must hold for an event to occur and the post-conditions that exist after an event has occurred [135].

Some fundamental properties of events which may occur within systems and have either temporal or structural relations are identified below. Their GSPN models are shown along with the ICE code which provides equivalent functionality. The inability to implement any of these events would indicate an area for concern within the language as it would not be universally suitable for all systems that we may in future wish to model.

6.9.1 Dependency

When one event is dependant upon the pre-occurrence of other events. Shown in the GSPN of figure 6.15, transition t_O is dependant upon the firing of transitions t_{i1} and t_{i2}. This property is utilised in applications such as software modelling [136].

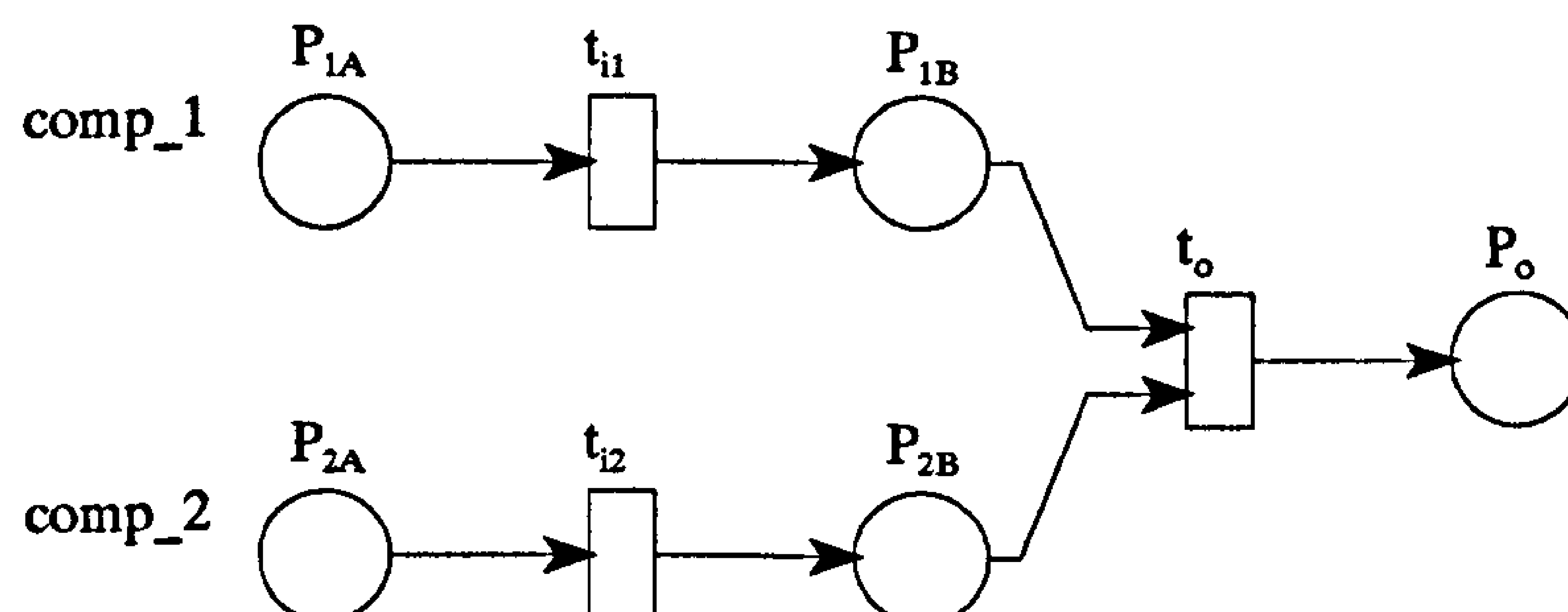


Figure 6.15 GSPN of dependency

ICE equivalent :

```
SYSTEM dependent = ALL (comp_1.P_1B , comp_2.P_1A );

BEHAVIOUR be_comp {
        ...
        ON_EVENT dependent {
                exp(t_i1) P_1B -> P_o ;
                exp(t_i1) P_1B -> P_o ; }
        ...
}
```

6.9.2 Concurrency

More than one event occurs simultaneously but these events do not interfere or interact with one another. In the GSPN of figure 6.16 the transitions t_{i1} and t_{i2} may be simultaneously enabled but there is no interaction between the input and output places. This property is applicable to such systems as concurrent software applications [137].

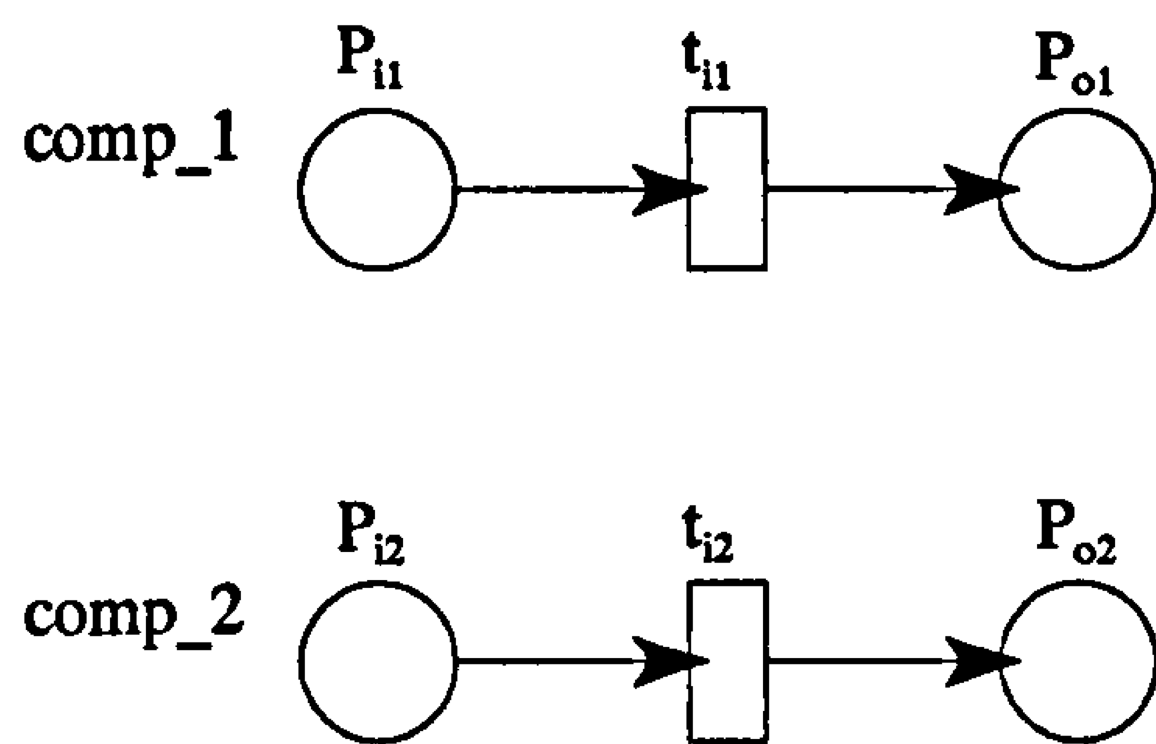


Figure 6.16 GSPN of concurrency

ICE equivalent :

```
BEHAVIOUR be_comp_1 {
        ...
        exp(t_i1) P_i1 -> P_o1 ;
        ...
}
BEHAVIOUR be_comp_2 {
        ...
        exp(t_i2) P_i2 -> P_o2 ;
        ...
}
```

6.9.3 Synchronisation

In a parallel application a number of tasks may have to synchronise at given stages of operation. In the GSPN of figure 6.17 the immediate transition t_S synchronises transitions t_1, t_2, ... , t_n. This may be used in systems where multiple resources are required to complete a given task such as safety protection systems.

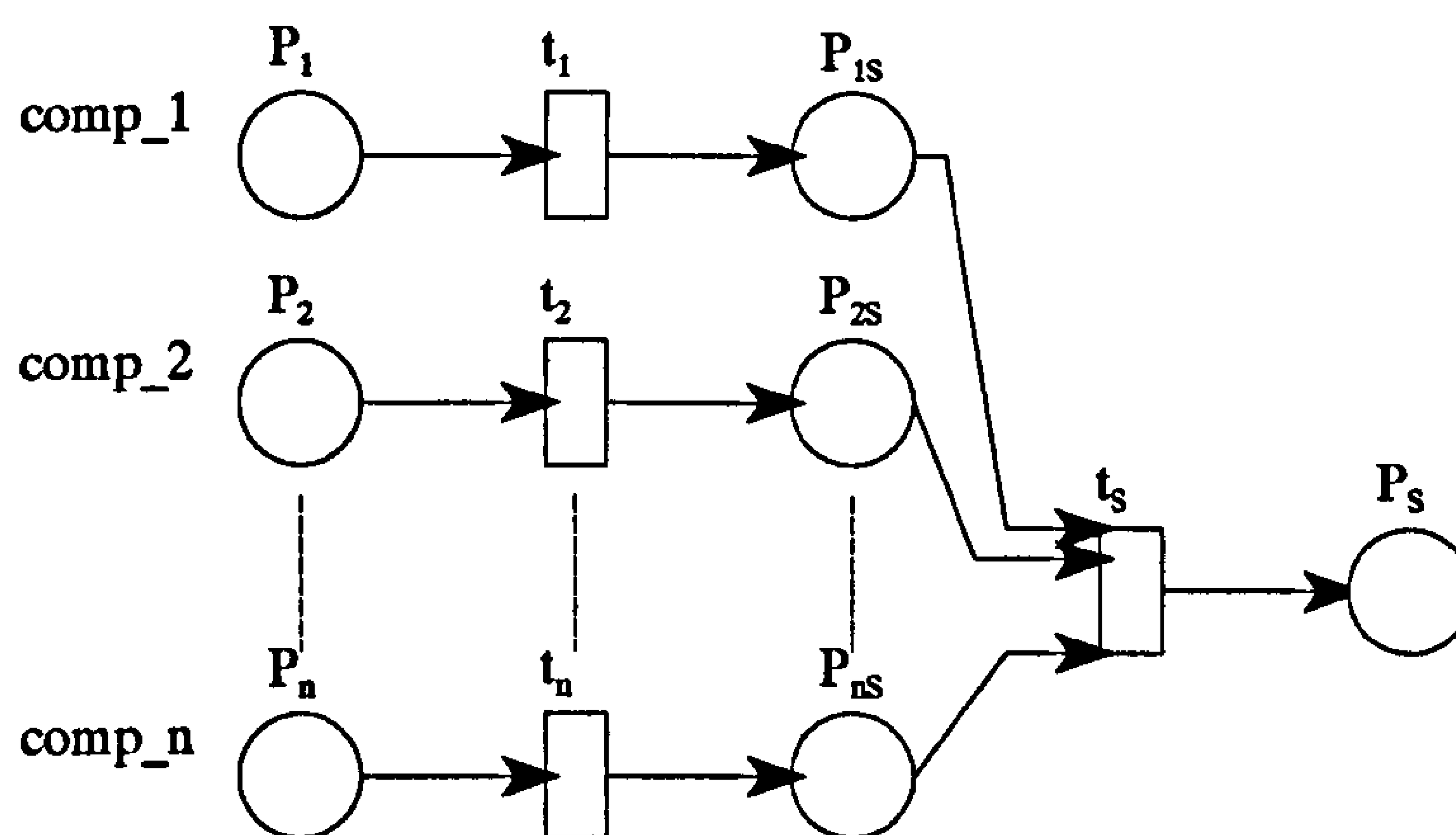


Figure 6.17 GSPN of synchronisation

ICE equivalent :

```
SYSTEM synchronise = ALL (comp_1.P1S ; comp_2.P2S ; ... ; comp_n.PnS ;)}

BEHAVIOUR be_comp {
        ...
        ON_EVENT synchronise {
                exp(tS) PX -> PS ; }
        ...
}
```

6.9.4 Conflict

In a system more than one event may be possible at the same time and the occurrence of one of these events may preclude the other. In the GSPN of figure 6.18 transitions t_1 and t_2 are simultaneously enabled. The firing of either will disable the other. This concept is used to model such things as resource conflict within systems.

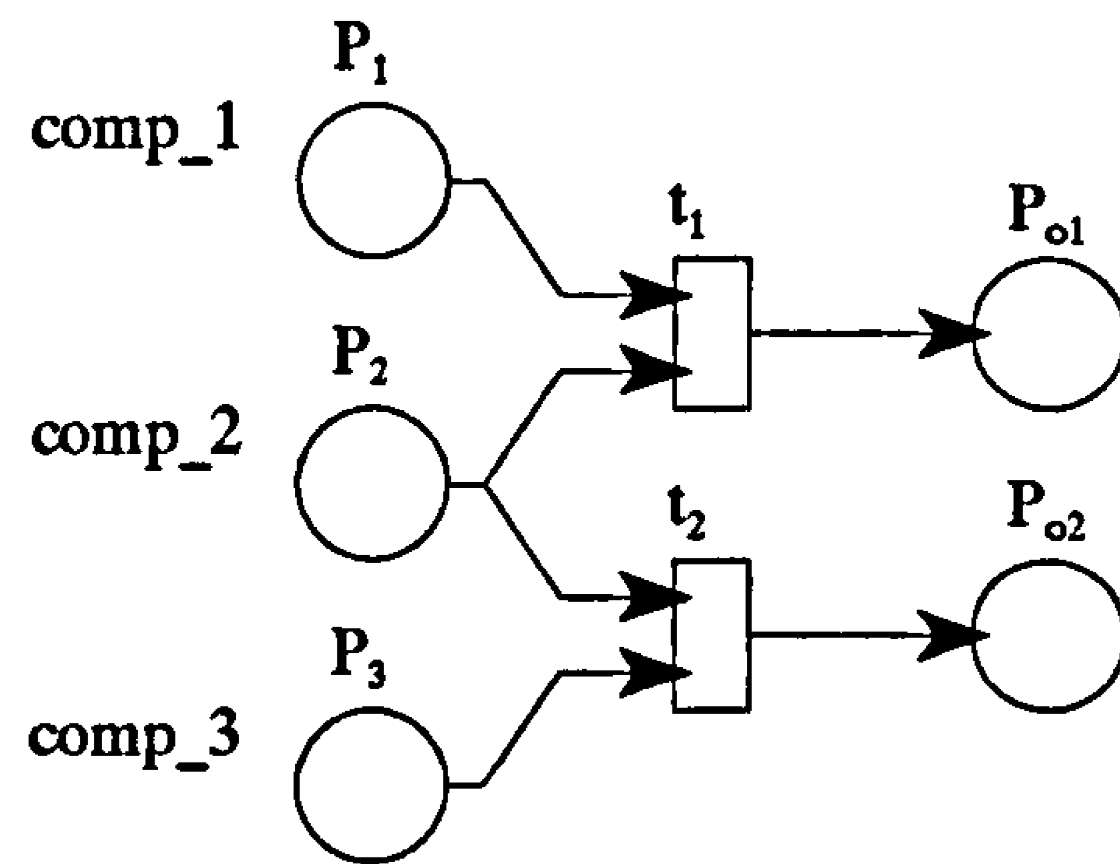


Figure 6.18 GSPN of conflict

ICE equivalent :

```
BEHAVIOUR be_comp_1 {
        ...
        IF comp_2.P2 {
                exp(t1) P1 -> Po1 ; }
        ...
}
BEHAVIOUR be_comp_2 {
        ...
        IF ANY {comp_1.Po1, comp_3.Po2} {
                0 P2 -> P2_next ; }
        ...
}
BEHAVIOUR be_comp_3 {
        ...
        IF comp_2.P2 {
                exp(t2) P3 -> Po2 ; }
        ...
}
```

Note that in the GSPN of figure 6.18 we assume that a transition firing policy is adopted whereby tokens are not removed from input places when a transition is enabled but when it is fired.

6.11 Conclusions

ICE components may be regarded as FSMs and are represented by PNs with a single token. These basic models are built upon when counters are added. Counters contribute a considerable increase in descriptive power and this is reflected by the increased complexity of the PN model.

ICE uses constructs termed SYSTEM statements that provide a simple means of relating what can be some very complex component interactions. It is not a trivial task representing even a relatively simple SYSTEM statement with PNs. These PNs are the most sophisticated of all that are used to model ICE and show how the apparently simple syntax of ICE is founded upon some involved semantics. This comparison in particular shows the modelling power that is achievable with ICE and which would require considerably more complex representation with PNs.

Passive resources are similar to semaphores and can be shown as simple additions to the component PNs. ICE attributes which are used to group together component states with some form of commonality are modelled and it is apparent how they add power and sophistication to behavioural statements.

Perhaps the most important constructs to be formally defined by PNs are behavioural statements. They contain some of the languages most complex semantics and the PN representation is a valuable aid in their understanding. Transition firing policies are of significant importance and the PN models give some detailed insight and are a source of notable discussion.

In producing all the PN models for ICE it was interesting to note that no single dissertation on PNs contained all the high level extensions that were required. All of the ICE constructs can be modelled with PNs, involving varying degrees of complexity and hence we cannot state that ICE is more powerful. However it is evident from some of the comparisons that the ICE implementations are considerably simpler.

We go on in the chapter to consider the challenges of dependency, concurrency, synchronisation and conflict. Any modelling technique must be able to reflect all of these if is to be of generic use in the modelling of systems. We derive both PN and ICE solutions which are equally as simple to implement.. The ICE implementation of each of these events is straightforward and does not involve any sophisticated manipulation of the language.

Chapter 7

An ICE Performance Model of an ATM Switch

7.1 Introduction

Performability as described earlier is a combined performance/reliability measure and as such comprises both pure performance and pure reliability measures. In this chapter we consider a pure performance model of an ATM Banyan switch.

Banyan switches have many constituent interacting components and are an ideal medium for illustrating the high level declarative nature of the ICE language. The individual components are described directly with a very low level of abstraction. Performance parameters of interest in switch modelling are those which will give a good indication of the theoretical Quality of Service (QoS) provided [138]. The ICE model is implemented in such a way as to give both relevant measures of performance and to demonstrate the flexibility of the model.

Banyan networks are recognised as being one of the most suitable architectures for ATM switches [139]. They have received much attention in their various forms for modelling by probabilistic means [140]. We have chosen one with a recognised buffering strategy so that the assumptions made when modelling may be compared.

7.2 Asynchronous transfer mode

The existing Integrated Services Distribution Network (ISDN) is currently evolving into the Broadband ISDN (BISDN). The BISDN will be required to transfer a much larger amount and variety of traffic than the existing ISDN and will therefore impose constraints in terms of throughput, delay, delay dispersion, reliability and sequenced delivery [141]. For this reason the CCITT formed Study Group XVIII to select and standardise a suitable transfer mode for the BISDN.

In 1988, CCITT selected Asynchronous Transfer Mode (ATM) as the transfer mode for BISDN. In 1990, a first set of recommendations [142] was agreed upon worldwide. ATM is being further refined and standardised by the ITU-T, ETSI and ANSI [143]. The universal flexibility of ATM guarantees that it can support any service from simple telephony to full multimedia, including HDTV, audio and high speed data [144].

ATM is based on switching small, fixed length packets of information (cells), and doing it extremely quickly. All cells consist of a 48 byte information field and a 5 byte header, according to CCITT recommendations [145]. A connection within the network is defined link-by-link by a label within the cell header, the virtual channel identifier (VCI) or virtual path identifier (VPI). ATM operates in a connection oriented mode, thus a connection is only established if sufficient resources are available. All cells are routed via one path to maintain the cell sequence. All network links are interconnected by some type of ATM switch or multiplexer. There has been considerable interest in the modelling of ATM multiplexers [146..150], here we shall limit our investigation to switches. For an excellent comprehensive treatise of ATM the text by Cuthbert and Sapanel [204] is recommended.

7.3 ATM switches

There has recently been much interest in the design of ATM switches [151, 152]. Tobagi [153] proposes three main types of ATM switch architectures, namely, (i) shared memory [154] (ii) shared-medium [155] and (iii) space-division [156]. In further work Tobagi goes on to explore the possibilities of combining architectures of different types [157]. All types are limited in both size and line speed, making it necessary to connect many stages together

to form a multistage configuration [158], originally proposed for multiprocessor applications, to produce practical switches. Many demands which must be adhered to are placed on ATM switch design, such as (1) modularity, (2) relaxed synchronisation, (3) guaranteed high performance without requiring internal speed-up and (4) maintenance of packet sequence integrity [159]. Switches must be able to handle both point to point to point connections and point to multipoint connections [160, 161] as required by a wide range of applications such as video-conferencing, entertainment video, LAN Bridging and data distribution. Both types of connection normally utilise a multistage interconnection network, with the latter also incorporating a proceeding copy network. For a comprehensive introduction to ATM switching the reader is referred to either Onvural [162] or Chen et al [163]. A common type of multistage interconnection network is the Banyan Network [164]. Banyan Network space-division switches are being seen to play a major role in ATM switching.

7.3.1 The architecture of banyan networks

There have been a number of detailed surveys of Banyan Networks in the literature. For a good introduction the user is referred to Perros [165]. Feng [166] describes Banyan Networks as having a dynamic topology belonging to the class of multistage interconnection networks. They are constructed by interconnection of a number of discrete Switching Elements (SEs). They have been comprehensively defined by Tubtiang et al's general ATM switch classification method [167]. The class of Banyan Networks can be divided into several subclasses, the most common, the one we will consider, is known as *L-level Banyan.*

In L-level Banyan only adjacent stages are connected by links, therefore each path from input to output leads through L stages. There are two types of L-level Banyans, *Regular* and *Irregular*. Regular Banyans are constructed from one basic type of SE with F inputs and S outputs, whereas the type of SEs in Irregular Banyans varies throughout the network. For economic reasons Regular Banyans are preferred as they lend themselves to straightforward VLSI implementation. Two subclasses of Regular Banyans are *CC-Banyans* and *SW-Banyans*. We will confine our examination to SW-Banyans as they cover nearly all existing implementations [168]. SW-Banyans are constructed from a number of basic crossbar SEs with F inputs and S outputs.

Delta Networks have the topological structure of SW-Banyans. They have N stages corresponding to the L levels. Each stage has a number of SEs, each with n inputs and m outputs, thereby giving the network n^N input ports and m^N output ports. *Rectangular Delta Networks* are constructed from SEs which have the same number of inputs as outputs ie $n = m$. The network will have a total of $N = log_2 n$ stages. It follows that the number of output ports, n, will be equal to the number of input ports and that the number of SEs per stage will be constant ie $^n/_2$. A Delta Network which has SEs with n inputs is known as a *Delta-n Network*. In hardware realisations n is limited if a single chip implementation is required. An SE with n = 16 has been fabricated and required 0.8 micron Bi-CMOS technology [169]. The most common form of Delta-n Network is the *Delta-2 Network* and this has been the basis for many ATM switch models [139, 140, 170..175]. Figure 7.1 shows a 16-input Delta-2 Network.

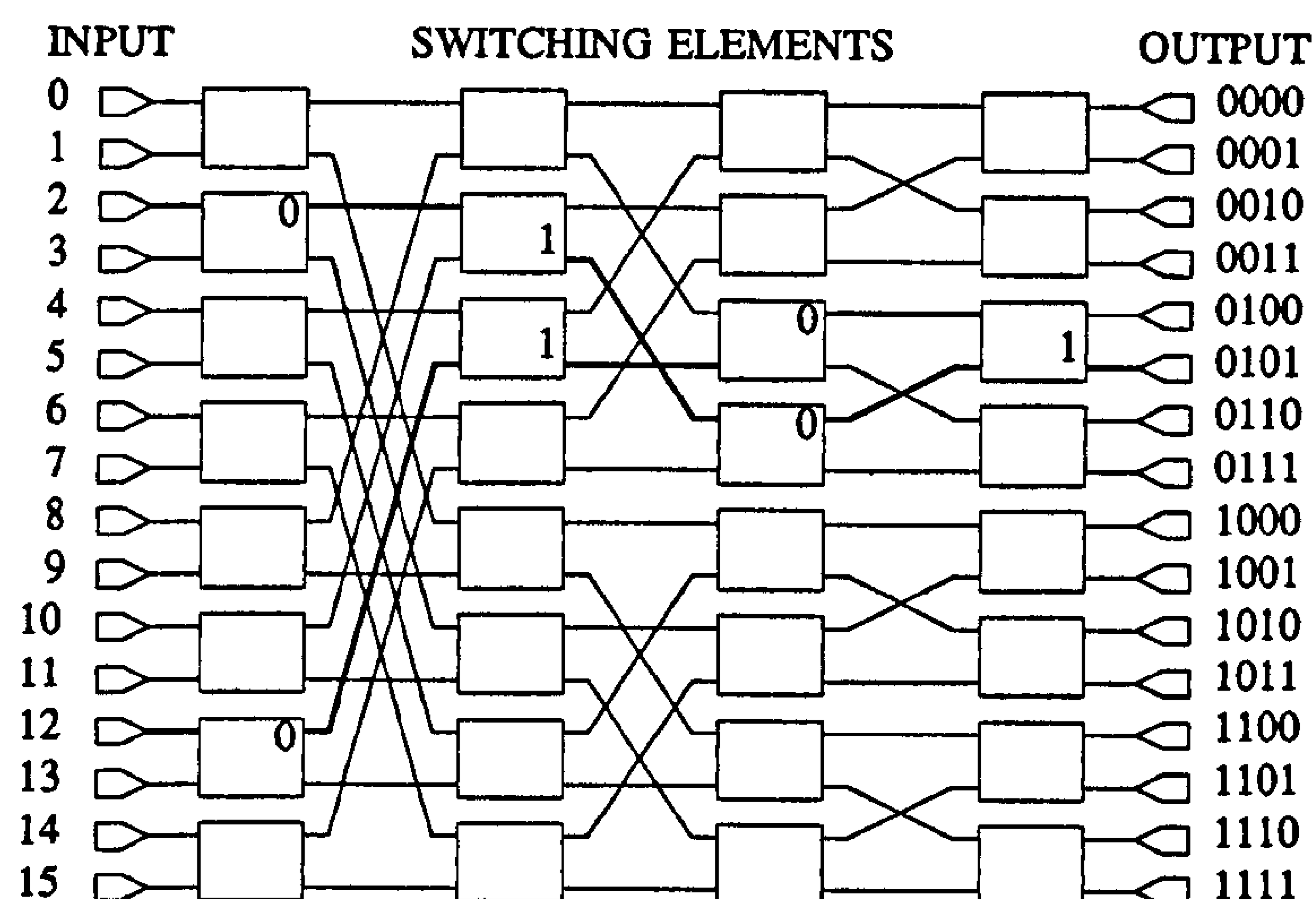


Figure 7.1 Delta-2 16x16 Banyan switch architecture

The Delta-2 network is used as a point to point switch with a *self routing algorithm.* The header of each incoming cell is given a n bit destination address relating to its virtual path or virtual channel address and routing inside the switch is done by decoding the header. For example, a node at stage k sends the cell out on either link-0 (up) or link-1 (down) according to the k *th* bit of the header. The topology of the network ensures that the path from any input to a given output is uniquely determined by the output address. The header is independent of input link, as illustrated by the two routings shown in Figure 7.1 from inputs 3 and 14 to output 0101. This type of routing also known as *digit controlled routing,* can be fully implemented in hardware. It becomes apparent from examining such a network that if any SE fails then certain paths which must utilise this SE will be blocked.

Techniques are available to increase the size of the network and thus provide alternative paths [176], for example by using parallel layers of interconnection networks [177, 178], shuffle exchange networks [179], bypass connections [180], turn-back networks [181] or dilated interconnection networks [182]. Whatever technique (if any) is used, switches should be of modular design [183] to facilitate expansion if network traffic demands dictate.

The Banyan Network is a blocking network [184], for two cells with different destination addresses may be routed through the same internal link at the same time. One solution to this problem is to sort arriving cells with distinct destination addresses into ascending or descending order before transmission through the switch. Time overheads are incurred however due to sorting time and only one cell per time slot being transmitted. It has become clear that to provide satisfactory speed and cell loss performance as required in ATM networks, it is necessary to provide buffering within the SEs. Cells are then transmitted directly through the switch, being buffered at each stage, and if a required link is busy between stages k and $k + 1$ then the cell is retained in the buffer at stage k until the link is free. Cells may also be retained in the buffer at stage k if the buffer at stage $k + 1$ is full. Buffers are normally also provided at the switch inputs and outputs to compensate for the difference in speeds between the switch fabric and network links. The buffers in the SEs can be located at the inputs, crosspoints, outputs or be shared as shown in Figure 7.2.

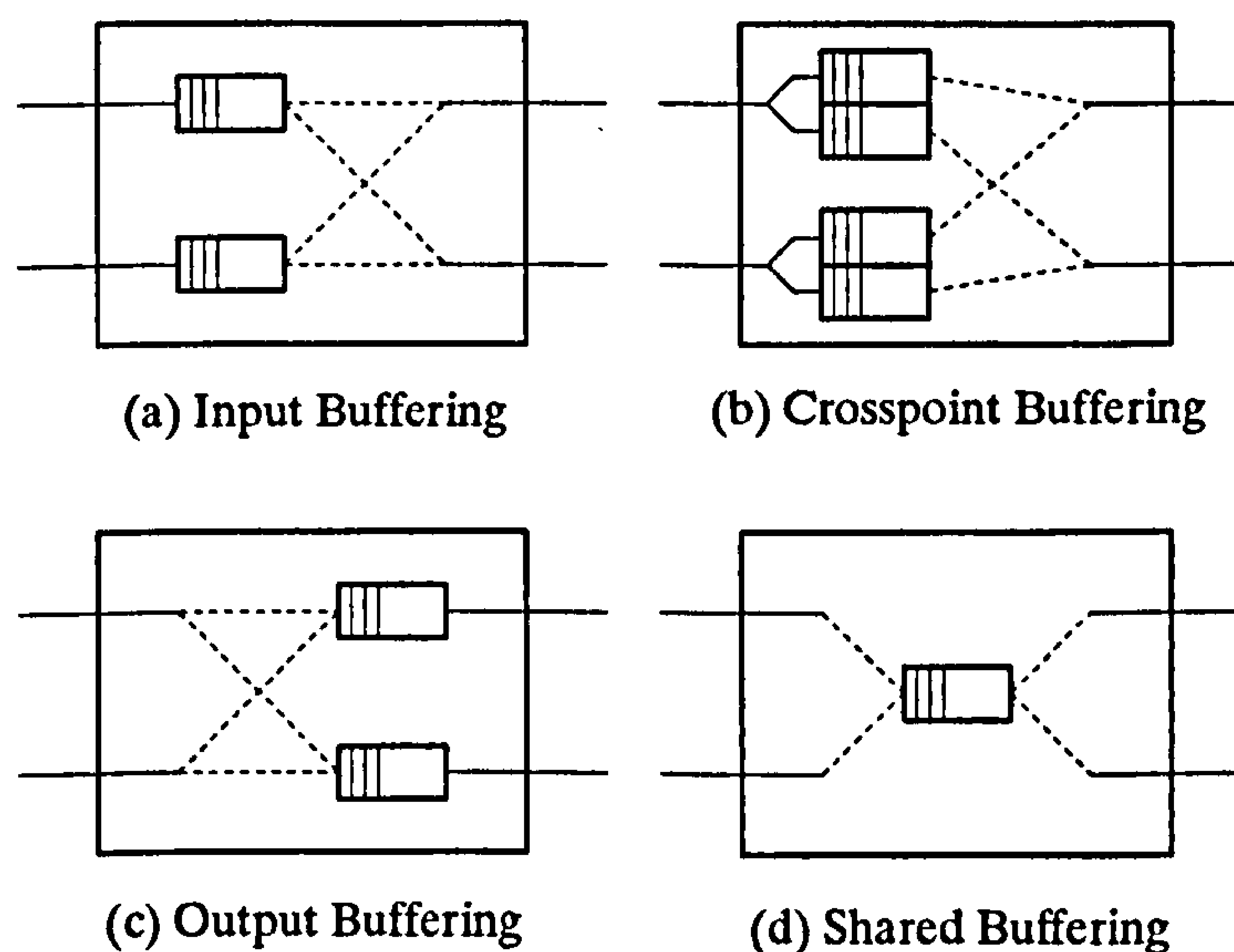


Figure 7.2 Switch buffering policies

In input buffering [185], buffers are located at the inputs of the SEs. When two cells in different buffers contend for the same output only one can move to the next stage. The cell loosing contention waits at the head of its input buffer blocking other cells which may be destined for the other, idle, output. This is known as head-of-line (HOL) blocking. It has been shown by Dias and Jump [186] that HOL blocking limits the maximum load to 0.75 in Delta-2 networks when input buffered SEs are used.

Output buffering [186] and shared buffering [158] allow higher loads by sharing the input cells between the different outputs. In output buffering, buffers are placed at each output. In a given time slot, n inputs may access the same output buffer, thus multi port buffers are required. In shared buffering, cells from different inputs and destined for different outputs share the same buffer. Therefore multi-port buffers are required of a size n times of that which is required for output buffering [187].

In crosspoint buffering, buffers are required for each input-output pair. Arriving cells at stage k are enqueued in the appropriate buffer according to the k *th* bit of their header. An example of crosspoint buffering is the PHOENIX fault tolerant SE implemented at AT&T Bell Laboratories [188].

7.4 The ICE model

In this section we discuss the development of the model, looking at each functional block individually.

7.4.1 Overview

The first decision to make is how to subdivide the switch into a suitable combination of components. A suitable balance is required between limiting complexity, which increases with number of components and fully representing the functionality. Four functional units are identified, namely the communications channel, input controllers, switching elements and output controllers. Each is modelled with a component, save for the switching elements which are best represented by four interacting components.

The design takes a modular approach both for simplicity and ease of expansion. In the literature there are models for a great range of sizes, however the intention was to make the ICE model representative of a practical switch. The size decided upon was 16x16, chosen as an optimum size as it provides a high enough number of channels and 16 interface cards fit well into one module mounted in a standard 19" telecommunications equipment rack [189].

A 16x16 Banyan switch has 4 levels of switching, each with 8 layers of switching elements. The popular assumption made is that all cells arriving from the input channels have output addresses which have an equal probability of being any of the output channels [190]. If we adopt this assumption then it is safe to conclude that we need only model one layer of switching elements and the behaviour of this layer will be representative of any other [191]. To model all layers would only require reiterating the ICE code a further 7 times and editing component names in the behaviour statements. Such expansion would not be practical in probabilistic models as the size of the expressions would become unmanageable.

Thus the ICE model describes all 4 levels of switching elements for one layer of a 16x16 switch. Figure 7.3 is a diagram of the switch with the ICE components marked.

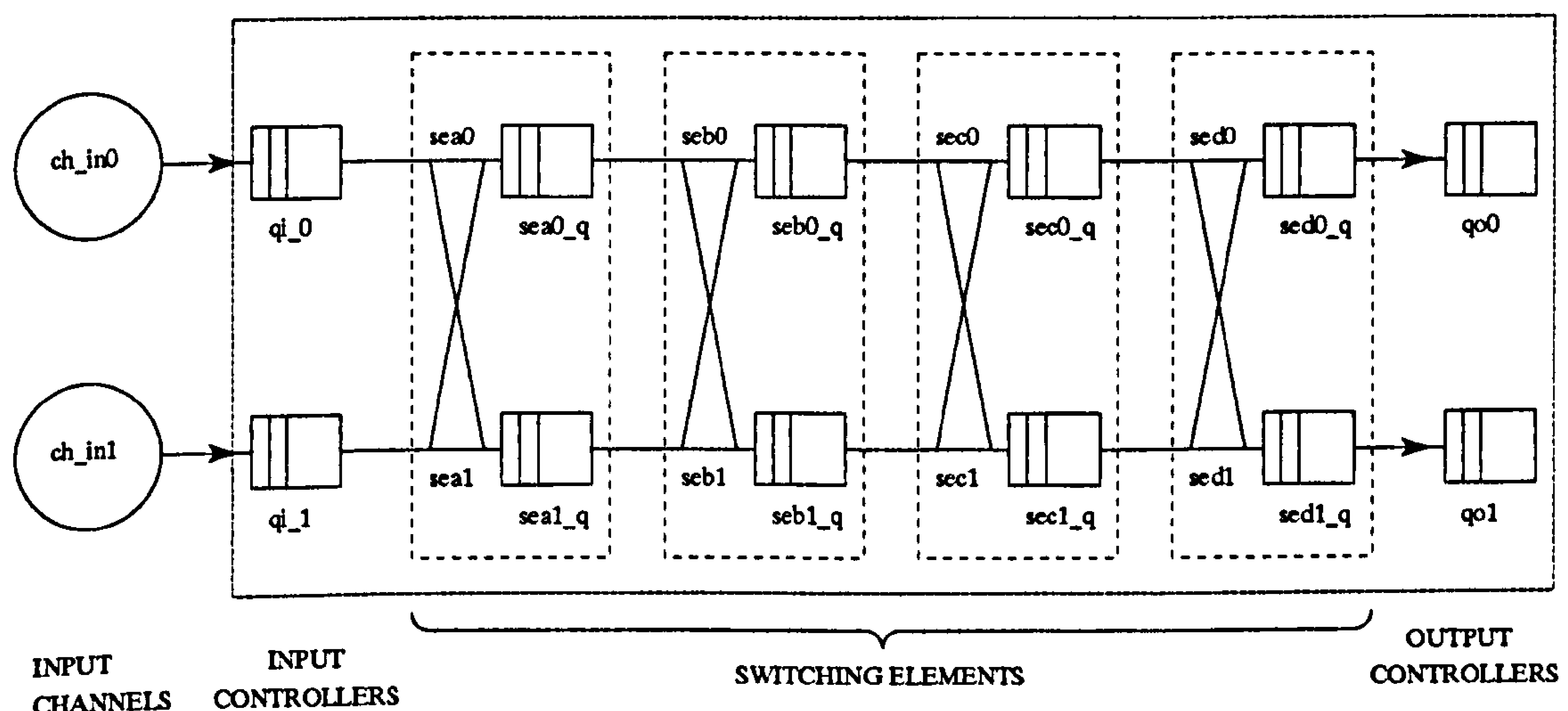


Figure 7.3 One layer of 16x16 switch with components marked

The complete ICE program for the switch model is listed in appendix D. Below we shall consider each type of constituent component.

7.4.2 The input traffic

The input traffic to the switch is described by modelling input channels that can either be in an *arrive* (cell slot occupied) or *quiet* (cell slot empty) state. It was initially thought that this could be incorporated as part of the behaviour of the input controllers but the requirement that the load be constant prevented this. To expand on this point, to give a true representation of an input channel the model must show a steady flow of cell slots with the probability that any slot is occupied being equal to the required load. If the two states that are described form part of a larger state set with other transitions, this would jeopardise that requirement

In mathematical modelling it is necessary to select some appropriate stochastic distribution that will closely reflect the behaviour of traffic. Uniform cell arrival rates may be represented by either the Poisson or Bernoulli distributions. These may be utilised in ICE by manipulation of the exponential transition firing rates. Bursty cell arrivals have been modelled in ATM networks by Interrupted Poisson Processes (IPP) [192] and Bulk Bernoulli Processes (BBP) [193]. Complex models of bursty traffic with both exponentially distributed quiet and bursty periods can be modelled using a Markov Modulated Poisson Process (MMPP) [192] as implemented in the BONeS simulator [194]. The MMPP can be implemented in ICE by building on the model for uniform cell arrival which is shown below in figure 7.4.

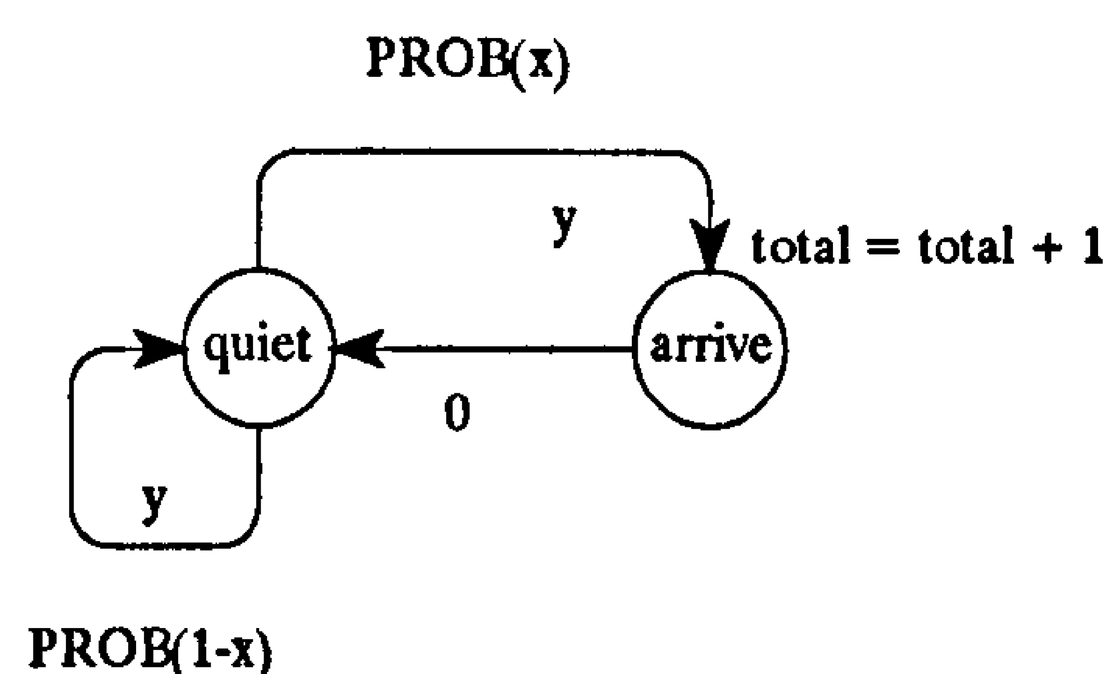


Figure 7.4 State diagram of an input channel

By assigning probabilities to the output transitions from the *quiet* state we can directly represent the channel in ICE with no level of abstraction. One component is used for each input channel. This component can only exists in the states *quiet* or *arrive* and will move between them with a probability equal to the load as shown by the behaviour statement in listing 7.1. This gives a very simple but very accurate model of the input traffic. The

counter *total* which is shown being incremented in the *arrive* state keeps a tally of the number of cells arriving. This is useful for validating loads during simulation.

```
BEHAVIOUR be_ch_in {
        1 quiet -> arrive PROB(0.6);
        1 quiet -> quiet PROB(0.4);
        0 arrive -> quiet;
}
```

Listing 7.1 Input channel BEHAVIOUR statement

7.4.3 The input controllers

The input controllers buffer the cells arriving from the input channels before transmitting them to the first switching elements. Each input controller is modelled as an individual component. The state diagram for is given in figure 7.5.

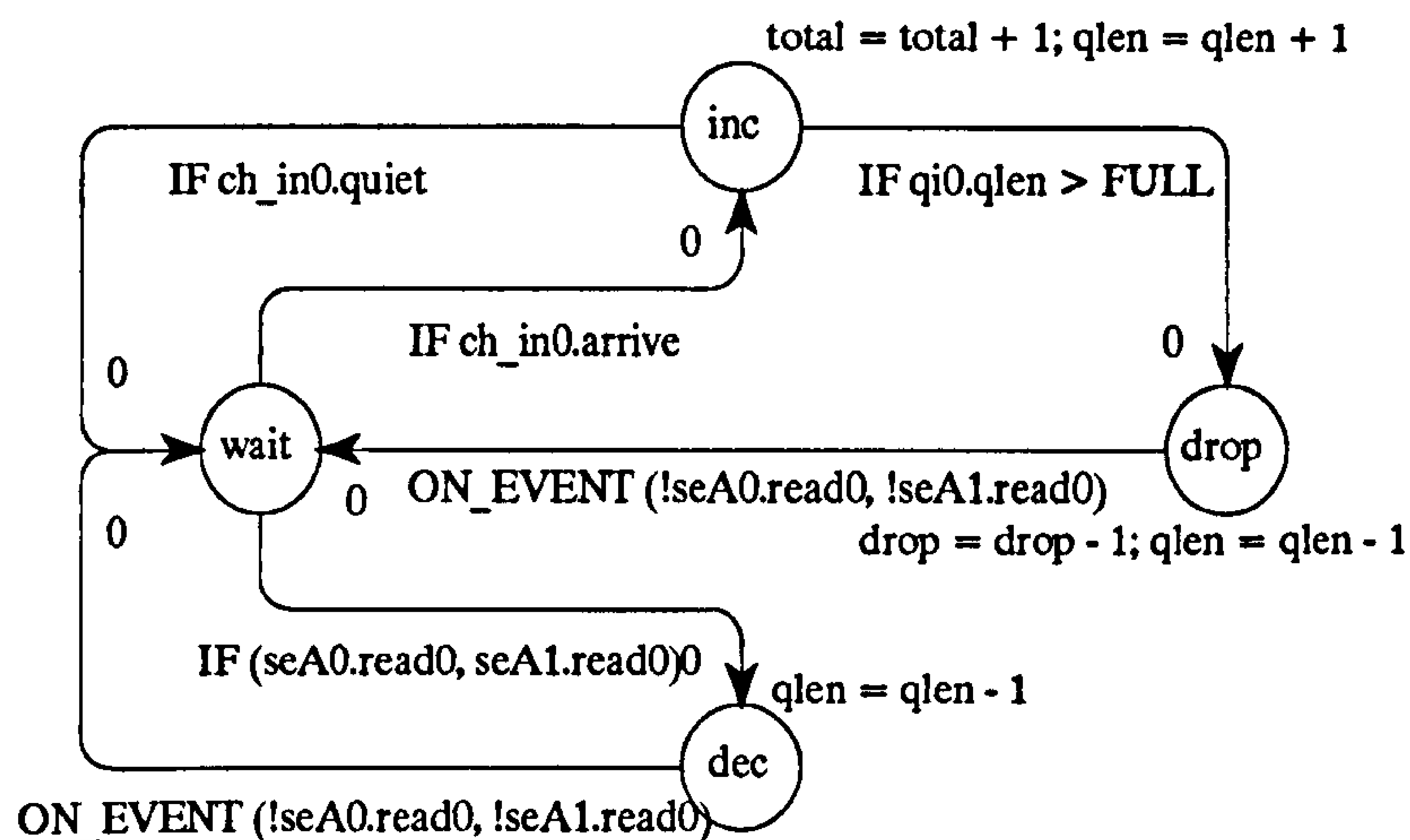


Figure 7.5 State diagram of input controller

There is one buffer per input controller so that all cells that arrive at the same input share the same buffer as in the architecture proposed by Del Re and Fantacci [195]. In ATM there is a priority flag in the header data that facilitates two priorities of traffic. If this model were expanded to have two buffers then behaviour for both priorities could be measured [196]. There are four states. The quiescent state is *wait.* When a cell arrives from the input channel it moves into state *inc* . If the buffer is already full the cell is dropped (state *drop*) otherwise the buffer is incremented and the component returns to *wait.* When a succeeding switching element reads a cell from the buffer it moves into state *dec*

and the buffer is decremented by one cell before returning to state *wait.*

Note that all of the transitions are immediate, this is in order to achieve synchronisation. For example, consider the state *inc*. This state is entered when a cell arrives on channel *ch_in0*. Cells arrive in one time unit. This requires the component to move into *inc* and back to *wait* in one time unit and hence this component would move into *wait* at the same time the input channel is moving out of *arrive*. Since these two transitions are happening in the same time unit the order cannot be guaranteed. If the *inc ->wait* occurs first, the input channel will still be in state *arrive* causing this component to re-enter *inc* and falsely record another cell arrival. By putting the transition condition that the component cannot move out of state *wait* until the input controller moves out of state *arrive* this error is prevented. The corresponding behaviour statement is shown in listing 7.2.

```
BEHAVIOUR be_qi0 {
        IF ch_in0.arrive {
                0 wait -> inc; }
        IF ANY(sea0.read0, sea1.read0) {
                0 wait -> dec; }
        IF qi0.qlen > FULL {
                0 inc -> drop; }
        ON_EVENT ch_in0.quiet {
                0 inc -> wait; }
        ON_EVENT ALL(!sea0.read0, !sea1.read0) {
                0 dec -> wait;
                0 drop -> wait; }
}
```

Listing 7.2 Input controllers BEHAVIOUR statement

The counter *total* stores the total number of cells that have arrived, *qlen* gives the instantaneous length of the queue and *drop* gives the number of cells that have overflown the buffer.

7.4.4 Operation of the cross-bar switches

From figure 7.3 it can be seen that each switching element contains a cross-bar switch. At each time slot these switches will either be in the *cross* or *bar* state and thus dictate which queue the switching element will be reading from. The operation of these switches is modelled by two components per switch. One component represents the top branch of the

switch and one the lower. Each can exist in the two states *cross* or *bar*. The state diagram is given in figure 7.6.

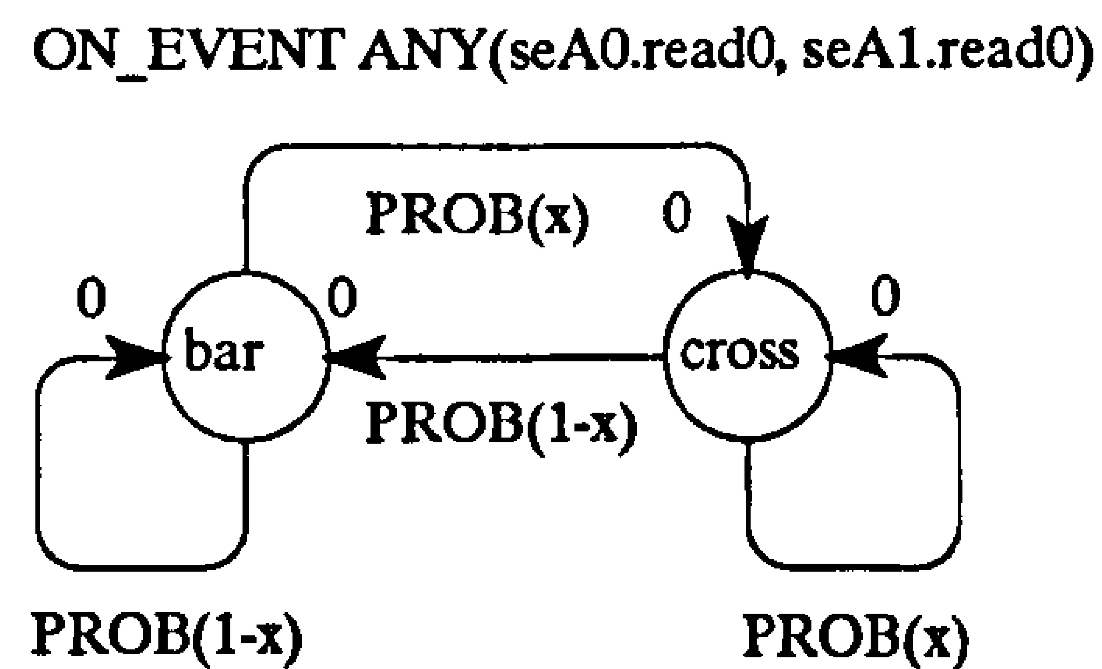


Figure 7.6 State diagram of cross-bar component

For a balanced routing all probabilities will be 0.5. By changing these probabilities the route can be altered. In the listing 7.3 of the BEHAVIOUR statement for an upper branch component the probabilities are set so that there is a bias for the *bar* position. This means there is more traffic arriving at the upper input destined for the upper output than for the lower output. The probabilities for the lower branch are all set to 0.5, hence traffic arriving at the lower input will have equal likelihood of being destined for either output. By adopting this approach, which allows flexibility in the balance of traffic, we can investigate Bruneeli and Wittevongel's [197] finding that queuing deteriorates in output buffered SEs as correlation in the routing gets higher.

```
BEHAVIOUR be_dest_in0 {
   ON_EVENT ANY(sea0.read0, sea1.read0) {
      0 cross -> bar PROB(0.65);
      0 cross -> cross PROB(0.35);
      0 bar -> cross PROB(0.35);
      0 bar -> bar PROB(0.65);
   }
}
BEHAVIOUR be_dest_in1 {
   ON_EVENT ANY(sea0.read1, sea1.read1) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}
```

Listing 7.3 BEHAVIOUR statement for cell routing

Note that the component changes state each time the switching element has read a cell from

the proceeding buffer.

7.4.5 The switching elements

In many mathematical models a general expression is derived which expresses the output conditions dependant upon the input and it is not possible to monitor the internal performance of the switch. For many applications this type of method is appropriate as loss probability is a comprehensive enough measure of performance [198]. In this model however we wish to monitor the behaviour of various queues within the interconnection network and switching elements are therefore modelled individually. Each switching element is represented by four components. This seems verbose on first inspection but when examined it allows for simplicity. Two components are required for the two queues. It would be possible to represent the two queues by two counters in one component but by using one counter each in separate components it allows the queues to function in parallel without state transitions being delayed. This reflects the operation of the hardware design.

Initially each queue and its operation was modelled by one component. This is restrictive as checks for read and write operations had to be made sequentially. The final implementation uses two components. The first is used to monitor whether a queue may read from a proceeding queue during each time slot and the second handles the actual updating of the queue. The state diagram of the first component is shown in figure 7.7.

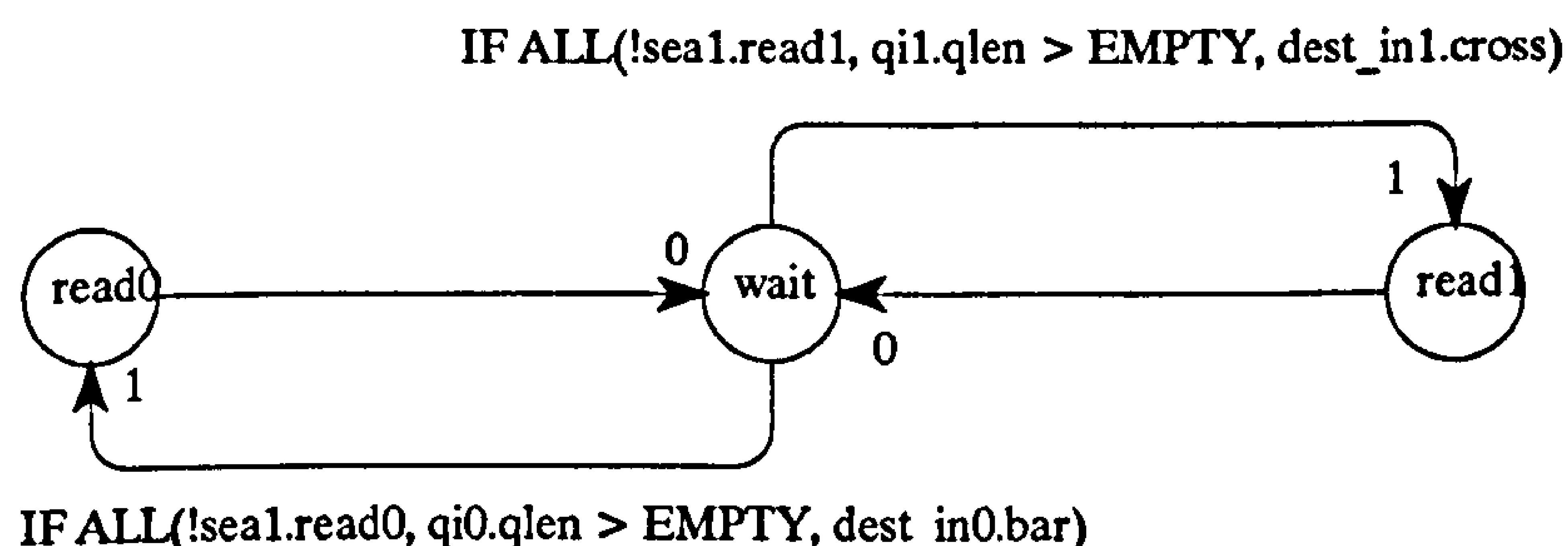


Figure 7.7 State diagram of SE queue reading component

From the state diagram it can be seen that the queue can read from either the proceeding 0 (upper) or 1 (lower) queue. The conditions for reading the proceeding 0 buffer are that the other queue in the SE is not currently reading from it, the queue is not empty and the

cross-bar is in the *bar* position. The conditions for reading from the proceeding lower queue are similar but the cross-bar must be in the *cross* position. The position of the cross-bar is determined by the cross-bar component discussed in section 7.4.4 which selects cell routing. Note that only one cell may be read in one time slot, following the operational procedure proposed by Jenq [191]. The behaviour statement for this component is shown in listing 7.4.

```
BEHAVIOUR be_sea0 {
      IF_ON ALL(!sea1.read0, qi0.qlen > EMPTY, dest_in0.bar) {
            1 wait -> read0; }
      IF_ON ALL(!sea1.read1, qi1.qlen > EMPTY, dest_in1.cross) {
            1 wait -> read1; }
      0 read0 -> wait;
      0 read1 -> wait;
}
```

Listing 7.4 BEHAVIOUR statement for queue reading component

The component will firstly check to see if there is a cell in the upper preceding buffer and if it is destined for the upper queue. If so it will read it, if not it will check the lower buffer. There is no read operation during the time slot if there are no cells available or if the queue is blocked by the complimentary queue reading from the required preceding buffer. By making the SE time slots faster than the networks time slots (say a speed-up factor of two) it would be possible for each SE queue to read from the same preceding buffer in the same SE time slot [199]. The model could be simply changed to encompass this feature by changing the timing on the transitions. Speed-up can also be accomplished at switch level [200] but is limited by the network speed.

The component that models the updating of the queues has the same state diagram as that for the input controllers shown in figure 7.5 and the behaviour is identical. The corresponding behaviour statement is given in listing 7.5.

```
BEHAVIOUR be_sea0_q {
      IF_ON ANY(sea0.read0, sea0.read1) {
            0 wait -> inc; }
      IF sea0_q.qlen > FULL {
            0 inc -> drop; }
      IF_ON ANY(seb0.read0, seb1.read0) {
            0 wait -> dec; }
      ON_EVENT ALL(!sea0.read0, !sea0.read1) {
            0 inc -> wait;
            0 drop -> wait; }
```

```
        ON_EVENT ALL(!seb0.read0, !seb1.read0) {
                0 dec -> wait; }
}
```

Listing 7.5 BEHAVIOUR statement for queue updating component.

Note that all the transition timings are again 0. This allows the transitions to be wholly determined by the queue reading components and facilitates the possibility of a cell being read into and read from the same queue within one time slot.

7.4.6 The output controllers

The output controllers present a new challenge within themselves. What is required is a suitable buffer on each output port with consideration of both capacity allocation and overflow. This will be largely dependant upon the network to which the switch is connected [201]. Our aim is to concentrate on the switches behaviour and thus we assume infinite capacity queues in the output controllers. This assumption is equivalent to assuming that the output network is available to read one cell per time slot. The space diagram for the output controllers is shown in figure 7.8.

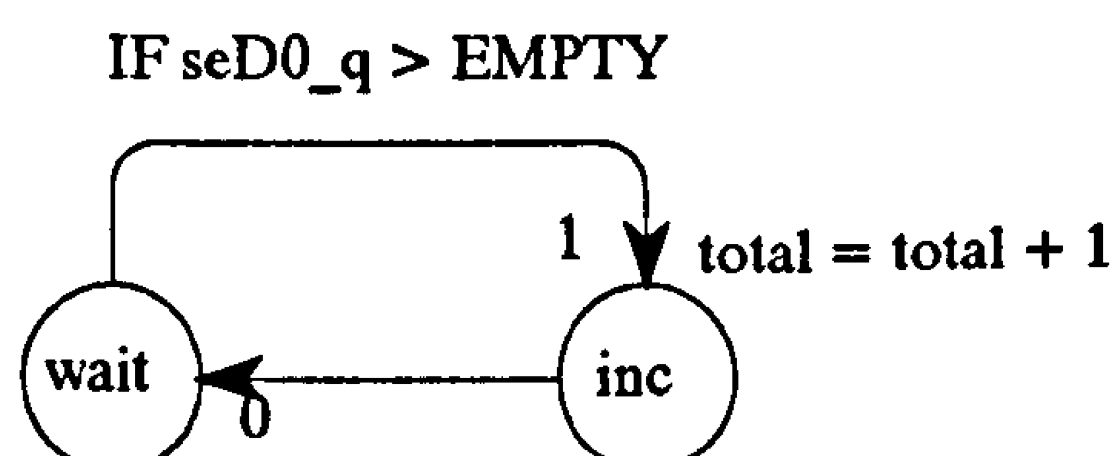


Figure 7.8 Space diagram of Output Controllers

The output controllers will read a cell from the final (4th) level switching elements each time slot if there is a cell to read. There is one output controller dedicated to each output port and hence there will be no blocking in these components. Counters have been associated to each output so that the total number of cells leaving the switch may be monitored. The behaviour statement is given in Listing 7.6.

```
BEHAVIOUR be_qo0 {
        IF_ON sed0_q.qlen > EMPTY {
                1 wait -> inc; }
        0 inc -> wait;
}
```

Listing 7.6 BEHAVIOUR statement of Output Controllers

7.5 Model validation

With a model of this size and complexity it is necessary to analyse its behaviour to ensure that it reflects correctly the operation of the system being modelled. The post processor *viz* is a suitable tool for this. Full validation required two steps, the first being to examine a textual event trace of a simulation of the model to ensure components behave as expected, and the second being to run a short simulation and examine the resulting counter values.

For the first step a short simulation (100 time slots) was run. The textual event trace for this simulation was obtained using *viz*. Each type of component was considered in turn. Every transition was examined for each component type to ensure the firing and timing corresponded to that which was expected. This step highlighted the timing problems that were discussed in section 7.4.3 and thus proved a valuable technique.

For the second step all the counter values were noted at the end of simulation from the previous event trace. These values are given in table 7.1. The validating technique is as follows. For each pair of components, eg sea0 and sea1, the sum of the *total* values minus the sum of the *qlen* values should be equal to the sum of the *total* values for the following pair of components. By performing this check for each pair of components correct counter operation can be confidently determined.

Cnt. Check	Components											
Counters	qi0	qi1	sea0	sea1	seb0	seb1	sec0	sec1	sed0	sed1	qo1	qo0
total	65	56	57	61	50	59	61	47	50	57	50	56
qlen	2	1	1	8	0	1	1	0	0	1	50	56

Table 7.1 Validation of counter operation

7.6 Results

In this section we present some of the results from simulating the model.

7.6.1 Performance parameters

The performance parameters used are the mean queue length of each queue in the input controllers and switching elements. These values can be directly determined from the I_SIM post processor *tpp*. The mean queue values are determined for various combinations of traffic load and routing balance.

The switch model has two distinct routing paths, the *upper* and *lower*. These may be considered as one for balanced routing conditions. In the case of unbalanced routing they are taken separately and compared to determine the effect of the routing.

7.6.2 Simulation parameters

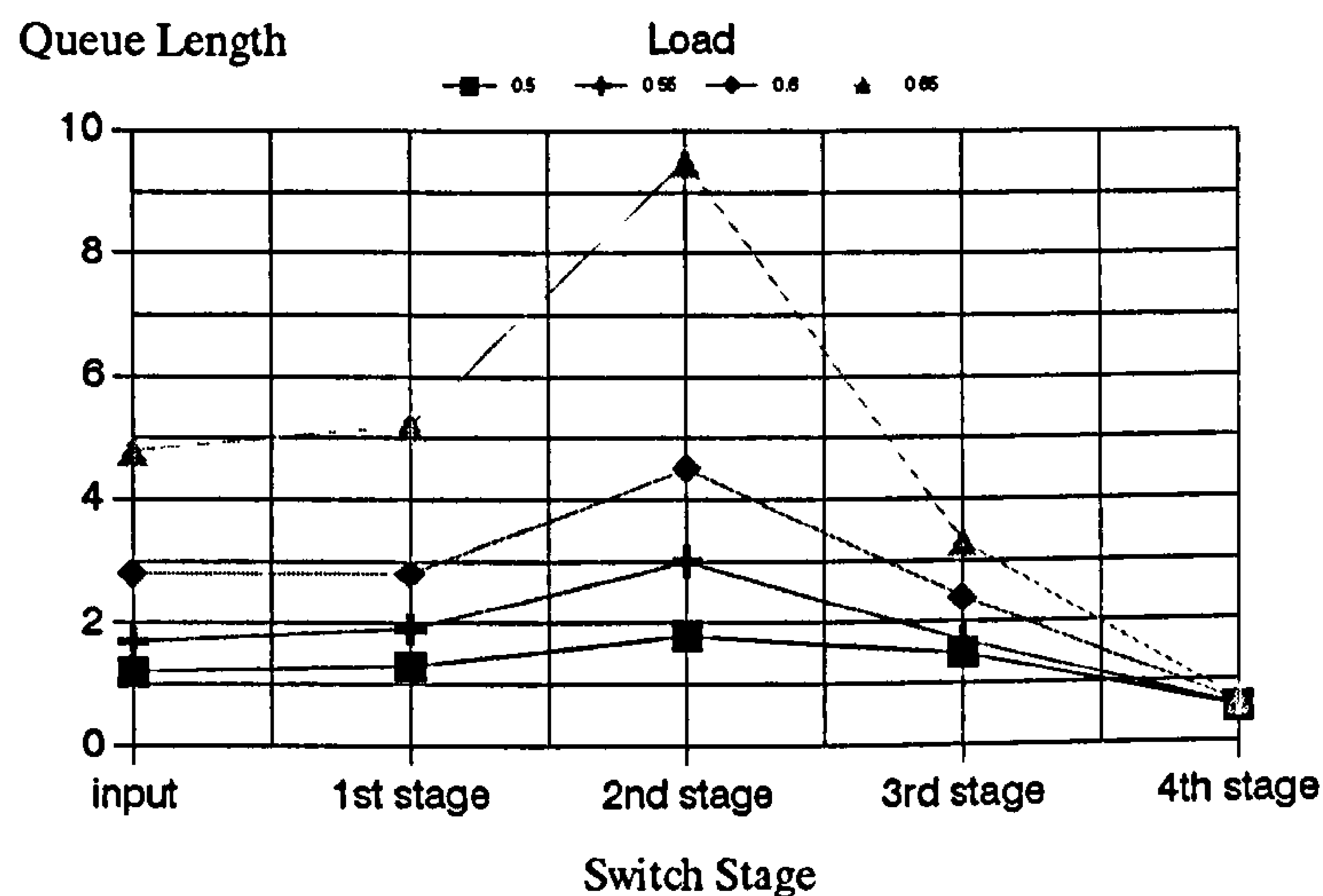
The model of the input traffic described in section 7.4.2 is manipulated to give the required variation in load. Referring to listing 7.1, the transitional probabilities govern the load value and are altered accordingly. The offered load is critical as cell loss is more sensitive to load than to queuing delay [202]. For the results presented below the load was varied from 0.50 to 0.65 in steps of 0.05.

The model of the cross-bar switches described in section 7.4.4 is manipulated to provide different routing balances. Referring to listing 7.3, the transitional probabilities reflect the routing balance. In this example input traffic on the upper link has a likelihood of 0.65 being destined for the upper output, whilst traffic from the lower input is balanced. For the results presented below the lower path is kept balance and the balance of the upper path is varied from 0.50 to 0.65 in steps of 0.05.

For each case the simulator was run for 10 trials of 10000 time slots each. Each group of runs was analysed by the I_SIM post processor *tpp.*

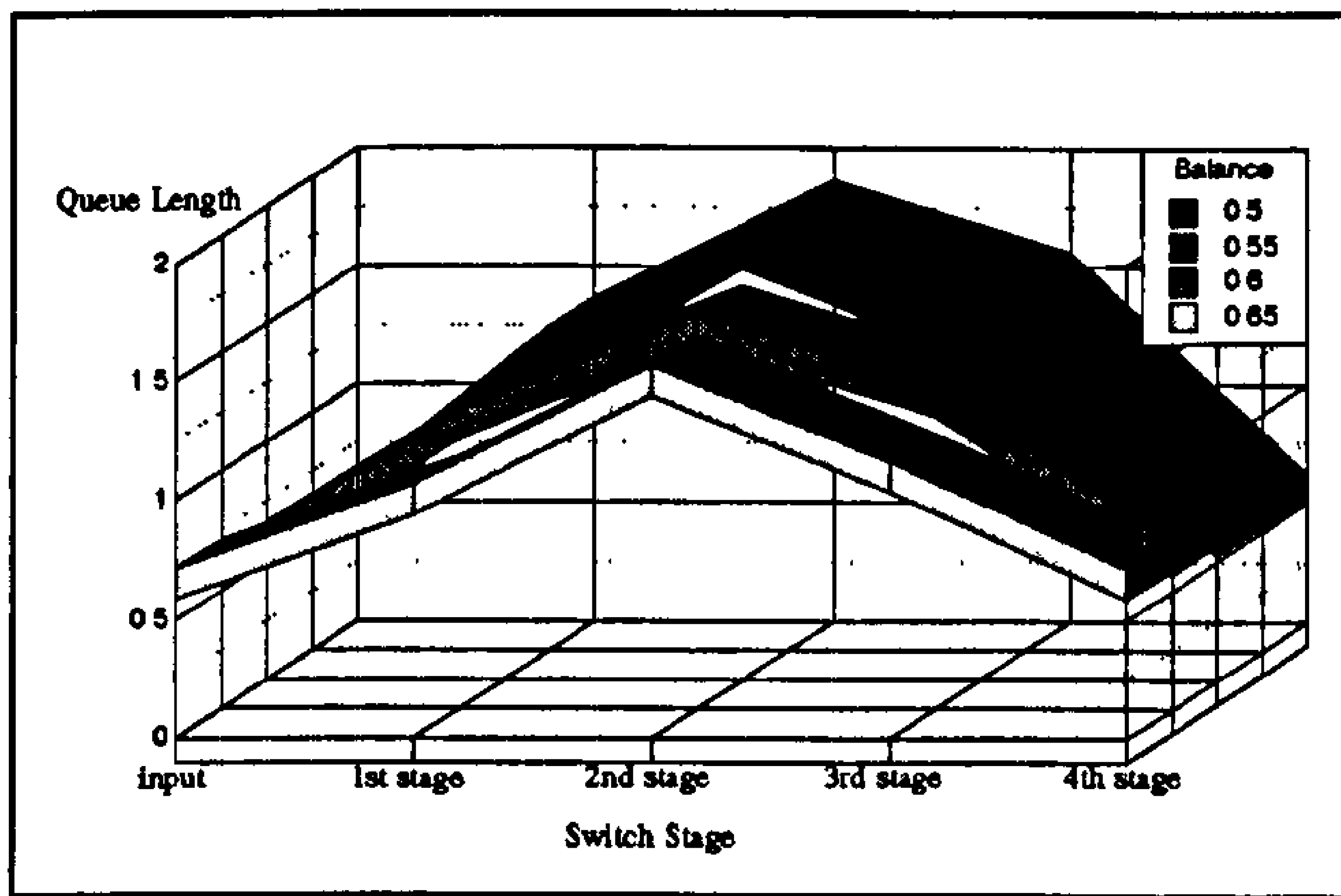
7.6.3 Simulation results and observations

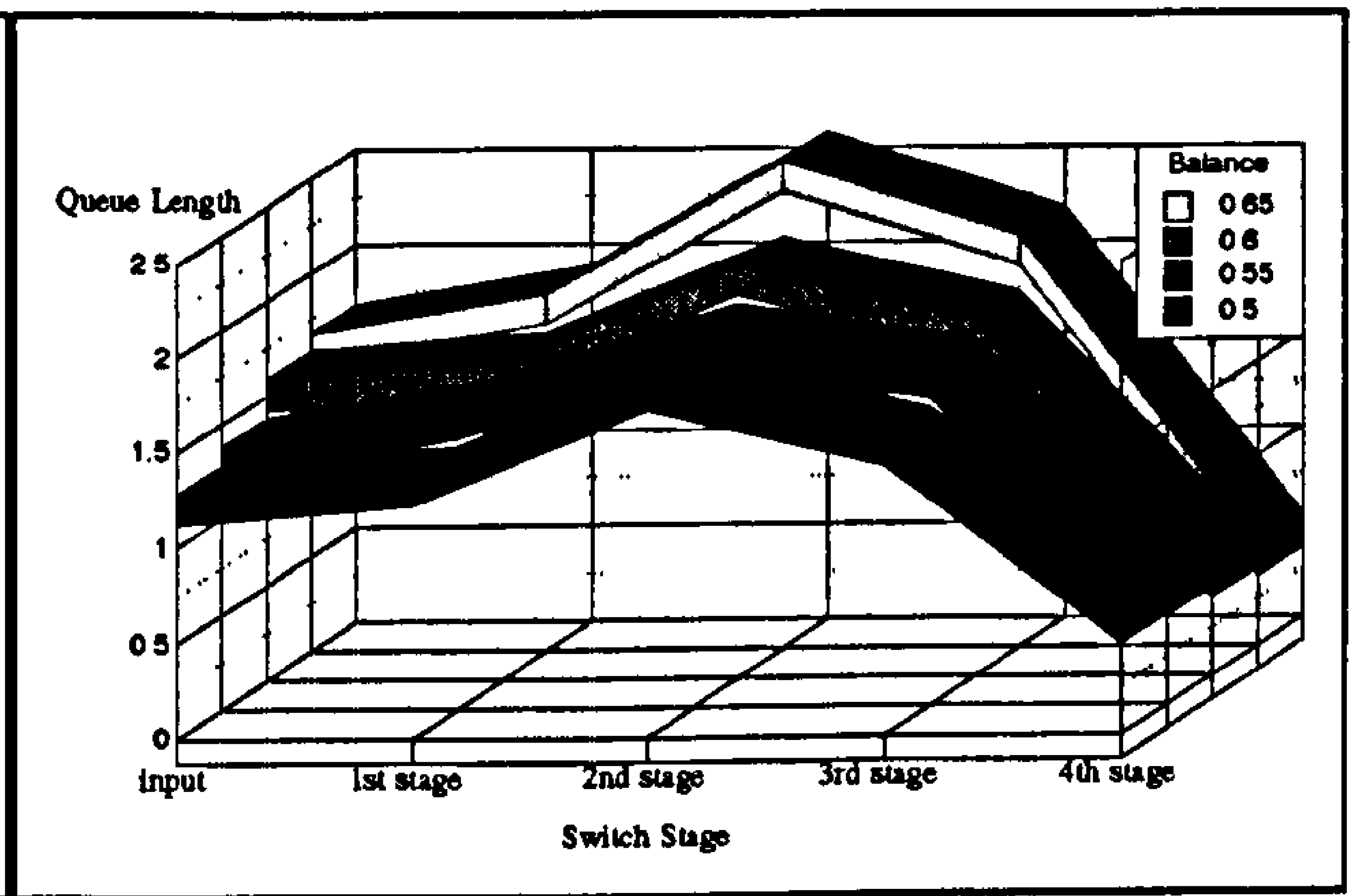
The graphs below present a selection of the simulation results. Graph 7.1 shows the mean queue lengths of the input controller and each stage for various traffic loads.



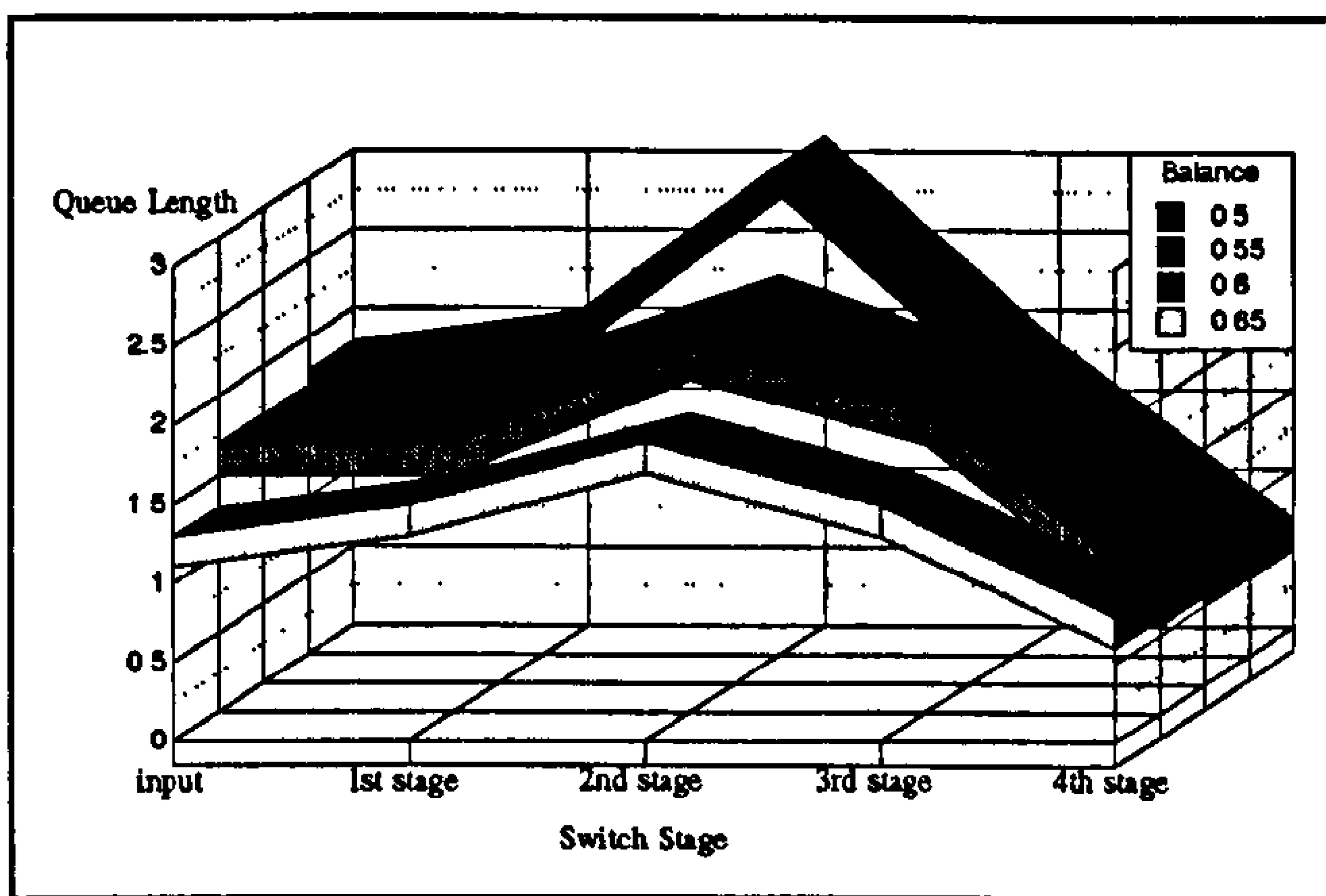
Graph 4.1 Queue length for varied balanced loads

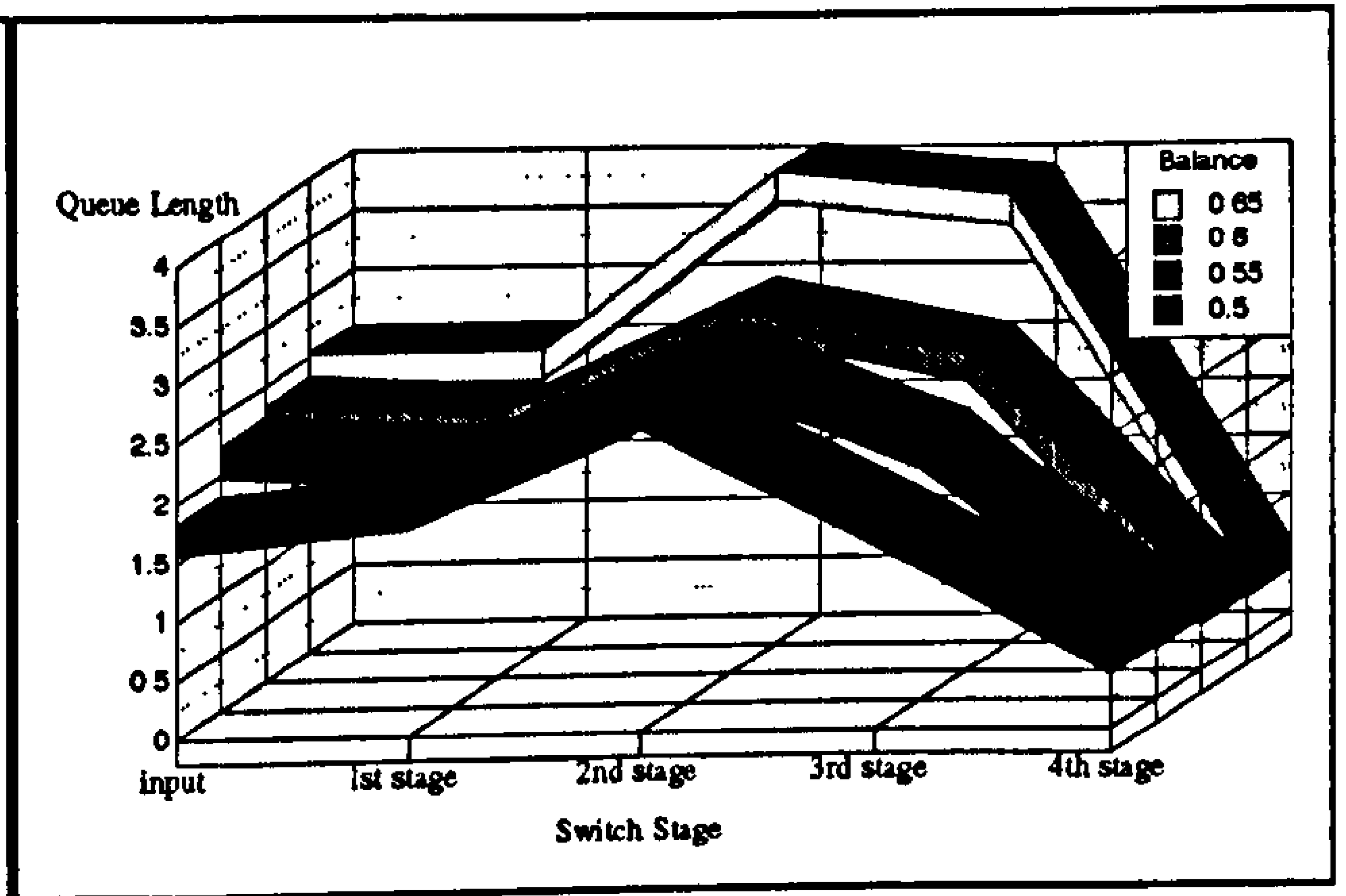
It is interesting to note the behaviour of the second stage queue and that the ratio of this queue length to the others increases with load. As expected all queue lengths increase with load save for the 4th stage which is modelled as feeding an infinite capacity network. The most marked increase in queue length is from a load of 0.60 to 0.65, reflective of the graduation towards maximum load. Each of the graphs 4.2 to 4.9 show the mean queue lengths for different traffic loads as a function of routing balance.



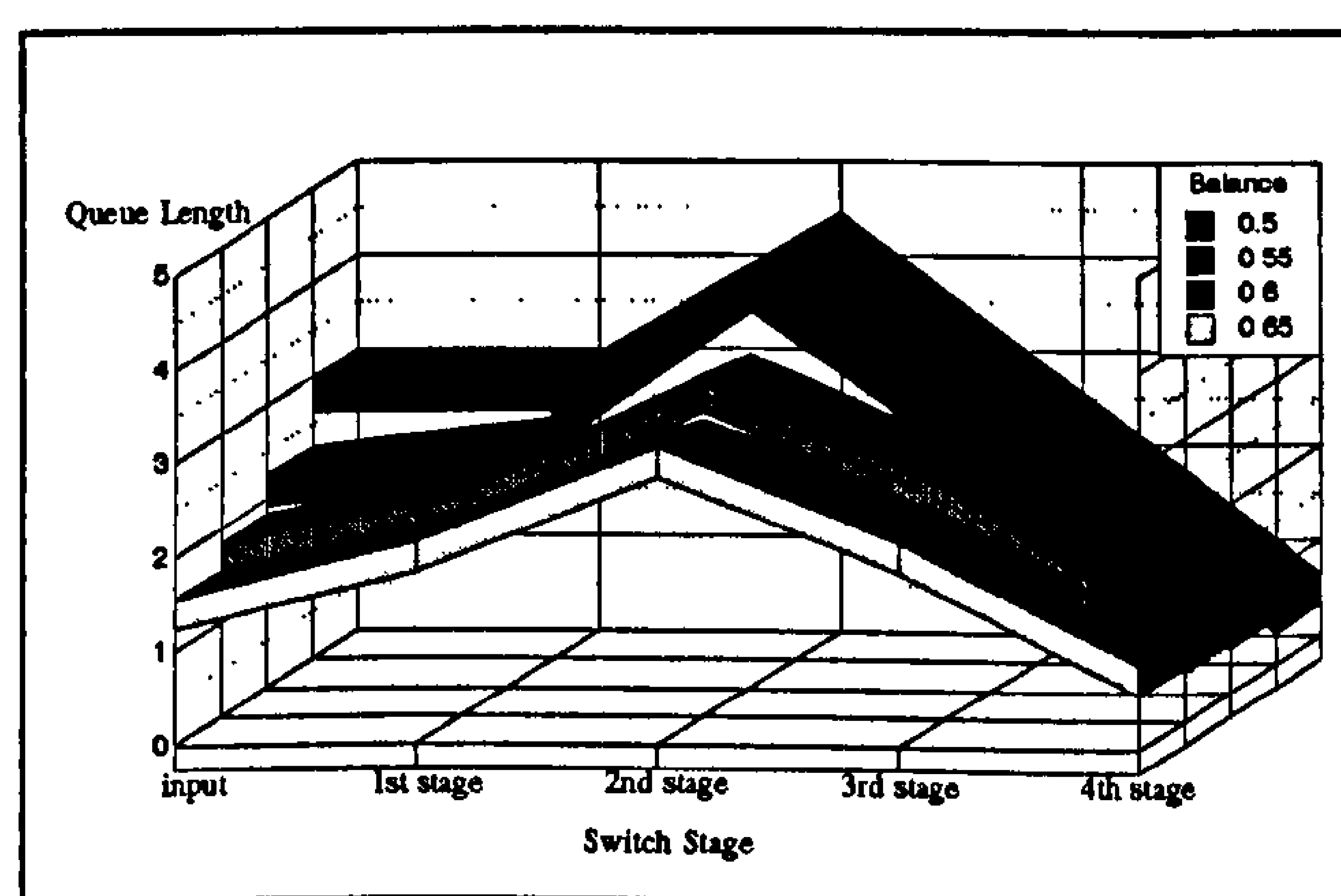


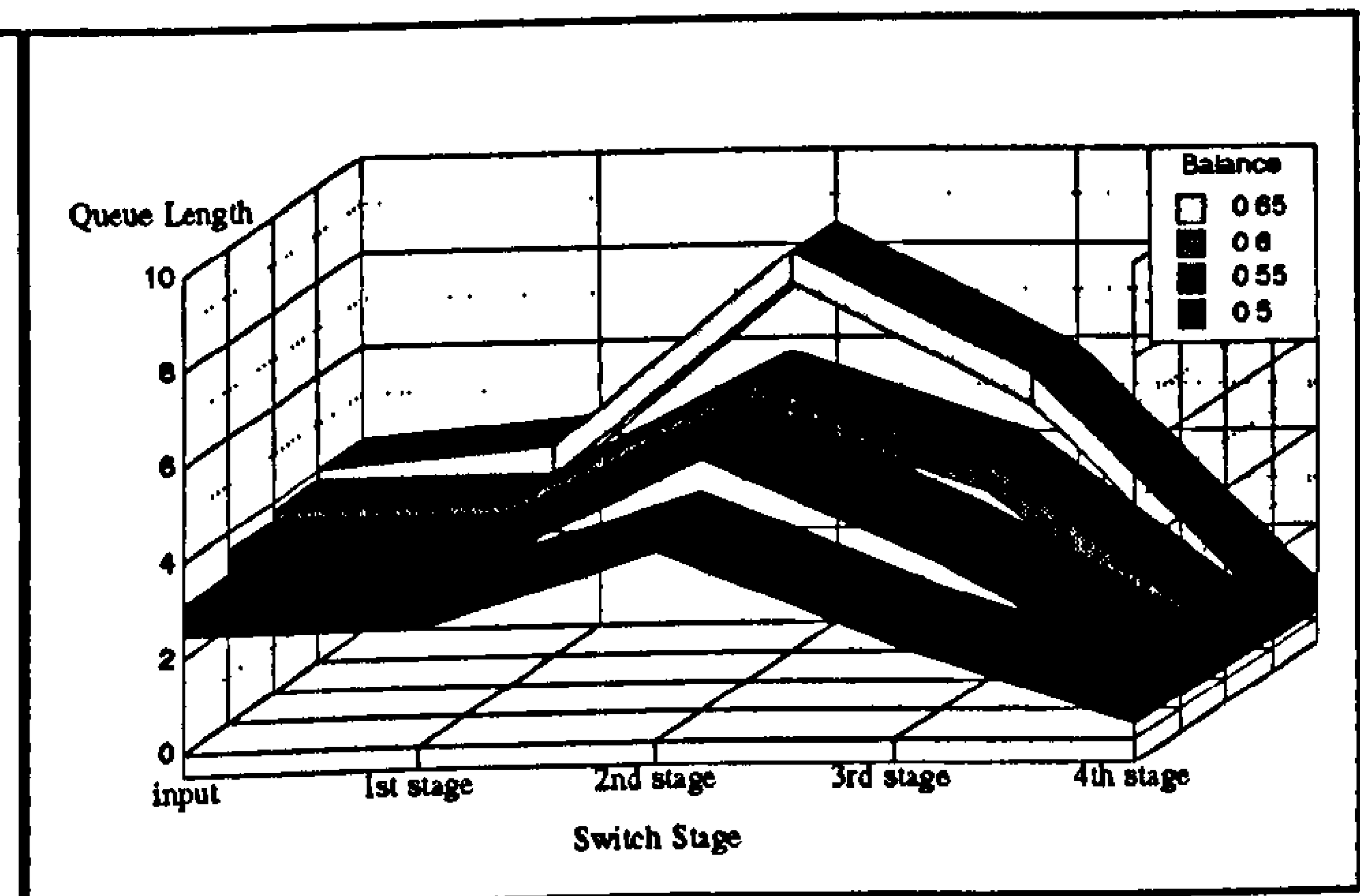
Graphs 4.2 &4.3 Unbalanced upper and lower paths for load of 0.50



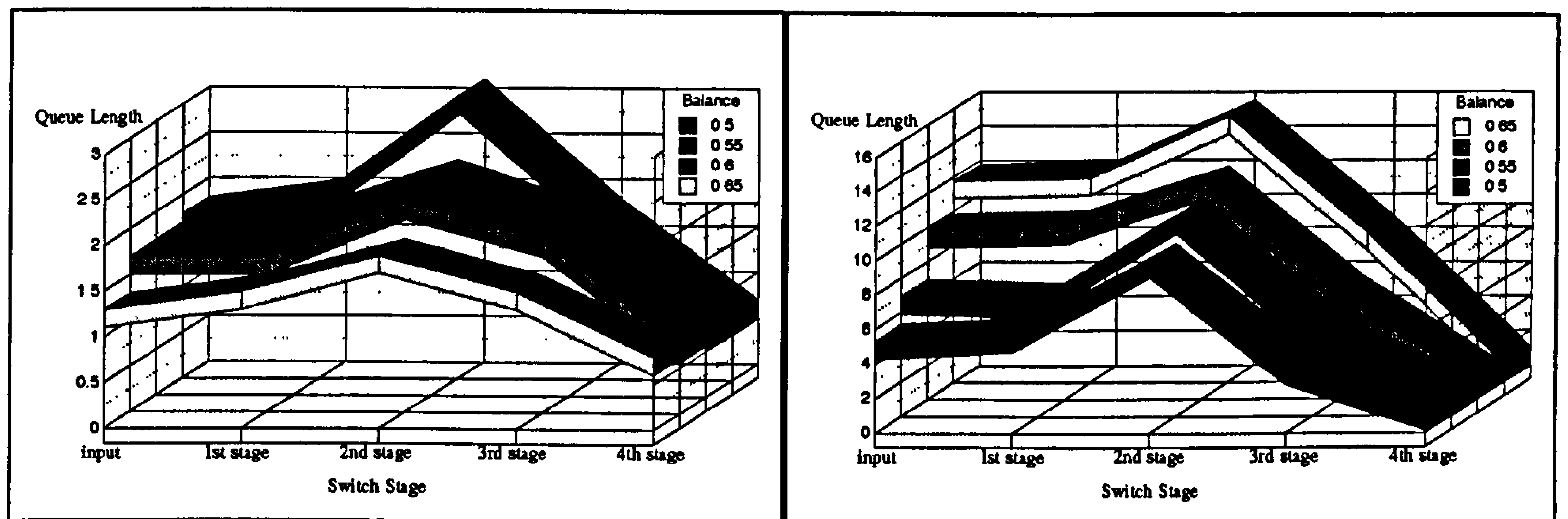


Graphs 4.4 & 4.5 Unbalanced upper and lower paths for a load of 0.55





Graphs 4.6 & 4.7 Unbalanced upper and lower paths for a load of 0.60



Graphs 4.8 & 4.9 Unbalanced upper and lower paths for a load of 0.65

These graphs again show the importance of the second stage queue, with the lower queue being slightly longer due to the imbalance of traffic routing. The third stage queue on the lower path shows a marked increase over its upper counterpart. This we can assume is due to the priority given to the upper path causing the lower cross-bar, cross path, to be subject to head of line blocking. As would be expected , the queue lengths increase with load. Interestingly the disparity between the upper and lower paths also increases with traffic load.

This model may be replicated to provide a two switch network and used to determine the effect of switching and queuing on network performance. This work has been proposed by Friesein and Wong [203] who modelled two switches with various source interacting across a network for loads from 0.6 to 0.8. Expansion on this to examine the effects of the network and other switches on the performance of the various stages of our switch would be an interesting area of further work.

7.7 Conclusions

A complex performance model of an ATM Delta-2 16x16 Banyan switch architecture has been developed in ICE. This has demonstrated that the ICE language is capable of modelling intricate systems comprising inter-dependant components and the I_SIM simulator's ability to simulate and analyse such systems. Analysis of the textual event trace was used to obtain a functually correct model and the results produced by the *tpp* post processor revealed some interesting insights into the switches behaviour under various conditions.

The tight timing restrictions of the model proved to be a rigorous testing ground for the language. Initial development revealed some challenges to obtain correct sequencing of simulation events. A solution was presented which facilitated correct synchronisation and the lessons learned prove useful for future modelling.

The amount of code required was reasonable (~300 lines, formatted) for the complexity of problem. The use of counters to model queues contributed considerably to the restriction of the code to this size. The generic nature of the language allowed for a very modular design with many constructs being reused with minimal editing.

The work presented in this chapter has been the subject of a published paper which is reprinted in appendix E.

Chapter 8

Performability modelling with ICE

8.1 Introduction

In this chapter we come to a significant testing point for ICE. The language's ability to model intricate systems comprising many interacting components has been demonstrated as has it's use in obtaining performance and dependability measures from such systems. We now turn our attention to the complex field of performability modelling.

Chapter 2 gave a detailed account of the development of performability modelling and the techniques employed were presented. In this chapter some of these techniques are adopted and adapted for ICE. We identify some of the important measures previously discussed and apply these to the ICE models.

Rather than develop specific models which may be limited to certain applications we propose a generic ICE performability framework. This framework utilises the proven concept of reward models. It is shown how this generic approach can be used to obtain suitable performability measures. The method is illustrated by the use of an example

model of a multiprocessor system.

8.2 Distribution of accumulated reward

Distribution of accumulated reward is the classic performability measure. It gives a revealing insight into a systems behaviour and facilitates a better understanding of its operation. Smith et al [205] give an example of a multiprocessor system whose behaviour is not fully described by expected values of reward but requires the comprehensive information contained in the distribution of accumulated reward.

The distribution of accumulated reward is the probability a system will complete a given number of tasks during the time interval [0,t). This is effectively an index of system productivity and allows appropriate system capability to be determined. Reward rates may be assigned to appropriate states. A task is taken to be completed once the cumulative reward of the system is equal to or greater than that required by the task.

Let $Y(t)$ be the instantaneous reward rate and $F(t,y) = P\{Y(t) \leq y\}$ denote the distribution function of the accumulated reward. The complementary distribution

$$F^C(t, y) = 1 - F(t,y) = P\{Y(t) > y\}$$

can be used to answer the important question : *What is the probability that an amount of work, y, will be completed by the system during the interval [0,t) ?*

$Y(\infty)$ is not defined for systems without absorbing states and hence when $t \rightarrow \infty$, the distribution of accumulated reward can be fully defined only for a system with imperfect repair. In this case Beaudry [52] proposes a method for calculating the distribution. The approach is based on transforming the original Markov chain, X, into an equivalent one, X', with the same state space but with the generic transition rate determined by dividing the transition rate of the original chain by the reward rate of the departing state. This effectively transforms a time domain representation of the system into a computation (or reward) domain representation. The model changes state after a certain amount of computation rather than a period of time. The accumulated reward is then

the analog of the time domain availability, i.e. rather than viewing the model as showing the mean time to failure it shows the mean reward accumulated. The limiting distribution of $Y(t)$ for X can be obtained from the transient state probabilities of X' as it corresponds with the distribution of its time to failure. Ciardo et al [206] expanded this technique removing the restriction of non-zero reward rates for transient states by employing semi-Markov reward models.

8.3 The ICE reward model

In a performability model it is possible to allocate a reward assignment equal to a simple function of the state index. As an example, in a simple multiprocessor reliability/availability model the reward rate may be taken as being directly proportional to the number of functioning processors [52]. This approach is employed in the example of section 2.4.3.2 using stochastic reward nets (SRNs) and accommodates a simple solution. However this assumption is often erroneous. More meaningful models will normally require a performance evaluation of each state in the state space to derive suitable reward rates. In this instance the reward rate of each combination of functioning units is individually defined and thus accurately reflecs the real systesm. This adds significant complexity to SRN and Markov performability models.

With the ICE reward model the objective is to facilitate an accurate method of assigning correctly evaluated reward rates to each component state. This is implemented by using COUNTERS to monitor reward rates. The generic framework is described below.

① An ICE model of the system of interest is developed as normal. The reward model will be an extension to this, as opposed to a seperate model.

② Each component within the system that we wish to obtain performability measures from has a reward counter associated with it. When the component enters a given state this counter is set to the reward rate of that state. The reward rate is the reward per time unit (the size of the time unit being set

appropriately by the modeller). This counter therefore monitors the *instantaneous reward rate* of the component. These counter values of different components may be summed to give the instantaneous reward rate of the system.

③ A separate component is used to monitor the *cumulative reward rate* of the system. At the end of every reward time unit the value of this counter is incremented by an amount equal to the instantaneous reward rate counter.

④ By accessing the *instantaneous reward rate* and *cumulative reward rate* counters any of the performability measures identified in section 2.4.2.1 can be determined.

This ICE generic performability modelling framework is best demonstrated by considering an example.

8.4 Multiprocessor example

In this example a typical multiprocessor system is modelled and performability measures obtained.

The system considered contains 4 processing units and can handle several tasks simultaneously. We are concerned with the processing of individual tasks that are submitted to the system. Each processing unit can exist in one of the two states *serve* or *busy*. In the *serve* state it is serving the specific task of interest. In the *busy* state it is serving another task or performing housekeeping duties and therefore not contributing towards the task of interest, i.e. it is a non-productive state. The state space of one of these units is shown in figure 8.1 and the corresponding ICE code in listing 8.1.

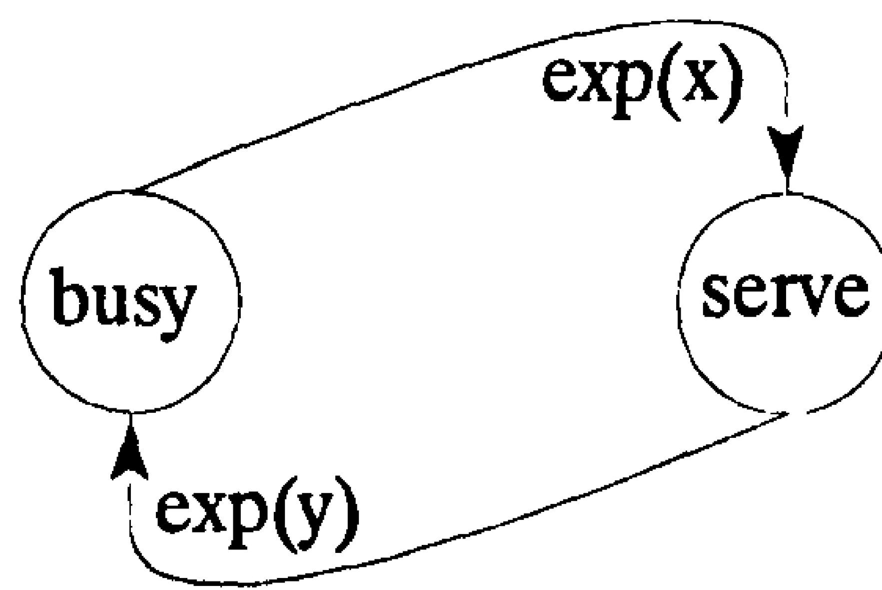


Figure 8.1 State space of *unit* component

```
STATE_SET ss_unit {
        COUNTERS ::;
        STATES {
                busy ::;
                serve ::;
        }
}

BEHAVIOUR be_unit {
        exp(x) busy -> serve ;
        exp(y) serve -> busy ;
}

COMPONENT unit1, unit2 ,unit3 ,unit4 {ss_unit; be_unit; busy;}
```

Listing 8.1 ICE code for *unit* component

To determine the total amount of work being done on the task of interest at any given time the SYSTEM statements in listing 8.2 are used.

```
SYSTEM up0 = ALL(!unit1.serve,!unit2.serve,!unit3.serve,!unit4.serve);
SYSTEM up1 = EXACTLY1(unit1.serve,unit2.serve,unit3.serve,unit4.serve);
SYSTEM up2 = EXACTLY2(unit1.serve,unit2.serve,unit3.serve,unit4.serve);
SYSTEM up3 = EXACTLY3(unit1.serve,unit2.serve,unit3.serve,unit4.serve);
SYSTEM up4 = EXACTLY4(unit1.serve,unit2.serve,unit3.serve,unit4.serve);
```

Listing 8.2 SYSTEM statements monitoring number of serving units

In each statement, *upN*, *N* denotes the number of processing units currently serving the task of interest.

The state of the system and the *instantaneous reward rate* is modelled by the *processor*

component. It has 6 states, the initial state *init* and states *proc0..proc4*, indicating how many units are currently serving the task of interest as determined by the SYSTEM statements of listing 8.3. This component has the counter *reward. Reward* is set on entry to each state to be equal to the reward rate for that state and hence contains the *instantaneous reward rate* of the system. The state space for the *processor* component is shown in figure 8.2 and the corresponding ICE code in listing 8.3.

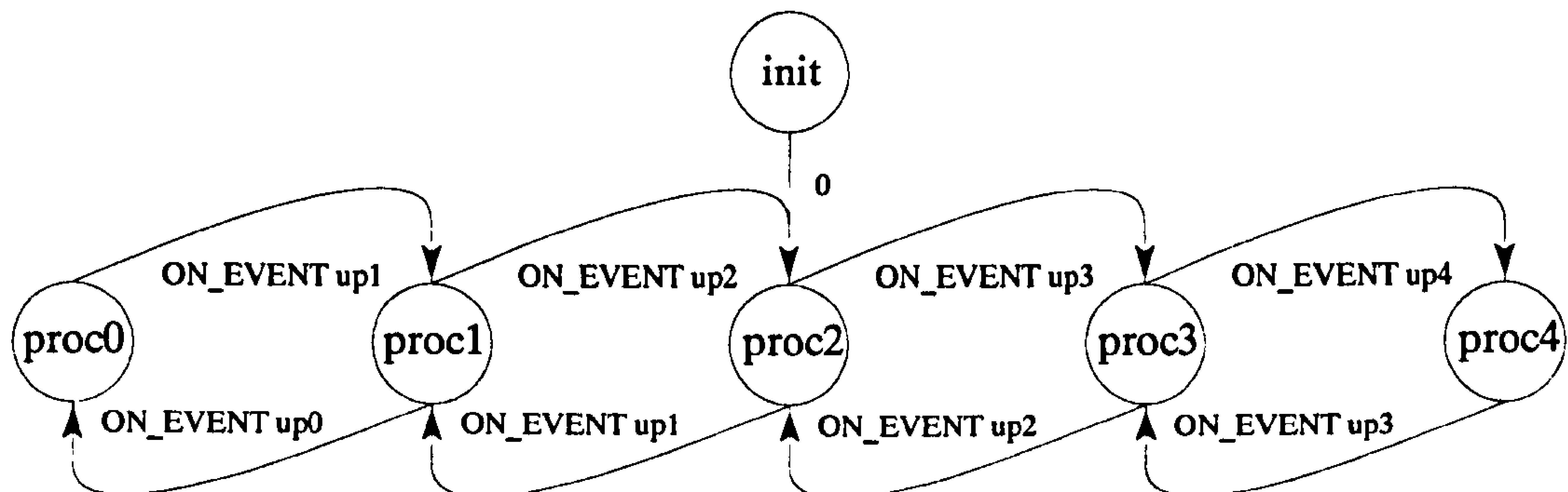


Figure 8.2 State space of *processor* component

```
STATE_SET ss_processor {
        COUNTERS : reward;
        STATES {
                init :;
                proc0 : {reward = 0};
                proc1 : {reward = 10};
                proc2 : {reward = 15};
                proc3 : {reward = 22};
                proc4 : {reward = 27};
        }
}
BEHAVIOUR be_processor {
        0 init -> proc2;
        ON_EVENT up0 {  0 proc1 -> proc0;}
        ON_EVENT up1 {  0 proc0 -> proc1;
                        0 proc2 -> proc1;}
        ON_EVENT up2 {  0 proc1 -> proc2;
                        0 proc3 -> proc2;}
        ON_EVENT up3 {  0 proc2 -> proc3;
                        0 proc4 -> proc3;}
        ON_EVENT up4 {  0 proc3 -> proc4;}
}
COMPONENT proc{ss_processor; be_processor; init(reward = 0);}
```

Listing 8.3 ICE code for *processor* component

The reward values shown in the STATE_SET statement of listing 8.3 are typical values for multiprocessor systems. These rewards are the *performance measures* of the system.

To obtain the *cumulative reward rate* a further component, *check*, is used. *Check* has a counter *record* and two states, *wait* and *inc*. At the beginning of a simulation *check* is in state *wait* and counter *record* is set to zero. At the end of every reward time unit the state *inc* is entered. *Record* is incremented by an amount equal to the value of counter *reward* of component *processor* (the *instantaneous reward rate* counter) and hence contains the *cumulative reward rate.* The state space of *check* is shown in figure 8.3 and the corresponding ICE code in listing 8.4.

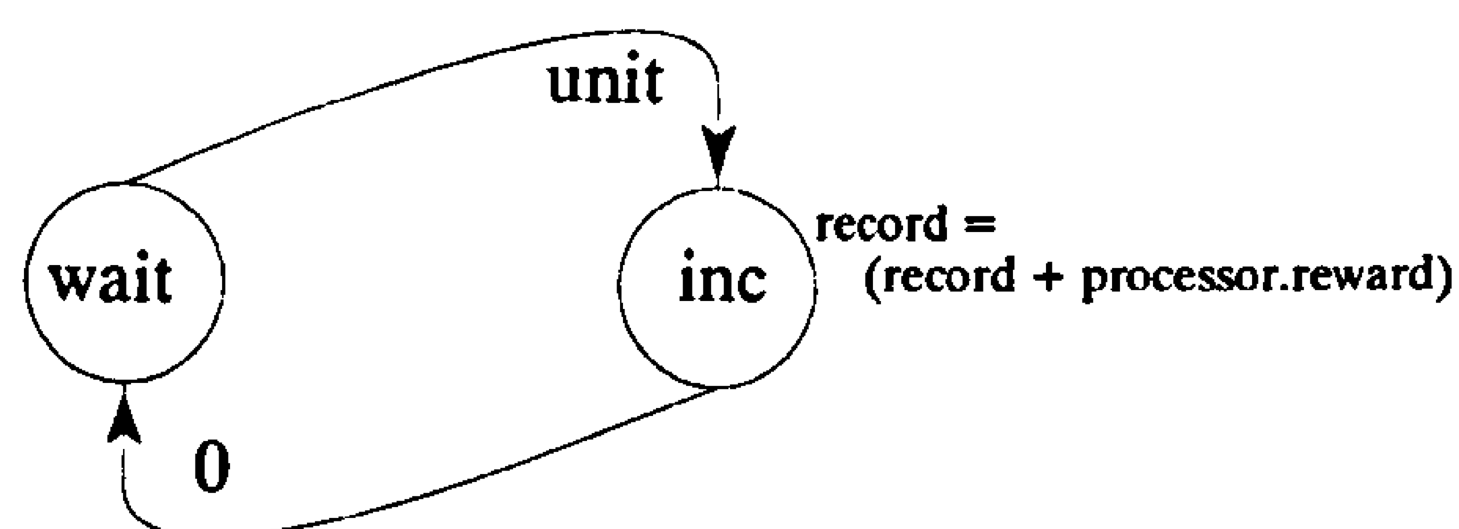


Figure 8.3 State space of *check* component

```
STATE_SET ss_check {
        COUNTERS : record;
        STATES {
                wait :;
                inc  :{record = record + proc.reward};
        }
}

BEHAVIOUR be_check {
        1 wait -> inc;
        0 inc  -> wait;
}

COMPONENT check {ss_check; be_check; wait(record = 0);}
```

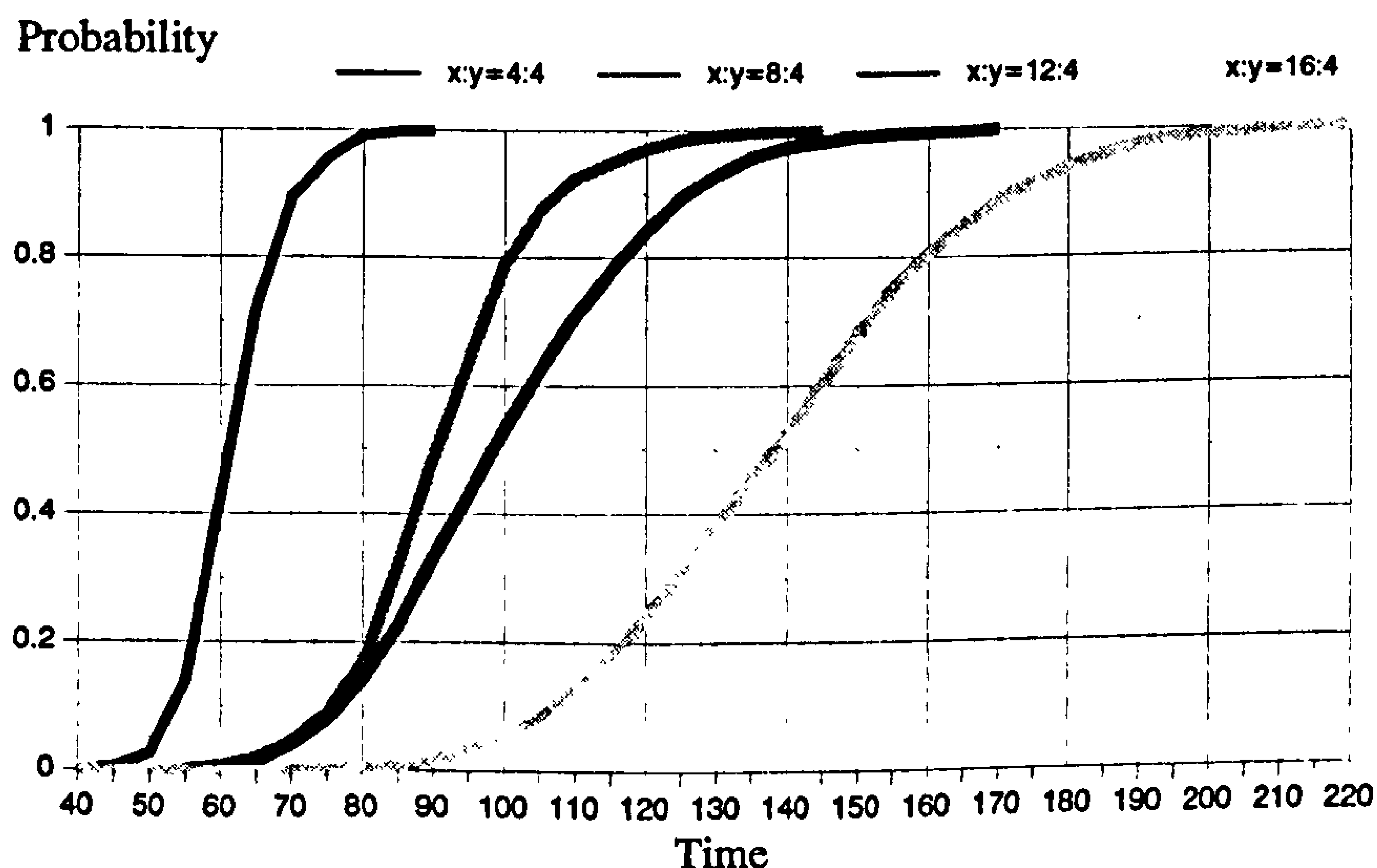
Listing 8.4 ICE code for *check* component

8.4.1 Performability measures

The multiprocessor model was simulated using I_SIM. The exponentially distributed sojourn times of the processing units determine the *availability* of the system. These sojourn times are dictated by the values of *x* and *y* in the *unit* component behaviour statements. The ratio of time spent in the *busy* and *serve* states is the ratio of *exp(x)* to *exp(y)*. This was varied from 4:4 to 16:4 to determine the impact on the performability measures. For each set of parameters 1000 runs were done. Statistical information on the values of the instantaneous and cumulative reward rate counters was obtained using the *tpp* analyzer. This information can be used to calculate the performability measures described in chapter 2.

Distribution of accumulated reward

For a known size of task the probability of completing the task within a given time can be determined from the distribution of accumulated reward. This distribution is obtained by recording the value of the cumulative reward counter at known time intervals during the simulations. Points are then plotted as the percentage of times the cumulative reward is equal to or greater than the total reward a given task requires. In this example the task size was 1000 reward units.



Graph 8.1 Distribution of accumulated reward for task size of 1000 reward units.

Graph 8.1 reveals that as the proportion of time the processing units spend serving the given task decreases then the time to task completion increases as would intuitively be expected. As the expected time to task completion increases so does the range of possible completion times. This is due to the variations in state sojourn time around the mean having an increased influence on the reward being accumulated.

Expected instantaneous reward

The expected instantaneous reward, $E[\Upsilon(t)]$, was shown in section 2.4.2.1 to be

$$E[\Upsilon(t)] = \sum_{i \in S} r_i P_i(x)$$

where S is the set of states being monitored for reward. In this example all the states of component *processor*. The subscript i denotes each state *proc0 .. proc4*. The reward rate of each of these states is denoted by r_i and $P_i(x)$ is the probability of *processor* being in state i at any time x during simulation. For each of the states the value of reward rate used in the example and the values of P_i obtained by *tpp* are given in table 8.1.

state	r_i	$P_i(x)$			
		x:y = 4:4	x:y = 8:4	x:y = 12:4	x:y = 16:4
proc0	0	0.063	0.193	0.313	0.401
proc1	10	0.256	0.390	0.416	0.410
proc2	15	0.374	0.303	0.217	0.160
proc3	22	0.247	0.101	0.050	0.027
proc4	27	0.066	0.013	0.004	0.002

Table 8.1 Selected values of reward and calculated state occupancy probabilities

$E[\Upsilon(t)]$ can hence be calculated giving the results shown in table 8.2.

sojourn times	4:4	8:4	12:4	16:4
$E[\Upsilon(t)]$	15.386	11.018	8.623	7.148

Table 8.2 Calculated values of expected instantaneous reward

The expected instantaneous reward decreases in an exponential manner as the ratio of time the processing units spend servicing the given task decreases. If a number of task were submitted to the system with known reward requirements this would be a very useful measure for determining the processing unit service times required to complete the tasks without incurring a backlog.

Expected accumulated reward

The expected accumulated reward, $E[\phi(t)]$, was shown in section 2.4.2.1 to be

$$E[\phi(t)] = \sum_{i \in S} r_i L_i(t)$$

where L_i is the total time spend in state i during the period $[0, t]$. The subscript i again denotes states *proc0..proc4* in this example. Since this system is non-absorbing and each processing unit follows a simple cyclic behaviour, L_i, may be calculated as $L_i = P_i \times t$, other than for small values of t ($< 2x$) in which case L_i may be obtained by simulation. It follows that $E[\phi(t)] = E[\Upsilon(t)] \times t$, and thus may be simply determined for any value of t.

8.5 Conclusions

In this chapter we have applied ICE to performability modelling. This has been a significant testing area for the language to which it has proven well suited.

A generic ICE reward model framework has been proposed. This involves developing a complete ICE model of a system and then adding counters to monitor instantaneous and cumulative reward. The increase in complexity of the model due to the counters is negligible. After simulation by I_SIM the tpp analyzer can be used to obtain statistical information on the reward counters. With this information all of the performability measures identified in section 2.4.2.1 can be determined. The distribution of accumulated reward may be plotted giving valuable insight into a system's behaviour.

Detailed performability measures were obtained for a multiprocessor system example from a relatively simple ICE model. The operation of the system was investigated by observing the effect that altering model parameters had on these measures.

A hybrid approach is taken to performability modelling. This is a combination of measurement and model based evaluation. Performance parameters were measure based in the form of the processor reward rates. Availability parameters were obtained from simulating the system and hence are model based. This approach is flexible and should be modified to suit the particular application.

In the ICE framework appropriate reward rates are assigned to each state of interest. This is a notable benefit over Stochastic Reward Net and Markov techniques, where it is significantly simpler to assign reward rates that are proportional to some system parameter, such as number of functioning units, rather than on an individual state basis.

Chapter 9

Discussions and Conclusions

9.1 Overview

In this chapter we critically discuss some of the main issues presented in the thesis with an emphasis on how the work measures against the original objectives and ways in which it could be progressed. The conclusions given are additional to those at the end of each preceding chapters.

There are 5 main areas that are considered, these are

1. The ICE language.
2. Implementation of the simulator.
3. Random number generation.
4. The computational models of ICE.
5. Performability modelling with ICE.

Following the general discussion on these areas some specific ideas for further research are highlighted.

9.2 The ICE language

The objective of developing a simple, intuitive and powerful description and simulation formalism has been successfully met.

Perceived complexity and modelling power

With any descriptive language there is a trade-off between volume and complexity. A certain number of syntax constructs are required to provide a suitable descriptive space and as this number increases so also does the complexity of the language. It is felt that with ICE a good balance has been achieved. The syntax is relatively concise but yet powerful. The use of COUNTERs within the language provides a novel means of considerably increasing the descriptive space without increasing the state space of the model. The additional features required for the counters has been kept to a minimum and maintains continuity with the other constructs. The added burden to the syntax is far outweighed by the increased descriptive power. ICE models may be viewed graphically by state transition diagrams, as has been illustrated by many of the examples in this text. Component states are simply represented as circles and transitions by arcs. The addition of counters requires that extra inscriptions are required for the states to show where and how they are updated. This is analogous of the development of coloured Petri nets (CPNs) from Petri nets (PNs). CPNs, using coloured tokens, provide a significant increase in descriptive power over PNs although extra net inscriptions are required to describe the tokens. With CPNs the use of inscriptions is flexible. The modeller may use small nets with many inscriptions to describe the functionality or large nets with few inscriptions that may be less tractable but where the functionality is more easily apparent. With ICE the use of inscriptions is fixed and detracts relatively little from the understanding of the state transition diagrams. Due to the intuitive nature of counters and their expansive descriptive power modellers are therefore encouraged to use them to describe as much system functionality as is practicable.

The ICE approach

The degree of complexity of any modelling technique as presented to the human modeller should always be a primary concern. Many techniques may only be successfully adopted by 'experts'. It is immediately obvious that ICE is inherently far simpler than formalisms such as PNs. It's intuitive syntax and the reduced state space of the model should mean

that in can be learned and applied by a novice with only a few hours training.

As has been hinted at above, many practitioners find it easier to think in two dimensions and resultingly their first approach to modelling all but very simple systems is to make a spatial sketch. It would therefore be fair to make the comment that ICE is still just a documentation of a diagram. The author would personally support this view rather than defend against it. Recognising that when a state space approach to a problem is adopted the simple first step of modelling is to represent the acquired information in some sort of state transition diagram, the syntax and structure of ICE has been designed to make the translation between diagram and formalised description as simple and fluid as possible. This may indeed be seen as one of the languages main strengths. It captures what is to many modellers the natural, intuitive approach. This being the case, the question may then be raised, should some form of graphical user interface be provided whereby the user's initial diagrammatic representation may be automatically compiled and converted into ICE code ? At first this appeared to be a logical step but extended application of the language now suggests that it would not be worthwhile. Considering the problems of layout and connectivity experienced when using graphical software packages only very simple models could be drawn easily and with such models it is easier to go straight to an ICE implementation. The speed and ease with which 'paper and pen' diagrams can be drawn, edited and converted into ICE suggests that even with a graphical interface, modellers would revert back to such a technique.

Syntax and semantics

The syntax of the ICE constructs are mostly simple and intuitive although in a couple of cases they hide some rather complex semantics. It is a fair criticism of the language that a computational model should not be required to clarify the semantics. Ideally, all semantics should be obvious from the syntax. This is largely true with the exception of the conditional transition statements. If timing was removed from these statements then their meaning would be wholly obvious. However it is felt that the timing is important as it allows for more compact descriptions and ensures that all transitions follow a standard format. The detrimental implication is that timing does add complexity. It is no longer clear, exactly when transitions occur and what priority they take. The precise meaning of the IF and ON_EVENT transitions now become implementation dependent. The sequencing of events is fully explained in chapter 3 and it is crucial that a modeller is au

fait with this. The approach taken follows what it is thought would be intuitive, although without explanation it may be open to misunderstanding. It may be argued that in this respect PNs have an advantage over ICE. They give an exact representation of a system and leave no room for ambiguity in understanding. This consequently was one of the reasons that PNs were used to develop a computational model for ICE. However this argument is not quite correct when it is considered that for PNs a transition firing policy may be required. This states when a transition is enabled and when tokens are removed from input places and put in output places. Such factors are not apparent from the PN diagram but must be explicitly given. It is conceded that the complexity of the semantics is a necessary weakness of the language, however since it is relatively limited and they are explicitly defined through the computational model, it is not considered to be a major issue. The ethos of the language has encouraged a structure, style and choice of syntax that is intuitive and simple. Most of the discussion of the language in this thesis has centred around the syntax, which as we have discussed, does not fully define the language. It is therefore suggested that any future discussions regarding modifying ICE should be based equally around the computational model.

Static analysis

With PNs it is possible, depending upon the high level extensions used, to reduce the net to an underlying Markov chain and perform static analysis. The ability to perform such analysis on ICE models, e.g. the reachability of component states, would be a significant benefit. To achieve this it was considered whether ICE or a meaningful sub-set of ICE could be identified that would be reducible to a Markov chain. Firstly we shall examine the most simple of ICE examples. A two component state transition model is shown on the left in figure 9.1. Each of the components *comp1* and *comp2* have only two states and the transitions between the states are all deterministic. On the right in figure 9.1 an attempt is shown at the construction of an equivalent Markov chain.

It can be seen that the attempted Markov chain has two *AD* states and that there is more than one possible transition between many of the states. For clarity this figure only shows the first three iterations but it is easy to project how further iterations would increase the complexity. This state space diagram is clearly not memoryless and is therefore not a Markov chain. It is worth noting that we get this duplication of a state and multiple paths for even the most simple of ICE models. This suggests that the only possible sub-set of

ICE that could be reduced to an equivalent Markov chain would be for a single component model. ICE is however designed for multiple component systems and this would not be worthwhile. This example hints at the relative power of ICE. Many high-levels extensions to PNs cannot be reduced to Markov chains and as was shown in chapter 6, a considerable range of these extensions are required to represent the ICE syntax. This suggests that ICE would be irreducible, and it is interesting that we cannot reduce even a very basic structure.

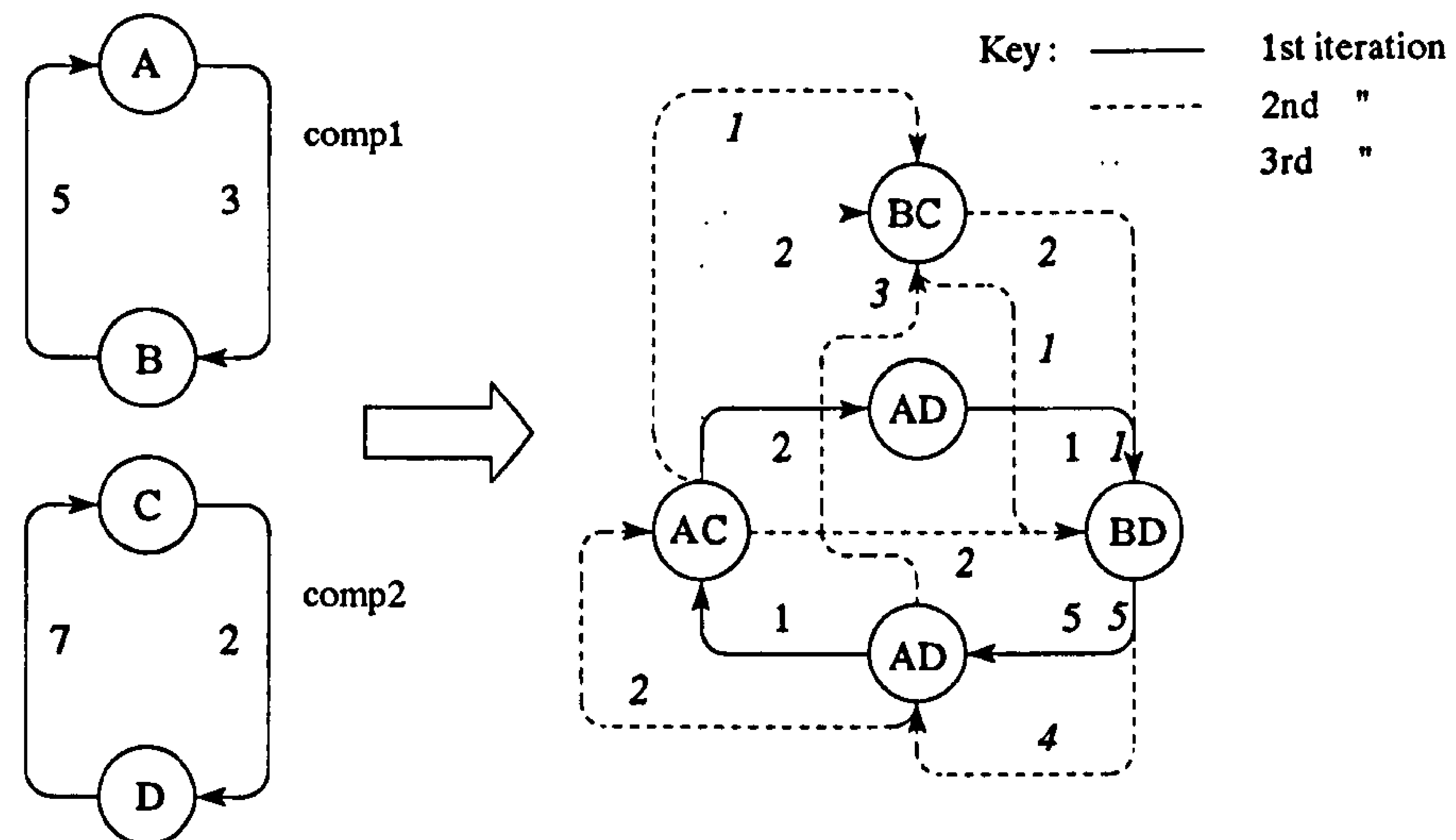


Figure 9.1 Conversion of simple state transition diagrams to a single diagram

A worthy area of further work would be to investigate other possible means of implementing static analysis on ICE model. An additional foreseeable challenge however would be ICE's stochastic nature. Reachability graphs can be drawn of PNs with deterministic transition firing rates, however the theoretical number of state consequences becomes infinite when stochastic transition rates are allowed. In ICE, where transitions are dependant upon the stochastic transitions of other components a similar problem would be encountered. One possible way of accommodating this would be to place limits on all stochastic transitions. For example, if a transitions timing was $t = exp(30)$, we could set the limits so that the transition must occur at time $t \in \{5,10,15,20,25,30,35,40,45,50,55\}$. This would theoretically make the reachability graph obtainable, though if there were more than a few stochastic transitions it may not be tractable.

ICE as a descriptive formalism

Often formalisms are used to capture the description of a system and this representation is then converted into a format suitable for simulation. ICE is beneficial in that it can be

used for both stages. The whole ethos of ICE dictates that there should be no need to describe a system in terms of another formalism before creating a model. In this thesis we have concentrated on using ICE to model systems for simulation but the language should also be viewed as a formal tool for producing unambiguous system descriptions. This may be suited for an application such as risk assessment where it may not be desired to simulate a system but a complete and consistent means of description is essential.

Using the #include command in ICE it is possible to build up libraries of ICE code that can be reused in different models. A criticism often made of simulation libraries that they are too simple or if the modules are not in the format of source code, it is not known exactly what they contain. As a result, libraries are often not used, modellers preferring to develop their own modules. ICE libraries are of ICE source code and owing to the simple syntax they should be inherently easy to comprehend and analyze, thus negating the latter problem. The former is also manageable as the libraries may be copied into new files and edited to suit a users requirements.

9.3 Implementation of the ICE simulator

The design and implementation of the ICE simulator, described in chapter 4, followed an iterative process. The criticism could be made that a full system specification should have been completed before the implementation began. This however was not possible as the syntax of the language had not been finalised at that stage. In truth, this iterative process proved to be valuable. Design of the simulation algorithm in particular led to some interesting insights into further possible interpretations of the language's semantics and transition timing issues. The exact definition of the algorithm required an exhaustive examination of all possible combinations of event enabling and sequencing and following from this an optimal understanding of the semantics was obtained. This understanding is presented in the computational model of ICE.

The simulator is implemented in quite complex object oriented code. Due to the level of abstraction inherent within object oriented programming and the number of data objects required for an application such as this, obtaining an understanding of the code can be quite daunting. The example presented in appendix B should prove useful and save much time for anyone aiming to understand the software with a view to modifying it. Extensive use

of reusable libraries and dynamic data structures, whilst perhaps not optimal due to the iterative design procedure, ensure that the code is relatively compact.

One weakness of the simulator is that it is subject to much possible conflict between simulation events. A consequence tree is used to fire events that result from the occurrence of previous events. This can result in many conflicting transitions being scheduled for the same time. In such an instance, the order in which these transitions occur should be random, however in practice they are dictated by the order in which they were submitted to the event list. As a result, some of the supposedly random events are predictable. To counteract this a random event scheduler is required as part of the simulation algorithm. This should be able to identify occurrences of conflict and randomly prioritise the conflicting events.

The range of statistics available from the analysis post processor is reasonably comprehensive and easily expanded. This processor has efficiently produced meaningful results from many of the models presented in this thesis. The flaw in this tool however is the time consuming way in which the user must enter the information that is required. The simulator has been written to run on a standard DOS environment. It would benefit greatly from a Windows implementation. As this thesis is being written, analysis of the software's graphical interface code is being done and a specification is being written for a new project that will develop a Windows version of the software. As well as allowing such things as the multiple editing of files and easy access to all functions this will facilitate a user friendlier interface to the post processors.

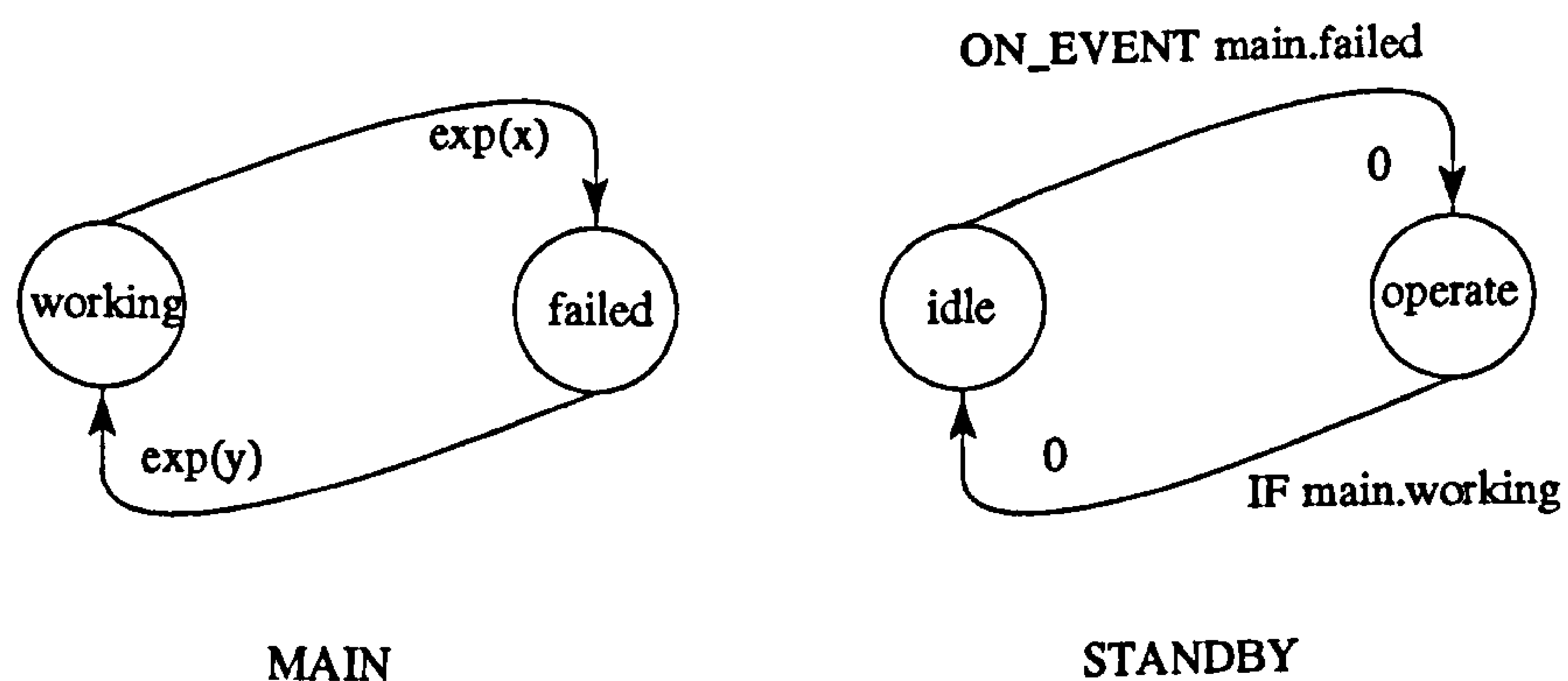


Figure 9.2 State transition diagram of standby redundant system.

Speed is of importance to the modeller, both during the model specification and simulation phases. It has been shown how the nature of ICE allows for the relatively quick development of models by 'non-experts'. The event scheduling nature of the simulator enables impressive run times. To illustrate this we shall consider a simple example. Figure 9.2 shows the state transition diagram of a standby redundant system. The component *main* moves between the two states *working* and *failed* with failure and repair rates *exp(x)* and *exp(y)* respectively. The component *standby* normally in state *idle* moves into state *operate* on the event that *main* moves into state *failed.* When *main* returns to state *working, standby* returns to state *idle.* This example thus has instances of all types of stochastic and conditional statements and gives a true representation of typical tun times. When implemented in ICE and simulated on a 486 DX4, 100 MHz PC the time taken to process one million events was 590 seconds. A large proportion of this time is due to the generation of the stochastic distributions for the exponential transitions. If $exp(x)$ and $exp(y)$ are replaced with the deterministic values x and y, the time taken to process one million events is reduced to 83 seconds. This clearly demonstrates the value of reducing the generation time of the pseudo random numbers.

9.4 Random number generation

The area of work on random number generation was not identified in the original specification but was suggested during the development of the simulator. It looked to be a fruitful area of research and was therefore pursued. The generator originally implemented was chosen from the literature for its proven statistical properties. Further research however revealed that these had been obtained for only long runs of generation. It was shown in chapter 5 that the properties of a pseudo randomly generated sequence will not necessarily be shared by sub-sequences. This implies that all sources within an application that require a supply of random numbers must be supplied from independent sequences. The length of sequences generated may then be of greatly varying ranges and the statistical properties must remain good for runs of a few numbers to many thousands of numbers.

In specifying that all lengths of sequences must possess good statistical properties we must be careful to recognise that this is on average. If each individual sequence generated possessed good properties, then this, paradoxically, would be non-random, as we expect by the law of probabilities to obtain some bad results. The longer a sequence that is used then

the more likely all random properties will be observed and the truer reflection of the actual system will be obtained. In many instances of simulating a model, many of the sequences used for the model will be of orders of magnitude longer than other sequences. To give credible results it is therefore necessary to do many runs of the simulation.

The new fast shift register generator proposed in chapter 5, has similar statistical properties to the proven industry standard generator but is over four times faster in operation. This represents a significant saving in time intensive simulation environments with many stochastic processes.

9.5 Computational models for ICE

Computational models for ICE were developed to both provide a concise description of the language's semantics and to compare it to a recognised performability modelling technique. ICE components may be regarded as finite state machines and are represented by PNs with a single token. These basic models are built upon when counters are added. Counters contribute a considerable increase in descriptive power and this is reflected by the increased complexity of the PN model. Within ICE, SYSTEM statements provide a simple means of relating what can be very complex component interactions. It is a non trivial task representing even a relatively simple system statement with PNs. These PNs are the most sophisticated of all that are used to model ICE and show how the simple syntax of the language can hide some surprisingly powerful descriptive capabilities. The counters and system statements make it possible to view ICE as a high level PN in the sense that due to these constructs there is a significant reduction in state space complexity of some ICE models relative to their PN equivalent.

In most of the PN models the transition firing policies are arbitrary though in a few they have to be stated so that the model accurately reflects the ICE equivalent. It is in this one aspect that the operation of the PNs is not wholly deterministic from the schematic alone. The conflict inherent in some of the ICE constructs is modelled by the use of inhibitor arcs and test arcs. This conflict results from the PNs being an exact representation of the simulation algorithm described in chapter 4. These PNs help to quantify and provide valuable insight into the problem of conflict that causes determinism in a random process,

as was discussed in section 9.3.

It would have been normal and perhaps beneficial to have developed a computational model before implementing the language. Had this been done, the iterative procedure by which the simulator was written would have been avoided. However, the learning process that did take place was very beneficial and it is not felt that the order in which the project progressed was detrimental in any way to the development of the language or simulator.

9.5.1 A macroscopic view

In chapter 6 all of the ICE constructs were modelled by PN equivalents in isolation. An interesting exercise would be to produce macroscopic PN representations of complete ICE models, though it is intuitively felt that these may quickly become unmanageably large. A number of primitives could be constructed, existing mostly of the already defined PNs, and these could be used to construct more complex features of a model. Such a macroscopic model would give a truer comparison of the state space of ICE and PN models. This exercise would identify the high and low level language features. By high level, we denote constructs that require a number of places and arcs to build up a PN model, such as the SYSTEM statement. Low level signifies where a construct can be modelled by a one to one equivalent of language items to PN places such as the RESOURCE statement. When these features are identified, it is the high level features that represent a significant saving in state space of ICE over PNs.

Some of the computational models, such as the SYSTEM statement, are rather complex and this defeats the purpose of using them to clarify the language's semantics. Perhaps the most important constructs to be defined by the PNs are the behavioural statements as it is in these that understanding of the semantics is open to misinterpretation. These PNs are of a reasonable size and the semantics are clearly illustrated. Although a macroscopic view of a system created by combining many primitive PNs may give a large and unmanageable view of the modelled system, this may not be the optimal PN representation. The ICE and PN approaches to system description are inherently different and therefore a direct comparison between the ICE constructs and the equivalent PN models will not in all cases give a definitive measure of relative model sizes.

Although it has been proven that ICE offers features that give a significantly more compact descriptive space than is possible with PNs we cannot state that ICE is more powerful than extended high level PNs. This is due to the ability to model all of the ICE syntax with equivalent PNs. However it is very interesting to note that in producing these PNs, no one dissertation on extended PNs contained all the high level extensions that were required.

The challenges to modelling of dependency, concurrency, synchronisation and conflict were all considered and ICE solutions derived. These solutions took the form of very basic ICE models, illustrating the practical nature of the language and giving confidence in its ability to model more complex systems.

9.6 Performability modelling

From the background research presented in chapter 2 it can be concluded that performability is a very important measure when analysing the behaviour of a system. It often presents more complex challenges to the modeller than pure performance or dependability measures. It was found that the techniques being developed to facilitate performability modelling often involved rather complex definitions and solutions of the problem suggesting that there is a need for an approach where the problem description involves a low level of abstraction but is still powerful. Performability was chosen as a good and thorough test bed for ICE and the work showed that ICE is well suited to this type of problem.

The ATM switch model presented in chapter 7 illustrates ICE's approach and ability to model complex system comprising many interacting components. It also demonstrates the very low level of abstraction involved which consequently enables detailed and interesting analysis of the switches behaviour. Such a model as this involves many simultaneous event transitions and requires a lot of careful consideration. It is difficult to see how this problem could be eased as the complexity is in the problem not the modelling technique. For such complex models errors may be hidden and validation is essential before we can have confidence in the results obtained. The textual and statistical post processors of the simulator provide an efficient means of conducting this validation. Although ICE was well suited to this application it is recognised that a state space approach may not always be the most efficient means of pure performance modelling and when performance measures are

the only requirement other techniques should not be ignored.

The generic ICE performability modelling framework proved effective for obtaining meaningful performability measures from a modelled system. This approach was based on reward model techniques and was demonstrated to be stronger than either Markov reward models or stochastic reward nets, in respect that the modelling of accurate non linear reward involved no added complexity over the modelling of linear reward.

9.6.1 Problems encountered in modelling systems

Consideration of the literature on performability modelling revealed three recurrent problems that are encountered, namely largeness, stiffness and non exponential behaviour. In chapter 2 we considered the implications of these and some of the existing solution techniques. Here be briefly consider how they can be handled by ICE.

Largeness

This problem can be elegantly handled by ICE in two ways. Firstly the use of counters can greatly reduce the size of a model's state space hence possibly removing any requirement for *lumping* and the consequent approximation. Secondly there is no requirement to detail every possible combination of states, only the states for each individual component and their interactions. This may avoid the need for *truncation* and again the approximation that this involves. These techniques suggest that for many applications the problem of largeness will not be significant when modelling with ICE.

Stiffness

The ability to reuse code libraries and perform decomposition within ICE models by adopting hierarchical techniques will in some instances solve the problem of stiffness. However some applications still present degrees of stiffness that are not possible to model without some degree of approximation. Take as an example a typical ATM system. If we assume a line speed of 155 Mbits^{-1} and 30% load, an error rate of 10^{-8} would represent one erroneous cell per day ! Also errors tend to occur in bursts and this is a high error rate.

I_SIM suffers from the weakness common to all simulators of not being able to handle this dynamic range. For such situations analytical techniques will need to be applied.

Non exponential behaviour

This is not an issue with ICE as many types of stochastic distributions are facilitated and any that are not currently implemented can be easily incorporated.

9.7 Suggested areas of further work

In this section possible areas of further work that have not been discussed elsewhere in this chapter are suggested.

The ICE language

ICE is thought to be fairly optimal in its current state. A couple of changes have become apparent during the performability modelling that would not add extra power to the language but would assist in more elegant model implementations. These changes would not increase the complexity of the syntax, fitting with the adopted ethos.

When using the generic performability framework a component is used to monitor the cumulative reward after every reward time unit. In models where states have significantly different sojourn times this becomes computationally intensive. It is suggested that counters be given access to the system clock during simulation. With this facility it would be possible to check state entry and exit times and thus determine the total accumulated reward for that state when it is exited. This would save computation and therefore time during simulation.

If the ATM switch model is examined it will be recognised that much of the code is repetitive with some constructs being repeated many times over, the only difference being some of the state names. Much editing could be saved if a functional approach was adopted. By this we mean the introduction of variables. A construct that will be used more than once could be defined and the changing state names replaced by these variables.

```
BEHAVIOUR be {     2 idle -> var1;
                   3 var1 -> busy;
                   1 busy -> var2;
                   4 var2 -> idle; }

be ( working_1; standby_1 );
be ( working_2; standby_2 );
```

Listing 9.1 Example of functions

Consider as an example the BEHAVIOUR statement of listing 9.1. The statement contains the two variables *var1* and *var2*. It may then be used many times over by giving the statement name and in brackets afterwards the state names that the variables should be substituted with. This is analogous to the use of functions in the C programming language. If such an approach were used in the ATM switch model the amount of code would be reduced by about 80%.

A third change to the language which would add complexity and significant power would be the incorporation of intelligence. ICE can now be used to obtain performability measures. A suitable progression would be the ability to optimise performability. This would involve the capability for the dynamic adjustment of components and resources during simulation so that different configurations could be tested based on intelligence of the operation status of the system. It is envisaged that with such an implementation desirable optimal states could be identified and it would be possible to dynamically re-configure the model to obtain these states. It should also be capable of monitoring for the occurrence of given states and work backwards to identify possible causes. This would be an extremely useful tool for many applications where the emphasis would be shifted from mitigation to prevention thus potentially offering significant savings in design time and cost.

The ICE simulator

The conversion of the ICE simulator to a Windows based version, discussed in section 9.3, will offer many benefits. It is necessary to analyse most models that are simulated to obtain understanding of the system in question. The simulators post processors could be improved to facilitate more options for this analysis.

Currently the analysis post processor will only perform analysis on whole runs of a simulation. It would be beneficial to be able to select exact periods during simulation for which analysis is required. This would save having to simulate the model to determine steady state conditions or conditions at the start of the desired time of analysis which are then fed back into the model.

Many of the results in this thesis have been presented in graphical format. This was achieved by obtaining statistics from the post processors and then using these in a standard drawing package. Often features and trends of the results do not become apparent until a graphical plot is generated. As part of the revised post processor a graphing facility should be provided, to plot any of the statistics obtained and be in a format suitable for direct printing or import into a word processing or drawing package. An extension of this would be on-line visualisation of results. Often it would be of great advantage to be able to view the behaviour of a system as it is being simulated. This would involve much computation, sacrificing simulation speed and therefore should be optional. On-line visualisation should allow any of the obtainable statistics to be plotted as the simulator is running. These plots could be updated after given time periods or by using a pause option, where the simulation would be stopped at the user's discretion, plots updated then the simulation recommenced.

9.8 Conclusions

The achievements of the project are listed in section 1.4.

Overall the project has developed :

1. A generic description and simulation language that is intuitive and simple to apply but powerful.

2. A tool that facilitates the attainment of performability measures and which may be applied by an expert in the field rather than an expert in performability.

References

[1] Silva M; *Interleaving Functional and Performance Structural Analysis of Net Models*, 14th Conf. on the Application and Theory of Petri Nets, Springer-Verlag, pp16-23, June 1993.

[2] Buchholz P; *Hierarchies in Coloured GSPNs*, 14th Conf. on the Application and Theory of Petri Nets, Springer-Verlag, pp106-25, June 1993.

[3] Trivedi K S, Ciardo G, Malhotra M and Garg S ; *Dependability and Performability Analysis Using Stochastic Petri Nets*, Proc. of 11th International Conf. on Analysis and Optimization of Systems - discrete event systems, pp144-57, Sophia Antiplois, France, June 1994, Springer-Verlag.

[4] Jensen K; *Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use, volume 1*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[5] Deng Y, Chang S K, de Figueired J C A and Perkusich A; *Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems*, 14th Conf. on the Application and Theory of Petri Nets, Springer-Verlag, pp106-25, June 1993.

[6] Miller A J, Wells R and Walker K S; *Simulation Using the Reliability Description Language*, Reliability '91, ed R H Matthews, 1991.

[7] Talib B; *The theory and design of a stochastic reliability simulator for large scale systems*, PhD Thesis, Robert Gordon's Institute of Technology, 1988.

[8] Deans N D, Miller A J, Mann D; *A reliability simulation language for reliability analysis*, In Proc. Eurodata Conference on Reliability Data Collection and Use in

Risk Availability Assessment, Heidelberg, 1986.

[9] Deans N D, Miller A J; *Study of the feasibility of using a formalised language for the description of reliability problems*, Technical Report RGIT, EEC contract number EC1-1479-B7222-86-UK, 1987.

[10] Scrase A S; *On RDL and its Application to The Performability of Communication Networks*, PhD Thesis, The Robert Gordon University, 1991.

[11] Smith W; *Design and Implementation of a Simulator for the Performance Analysis of Manufacturing Systems*, PhD Thesis, The Robert Gordon University 1991.

[12] Abdelmonem A H, Van Ryzin G J; *A method for evaluating the effects of performance degradations on communications network applications*, ICC 90: Proc. of IEEE Int. Conf. on communications, pp1307-13, Atlanta, USA, April 1990.

[13] Riebman A L; *Modeling the Effect of Reliability on Performance*, IEEE Trans. on Reliability, Vol. 39, No. 3, pp314-9, Aug. 1990.

[14] Meyer J. F. *Performability: a retrospective and some pointers to the future* Performance Evaluation, Vol. 14, pp139-56, Elsevier Science Pub. B.V. 1992.

[15] Meyer J F; *On evaluating the performability of degradable computing systems* Proceedings of the 8th International Symposium on Fault-Tolerant Computing, Toulouse, France, IEEE Computer Society Press, Silver Spring, MD, pp44-49, 1978.

[16] Meyer J F; *On evaluating the performability of degradable computing systems* ,IEEE Trans. on computers, Vol. 29, No. 8, pp720-31, Aug. 1980.

[17] Meyer J F; *Computation-based reliability analysis* IEEE Trans. Computers, Vol. C-25, No. 6, pp578-584, 1976.

[18] Borgerson B R, Freitas R F; *A reliability model for gracefully degrading and standby-sparing systems* IEEE Trans. Computers, Vol. C-24, No. 5, pp517-525, 1975.

[19] Meyer J. F. *A model hierarchy for evaluating the effectiveness of computing systems* Proc. 3rd National Reliability Symposium, Perros-Guirec, France, CNET France, pp539-555, 1976.

[20] Trivedi K S, Haverkort B R, Rindos A and Mainkar V; *Techniques and Tools for Reliability and Performance Evaluation : Problems and Perspectives*, 7th Int.Conf. on Techniques and Tools for Computer Performance Evaluation, Published as Lecture notes in computer science 794, Ed. Haring G and Kotsis G, Springer Verlag, 1994.

[21] Papoulis A; *Probability, Random Variables and Stochastic Processes*, Chapter 16, McGraw Hill International Editions, 1991.

[22] Sahner R A, Trivedi K S; *Reliability Modelling using SHARPE*, IEEE Trans. on Reliability, Vol. 36, No. 2, pp186-193, 1987.

[23] Hsueh M C, Iyer R K, Trivedi K S; *Performability Modelling Based on Real Data: A Case Study*, IEEE Trans. on Computers, Vol. 37, No. 4, pp478-84, April 1988.

[24] Trivedi K S *Probability and Statistics with Reliability, Queuing and Computer Science Applications* Prentice Hall, Englewood Cliffs, NJ, USA, 1982.

[25] Howard R A; *Dynamic Probabilistic Systems, Vol II; Semi-Markov and Decision Processes*, Wiley, New York, 1971.

[26] Gelenebe E, Pujolle G *Introduction to Queueing Networks* Wiley, UK, 1987.

[27] Sahner R A, Trivedi K S, Puliafto A; *Performance and reliability analysis of computing systems*, Chapter 4, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1996.

[28] Chou T.C.K., Abraham J.A. *Performance/availability model of shared resource multiprocessors* IEEE Trans. Reliability Vol. 29, No. 1, pp70-6, 1980.

[29] Trivedi K S et al *Composite performance and dependability analysis* Performance Evaluation, Iss. 14, pp197-215, North Holland, 1992.

[30] de Souza e Silva E, Gail H R; *Performability analysis of computer systems*, Performance Evaluation, Vol. 14, Parts 3-4, pp157-196, 1992.

[31] Malhotra M, Trivedi K S *A methodology for formal expression of hierarchy in model solution* Proc. Petri Nets and Performance Models 1993, pp258-67 Toulouse, France, Oct. 19-22, 1993.

[32] Ajmone Marsan M, Conte G, Balbo G; *On Petri nets with stochastic timing*, In proc. Int. workshop on Timed Petri nets, IEEE, Torino, Italy, 1985.

[33] Meyer J F, Movaghar A, Sanders W H; *Stochastic Activity Networks : Structure, Behaviour and Application*, In proc. Int. workshop on Timed Petri nets, pp106-15, IEEE, Torino, Italy, 1985.

[34] Peterson J L; *Petri Net Theory and the Modelling of Systems*, Prentice Hall, N.J. 1981.

[35] Molloy M *Performance analysis using stochastic Petri nets* IEEE Trans. on Computers, Vol. C-31, No. 9, pp913-17, Sept. 1982.

[36] Ciardo G et al *Automated generation and analysis of Markov reward models using stochastic Petri nets* Linear Algebra, Markov Chains and Queuing Models, editors : Mayer C and Plemmons R J, IMA Volumes in Mathematics and its Applications, Vol. 48, pp145-91, Springer Verlag, Heidelberg, 1993.

[37] Catania V, Puliafito A, Vita L; *A Modeling Framework To Evaluate Performability*

Parameter In Gracefully Degrading Systems, IEEE Trans. on Industrial Electronics, Vol. 40, No. 5, pp461-72, Oct. 1993.

[38] van der Aalst W M P; *Interval Timed Coloured Petri Nets and their Analysis*, Application and Theory of Petri Nets, 14th International Conference, pp453-71, Chicago, Springer-Verlag, June 1993.

[39] Berthelot G, Boucheneb H; *Occurrence graphs for interval timed coloured nets*, Application and Theory of Petri Nets, 15th International Conference, pp78-98, Spain, Springer-Verlag, June 1994.

[40] Christensen S, Hansen N D; *Coloured Petri Nets Extended with channels for Synchronous Communication*, Application and Theory of Petri Nets, 15th International Conference, pp159-77, Spain, Springer-Verlag, June 1994.

[41] Christensen S, Hansen N D; *Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs*, Application and Theory of Petri Nets, 15th International Conference, pp186-205, Chicago, Springer-Verlag, June 1993.

[42] Evans J B; *The Devnet : A Petri net for Discrete Event Simulation*, Advances in Petri nets, Ed. Rozenburg G, pp91-125, 1993.

[43] Sanders W H, Meyer J F; *Performance variable driven construction methods for stochastic activity networks*, In Ed. Coutois P J, Iazeolla G, Boxma O J, Elsevier Science Publishers B.V. (North Holland), 1988.

[44] Sanders W H, Meyer J F; *Performability evaluation of distributed systems using stochastic activity networks*, Ed. Bonatti M, Teletraffic Science, Aug. 1987.

[45] Sanders W H, Meyer J F; *METASAN : a performability evaluation tool based on stochastic activity networks*, In Proc. ACM-IEEE Computer Society Fall Joint Comp. Conf. Dallas, Texas, Nov. 1986.

[46] Ciardo G, Trivedi K S; *Decomposition Approach for Stochastic Reward Net Models*, Performance Evaluation, Vol. 18, No. 1, pp37-59, July 1993.

[47] Balbo G, Bruell S C, Ghanta S; *Combining Queueing Networks and Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems*, IEEE Trans. on Computers, Vol. 37, No. 10, pp1251-1268, 1984.

[48] Szczerbicka H; *A Combined queueing network and stochastic Petri-net approach for evaluating the performability of fault-tolerant computer systems*, Performance Evaluation, Vol. 14, pp217-26, North-Holland, 1992.

[49] Lopez-Benitez N, Trivedi K S; *Multiprocessor Performability Analysis*, IEEE Trans. on Reliability, Vol. 42, No. 4, pp579-87, 1993.

[50] Haverkort B R, Niemegeers I G; *Using Dynamic Queueing Networks as a Tool for*

Specifying Performability Models, ACM Performance Evaluation Review, Vol. 17, No. 1, pp225, 1989.

[51] Haverkort B R; *Approximate performability and dependability analysis using generalised stochastic Petri nets*, Performance Evaluation, Vol. 18, pp61-78, North-Holland, 1993..

[52] Beaudry M D; *Performance-related reliability measures for computer systems* IEEE Trans. on Computers, Vol C29, No. 6, pp501-9, 1978.

[53] Iyer B R, Donatiello L, Heidelberger P; *Analysis and Performability for Stochastic Models of Fault-Tolerant Systems*, IEEE Trans. on Computers, Vol. 35, No. 10, pp902-7, Oct. 1986.

[54] Meyer J F, Furchtgott D G, Wu L T; *Performability evaluation of the SIFT computer* IEEE Trans. Computers, Vol. C29, No. 16, pp501-9, 1980.

[55] Meyer J F; *Closed-Form Solutions of Performability*, IEEE Trans. on Computers, Vol. 31. No. 7, pp648-57, July 1982.

[56] Couvillion J et al; *Performability Modelling with UltraSAN*, IEEE Software, pp69-80, Sept. 1991.

[57] Prodromites K H, Sanders W H; *Performability Evaluation of CSMA/CD & CSMA/DCR Protocols Under Transient Fault Conditions*, IEEE Trans. on Reliability, Vol. 42, No. 1, pp116-27, March 1993.

[58] Ciciani B, Grassi V; *Performability Evaluation of Fault-Tolerant Satellite Systems*, IEEE Trans. on Communications, Vol. 35, No. 4, pp403-9, April 1987.

[59] Li V O K, Silvester J A; *Performance analysis of networks with unreliable components*, IEEE Trans. on Communications, Vol. 32, No. 10, pp1105-10, Oct.1984.

[60] Goyal A, Tanawi A N; *Evaluation of performability for degradable computer systems*, IEEE Trans. on Computers, Vol. 36, No. 6, pp738-44, June 1987.

[61] Brenner A; *Performance and Dependability Analysis of Fault-Tolerant Networks*, Microelectronics Reliability, Vol. 36, No. 3, pp307-21, Elsevier Science Ltd, 1996.

[62] Rezal Islam S M, Ammar H H; *Performability of the Hypercube*, IEEE Trans. on Reliability, Vol. 38, No. 5, pp518-25, Dec. 1989..

[63] van Dijk N M; *Simple bounds for queuing systems with breakdowns*, Performance Evaluation, No. 8, pp117-28, 1988.

[64] van Dijk N M; *Simple performability bounds for communications networks*, In Proc. Queueing, Performance and Control in ATM Networks, pp245-8, Elsevier Science Publishers B.V. North Holland 1991.

[65] Jones D R, Malec H A; *Communications systems performability: New horizons*, In proc. World Prosperity Through Communications: International Conference, Boston, pp15-23, IEEE, June 1989.

[66] Yang C-L, Kubat P; *Efficient Computation of Most Probable States for Communications Networks with Multimode Components*, IEEE Trans. on Communications, Vol. 37, No. 5, pp535-8, May 1989.

[67] Koren I, Koren Z; *On Gracefully Degrading Multiprocessors With Multistage Interconnection Networks*, IEEE Trans. on Reliability, Vol. 38, No. 1, April 1989.

[68] Bhattacharya A, Rao R R, Lin T-T Y; *Cumulative performance measure for gracefully degradable multistage interconnection networks*, In Proc. 1st International Workshop on Parallel Processing, pp234-9, Dec. 1994.

[69] Feldbrugge F; *Petri net tool overview 1992*, Advances in Petri Nets 1993, Editor Rozenberg G, Springer Verlag, Berlin, 1993.

[70] Lakos C A, Keen C D; *Modelling Layered Protocols in LOOPN*, IEEE pp106-15, 1991.

[71] Haverkort B R, Niemegeers I G, Veldhuyzen van Zanten P; *DyQNtool: a performability tool based on the dynamic queueing network concept*, Modelling Techniques and Tools for Computer Performance, Editor: Balbo G, North-Holland, Amsterdam, 1991.

[72] Lindenmann C; *Performance Modeling using DSPNexpress*, In Proc. Measurement and modelling of computer systems conference, Santa Clara, May 1993, Springer Verlag, Berlin. 1993.

[73] Bavuso S J et al; *Analysis of Typical Fault-Tolerant Architectures using HARP*, IEEE Trans. on Reliability, Vol. 36, No. 2, pp176-85, June 1987.

[74] Carrasco J A, Figueras J; *METFAC: Design and implementation of a software tool for modeling and evaluation of complex fault-tolerant computing systems*, In Proc. FTCS 16, IEEE Computer Society Press, pp424-29, 1986.

[75] Constantinescu C; *Predicting Performability of a Fault-Tolerant Microcomputer for Process Control*, IEEE Trans. on Reliability, Vol. 41, No. 4, pp558-63, Dec. 1992.

[76] Trivedi K S, Kulkarni V G; *FSPNs: Fluid Stochastic Petri Nets*, Lecture Notes in Computer Science, No. 691, Springer-Verlag, 1993.

[77] Trivedi et al; *Dependability & Performability Analysis*,, Measurement & Modelling of Computer Systems Conference, Santa Clara, pp589-612, Springer Verlag, May 1993.

[78] Puttipati K R, Shah S A; *On the Computational Aspects of Performability Models*

of Fault-Tolerant Computer Systems, IEEE Trans. on Computers, Vol. 39, No. 6, pp832-36, June 1990.

[79] Bobbio A, Trivedi K S; *An aggregation technique for the transient analysis of stiff Markov chains*, IEEE Trans. on Computers, Vol. 35, No. 9, pp803-14, Sep. 1986.

[80] Bruel S C, Chen P Z, Balboa G; *Alternative methods for incorporating non-exponential distributions into stochastic timed Petri nets.* In Proc. 3rd Int. Workshop on Petri nets and Performance Models, Kyoto, Japan, 1989.

[81] Zeigler B P; *Theory of Modelling and Simulation*, John Wiley & Sons, New York, 1976.

[82] Law A M, Kelton W D; *Simulation modelling and analysis*, McGraw Hill, New York, 1982.

[83] Schriber T; *Perspectives on Simulation using GPSS*, In Proc. 1988 Winter Simulation Conf. San Diego, Dec. 12-14, pp71-84.

[84] O'Rielly J J, Lilegdon W R; *Slam II Tutorial*, In Proc. 1988 Winter Simulation Conf. San Diego, Dec. 12-14, pp85-9.

[85] Delfossee C M; *Continuous simulation and combined simulation in Simscript II.5*, CACI, Arlington Virginia, 1976.

[86] Davis D A, Regden C D; *Introduction to SIMAN*, In Proc. 1988 Winter Simulation Conf. San Diego, Dec. 12-14, pp61-70.

[87] Kreutzer W; *System Simulation - Programming Styles and Languages*, Addison Wesley, 1986.

[88] McHaney R; *Computer Simulation, A Practical Perspective*, Academic Press Inc. 1991.

[89] Stroustrup B; *The C++ Programming Language*, Addison Wesley, 1986.

[90] Birtwistle G et al; *Simula Begin*, Petrocelli/Charter, New York, 1977.

[91] Goldberg A, Robson D; *Smalltalk-80 The language and it's Implementation*, Addison Wesley, Mass. 1986.

[92] Pohl I; *C++ for C programmers*, Benjamin Cumming, 1989.

[93] Bratley P, Fox B L, Scharge L E; *A Guide to Simulation*, 2nd Ed. Springer Verlag, New York, 1987.

[94] D E Knuth *The Art of Computer Programming Vol 2, Seminumerical Algorithms*, 2nd. Ed. Addison Wesley, Mass. USA, 1981.

[95] Vattulainen I, Kankaala K, Saarinen J, Ala-Nissila T *A Comparative Study of some Pseudorandom Number Generators* Computer Physics Communications, Vol. 86, pp209-26, 1995.

[96] D H Lehmer, *Mathematical Methods in Large Scale Computing Units,* Annu. Comp. Lab. Harvard University. 26[1951], [[141-146.

[97] Kwok H C et al *Design and Analysis of parallel random number generators* Proc. 31st Annual Southeast Conf. pp111-18, ACM, New York, 1993.

[98] S K Park, K W Miller, *Random Number Generators : Good ones are hard to find*, CACM. Vol 31 No 10 pp1192-1201.

[99] P A Lewis, A S Goodman & J M Miller, *A Pseudo-Random Number Generator for the System /360* IBM Systems Journal No 8 Vol 2, pp136-146.

[100] S K Park & K W Miller *Technical Correspondence - Remarks on Choosing and Implementing Random Number Generators*, CACM Vol 36 No 7, pp108-110.

[101] Marsaglia G *The Structure of Linear Congruential Generators* Applications of Number Theory to Numerical Analysis, Academic Press Inc. New York, 1992.

[102] Deng L-Y, Chan K H, Yuan Y *Random Number Generators for Multiprocessor Systems* International Journal of Modelling and Simulation, Vol. 14, No. 4, pp185-91, 1994.

[103] Makino J *On the structure of parellelized random number sources* Computer Physics Communications, Vol. 78, Iss. 1-2, pp 105-12, Dec. 1993.

[104] Burton F W & Page R L *Distributed Random Number Generation* Journal of Functional Programming, Vol. 2, No. 2, pp203-12, April 1992.

[105] L'Ecuyer P, Côté S *Implementing a Random Number Package with Splitting Facilities* AACM Trans. Mathematics Software, Vol. 17, pp98-111, 1991.

[106] Eichenauer J, Lehn J *A Non-linear Congruential Pseudorandom Number Generator* Statistical Papers, No. 27, pp 315-26, 1986.

[107] Eichenauer J *Inverse Congruential Pseudorandom Numbers : A Tutorial* International Statistics Review, No. 60, pp167-76, 1992.

[108] Neiderreiter H *On a New Class of Pseudorandom Number Generators for Simulation Methods* Journal of Computational and Applied Mathematics, No. 56, pp159-67, 1994.

[109] Golom S W *Shift Register Sequences* Holden-Day Inc. San Francisco, 1967.

[110] Zierler N *Primitive Trinomials whose Degree n is a Mersenne Exponent* Int. Control, Vol. 15, pp67-9, 1969.

[111] Zierler W *On Primitive Trinomials (mod 2)* Part 1 - Int. Control, Vol. 13, pp541-54 1968; Part 2 - Int. Control, Vol. 14, pp566-69, 1969.

[112] Hoffman de V *Binary Sequences* English Universities Press Ltd. 1971.

[113] Arazi B *Decimation of m-sequnces leading to any desired phase shitf* Electronics

Letters, Vol. 13, No. 7, pp213-22, Mar. 1977.

[114] Tomlinson G H, Galvin P *High Speed Generation of Long m-sequences* Electronics Letters, Vol. 14, No. 7, pp212-13, 1978.

[115] Hately M G *Digital Simulation Methods* Peregrinus Ltd. 1975.

[116] Birolini A *Hardware Simulation of Semi Markov and Related Processes, Part 1: A Versatile Generator* Mathematics and Computers in Simulation, XiX, pp75-97, 1977.

[117] Tausworthe R C *Random Numbers Generated by Linear Recurrence Modulo 2* Mathematics and Computing, Vol. 19, pp201-9, 1965.

[118] Tootill J P R, Robinson W D, Adams A G *The Runs Up an Down Performance of Tausworthe Pseudorandom Number Generators* Journal of the Association of Computing Machinery, Vol. 18, No. 3, pp381-39, 1971.

[119] Lewis T G, Payne W H *Generalised Feedback Shift Register Pseudorandom Number Algorithm* Journal of the Association for Computing Machinery, Vol. 20. No. 3, pp456-68, 1973.

[120] Bright H S, Enison R L *Quasi-Random Number Sequences* Computing Surveys, Vol. 11, No. 4, pp359-370, 1979.

[121] Hamilton K G *A Universal Random Number Generator for Personal Computers* Computer Physics Communications, Vol. 85, pp127-52, 1995.

[122] Fushimi M, Tezuka S *The k-distribution of Generalised Feedback Shift Register Pseudorandom Numbers* Communications of the ACM, Vol. 26, No. 7, pp516-23, 1983.

[123] Arvillas A C, Maritsas D G *Toggle-Register Generating in Parallel kth Decimations of m-sequences* $x^P + n^k + 1$*. Design Tables* IEEE Transactions on Computers, Vol. C-28, No. 2, 1979.

[124] Arvillas A C, Maritsas D G *Partitioning the Period of a Class of m-sequences and Application to Pseudorandom Number Generation* Journal of ACM, Vol.25, No. 4, pp675-686, 1978.

[125] Barel M *Fast Hardware Random Number Generator of r the Tausworthe Sequence* 16th Annual Simulation Symposium, Tampa, pp121-34, 1983.

[126] Miller A J, Mars P *Theory and Design of a Digital Stochastic Computer Random Number Generator* Mathematics and Computers in Simulation XIX, pp198-216, 1977.

[127] L'Ecuyer P *Uniform Random Number Generation* Annals of Operations Research, Vol. 53, pp77-120, J C Baltzer A.G. Science Publishers, 1994.

[128] L'Ecuyer P *Testing Random Number Generators* Proc. Winter Simulation Conf. pp305-313, 1992.

[129] Niederreiter H *Random Number Generation and Quasi-Monte Carlo Mathods* SIAM, CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 63, 1992.

[130] Ripley J D *Thoughts on Pseudorandom Number Generators* Journal Computing and Mathematics, Vol. 63, 1992.

[131] Law A M, Kelton W D *Simulation Modelling and Analysis* 2nd Ed, McGraw Hill, 1991.

[132] James F *A Review of Pseudorandom Number Generators* Computer Physics Communications, Vol. 60, pp329-44, 1990.

[133] Durst M J *Using Linear Congruential Generators for Parallel Random Number Generation* Proc. Winter Simulation Conference, pp462-466, 1989.

[134] De Matteis A, Pagiutti S, Numerical Mathematics, Vol. 53, pp595, 1988.

[135] Trivedi K S et al; *Dependability and performability analysis using stochastic Petri nets* 11th Int. Conf. on Analysis and Optimisation of Systems - discrete event systems, Sophia-Atipolis, France, June 1994, Springer Verlag, pp144-57, 1994.

[136] Ammar H H, Islam S M R, Deng S; *Performability analysis of parallel and distributed algorithms* Proceedings of Petri nets and Performance Models, Kyoto, Japan, pp240-8, 1989.

[137] Ciardo G, Muppala J K, Trivedi K S; *Analyzing concurrent and fault tolerant software using stochastic Petri nets* Journal of parallel and distributed computing, Vol. 15, pp255-69, 1992.

[138] Cochrane D, *Quality of Service*, Proc. 5th RACE TMN Conference, ppD31P/1/p1-18, London , Nov. 1993, Cray Communications Watford, 1993.

[139] Santamaria M L, Puigjaner R; *Banyan ATM switch: grade of service under unbalanced load*, Computer Networks, Architecture and Applications (C-13), Elsevier Science Publishers B.V. pp261-70, 1993.

[140] Morris T D, Perros H G; *Performance modelling of a multi-buffered Banyan switch under bursty traffic*, INFOCOM 92, Florence, IEEE, pp3D.2.1-10, 1992.

[141] Pujolle G, Fayet C; *On a new ATM architecture for the local area*, ICCT'92 Proc. of the 1992 International conference on communication theory, Beijing China, Sept. 1992, pp16.01.1-4, International Academic Publishers, 1992.

[142] De Prycker M, Horwood E; *Asynchronous Transfer Mode : Solution for Broadband ISDN*, Prentice Hall, London, Nov. 1993.

[143] Jeffry M; *Asynchronous transfer mode: the ultimate broadband solution ?*, Electronics and Communications Engineering Journal, Vol. 6, No. 3, Jun. 1994, IEE, pp143-51.

[144] De Prycker M; *Evolution to BISDN based on ATM*, ICCT'92 Proc. of the 1992 International conference on communication theory, Beijing China, Sept. 1992, pp12.01.1-5, International Academic Publishers, 1992.

[145] CCITT Recommendations I.361, SG XVII, 1991.

[146] Sun Z, Cosmas J, Cuthbert L G; *Simulation Studies of Multiplexing and Demultiplexing Performance in ATM Switch Fabrics* 10th UK Teletraffic Symposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp21/1-21/5.

[147] Baiocchi A, Bléfari-Melazzi N, Roveri A, Salvatore F; *Stochastic Fluid Analysis of an ATM Multiplexer Loaded with Heterogeneous ON-OFF Sources: an Effective Computational Approach* Infocomm 92, Florence 1992 IEEE, pp3C.3.1-10.

[148] Elwalid A I, Mitra D; *Fluid Models for the Analysis and Design of a Statistical Multiplexing with Loss Priorities on Multiple Classes of Bursty Traffic* Infocomm 92, Florence 1992 IEEE, pp3C.4.1-11.

[149] Karol M J, Eng K Y; *Performance of Heirachical Multiplexing in ATM Switch Design* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp311.4.1-7.

[150] Yegani P; *Performance Models for ATM Switching of Mixed Continuous-Bit-Rate and Bursty Traffic with Threshold-Based Discarding* Supercomm 92, Chicago, June 1992 IEEE, pp354.3.1-7.

[151] Larsson M, ljungberg M, Rooth J; *The ATM Switch Concept and the ATM Pipe Switch* Ericsson Review, Vol 70, Part 1, 1993, pp12-20.

[152] Guangdong D, Zengji L, Zheng H; *Design of ATM switches*, ICCT'92 Proc. of the 1992 International conference on communication theory, Beijing China, Sept. 1992, pp12.04.1-5, International Academic Publishers, 1992.

[153] Tobagi F A; *Fast packet switch architectures for broadband integrated services digital network*, Proc. IEEE, Vol. 78, No. 1, pp133-167, 1990.

[154] Wei S H, Kumar V P;*On the Multiple Shared Memory Module Approach to ATM Switching* Infocomm 92, Florence 1992 IEEE, pp1D.2.1-8.

[155] M E Bashai, E A Munter *A Rotating-Access ATM Switch* Queuing, Performance and Control in ATM, Elsevier Science Publishers BV (North Holland), 1991.

[156] Kim H S; *Multinet Switch: Multistage ATM SWitch Architecture with Partially Shared Buffers* Infocomm 93, San Francisco, 28.3-1.4 1993, IEEE, pp4c.3.1-8.

[157] Chiussi F M, Tobagi F A; *A Hybrid Shared-Memory/Space-Division Architecture for Large Fast Packet Switches* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp332.5.1-7.

[158] Monterosso A, Pattavina A; *Performance analysis of multistage interconnection networks with shared-buffered switching elements for ATM switching*, INFOCOM 92, Florence, IEEE, pp1D.3.1-8, 1992.

[159] Gupta A K, Barbosa L O, Georganas N D; *Limited Intermediate Buffer Switch Modules and their Interconnection Networks for B-ISDN* Supercomm 92, Chicago, June 1992 IEEE, pp354.7.1-5.

[160] Turner J S; *Design of a Broadcast Packet Switching Network* IEEE Transactions on Communications, Vol. 36, No. 8, June 1988, pp734-43.

[161] Lee T T; *Nonblocking Copy Networks for Multicast Packet Switching* IEEE Journal on selected areas in Communications, Vol. 6, No. 9, Dec. 1988, pp1455-66.

[162] Onvural R O; *Asynchronous transfer mode networks : performance issues*, chapter 5, 2nd edition, Artech House, UK, 1995.

[163] Chen T M, Liu S S; *ATM Switching Systems*, Artech House Inc. MA, USA, 1995.

[164] Venkatesan R, Mouftah H T; *Performance Analysis of Multipath Banyan Networks*, Supercomm 92, Chicago, USA, IEEE, pp332.6.1-5, June 1992.

[165] Perros H; *ATM Switch Architectures* First UK Workshop on Performance Modelling and Evaluation of ATM Networks, Bradford, 28-29th June 1993, pp tutorial 14-26.

[166] Feng T; *A survey of interconnection networks*, IEEE Trans. Computing, Vol. 14, No. 12, pp12-27, Dec. 1981.

[167] Tubtiang A, Kwon H I, Pujolle G; *A method for ATM switches classification*, Proc. of ICCT'92, IEEE, pp12.03.01-05. 1992.

[168] Theimer T H, Rathbeg E P, Huber M N; *Performance analysis of buffered banyan networks*, IEEE Trans. on Communications, Vol. 39, No. 2, Feb. 1991.

[169] Knorr R; *Realization of a 16 to 16 ATM-switching element for 680 MBit/s*, ICCT'92 Proc. of the 1992 International conference on communication theory, Beijing China, Sept. 1992, pp16.09.1-4, International Academic Publishers, 1992.

[170] Seman K, Smith D G; *Performance Analysis of an ATM Switch Capable of Supporting Multiclass Traffic* 10th UK Teletraffic Symposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp20/1-20/7.

[171] Fan Y, Wang J, Wang C; *Performance Analysis of Banyan Network Based ATM*

Switches IEEE International Conference on Communications, Chicago 1992, Vol. 3&4, pp354.1.1-5.

[172] Itoh A; *A Fault-Tolerant Switching Architecture for ATM Networks* Supercomm 92, Chicago, June 1992 IEEE, pp354.6.1-7.

[173] Gianatti S, Pattavina A; *Performance Analysis of Shared-buffered Banyan Networks under Arbitrary Traffic Patterns* Infocomm 93, San Francisco 28.3-1.4 1993, IEEE, pp8b.3.1-10.

[174] Sibal S, Zhang J; *On a Class of Banyan Networks and Tandem Banyan Switching Fabrics* Infocomm 93, San Francisco 28.3-1.4 1993, IEEE, pp4c.4.1-7.

[175] Wong P C, Yeung M S; *Pipeline Banyan - A Parallel Fast Packet Switch Architecture* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp332.1.1-6.

[176] Venkantesan R, Mouftah H T; *Performance Analysis of Multipath Banyan Networks* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp332.6.1-5.

[177] Aramaki T, Suzuki H, Hayano S, Takeuchi T; *Parallel "ATOM" Switch Architecture for High Speed ATM Networks* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp311.11-4.

[178] Kim H S; *Multichannel ATM Switch with Preserved Packet Sequence* Supercomm 92, Chicago, June 1992 IEEE, pp354.5.1-5.

[179] Liew S C, Lee T T; *N log N Dual Shuffle-Exchange Network with Error-Correcting Routing* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp311.3.1-6.

[180] Jou Y F, Nilsson A A, Lai F; *Approximate Analysis of an ATM Switching System with Bursty Arrivals and Finite Capacity* Modelling and Performance Evaluation of ATM Technology (C-15), Elsevier Science Publishers B.V. (North Holland), 1993, pp23-39.

[181] Lin T, Kleinrock L; *Performance Analysis of the Finite-Buffered "Turn-Back" Multistage Interconnection Network* Modelling and Performance Evaluation of ATM Technology (C-15), Elsevier Science Publishers B.V. (North Holland), 1993, pp3-22.

[182] Lee T T, Liew S C; *Broadband Packet Switches based on Dilated Interconnection Networks* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp311.2.1-7.

[183] Ferrari G, Lenti M, Pattavina A; *A New Architecture for Modular Growability of ATM Switches* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp311.5.1-5.

[184] Coppo P, D'Ambrosio M;, R Melen *Optimal Cost/Performance DEsign of ATM Switches* Infocomm 92, Florence 1992 IEEE, pp3D.3.1-13.

[185] Karol M J, Eng K Y;, H Obara *Improving the Performance of Input-Queued ATM*

Packet Switches Infocomm 92, Florence 1992 IEEE, pp1D.1.1-6.

[186] Dias D M, Jump J R; *Analysis and Simulation of Buffered Delta Networks* IEEE Transactions on Computers, Vol. C-30, No. 4, April 1981, pp273-283.

[187] Meyer J F, Montagna S, Paglino R; *Dimensioning of an ATM switch with shared buffer and threshold priority* Computer Networks and ISDN systems, Elsevier Science Publishers BV (North Holland), Vol. 26, Part 1, pp95-108.

[188] Goli P, Kumar V; *Performance of a Crosspoint Buffered ATM Switch Fabric* Infocomm 92, Florence 1992 IEEE, pp3D.1.1-10.

[189] DV2, ATM Switch, Technical Brochure, Netcomm Ltd, Essex, UK, 1993.

[190] Tubtiang A, Kwon H I, Pujolle G; *A Simple ATM Switching Architecture for Broadband-ISDN and its Performance* Modelling and Performance Evaluation of ATM Technology (C-15), Elsevier Science Publishers B.V. (North Holland), 1993, pp361-371.

[191] Jenq Y-C; *Performance Analysis of a Packet Switch Based on Single-Buffered Banyan Network* IEEE Journal on Selected Areas in Communications, Vol. SAC-1 No. 6, December 1983, pp401-8.

[192] Santamaria M L, Puigjaner R; *Analysis of Grade of Service in an ATM Switch* Computer and Information Sciences VI, Elsevier Science Publishers BV (NOrth Holland), 1991, pp515-23.

[193] Parr G P, Wright S, Marshall A; *Modelling ATM Switch-Fabric Based on the Knockout Principle* 10th UK Teletraffic Symposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp22/1-22/8.

[194] Kouvatsos D D, Tabet-Aouel N M, Denazis S G; *A Discrete-Time Queuing Model of a Shared Buffer ATM Switch Architecture with Bursty Arrivals* 10th UK Teletraffic Symposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp19/1-19/9.

[195] Del Re E, Fantacci R; *Efficient fast packet switch fabric with shared input buffers* IEE Proceedings-I, Vol. 140, No. 5, October 1993.

[196] Dagiuklas A K, Ghanbari M; *Priority Queuing Disciplines in ATM Switches Carrying Two Layer Video Traffic* 10th UK Teletraffic Symposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp4/1-4/6.

[197] Bruneel H, Wittenvrongel S; *Analytic performance study of ATM switching elements with on/off sources and correlated routing* Modelling and Performance Evaluation

of ATM Technology (C-15), Elsevier Science Publishers B.V. (North Holland), 1993, pp41-59.

[198] Schulzrinne H, Kurose J F, Towsley D F; *Loss Correlation for Queues with Bursty Input Streams* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp308.4.1-6.

[199] Xiong Y, Bruneel H; *Approximate Analytic Performance Study of an ATM Switching Element with Train Arrivals* Supercomm 92, Chicago, June 1992 IEEE, pp354.2.1-7.

[200] Chen D X, Mark J W; *A Buffer Managment Scheme for the SCOQ Switch Under Nonuniform Traffic Loading* Infocomm 92, Florence 1992 IEEE, pp1D.4.1-9.

[201] Chan D X, Mark J W; *Delay and Loss Control of An Output Buffered Fast Packet Switch Supporting Integrated Services* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp335A.1.1-5.

[202] Esaki H; *Call Admission Control in ATM Networks* Supercomm 92, Chicago, June 1992 IEEE, pp354.4.1-6.

[203] Freisen V J, Wong J W; *The Effect of Multiplexing, Switching and Other Factors on the Performance of Broadband Networks* Infocomm 93, San Francisco 28.3-1.4 1993, IEEE, pp10a.4.1-6.

[204] Cuthbert I G, Sapanel J C; *ATM the broadband telecommunications solution*, IEE, London, UK, 1993.

[205] Smith R M, Trivedi K S, Ramesh A V; *Performability analysis: measures, an algorithm and a case study*, IEEE Trans. on Computers, Vol. 37, No. 4, pp406-17, 1988.

[206] Ciardo G, Marie R, Sericola B and Trivedi K S; *Performability Analysis using Semi-Markov Reward Processes*, IEEE Trans. on Computers, Vol. 39, No. 10, pp1251-64, 1992.

APPENDIX A

The ICE language

A.1 Introduction

In this appendix we present the full syntax of ICE. The background of the language is described in section 1.2 The semantics are further discussed in chapter 3, along with the philosophy of some of the constructs. In chapter 6 we show the language to have the same descriptive power as extended high-level Petri nets.

A.2 Overview

The language has a declarative style that is based upon describing systems in terms of their constituent interacting discrete state components. Each COMPONENT in a system has a set of operational states. The component moves between the various states in its STATE_SET according to its predefined BEHAVIOUR. The transitions can be governed by :

- Time delays.
- Status of one or more components.
- Behaviour, ie transition event of one or more components or component counters.

Components may also have an associated AGE which can be used to manipulate their behaviour.

Components may also have COUNTERS associated with them. Counters are used to help counteract the problem of state explosion. For example, if we wished to model a buffer with 100 spaces, we could do so by using 101 states, ie 1 state for the empty condition and 100 for each of the levels of occupancy. Alternatively, we could use one state to represent the buffer and a counter which may take any value [0,100] to represent the levels of occupancy. This clearly allows the state complexity of models to be greatly reduced.

To fully define a component, three statements are required :

- STATE_SET, which lists the finite set of states that the component can exist in and any counters belonging to the component.
- BEHAVIOUR, which defines all possible transitions that can be made between

states.

- COMPONENT, which defines a component with a specified STATE_SET and BEHAVIOUR. It also defines an initial state and optionally an initial age and counter values of the component.

As well as components we can also describe passive resources which may be allocated to components. Resources may be consumable or non-consumable and are specified as STOCK and RESOURCE respectively. The WAIT_FOR statement allows dynamic creation of components and the explicit manipulation of free component and resource levels during the simulation.

The language is free format in the sense that blank space (spaces, tabs, new lines etc) are ignored. The order of statements is unimportant, unless this would cause a semantic conflict. This point is expanded in section A.3.5.

A.3 Language Syntax

In the examples of syntax given below the following conventions are used :

Keywords are shown in CAPITAL letters.
User defined names are shown in *italics.*
Optional syntax is shown in [square brackets].

A.3.1 COMPONENT

The COMPONENT statement defines one or more components which share the same STATE_SET and BEHAVIOUR (both defined later). It also defines the initial state of the component. It may define any initial counter values and the initial age of the component.

The general form of the statement is :

```
COMPONENT component _list
{
[STATE_SET :]        stateset_name ;
[BEHAVIOUR :]        behaviour_name ;
[INIT_STATE :]       state_name [ ( counter_init_list ) ];
[INIT_AGE :]         [age] ;
}
```

The *component_list* can consist of one or more names separated by commas. Alternatively, each name may be in array form, eg *switch[10],* would define ten components with the name switch0...switch9.

The *counter_init_list* can consist of one or more names which are assigned integer values separated by commas, for example

(buffer_a = 0, buffer_b = 7)

A.3.2 STATE_SET

The STATE_SET statement is used to define a finite set of states that a component can exist in and the counters belonging to the component. It also shows the operations which are performed on the counters when the component enters any of the given states. The statement is of the general form :

```
STATE_SET name {
        COUNTERS : [ counter_list ] ;
        STATES {
                [attribute] [%] statename  : [ { counter_modifier_list } ] ;
                ...
        }
}
```

An *attribute* is used to define a sub-set of states within the state set and is specified by one or more capital letters. The *%* indicates that a component is ageing when it is in the given state. A components age is a positive integer number that is incremented by the amount of simulation time that elapses while the component is in the ageing state.

The *counter_list* is a list of names of counters sepperated by commas. The

counter_modifier_list describes how counters are modified when the component *enters* the associated state. It consists of one or more *counter_modifiers* sepperated by commas. A *counter_modifier* will take one of the three general forms

- *counter_name = any_counter_name*
- *counter_name = any_counter_name expr any_b_counter_name*
- *counter_name = any_counter_name expr integer_value*

The *counter_name* is the name of the counter to be modified. *Any_counter_name* and *any_b_counter_name* are the names of the same counter or any other counters belonging to the same component. *Expr* is one of the arithmetical expressions in the set { +, -, *, / }, representing addition, subtraction, multiplication and division respectively.

For example, the statement

```
STATE_SET switching_element {
        COUNTERS : tx, rx, total;
        STATES {
                A       transmit : ( tx = tx + 1 ) ;
                A       receive :  ( rx = rx + 1 ) ;
                        clear :    ( tx = 0, rx = 0 ) ;
                        update :   ( total = tx + rx ) ;
                %       idle;
        }
}
```

defines the STATE_SET with name *switching_element* which has three counters *tx, rx* and *total* and five states *transmit, receive, clear, update* and *idle.* The attribute A is logically true when the component resides in the 'active' states transmit or receive. The component will age while in the state *idle.* The counters *tx* and *rx* will be incremented when the component enters states *transmit* and *receive* respectively and they will both be set to 0 when the component enters state *clear.* The counter *total* will be set to the sum of the counters *tx* and *rx* when the component enters state *update.*

A.3.3 SYSTEM

The SYSTEM statement is used to form complex boolean functions of component state pairs. The statement is of the form

SYSTEM *system_name = function (namelist);*

There are three types of *function*

- ANY*n*, where *n* is a positive integer. This will take the value 1 if at least *n* of the members of *namelist* are true, else it will take the value 0.
- EXACTLY*n*, where *n* is a positive integer. This will take the value 1 if exactly *n* of the members of *namelist* are true, else it will take the value 0.
- ALL. This will take the value 1 if all the members of *namelist* are true, else it will take the value 0.

A *namelist* is of the form

[!]name1, [!]name2, ..., [!]nameN

where *name* is one of the formats

- A component state pair of the form *component.state.* Takes the value 1 when *component* is in *state,* else it takes the value 0.
- A component attribute pair of the form *component.attribute.* Takes the value 1 when the *component* is in a state which posseses the *attribute*, else it takes the value 0.
- A component counter expression of the form *component.counter expr value.* Where *expr* ∈ { =, <, > } and *value* may be either an integer value or a component counter value. Takes the value 1 when the expression is true else it takes the value 0.
- A SYSTEM function of the form ANY*n*(*namelist*); ALL(*namelist*); EXACTLY*n*(*namelist*).
- A SYSTEM *name* that is defined elsewhere.

A preceding *!* would indicate a logical NOT.

The final SYSTEM format allows SYSTEM statements to be build up recursively. Consider for example

```
SYSTEM  level1  =  ANY2 ( comp1.receive, comp2.buffer > 7, comp3.A );
SYSTEM  level2  =  ALL ( level1, EXACTLY2 ( comp4.3, comp5.3, comp6.3 );
```

Level1 has value 1 if any two (or more) of the elements in the list have value 1. That is if two of the following are true : *comp1* is in state *receive*, the counter *buffer* belonging to *comp2* has a value greater than *7* or *compB* is in a state which posseses the attribute *A*. Level2 has value 1 if all the elements in the list have value 1. That isif level1 = 1 and exactly two of the components *comp4, comp5* and *comp6* are in a state which possesses the attribute A.

A.3.4 RESOURCE

Resources are passive entities which can be allocated and deallocated to and from components to modify their behaviour. Consumable resources are specified by the STOCK statement, non-consumable resources are specified by the RESOURCE statement. These statements take the form

RESOURCE { *name-quantity list* };

The *name-quantity list* is of the form

name1:quantity1, name2:quantity2, ..., nameN:quantityN

this indicates that resource *name1* has the initial value *quantity1* etc. . The form of the STOCK statement follows the same syntax.

A.3.5 BEHAVIOUR

A BEHAVIOUR statement contains a set of statements that describe the transitions between states in a STATE_SET. A BEHAVIOUR statement takes the form

```
BEHAVIOUR name {
[precondition {]   time   initial_state   ->   final_state   [PROB(n)] [}];
...
}
```

The component will stay in *initial_state* exactly *time* time units after entering and then immediately moves to *final_state*.

The *time* function can be of the two forms

- λ, a deterministic delay, where $\lambda \geq 0$.
- *distribution*(λ,[μ..]), where the *distribution* is one of a defined set, eg *exp, Weibull, gamma, beta* etc.

The *precondition* is of the form

PRECONDITION_TYPE condition

Where the *PRECONDITION_TYPE* can be one of

- ON_EVENT
- IF
- ON_RESOURCE
- ON_AGE

These are described in turn below

ON_EVENT *condition*

This means that if the component is in *initial_state* when the *condition* becomes true then it will make the timed transition into *final_state*. Note that if the *time* of the transition is 0 then this acts as a forced transition. The condition can be any valid *name*. The *condition* is an event generated by a state transition. Therefore if the component enters the *initial_state* after the *condition* becomes true then no transition is made, the event has passed.

For example, consider the following transitions

```
exp(70)    first    ->    alt_1;
ON_EVENT other.fail {    exp(50)    first    ->    alt_2; }
```

If the component described by the transitions enters the state *first* then normally it would move to state *alt_1* after a random time, T_1, determined by the exp(70) function. However if after any time t during the components existence in state *first* the component *other* moves to state *fail* then a random time, T_2, is generated by the function exp(50) and this component will move into *alt_2* if $T_2 < T_1 - t$.

IF ***condition***

This is a conditional statement which contains two basic parts. Firstly, if the *condition* is true upon the component entering the *initial_state* then the timed transition is made. Secondly, on the event the *condition* becomes true at any point while the component is in the *inital_state* then the timed transition is then made. The *condition* is the same as that in the ON_EVENT pre-condition.

For example, consider the following transition

```
IF other.fail {    7    first    ->    second; }
```

When this component described by the transition moves into state *first*, if the component *other* is in state *fail* then this component will move into state *second* after a time delay of 7. If however, component *other* is not in state *fail* when state *first* is entered but consequently moves into *fail* at time *t,* while state *first* is still occupied , the transition to state *second* will occur at time $t + 7$.

ON_RESOURCE ***condition***

The ON_RESOURCE pre-condition describes transitions that are enabled by the allocation of resources to a component. The *condition* is a resource list which takes the form

resource1:quantity1, resource2:quantity2 ,..., resourceN:quantityN
where $N \geq 1$.

When the component enters *initial_state* then it demands the resources detailed in the resource list. If all the resources are available then they are allocated to the component and it moves into state *final_state*. If the resources are not available, then the component will wait in state *initial_state* until they become free. If the component moves out of *initial_state* meantime then the request for resources is cancelled.

Note that the ON_RESOURCE condition is forced and hence the accompanying transitions cannot be timed or have associated probabilities.

ON_AGE *condition*

The *condition* specifies an age value for the component. If the component reaches this age when in the *initial_state* then it is forced into *final_state*. Note that since this is forced, the accompanying transitions cannot be timed or have associated probabilities.

Alternative transitions

Alternative Transitions may be made from an *initial_state* in one of two ways.

- With the PROB(n) keyword; where $0 \leq n \leq 1$.
- By specifying two or more unconditional transitions from an *initial_state*.

The PROB keyword allows us to list alternative transitions out of a state and the relative probabilities with which they will occur. The probabilities of transitions from an *initial_state* within a single pre-condition statement or BEHAVIOUR statement must sum to 1. When a component enters the *initial_state* the transition is selected randomly from the alternatives and the time of the transition is then calculated.

The second way of specifying alternatives is to list more than one transition from the same state without specifying any probabilities. In this instance the time for each transition is calculated and the one with the shortest time is made. If more than one time is equal then the transition declared first is made.

A.3.6 WAIT_FOR

The WAIT_FOR statement allows dynamic creation of components and explicit manipulation of free resource and stock levels during simulation. It takes the general form

WAIT_FOR *condition* { *action* }

The *condition* is one of two types

- *fixed_time*
- *component.state*

The *fixed_time* option allows an action to be enabled at a defined time during the simulation. The time is specified by an integer or a symbolic constant.

The *component.state* option behaves like an ON_EVENT precondition. That is at the instant the *component* enters the *state* the *action* is enabled.

The *action* may either be a component decleration, using the COMPONENT statement or it may be the increment or decrement of free resources or stock levels, taking the general form

RESOURCE { *name_quantity_list* }

The *name_quantity_list* will consist of one or more lines of the form

name : [*sign*] *quantity* ;

The *name* is any resource name. The sign is either + or - which represents increment of decrement respectively, the default being +. The *quantity* may be either an integer value or a symbolic constant. Note that STOCK levels are manipulated in exactly the same way.

Appendix B

Data Structures, Objects and Files created during Simulation

B.1 An Ice program - 'logger.ice'

Figure B.1 shows the block diagram of a simple data logger system. The *logger* records incoming datB. At given time intervals the *controller* polls the *logger* and the recorded data is down-loaded to the *controller.* The state diagram of the system operation we wish to consider is given in figure B.2.

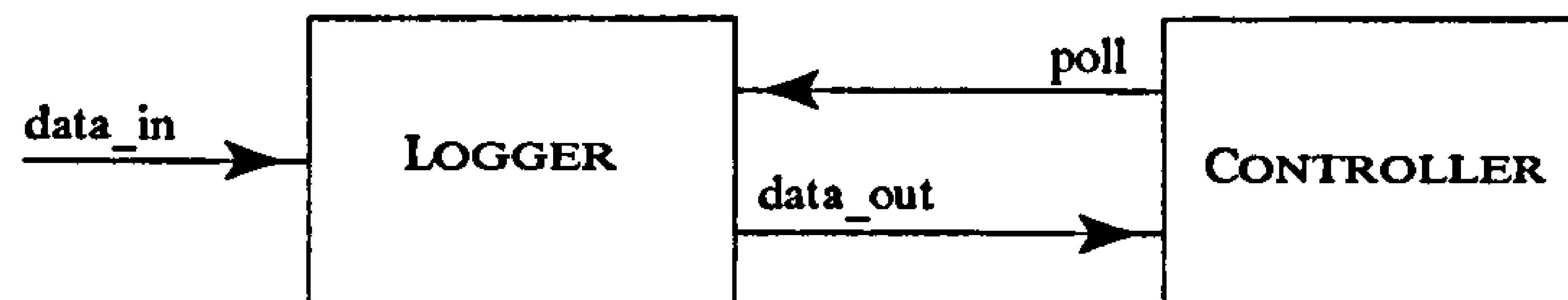


Figure B.1 Block diagram of Data Logger and Controller

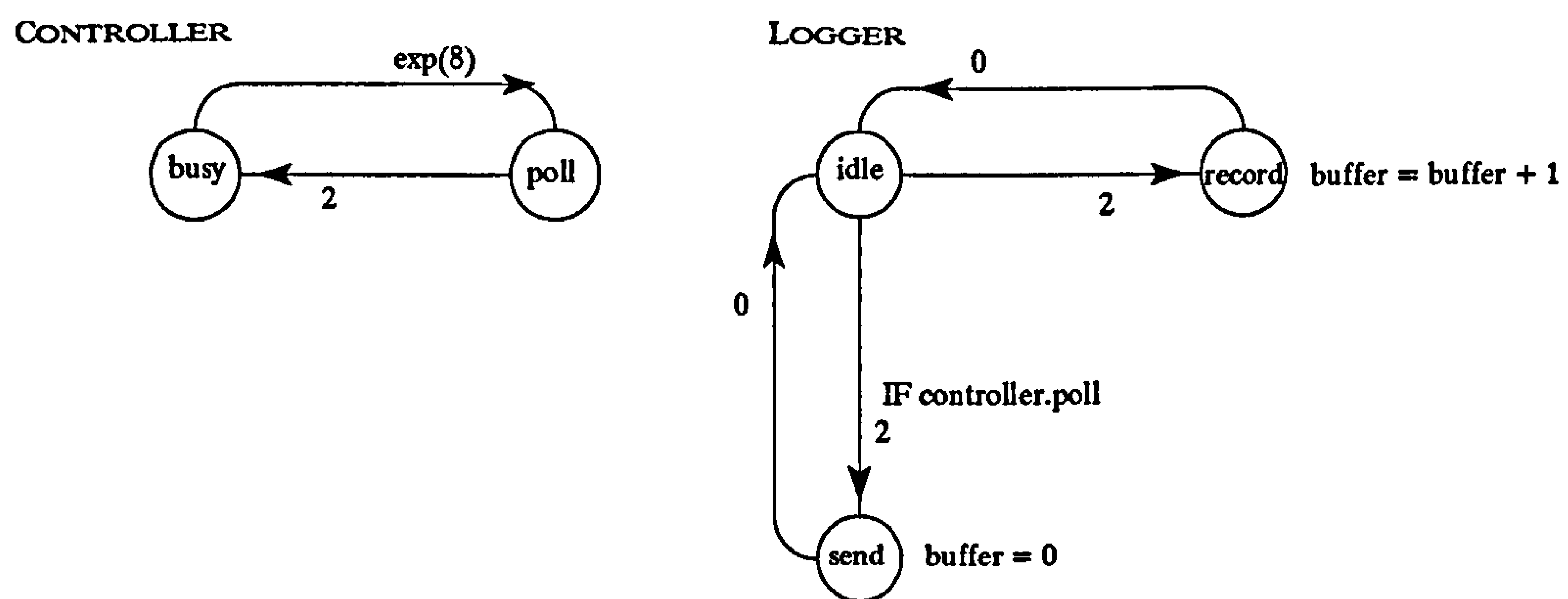


Figure B.2 State diagram of Data Logger and Controller

From the state diagram we can model the system in Ice using the code given in listing B.1. Note that line numbers are for reference and not part of the code.

```
01 // File              : logger.ice
02 // Author            : GAC
03 // Date              : 21.06.95
04 // Purpose           : Models a simple data logger. Used to illustrate
05 //                   : timed conditional transitions and Counters.
06
07 // Value above which data load considered high
08 CONSTANT {HIGH_VAL = 4;}
09
10 STATE_SET ss_control {
11        COUNTERS : ;
12        STATES {
13                busy : ;
14                poll : ;
15        }
16 }
17 BEHAVIOUR be_control {
18        exp(6)  busy -> poll ;
19        2       poll -> busy ;
20 }
21 COMPONENT control {ss_control; be_control; busy;}
22
23 STATE_SET ss_logger {
24        COUNTERS : buffer, empty ;
25        STATES {
26                idle    : ;
27                record  : { buffer = buffer + 1 } ;
28                send    : { buffer = empty } ;
29        }
30 }
31 BEHAVIOUR be_logger {
32        2 idle -> record ;
33        0 record -> idle ;
34        IF_ON control.poll {
35                3 idle -> send ; }
36        0 send -> idle ;
37 }
38 COMPONENT logger {ss_logger; be_logger; idle(buffer = 0, empty = 0); }
39
40 SYSTEM data_high = (logger.buffer > HIGH_VAL);
41
42 RUN(1);
43 STOPTIME(1000);
```

Listing B.1 Ice code to model the data logger system - 'logger.ice'

B.2 Parsing of 'logger.ice'

Figures B.3 - B.5 show the links between the data structures produced when the 'logger.ice' code is parsed. Section B.2.1 lists the actual data structures. Every data structure is given a name shown in bold. It is these names that are used in the linking diagrams. It is important to note that for clarity only partial data structures are shown. Whenever a structure utitlises a *Union* ie a single variable which may take on different types, only the type used in the particular instance is given.

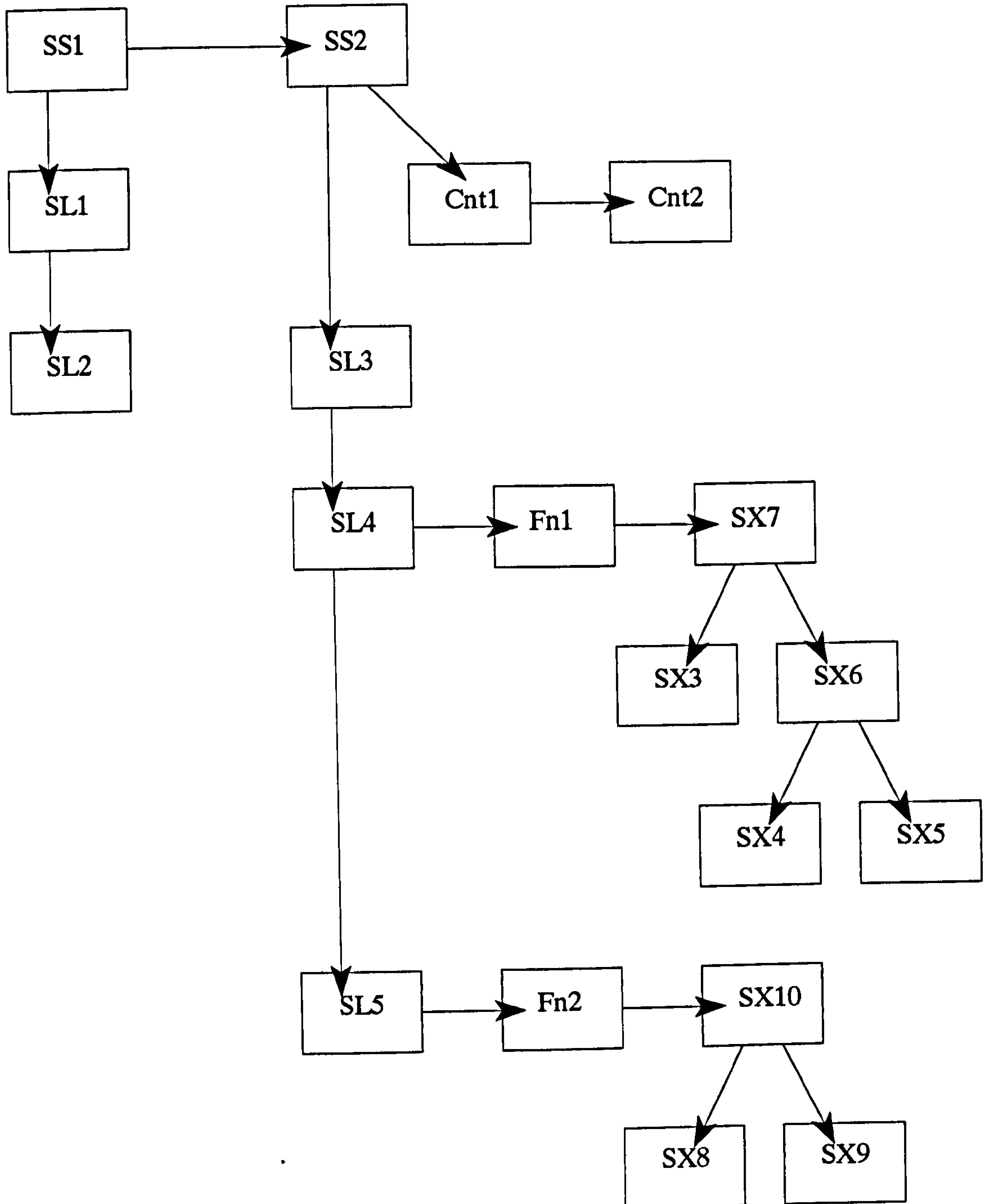


Figure B.3 STATE_SET and related structures

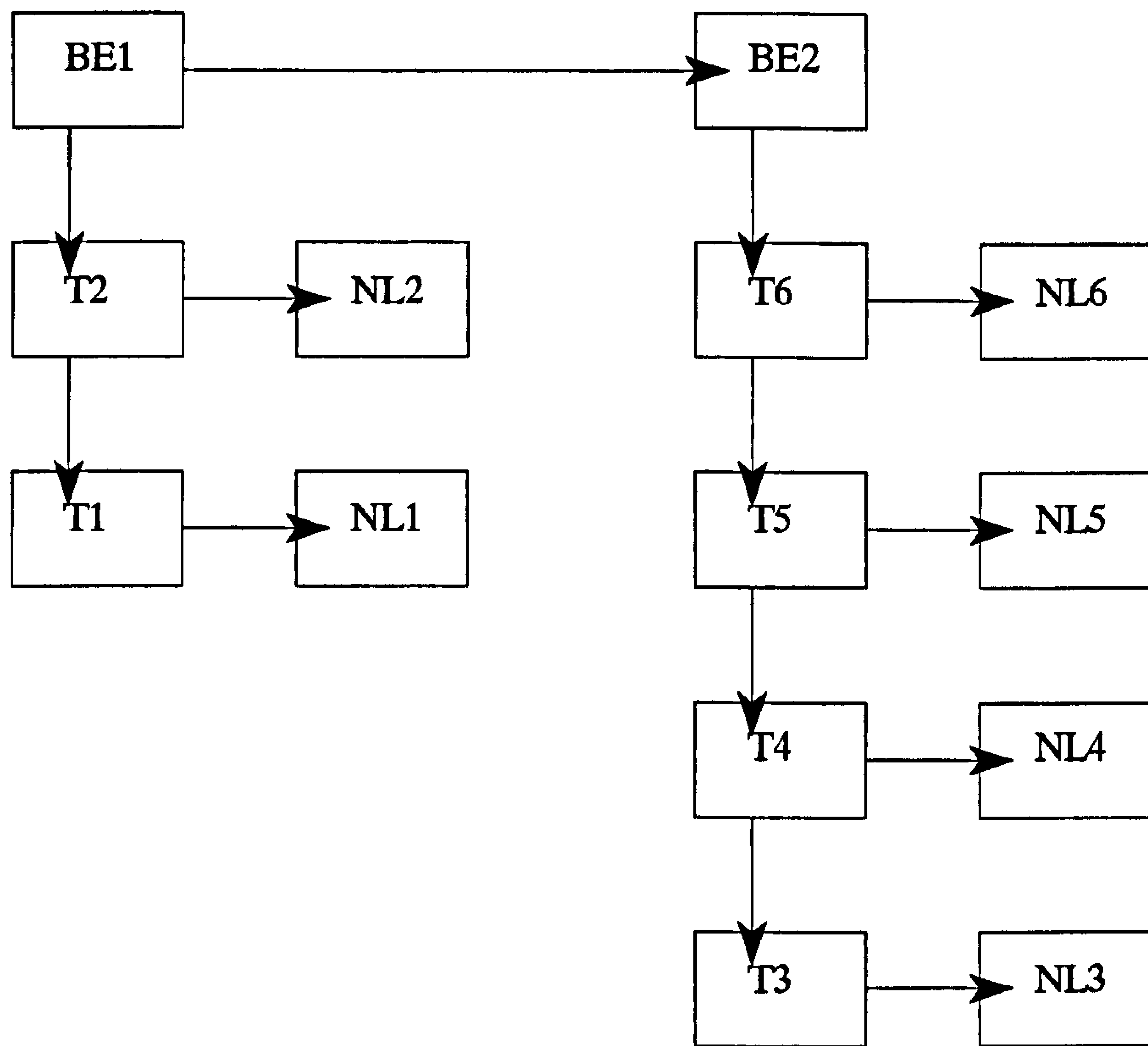
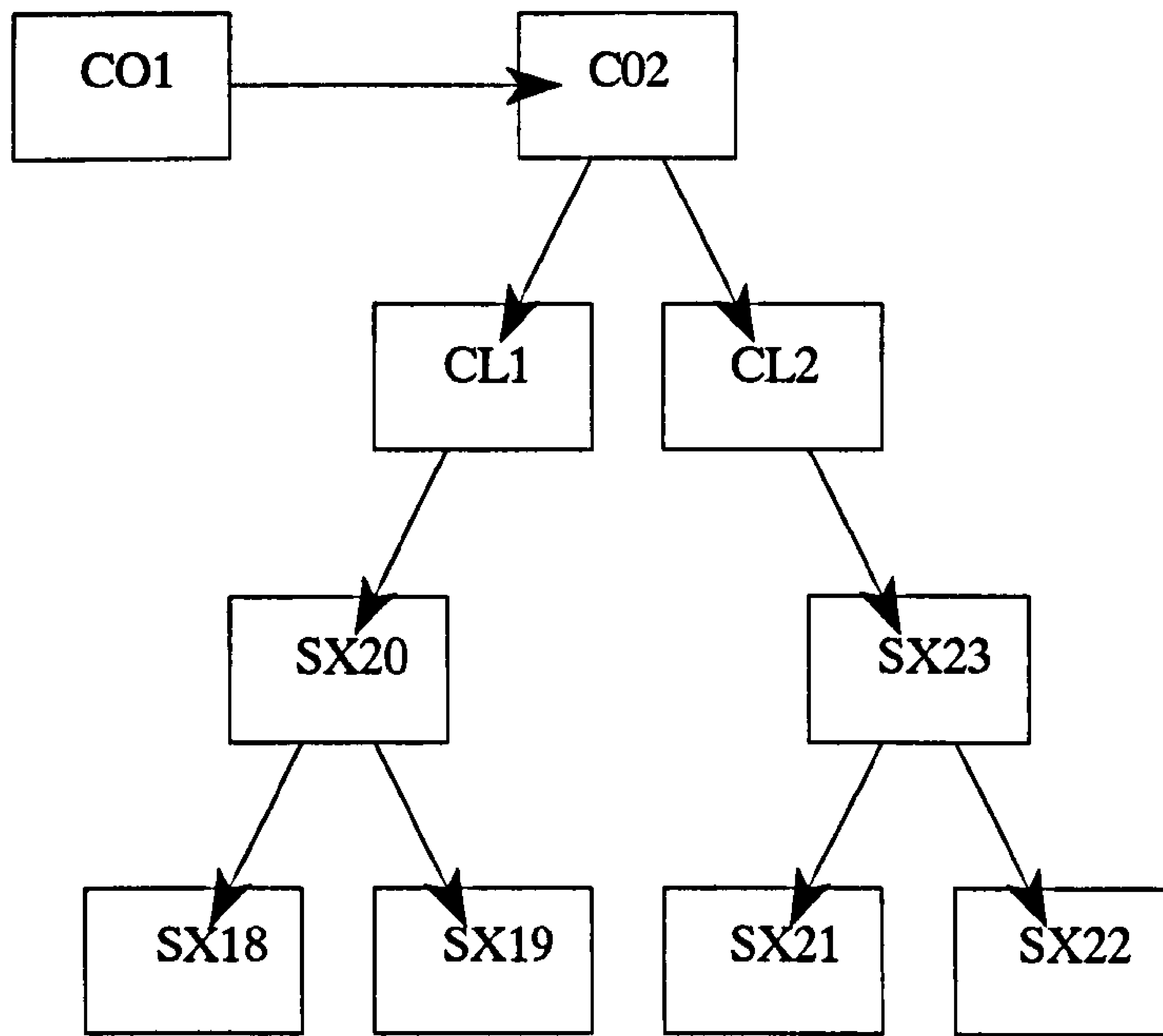


Figure B.4 BEHAVIOUR and related structures



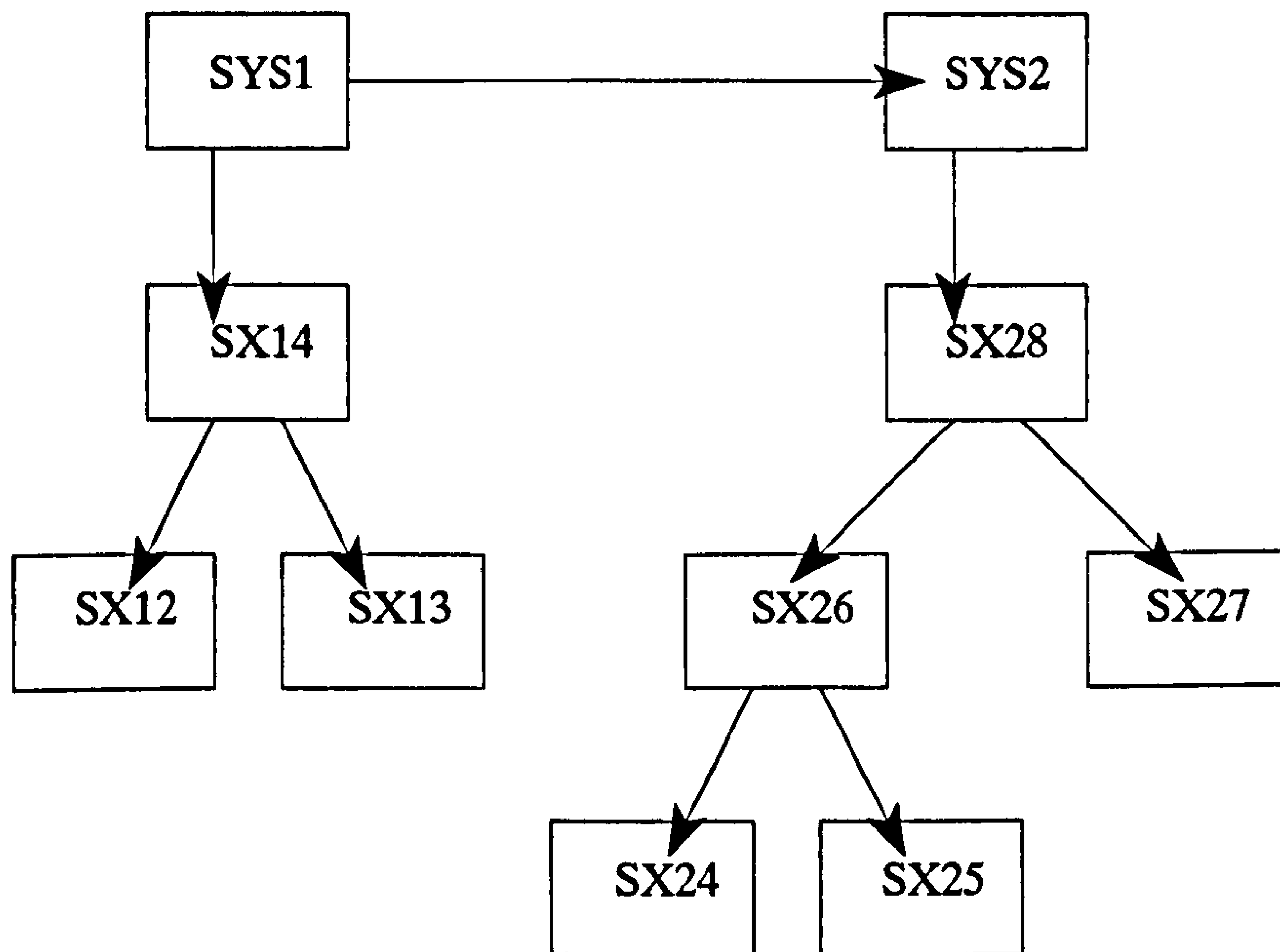


Figure B.5 COMPONENT, SYSTEM and related structures

B.2.1 Simulation data structures

struct symtabentry SS1

char	*name		ss_control
int	lineno		16
int	objno		-
int	flags		-
struct symtabentry	*next		symtab[ss_control]
struct symtabentry	*nextt		SS2
Symtypes	type		Sym_state_set
Union U	struct states	int	2
		StateList *list	SL1
		StateList **hashtable	-
		Counter *clist	NULL
		Counter **counterhtable	NULL
		int ncounters	0

struct StateList SL1

struct StateList	*next	SL2
struct StateList	*hnext	htable[SL1]
char	*name	busy
long	attribs	0
int	flow	0
int	no	0
FnExp	fnlist	NULL

struct StateList SL2

struct StateList	*next	NULL
struct StateList	*hnext	htable[SL2]
char	*name	poll
long	attribs	0
int	flow	0
int	no	1
FnExp	fnlist	NULL

struct symtabentry **SS2**

char	*name		ss_logger
int	lineno		23
int	objno		-
int	flags		-
struct symtabentry	*next		symtab[ss_logger]
struct symtabentry	*nextt		NULL
Symtypes	type		Sym_state_set
Union U	struct states	int	3
		StateList *list	**SL3**
		StateList **hashtable	-
		Counter *clist	Cnt1
		Counter **counterhtable	countertable_SS2
		int ncounters	2

struct Counter **Cnt1**

struct Counter	*next	**Cnt2**
char	*name	buffer
int	value	-
struct Counter	*hnext	counthtable[1]
int	no	0

struct Counter **Cnt2**

struct Counter	*next	NULL
char	*name	empty
int	value	-
struct Counter	*hnext	counthtable[2]
int	no	1

struct StateList **SL3**

struct StateList	*next	SL4
struct StateList	*hnext	htable[SL3]
char	*name	idle
long	attribs	0
int	flow	0
int	no	0
FnExp	fnlist	NULL

struct StateList **SL4**

struct StateList	*next	SL5
struct StateList	*hnext	htable[SL4]
char	*name	record
long	attribs	0
int	flow	0
int	no	1
FnExp	fnlist	**Fn1**

struct FnExp **Fn1**

struct FnExp	*nextfn	NULL
char	*name	function1
statexp	*fexp	SX7
int	counterno	0

struct Statexp **SX7**

int lineno		26
SXnodes type		SXassign
Union U struct op	struct Statexp *left	SX3
	struct Statexp *right	SX6

struct Statexp **SX3**

int lineno		26
SXnodes type		SXstr
Union U	char *str	buffer

struct Statexp **SX6**

int	lineno		26
SXnodes	type		SXplus
Union U struct op		struct Statexp *left	**SX4**
		struct Statexp *right	**SX5**

struct Statexp **SX4**

int	lineno		26
SXnodes	type		SXstr
Union U		char *str	buffer

struct Statexp **SX5**

int	lineno		26
SXnodes	type		SXconst
Union U		double dconst	1

struct StateList **SL5**

struct StateList	*next	NULL
struct StateList	*hnext	htable[SL5]
char	*name	send
long	attribs	0
int	flow	0
int	no	2
FnExp	fnlist	**Fn2**

struct FnExp **Fn2**

struct FnExp	*nextfn	NULL
char	*name	function2
statexp *fexp		**SX10**
int	counterno	0

struct Statexp **SX10**

int lineno		27
SXnodes type		SXassign
Union U struct op	struct Statexp *left	**SX8**
	struct Statexp *right	**SX9**

struct Statexp **SX8**

int lineno		27
SXnodes type		SXstr
Union U	char *str	buffer

struct Statexp **SX9**

int lineno		27
SXnodes type		SXstr
Union U	char *str	empty

struct symtabentry **BE1**

char	*name		be_control
int	lineno		9
int	objno		-1
int	flags		-
struct symtabentry	*next		symtab[be_control]
struct symtabentry	*nextt		**BE2**
Symtypes	type		Sym_behaviour
Union U		Tran *trans	**T2**

struct Tran T2

struct Tran	*next	**T1**
int	lineno	18
float	prob	0.0
char	*from_name	busy
char	*to_name	poll
int	to_state	1
Trantypes	type	TR-exp
Condtypes	cond	TR-notcond
NumList	params	**NL1**

struct NumList **NL2**

struct Numlist	*next	NULL
float	f	4

struct Tran **T1**

struct Tran	*next	NULL
int	lineno	19
float	prob	0.0
char	*from_name	poll
char	*to_name	busy
int	to_state	0
Trantypes	type	TR_fixed
Condtypes	cond	TR_notcond
NumList	params	**NL2**

struct NumList **NL1**

struct Numlist	*next	NULL
float	f	2

struct symtabentry **BE1**

char	*name	be_logger
int	lineno	32
int	objno	-1
int	flags	-
struct symtabentry	*next	symtab[be_logger]
struct symtabentry	*nextt	NULL
Symtypes	type	Sym_behaviour
Union U	Tran *trans	**T6**

struct Tran **T6**

struct Tran	*next	T5
int	lineno	32
float	prob	0.0
char	*from_name	idle
char	*to_name	record
int	to_state	1
Trantypes	type	TR_fixed
Condtypes	cond	TR_notcond
NumList	params	**NL6**

struct NumList **NL6**

struct NumList	*next	NULL
float	f	2

struct Tran **T5**

struct Tran	*next	**T4**
int	lineno	33
float	prob	0.0
char	*from_name	record
char	*to_name	idle
int	to_state	0
Trantypes	type	TR_fixed
Condtypes	cond	TR_notcond
NumList	params	**NL5**

struct NumList **NL5**

struct Numlist	*next	NULL
float	f	0

struct Tran **T4**

struct Tran	*next	**T3**
int	lineno	35
float	prob	0.0
char	*from_name	idle
char	*to_name	send
int	to_state	2
Trantypes	type	TR_fixed
Condtypes	cond	TR_ifon
NumList	params	**NL4**

struct NumList **NL4**

struct Numlist	*next	NULL
float	f	2

struct Tran **T3**

struct Tran	*next	NULL
int	lineno	36
float	prob	0.0
char	*from_name	send
char	*to_name	idle
int	to_state	0
Trantypes	type	TR_fixed
Condtypes	cond	TR_notcond
NumList	params	**NL3**

struct NumList **NL3**

struct Numlist	*next	NULL
float	f	0

struct symtabentry **CO1**

char	*name			control
int	lineno			21
int	objno			0
int	flags			-
struct symtabentry	*next			symtab[control]
struct symtabentry	*nextt			**CO2**
Symtypes	type			Sym_component
Union U	struct comp	char	*ss	ss_control
		char	*be	be_control
		char	*is	busy
		int	p1	0
		int	nparams	NULL
		CounterExp	initclist	NULL

struct symtabentry **CO2**

char	*name			logger
int	lineno			38
int	objno			2
int	flags			-
struct symtabentry	*next			symtab[logger]
struct symtabentry	*nextt			NULL
Symtypes	type			Sym_component
Union U	struct comp	char	*ss	ss_logger
		char	*be	be_logger
		char	*is	idle
		int	p1	2
		int	nparams	NULL
		CounterExp	initclist	**CL1**

struct CounterExp **CL1**

struct CounterExp	*next	**CL2**
char	*name	counter0
Statexp *exp		**SX20**
int	Counterno	0

struct Statexp **SX20**

int lineno		38
SXnodes type		SXassign
Union U struct op	struct Statexp *left	**SX18**
	struct Statexp *right	**SX19**

struct Statexp **SX18**

int lineno		38
SXnodes type		SXstr
Union U	char *str	buffer

struct Statexp **SX19**

int lineno		38
SXnodes type		SXconst
Union U	double dconst	0

struct CounterExp **CL2**

struct CounterExp	*next	NULL
char	*name	counter1
Statexp *exp		**SX23**
int	Counterno	1

struct Statexp **SX23**

int lineno		38
SXnodes type		SXassign
Union U struct op	struct Statexp *left	**SX21**
	structStatexp*right	**SX22**

Struct Statexp **SX21**

int	lineno		38
SXnodes	type		SXstr
Union U		char *str	empty

struct Statexp **SX22**

int	lineno		38
SXnodes	type		SXconst
Union U		double dconst	0

struct symtabentry **SYS1**

char	*name		<anon system 1>
int	lineno		34
int	objno		1
int	flags		NO_TRACE
struct symtabentry	*next		-
struct symtabentry	*nextt		**SYS2**
Symtypes	type		Sym_system
Union U		Statexp *sexp	**SX14**

struct Statexp **SX14**

int	lineno		34
SXnodes	type		SXdot
Union U struct op		struct Statexp *left	**SX12**
		structStatexp*right	**SX13**

struct Statexp **SX12**

int	lineno		34
SXnodes	type		SXstr
Union U		char *str	control

struct Statexp **SX13**

int	lineno		34
SXnodes	type		SXstr
Union U		char *str	poll

struct symtabentry **SYS2**

char	*name		data_high
int	lineno		40
int	objno		3
int	flags		-
struct symtabentry	*next		symtab[data_high]
struct symtabentry	*nextt		NULL
Symtypes	type		Sym_system
Union U		Statexp *sexp	**SX28**

struct Statexp **SX28**

int	lineno		40
SXnodes	type		SXgt
Union U struct op		struct Statexp *left	**SX26**
		structStatexp*right	**SX27**

struct Statexp **SX26**

int	lineno		40
SXnodes	type		SXdot
Union U struct op		struct Statexp *left	**SX24**
		structStatexp*right	**SX25**

struct Statexp **SX24**

int	lineno		40
SXnodes	type		SXstr
Union U		char *str	logger

struct Statexp **SX25**

int	lineno		40
SXnodes	type		SXstr
Union U		char *str	buffer

struct Statexp **SX27**

int	lineno		40
SXnodes	type		SXstr
Union U		char *str	HIGH_VAL

B.3 Compilation

This section gives the simulation objects that are created when the simulation data structures listed in section B.2.1 are compiled.

The *Component* objects **OBJ0** and **OBJ2** are created by the compilation of symtabentries **CO1** and **CO2**.

Object Component **OBJ0, OBJ2**

			OBJ0	**OBJ2**
	long	next_time	0	0
	int	next_state	0	0
	int	p1	0	2
	ResList	*resources	NULL	NULL
public:				
	Statespace	*space	**SP1**	**SP2**
	int	flags	-	-
	char	*name	control	logger
	int	objno	0	2
	int	init_state	0	0
	int	state	0	0
	int	flow	0	0
	long	attribs	0	0
CqGraph			**CQ1**	**CQ2**
	CqGraph	*pprev	this	this
	CqGraph	*pnext	this	this
	CqGraph	*cprev	this	this
	CqGraph	*cnext	this	this
	int	pno	-1	-1
	int	cno	-1	-1
	Counter	*counters	NULL	**[Cnt1|Cnt2]**
	FnExp	**functions	NULL	**[Fn1|Fn2]**

Note that **Cnt1, Cnt2, Fn1** and **Fn2** are the data structures shown in section B.2.1. The **fexp* fields of Fn1and Fn2 will have been changed to SX35 and SX38 respectively, these structures are shown below.

struct Statexp SX35

int lineno		{
SXnodes type		SXplus
Union U struct op	struct Statexp *left	SX33
	structStatexp*right	SX34

struct Statexp SX33

int lineno		{
SXnodes type		SXcounter
Union U struct op	struct Statexp *left	SX31
	structStatexp*right	SX32

Struct Statexp SX31

int lineno		{
SXnodes type		SXiconst
Union U	int iconst	2

struct Statexp SX32

int lineno		{
SXnodes type		SXiconst
Union U	int iconst	0

struct Statexp SX34

int lineno		{
SXnodes type		SXiconst
Union U	int iconst	1

struct Statexp SX38

int lineno		{
SXnodes type		SXcounter
Union U struct op	struct Statexp *left	SX36
	structStatexp*right	SX37

Struct Statexp SX36

int lineno		{
SXnodes type		SXiconst
Union U	int iconst	2

struct Statexp **SX37**

int lineno		{
SXnodes type		SXiconst
Union U	int iconst	1

The Sexp objects **OBJ1** and **OBJ3** are created by the compilation of the symtabentries **SYS1** and **SYS2.**

Object Sexp **OBJ1, OBJ3**

		OBJ1	**OBJ3**
Statexp	*sexp	**SX14**	**SX28**
public:			
int	flags	NO_TRACE	NO_TRACE
char	*name	<anon system 1>	data_high
int	objno	1	3
int	initstate	0	0
int	state	0	0
int	flow	0	0
long	attribs	-	-
CqGraph		**CQ3**	**CQ4**
CqGraph	*pprev	this	this
CqGraph	*pnext	this	this
CqGraph	*cprev	this	this
CqGraph	*cnext	this	this
int	pno	-1	-1
int	cno	-1	-1

The Statespace structures **SP1** and **SP2** are created by the compilation of symtabentries **BE1** and **BE2.** The state objects **S1** and **S2** are created by the compilate of symtabentries **SS1, SL1** and **SL2.** The state objects **S3, S4,** and **S5** are created by the compilation of symtabentries **SS2, SL3, SL4** and **SL5.**

Object StateSpace **SP1, SP2**

		SP1	**SP2**
char	*name	be_control	be_logger
int	nstates	2	3
int	nparams	0	0
State	*states	[S1|S2]	[S3|S4|S5]

Object State **S1,S2,S3,S4,S5**

		S1	**S2**	**S3**	**S4**	**S5**
Tran	*trans	**T2**	**T1**	**T4**	**T5**	**T3**
FnExp	*fnlist	Null	Null	Null	**Fn1**	**Fn2**
public:						
char	name	busy	poll	idle	record	send
int	no	0	1	0	1	2
int	flow	0	0	0	0	0
long	attribs	0	0	0	0	0

B.3.1 Consequence graphs

Consider the conditional statement in the BEHAVIOUR statement of component logger of the previous exampl.

```
34      IF control.poll {
35              2 idle -> send; }
```

As we saw during compilation, three objects were created which relate to this statement :

OBJ0 for component *control*
OBJ1 for the anonymous system statement *control.poll*
OBJ2 for component *logger*

These objects would be permanently linked together during compilation as shown in figure B.6. Note that for clarity only the consequence graph part of the objects is shown.

During the simulation phase, when component *control* is not in state *poll* then this would be the only link. Whenever it is in state *poll* a temporary (or dynamic) link must be provided so that the transition in component *logger* from state *idle* to state *send* may be implemented. This dynamic link that would be created is shown in figure B.7. In this instance the derived class *OnNode* is used.

In this example the conditional statement was straightforward and only dependant upon one external component. In the case of complex conditional statements links must be provided to all interacting objects. This is facilitated by the use of the consequence graphs parent and child, previous and next pointers which would indicate each nodes position in a chain of linked parent and child objects.

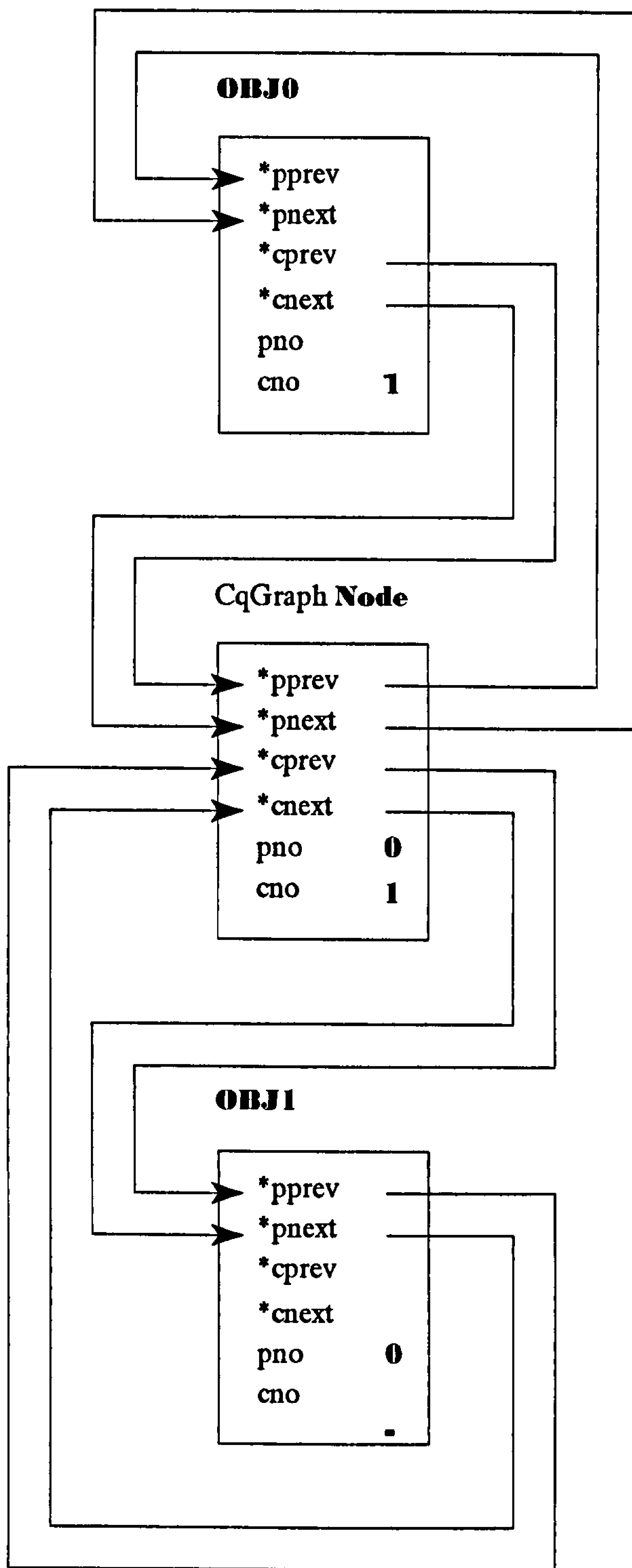


Figure 4.6 The permanent consequence graph links created at compile time between the component *logger* and the *IF* statement.

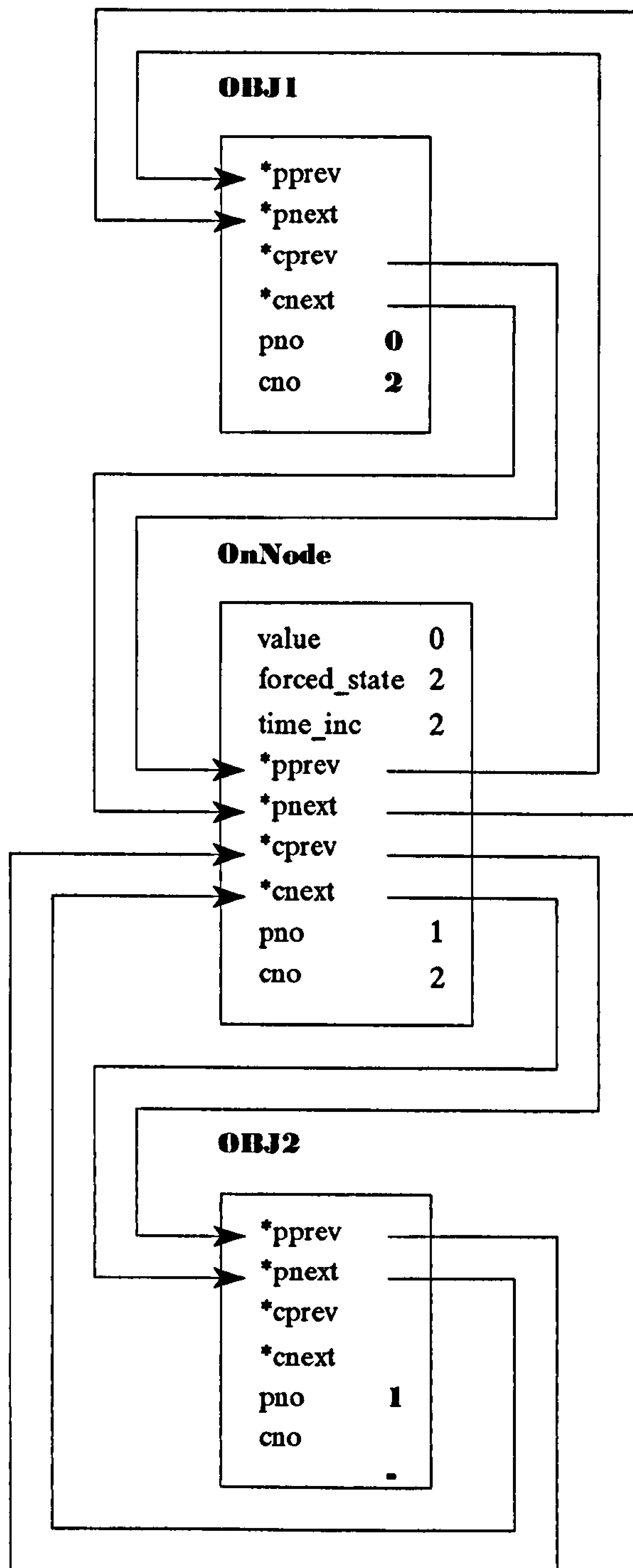


Figure B.7 The dynamic consequence graph links created during simulation when component *control* enters state *poll.*

B.4 Simulation - The event data file

The listing in figure B.8 shows the start of the event data file for the previous example. All simulation object data is shown as well as the first few events. Note that the control codes *ED_xxxxxx* are used by the analyser when reading the file to determine the nature of the data on the preceding line. Note that the comments are shown to aid understanding and are not part of the actual file.

TECSIM.EVT

```
1    ED_MAGIC 4 1000                          // start string, number of objects, runs
2    ED_STATESET ss_control                   // state_set name
3    ED_STATE 0 busy                          // state attributes, name
4    ED_STATE 0 poll                          // "
5    ED_STATESET ss_logger                    // state_set name
6    ED_COUNTER buffer                        // counter name
7    ED_COUNTER empty                         // "
8    ED_STATE 0 idle                          // state attributes, name
9    ED_STATE 0 record                        // "
10   ED_STATE 0 send                          // "
11   ED_COMPONENT 0 contorl ss_control        // component object no, name, state_set
12   ED_COMPONENT 2 logger ss_logger          // "
13   ED_SYSTEM 1 <anon_system 1>              // system object no, name
14   ED_SYSTEM 3 data_high                    // "
15   ED_START 1000                            // start of simulation, stop time
16   ED_TIME 0 ED_Trans 1 0                   // time, transition object no, state
17   ED_TIME 0 ED_Trans 3 0                   // "
18   ED_TIME 0 ED_Trans 0 0                   // "
19   ED_TIME 0 ED_Trans 2 0                   // "
20   ED_TIME 2 ED_Trans 2 1 ED_CTrans 2 0 1   // ", counter object no, counter no, value
...
```

B.5 Statistical analysis data

This section shows the data objects that are created from the event data file shown in section B.4 and the user analysis specification in figure 4.9.

B.5.1 Objects created from the event data file

Shown below are the *TDthing* data objects that are created from the STATE_SET, COMPONENT and SYSTEM information in the event data file. The format used is to give the line of the event data file and the corresponding data structures.

```
2    ED_STATESET ss_logger
3    ED_STATE 0 busy
4    ED_STATE 0 poll
```

struct TDthing **tdt1,st1,st2**

		tdt1	st1	st2
char	*name	ss_control	busy	poll
long	value	0		
int	monitor	0		
int	objno	-	-	-
enum TDthing	type	TDstateset	TDstates	TDstates
union {				
struct TDthing	**states	[st1,st2]		
char	*attribs			
int	decvalue}			

5 ED_STATESET ss_logger
6 ED_COUNTER buffer
7 ED_COUNTER empty
8 ED_STATE 0 idle
9 ED_STATE 0 record
10 ED_STATE 0 send

struct TDthing **tdt2,co1,co2**

		tdt2	co1	co2
char	*name	ss_logger	buffer	empty
long	value	2		
int	monitor	0		
int	objno	-	-	-
enum TDthing	type	TDstateset	TDcounter	TDcounter
union {				
struct TDthing	**states	[co1,co2,st3,st4,st5]		
char	*attribs			
int	decvalue}			

struct TDthing **st3,st4,st5**

		st3	st4	st5
char	*name	idle	record	send
long	value			
int	monitor			
int	objno	-	-	-
enum TDthing	type	TDstates	TDstates	TDstates
union {				
struct TDthing	**states			
char	*attribs			
int	decvalue}			

```
11      ED_COMPONENT 0 contorl ss_control     // component object no, name, state_set
12      ED_COMPONENT 2 logger ss_logger              // "
13      ED_SYSTEM 1 <anon_system 1>                  // system object no, name
14      ED_SYSTEM 3 data_high                        // "
```

struct TDthing **tdt3,tdt4,tdt5,tdt6**

		tdt3	tdt4	tdt5	tdt6
char	*name	control	logger	<anon system 1>	data_high
long	value	0	0	0	0
int	monitor	1	1	0	1
int	objno	0	2	1	3
enum TDthing	type	TDcomps	TDcomps	TDsyss	TDsyss
union {					
struct TDthing	**states				
char	*attribs				
int	decvalue}	0	1	-	-

B.5.2 Objects created from the analysis file

Shown below are the *ANA*, *Statexp*, *stats* and *Stats* objects created from the statistical analysis information given in the analysis file data structures shown in figure 3.10.

struct ANA **ana1,ana2,ana3**

	ana1	ana2	ana3
Statexp *sexp	sexp1	sexp2	sexp3
int countval	-1	-1	0
int cval	0	0	0
TDthing *obj			
int flags	STAT2	IGNORESPIKES	
ANstats format[]	[ANqstr| ANmean| ANnull|... ..ANpercentage]	[ANqstr| ANpercentage| ANnull]	[ANqstr| ANmean| ANnull]
int formatparams[]			
char *formatqstrs[]			
char *headings[]			
char *title			
char *filename			
double classify			
long account			
int conf	95	95	95
long ovalue	0	0	0
long otime	0	0	0
stats *stat1	stan1	statsB	statsC
stats *stat2	statsA		
Flist results	FLres1	FLres2	FLres3
Flistelem *current_result	eA	eB	eC
long soacc	0	0	0
long eoacc	0	0	0

struct Statexp**sexp1,tmp,tmp2**

			sexp1	**tmp**	**tmp2**
SXnodes type			SXdot	SXobj	SXint
Union U	struct op	struct Statexp *left	**tmp**		
		struct Statexp *right	**tmp2**		
	TDthing *obj			**tdt3**	
	int val				1

struct Statexp**sexp2,tmpa,tmp2a**

			sexp2	**tmpa**	**tmp2a**
SXnodes type			SXdot	SXobj	SXint
Union U	struct op	struct Statexp *left	**tmpa**		
		struct Statexp *right	**tmp2a**		
	TDthing *obj			**tdt6**	
	int val				1

struct Statexp**sexp3,tmpb,tmp2b**

			sexp3	**tmpb**	**tmp2b**
SXnodes type			SXdot	SXobj	SXint
Union U	struct op	struct Statexp *left	**tmpb**		
		struct Statexp *right	**tmp2b**		
	TDthing *obj			**tdt4**	
	int val				0

Note that in the *stats* structures the *union* fields that are not used in this exampl are not shown.

struct stats **stat1,stat2,stat3**

	stat1	**stat2**	**stat3**
char *object	control	data_high	logger
char *state	poll	T	buffer
ANstats type	ANmean	ANpercentage	ANmean
ANstats stats_on	ANpercentage	-1	-1

Note that only one of the three *Stats* structures is shown. The other two would be similar. This example has been shown with the fields being filled in with the appropriate data as a sample simulaiton is run.

struct Stats **stan1**

long nsamples		~~0~~ ~~1~~ ~~2~~ 3
double sum		~~0~~ ~~2~~ 4 6
double SumSq		~~0~~ 4 ~~8~~ 12
double SumExp		0
double max		2
double min		2
start save_data		0
float *data[]		[2|2|2...]
float *cdata		[0|0|0...]
int sizeof_data		256
short sorted		0

Appendix C

ICE listing of communication network sources model

// File : rantest.ice

// Author : GAC

// Date : 06.11.95

// Purpose : Checks effect on queues of random number streams

// Counter reference values

```
CONSTANT {EMPTY = 0; FULL = 20;}
```

// Model of network traffic

```
STATE_SET ss_ch {
   COUNTERS : total;
   STATES {
      quiet :;
      arrive : {total = total +1};
   }
}
BEHAVIOUR be_chA {
   1 quiet -> arrive PROB(0.4);
   1 quiet -> quiet PROB(0.6);
   0 arrive -> quiet;
}
BEHAVIOUR be_chB {
//   1 quiet -> arrive PROB(0.5);
//   1 quiet -> quiet PROB(0.5);
   0 arrive -> quiet;
}
COMPONENT chA {ss_ch; be_chA; quiet(total = 0);}
COMPONENT chB {ss_ch; be_chB; quiet(total = 0);}
```

// Model of a fifo queue

```
STATE_SET ss_q {
   COUNTERS : qlen, drop, total;
   STATES {
      wait :;
      inc : {qlen = qlen + 1, total = total + 1};
      dec : {qlen = qlen - 1};
      drop : {qlen = qlen - 1, drop = drop + 1};
   }
}

BEHAVIOUR be_qiA {
   IF_ON chA.arrive {
      0 wait -> inc; }
   IF_ON seA.read {
      0 wait -> dec; }
   IF qiA.qlen > FULL {
      0 inc -> drop; }
   ON_EVENT chA.quiet {
      0 inc -> wait; }
   ON_EVENT !seA.read {
      0 dec -> wait;
        0 drop -> wait; }
}
```

```
BEHAVIOUR be_qiB {
   IF_ON chB.arrive {
      0 wait -> inc; }
   IF_ON seB.read {
      0 wait -> dec; }
   IF qiB.qlen > FULL {
      0 inc -> drop; }
   ON_EVENT !chB.arrive {
      0 inc -> wait; }
   ON_EVENT !seB.read {
      0 dec -> wait;
        0 drop -> wait; }
}
```

// Model of network terminating units (NTUs)

```
COMPONENT qiA {ss_q; be_qiA; wait(qlen = 0, drop = 0);}
COMPONENT qiB {ss_q; be_qiB; wait(qlen = 0, drop = 0);}
```

// Model of switching elements

```
STATE_SET ss_se {
   COUNTERS :;
   STATES {
      wait :;
      read :;
   }
}
BEHAVIOUR be_seA {
   IF_ON qiA.qlen > EMPTY {
      2 wait -> read; }
   0 read -> wait;
}
COMPONENT seA {ss_se; be_seA; wait; }
```

// Model of switching element queues

```
BEHAVIOUR be_seA_q {
   IF_ON seA.read {
      0 wait -> inc; }
   IF seA_q.qlen > FULL {
      0 inc -> drop; }
   ON_EVENT !seA.read {
      0 inc -> wait;
      0 drop -> wait; }
}
COMPONENT seA_q {ss_q; be_seA_q; wait(qlen = 0, drop = 0);}
```

// Model of switching element

```
BEHAVIOUR be_seB {
   IF_ON qiB.qlen > EMPTY {
      2 wait -> read; }
   0 read -> wait;
}
COMPONENT seB {ss_se; be_seB; wait; }
```

// Model of switching element queues

```
BEHAVIOUR be_seB_q {
   IF_ON seB.read {
      0 wait -> inc; }
   IF seB_q.qlen > FULL {
      0 inc -> drop; }
   ON_EVENT !seB.read {
      0 inc -> wait;
      0 drop -> wait; }
}

COMPONENT seB_q {ss_q; be_seB_q; wait(qlen = 0, drop = 0);}
```

// Simulation control

```
STOPTIME(1000);
RUN(1);
SEED(370829552);
```

Appendix D

ICE listing of Delta-2 Banyan switch architecture model

```
// File       : asw4a.ice
// Author     : GAC
// Date       : 13.02.96
// Purpose    : Models four layers of a 16i/p Delta-2 Banyan Switch with
//              o/p buffered SEs. Has both i/p and o/p controllers.


// Counter reference values

CONSTANT {EMPTY = 0; FULL = 20;}


// Model of network traffic

STATE_SET ss_ch_in {
   COUNTERS : total;
   STATES {
      quiet :;
      arrive : {total = total +1};
   }
}
BEHAVIOUR be_ch_in {
   1 quiet -> arrive PROB(0.65);
   1 quiet -> quiet PROB(0.35);
   0 arrive -> quiet;
}
COMPONENT ch_in1, ch_in0 {ss_ch_in; be_ch_in; quiet(total = 0);}


// Model of traffic routing through switch

STATE_SET ss_dest {
   COUNTERS :;
   STATES {
      cross :;
      bar :;
   }
}

BEHAVIOUR be_dest_in0 {
   ON_EVENT ANY(sea0.read0, sea1.read0) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}

BEHAVIOUR be_dest_in1 {
   ON_EVENT ANY(sea0.read1, sea1.read1) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}
```

```
BEHAVIOUR be_dest_sea0 {
   ON_EVENT ANY(seb0.read0, seb1.read0) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}

BEHAVIOUR be_dest_sea1 {
   ON_EVENT ANY(seb0.read1, seb1.read1) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}
BEHAVIOUR be_dest_seb0 {
   ON_EVENT ANY(sec0.read0, sec1.read0) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}
BEHAVIOUR be_dest_seb1 {
   ON_EVENT ANY(sec0.read1, sec1.read1) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}
BEHAVIOUR be_dest_sec0 {
   ON_EVENT ANY(sed0.read0, sed1.read0) {
      0 cross -> bar PROB(0.65);
      0 cross -> cross PROB(0.35);
      0 bar -> cross PROB(0.35);
      0 bar -> bar PROB(0.65);
   }
}
BEHAVIOUR be_dest_sec1 {
   ON_EVENT ANY(sed0.read1, sed1.read1) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}

COMPONENT dest_in0 {ss_dest; be_dest_sec0; bar;}
COMPONENT dest_in1 {ss_dest; be_dest_sec1; bar;}
COMPONENT dest_sea0 {ss_dest; be_dest_sec0; bar;}
COMPONENT dest_sea1 {ss_dest; be_dest_sec1; bar;}
COMPONENT dest_seb0 {ss_dest; be_dest_sec0; bar;}
COMPONENT dest_seb1 {ss_dest; be_dest_sec1; bar;}
COMPONENT dest_sec0 {ss_dest; be_dest_sec0; bar;}
COMPONENT dest_sec1 {ss_dest; be_dest_sec1; bar;}
```

// Model of Input Controllers

```
STATE_SET ss_q {
   COUNTERS : qlen, drop, total;
   STATES {
      wait :;
      inc : {qlen = qlen + 1, total = total + 1};
      dec : {qlen = qlen - 1};
      drop : {qlen = qlen - 1, drop = drop + 1};
   }
}
BEHAVIOUR be_qi0 {
   IF_ON ch_in0.arrive {
      0 wait -> inc; }
   IF_ON ANY(sea0.read0, sea1.read0) {
      0 wait -> dec; }
   IF qi0.qlen > FULL {
      0 inc -> drop; }
   ON_EVENT ch_in0.quiet {
      0 inc -> wait; }
   ON_EVENT ALL(!sea0.read0, !sea1.read0) {
      0 dec -> wait;
        0 drop -> wait; }
}
BEHAVIOUR be_qi1 {
   IF_ON ch_in1.arrive {
      0 wait -> inc; }
   IF_ON ANY(sea0.read1, sea1.read1) {
      0 wait -> dec; }
   IF qi1.qlen > FULL {
      0 inc -> drop; }
   ON_EVENT !ch_in1.arrive {
      0 inc -> wait; }
   ON_EVENT ALL(!sea0.read1, !sea1.read1) {
      0 dec -> wait;
      0 drop -> wait; }
}
COMPONENT qi1 {ss_q; be_qi1; wait(qlen = 0, drop = 0);}
COMPONENT qi0 {ss_q; be_qi0; wait(qlen = 0, drop = 0);}
```

// Model of Switching Elements

```
STATE_SET ss_se {
   COUNTERS :;
   STATES {
      wait :;
      read0 :;
      read1 :;
   }
}
BEHAVIOUR be_sea0 {
   IF_ON ALL(!sea1.read0, qi0.qlen > EMPTY, dest_in0.bar) {
      1 wait -> read0; }
   IF_ON ALL(!sea1.read1, qi1.qlen > EMPTY, dest_in1.cross) {
      1 wait -> read1; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT sea0 {ss_se; be_sea0; wait; }
```

```
BEHAVIOUR be_sea0_q {
   IF_ON ANY(sea0.read0, sea0.read1) {
      0 wait -> inc; }
   IF sea0_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON ANY(seb0.read0, seb1.read0) {
      0 wait -> dec; }
   ON_EVENT ALL(!sea0.read0, !sea0.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT ALL(!seb0.read0, !seb1.read0) {
      0 dec -> wait; }
}
COMPONENT sea0_q {ss_q; be_sea0_q; wait(qlen = 0, drop = 0);}
BEHAVIOUR be_sea1 {
   IF_ON ALL(!sea0.read1, qi1.qlen > EMPTY, dest_in1.bar) {
      1 wait -> read1; }
   IF_ON ALL(!sea0.read0, qi0.qlen > EMPTY, dest_in0.cross) {
      1 wait -> read0; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT sea1 {ss_se; be_sea1; wait; }
BEHAVIOUR be_sea1_q {
   IF_ON ANY(sea1.read0, sea1.read1) {
      0 wait -> inc; }
   IF sea1_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON ANY(seb0.read1, seb1.read1) {
      0 wait -> dec; }
   ON_EVENT ALL(!sea1.read0, !sea1.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT ALL(!seb0.read1, !seb1.read1) {
      0 dec -> wait; }
}
COMPONENT sea1_q {ss_q; be_sea1_q; wait(qlen = 0, drop = 0);}
BEHAVIOUR be_seb0 {
   IF_ON ALL(!seb1.read0, sea0_q.qlen > EMPTY, dest_sea0.bar) {
      1 wait -> read0; }
   IF_ON ALL(!seb1.read1, sea1_q.qlen > EMPTY, dest_sea1.cross) {
      1 wait -> read1; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT seb0 {ss_se; be_seb0; wait; }
BEHAVIOUR be_seb0_q {
   IF_ON ANY(seb0.read0, seb0.read1) {
      0 wait -> inc; }
   IF seb0_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON ANY(sec0.read0, sec1.read0) {
      0 wait -> dec; }
   ON_EVENT ALL(!seb0.read0, !seb0.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT ALL(!sec0.read0, !sec1.read0) {
      0 dec -> wait; }
}
COMPONENT seb0_q {ss_q; be_seb0_q; wait(qlen = 0, drop = 0);}
```

```
BEHAVIOUR be_seb1 {
   IF_ON ALL(!seb0.read1, sea1_q.qlen > EMPTY, dest_sea1.bar) {
      1 wait -> read1; }
   IF_ON ALL(!seb0.read0, sea0_q.qlen > EMPTY, dest_sea0.cross) {
      1 wait -> read0; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT seb1 {ss_se; be_seb1; wait; }

BEHAVIOUR be_seb1_q {
   IF_ON ANY(seb1.read0, seb1.read1) {
      0 wait -> inc; }
   IF seb1_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON ANY(sec0.read1, sec1.read1) {
      0 wait -> dec; }
   ON_EVENT ALL(!seb1.read0, !seb1.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT ALL(!sec0.read1, !sec1.read1) {
      0 dec -> wait; }
}
COMPONENT seb1_q {ss_q; be_seb1_q; wait(qlen = 0, drop = 0);}

BEHAVIOUR be_sec0 {
   IF_ON ALL(!sec1.read0, seb0_q.qlen > EMPTY, dest_seb0.bar) {
      1 wait -> read0; }
   IF_ON ALL(!sec1.read1, seb1_q.qlen > EMPTY, dest_seb1.cross) {
      1 wait -> read1; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT sec0 {ss_se; be_sec0; wait; }

BEHAVIOUR be_sec0_q {
   IF_ON ANY(sec0.read0, sec0.read1) {
      0 wait -> inc; }
   IF sec0_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON ANY(sed0.read0, sed1.read0) {
      0 wait -> dec; }
   ON_EVENT ALL(!sec0.read0, !sec0.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT ALL(!sed0.read0, !sed1.read0) {
      0 dec -> wait; }
}
COMPONENT sec0_q {ss_q; be_sec0_q; wait(qlen = 0, drop = 0);}

BEHAVIOUR be_sec1 {
   IF_ON ALL(!sec0.read1, seb1_q.qlen > EMPTY, dest_seb1.bar) {
      1 wait -> read1; }
   IF_ON ALL(!sec0.read0, seb0_q.qlen > EMPTY, dest_seb0.cross) {
      1 wait -> read0; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT sec1 {ss_se; be_sec1; wait; }
```

```
BEHAVIOUR be_sec1_q {
   IF_ON ANY(sec1.read0, sec1.read1) {
      0 wait -> inc; }
   IF sec1_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON ANY(sed0.read1, sed1.read1) {
      0 wait -> dec; }
   ON_EVENT ALL(!sec1.read0, !sec1.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT ALL(!sed0.read1, !sed1.read1) {
      0 dec -> wait; }
}
COMPONENT sec1_q {ss_q; be_sec1_q; wait(qlen = 0, drop = 0);}
BEHAVIOUR be_sed0 {
   IF_ON ALL(!sed1.read0, sec0_q.qlen > EMPTY, dest_sec0.bar) {
      1 wait -> read0; }
   IF_ON ALL(!sed1.read1, sec1_q.qlen > EMPTY, dest_sec1.cross) {
      1 wait -> read1; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT sed0 {ss_se; be_sed0; wait; }
BEHAVIOUR be_sed0_q {
   IF_ON ANY(sed0.read0, sed0.read1) {
      0 wait -> inc; }
   IF sed0_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON qo0.inc {
      0 wait -> dec; }
   ON_EVENT ALL(!sed0.read0, !sed0.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT !qo0.inc {
      0 dec -> wait; }
}
COMPONENT sed0_q {ss_q; be_sed0_q; wait(qlen = 0, drop = 0);}
BEHAVIOUR be_sed1 {
   IF_ON ALL(!sed0.read1, sec1_q.qlen > EMPTY, dest_sec1.bar) {
      1 wait -> read1; }
   IF_ON ALL(!sed0.read0, sec0_q.qlen > EMPTY, dest_sec0.cross) {
      1 wait -> read0; }
   0 read0 -> wait;
   0 read1 -> wait;
}
COMPONENT sed1 {ss_se; be_sed1; wait; }
BEHAVIOUR be_sed1_q {
   IF_ON ANY(sed1.read0, sed1.read1) {
      0 wait -> inc; }
   IF sed1_q.qlen > FULL {
      0 inc -> drop; }
   IF_ON qo1.inc {
      0 wait -> dec; }
   ON_EVENT ALL(!sed1.read0, !sed1.read1) {
      0 inc -> wait;
      0 drop -> wait; }
   ON_EVENT !qo1.inc {
      0 dec -> wait; }
}
COMPONENT sed1_q {ss_q; be_sed1_q; wait(qlen = 0, drop = 0);}
```

// Model of Output Controllers

```
BEHAVIOUR be_qo0 {
        IF_ON sed0_q.qlen > EMPTY {
                1 wait -> inc; }
        0 inc -> wait;
}
BEHAVIOUR be_qo1 {
        IF_ON sed1_q.qlen > EMPTY {
                1 wait -> inc; }
        0 inc -> wait;
}
COMPONENT qo0 {ss_q; be_qo0; wait(qlen = 0, total = 0, drop = 0);}
COMPONENT qo1 {ss_q; be_qo1; wait(qlen = 0, total = 0, drop = 0);}
```

// Simulation Control

```
SEED(21747007);
STOPTIME(10000);
RUN(10);
```

Appendix E

Reprint of published paper

A NOVEL APPROACH TO SIMULATING THE PERFORMANCE OF ATM SWITCHES

G A Corr and A J Miller

Abstract

This paper addresses the problem of the descriptive complexity presented by systems involving a high number of interacting components. The declarative language, ICE, is described and applied to an ATM switch of Banyan architecture. Results of simulating the ICE model are presented and discussed.

1 Introduction

Much work has been done on modelling the performance of complex ATM switch architectures [1,2,3]. Modelling space division architectures such as the popular Banyan network design [4] involves the complex task of describing the interaction of a set of inter-dependant switching elements which may incorporate more than one queuing strategy [5,6,7].

Stochastic Petri-Nets and related techniques have traditionally been used as a generalised approach to modelling concurrent systems [8]. Although there are now tools to support a hierarchial approach [9] to problem specification, the underlying philosophy is to produce a monolithic Markovian model describing the entire system. The resulting model can be of the order of many thousands of states although analytic procedures, with their underlying assumptions [10], are available to make the mathematics tractable.

An alternative approach is to use the formalisms offered by higher level Petri Nets such as coloured [11] and object oriented Petri Nets [12]. These however require to be simulated unless restrictive assumptions are made.

Complexity is a problem not only for computational analysis but for the system description itself. There is a need for a problem specification formalism which supports descriptions proportional to the size of the physical system rather than the overall system state space. This is an approach which addresses the problem of complexity as presented to the human modeller rather than the computing hardware. Whereas computing architecture continues to become increasingly more powerful, we can safely predict that human capability to intellectually grasp the operation of highly interconnected systems will remain relatively limited.

In order to simplify the problem description a novel specification language has been developed, known as the ICE (Interacting ComponEnts) language. Its attractiveness lies in its transparency, inherent simplicity and surprising descriptive power.

ICE is a general purpose language, not linked to any application area. Systems are described as a number of interacting components. Each component may be thought of as a finite state machine with external inputs. The basic concepts and the language structure are easily understood allowing complex models to be constructed within a relatively short time. Once models have been built they may be simulated using the bespoke package I_SIM. Simulations are in discrete time, the duration of time units being determined by the user. It is possible to examine every step of the simulation and to determine a range of statistics on the model's behaviour both at a component and system level.

The ICE language may be used to describe any type of system and some work has been done on the behaviour of communication protocols [13] and flexible manufacturing systems [14]. This paper covers the syntax and semantics of the language and demonstrates its descriptive power by application to the performance modelling of replicated Banyan switches.

The paper describes how a model of a 16 input/output switch can be constructed by representing the output buffered switching elements (SEs) as interacting components. This approach allows access to the buffers

within the SEs at all points during simulation. Our interest is in the study of queue behaviour at the different switch levels as a consequence of varying input loads. Results are presented for the case of both balanced and unbalanced traffic. Statistical analysis is performed on the buffers providing valuable information on relative buffer sizes that may be used in the design of stage dependant SEs.

2.0 ICE Language

2.1 Background

This language was originally conceived as a purely declarative language for modelling reliability problems. The syntax of the language was formally presented in [15]. Subsequent to this, the language compiler and simulator were developed [14]. Experience with the applications resulted in some development of the original syntax although this was adhered to as closely as possible.

The language was later applied to the modelling of both the performance and reliability of communication systems [13]. This proved to be a rigorous testing ground and highlighted further areas of improvement. The language underwent some significant modifications and the simulator was updated to facilitate these. However the description of components with a high number of states, for example queues, still presented a significant challenge. The desire to apply the language to communication networks encouraged a major development of the language and the simulator. The language ICE is the result of subsequent research.

The inherent simplicity of the syntax is one of the major strengths of the language however it has been shown, [13], to be equivalent in descriptive power to timed stochastic Petri Nets with inhibitor arcs.

2.2 Overview

The language has a declarative style that is based upon describing systems in terms of their constituent interacting discrete state components. Each COMPONENT in a system has a set of operational states. The component moves between the various states in its STATE_SET according to its predefined BEHAVIOUR. The transitions can be governed by :

- Time delays.
- Status, ie state occupancy of one or more components.
- Behaviour, ie transition event of one or more components or component counters.

Components may also have an associated AGE which can be used to manipulate their behaviour.

Components may also have COUNTERS associated with them. Counters are used to help address the problem of state explosion. For example, if we wished to model a buffer with 100 spaces, we could do so by using a component with 101 states, ie 1 state for the empty condition and 100 for each of the levels of occupancy. Alternatively, we could use one state to represent the buffer and a counter which may take any value [0,100] to represent the levels of occupancy. This clearly allows the state complexity of models to be greatly reduced.

To fully define a component, three statements are required :

- STATE_SET, which lists the finite set of states that the component can exist in and any counters belonging to the component.
- BEHAVIOUR, which defines all possible transitions that can be made between states.
- COMPONENT, which defines a component with a specified STATE_SET and BEHAVIOUR. It also defines an initial state and optionally an initial age and counter values of the component.

As well as components we can also describe passive resources which may be allocated to components. Resources may be consumable or non-consumable and are specified as STOCK and RESOURCE respectively.

The WAIT_FOR statement allows dynamic creation of components and the explicit manipulation of free stock and resource levels during the simulation.

The language is free format in the sense that blank space (spaces, tabs, new lines etc) are ignored the order of statements is unimportant. A full description of the languages syntax and semantics is given in [28], but an appreciation can be drawn from the ICE listing in Appendix A.

3.0 The Software Simulator

The language is translated using a lexical analyzer and a parser. The lexical analyzer is handwritten whereas the parser used is a DOS version of the UNIX YACC tool. A text file of language statements is converted to data structures . The lexical analyzer and parser check for syntax errors and a successful compilation produces linked lists of C structures, one list corresponding to each type of language statement.

Further software cross-checks the data structures for consistency (eg between STATE_SET and BEHAVIOUR statements). An error free input will produce lists of C++ objects which form the basis of the discrete event simulation.

To provide control over simulation runs, extra statements are required. These include the ability to define simulation runtimes and setting the seed of the random number generator. To gain statistical accuracy it is possible to specify a number of multiple runs of the simulator, each covering a lifetime of system operation.

Post simulation software allows the users to view an event trace of the entire simulation and to generate a range of statistics as to the behaviour of the components and systems. Work is proceeding on producing a facility for presenting statistics in a graphical format.

4.0 Example ; an ICE model of an ATM Delta-2 Banyan Switch

4.1 Overview

The first decision to make is how to subdivide the switch into a suitable combination of components. A suitable balance is required between limiting complexity, which increases with number of components and fully representing the functionality. Four functional units are identified, namely the communications channel, input controllers, switching elements and output controllers. Each is modelled with a component, save for the switching elements which are best represented by four interacting components.

The design takes a modular approach both for simplicity and ease of expansion. In the literature there are models for a great range of sizes, however the intention was to make the ICE model representative of a practical switch. The size decided upon was 16x16, chosen as an optimum size as it provides a high enough number of channels and 16 interface cards fit well into one module mounted in a standard 19" telecommunications equipment rack.

A 16x16 Banyan switch has 4 levels of switching, each with 8 layers of switching elements. The popular assumption made is that all cells arriving from the input channels have output addresses which have an equal probability of being any of the output channels [16]. If we adopt this assumption then it is safe to conclude that we need only model one layer of switching elements and the behaviour of this layer will be representative of any other [17]. To model all layers would only require reiterating the ICE code a further 7 times and editing component names in the behaviour statements. Such expansion would not be practical in probabilistic models as the size of the expressions would become unmanageable.

Thus the ICE model describes all 4 levels of switching elements for one layer of a 16x16 switch. Figure 4.1 is a diagram of the switch with the ICE components marked.

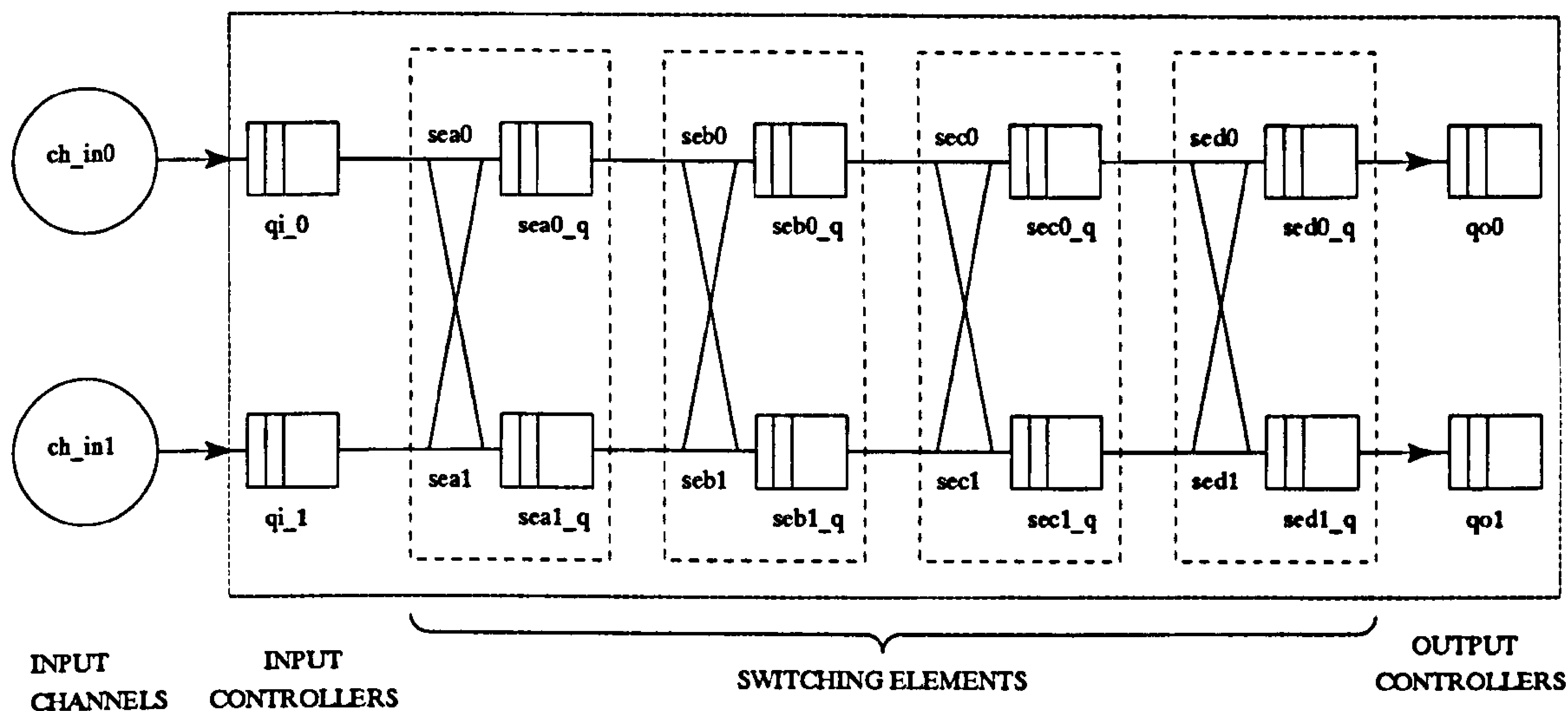


Figure 4.1 One layer of 16x16 switch with components marked

The complete ICE program for the switch model is listed in appendix A. Below we shall consider each type of constituent component.

4.2 The input traffic

The input traffic to the switch is described by modelling input channels that can either be in an *arrive* (cell slot occupied) or *quiet* (cell slot empty) state. It was initially thought that this could be incorporated as part of the behaviour of the input controllers but the requirement that the load be constant prevented this. To expand on this point, to give a true representation of an input channel the model must show a steady flow of cell slots with the probability that any slot is occupied being equal to the required load. If the two states that are described form part of a larger state set with other transitions, this would jeopardise that requirement

In mathematical modelling it is necessary to select some appropriate stochastic distribution that will closely reflect the behaviour of traffic. Uniform cell arrival rates may be represented by either the Poison or Bernoulli distributions. These may be utilised in ICE by manipulation of the exponential transition firing rates. Bursty cell arrivals have been modelled in ATM networks by Interrupted Poison Processes (IPP) [18] and Bulk Bernoulli Processes (BBP) [19]. Complex models of bursty traffic with both exponentially distributed quiet and bursty periods can be modelled using a Markov Modulated Poison Process (MMPP) [18] as implemented in the BONeS simulator [20]. The MMPP can be implemented in ICE by building on the model for uniform cell arrival which is shown below in figure 4.2.

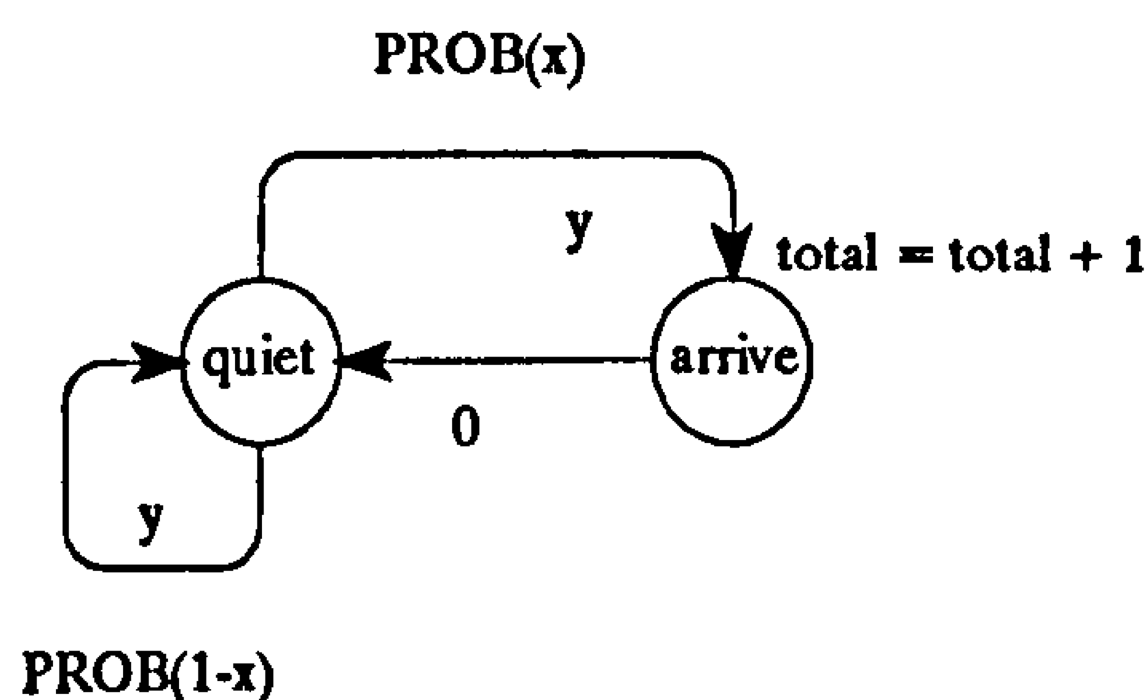


Figure 4.2 State diagram of an input channel

By assigning probabilities to the output transitions from the *quiet* state we can directly represent the channel

in ICE with no level of abstraction. One component is used for each input channel. This component can only exists in the states *quiet* or *arrive* and will move between them with a probability equal to the load as shown by the behaviour statement in listing 4.1. This gives a very simple but very accurate model of the input traffic. The counter *total* which is shown being incremented in the *arrive* state keeps a tally of the number of cells arriving. This is useful for validating loads during simulation.

```
BEHAVIOUR be_ch_in {
        1 quiet -> arrive PROB(0.6);
        1 quiet -> quiet PROB(0.4);
        0 arrive -> quiet;
}
```

Listing 4.1 Input channel BEHAVIOUR statement

4.3 The Input Controllers

The input controllers buffer the cells arriving form the input channels before transmitting them to the first switching elements. Each input controller is modelled as an individual component. The state diagram for is given in figure 4.3.

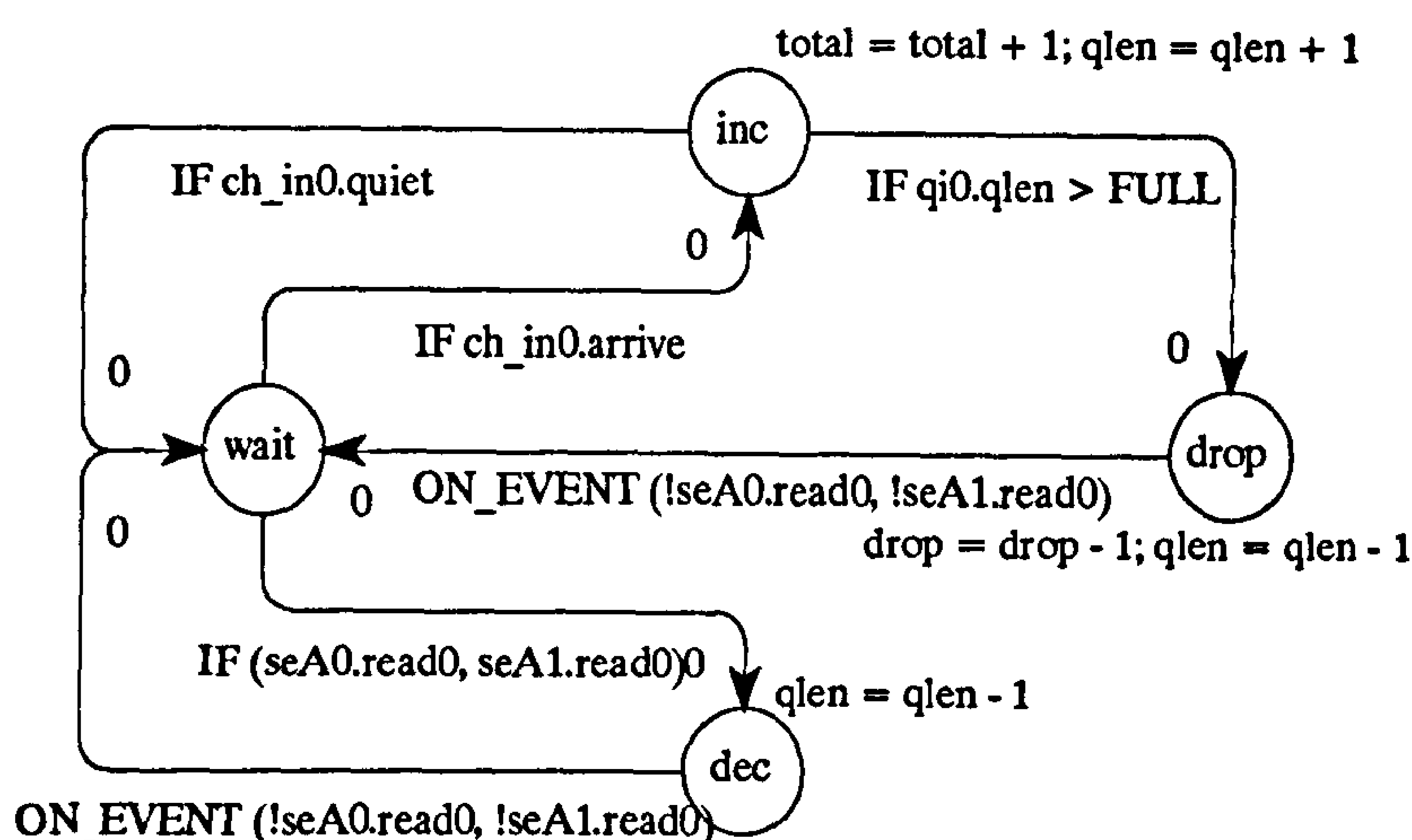


Figure 4.3 State diagram of input controller

There is one buffer per input controller so that all cells that arrive at the same input share the same buffer as in the architecture proposed by Del Re and Fantacci [21]. In ATM there is a priority flag in the header data that facilitates two priorities of traffic. If this model were expanded to have two buffers then behaviour for both priorities could be measured [22]. There are four states. The quiescent state is *wait*. When a cell arrives from the input channel it moves into state *inc* . If the buffer is already full the cell is dropped (state *drop*) otherwise the buffer is incremented and the component returns to *wait*. When a succeeding switching element reads a cell from the buffer it moves into state *dec* and the buffer is decremented by one cell before returning to state *wait*.

Note that all of the transitions are immediate, this is in order to achieve synchronisation. For example, consider the state *inc*. This state is entered when a cell arrives on channel *ch_in0*. Cells arrive in one time unit, this requires the component to move into *inc* and back to *wait* in one time unit and hence this component would move into *wait* at the same time the input channel is moving out of *arrive*. Since these two transitions are happening in the same time unit the order cannot be guaranteed. If the *inc ->wait* occurs first, the input channel will still be in state *arrive* causing this component to re-enter *inc* and falsely record another cell arrival. By putting the transition condition that the component cannot move out of state *wait* until the input

controller moves out of state *arrive* this error is prevented. The corresponding behaviour statement is shown in listing 4.2.

```
BEHAVIOUR be_qi0 {
        IF ch_in0.arrive {
                0 wait -> inc; }
        IF ANY(sea0.read0, sea1.read0) {
                0 wait -> dec; }
        IF qi0.qlen > FULL {
                0 inc -> drop; }
        ON_EVENT ch_in0.quiet {
                0 inc -> wait; }
        ON_EVENT ALL(!sea0.read0, !sea1.read0) {
                0 dec -> wait;
                0 drop -> wait; }
}
```

Listing 4.2 Input controllers BEHAVIOUR statement

The counter *total* stores the total number of cells that have arrived, *qlen* gives the instantaneous length of the queue and *drop* gives the number of cells that have overflown the buffer.

4.4 Operation of the Cross-Bar Switches

From figure 4.1 it can be seen that each switching element contains a cross-bar switch. At each time slot these switches will either be in the *cross* or *bar* state and thus dictate which queue the switching element will be reading from. The operation of these switches is modelled by two components per switch. One component represents the top branch of the switch and one the lower. Each can exist in the two states *cross* or *bar*. The state diagram is given in figure 4.4.

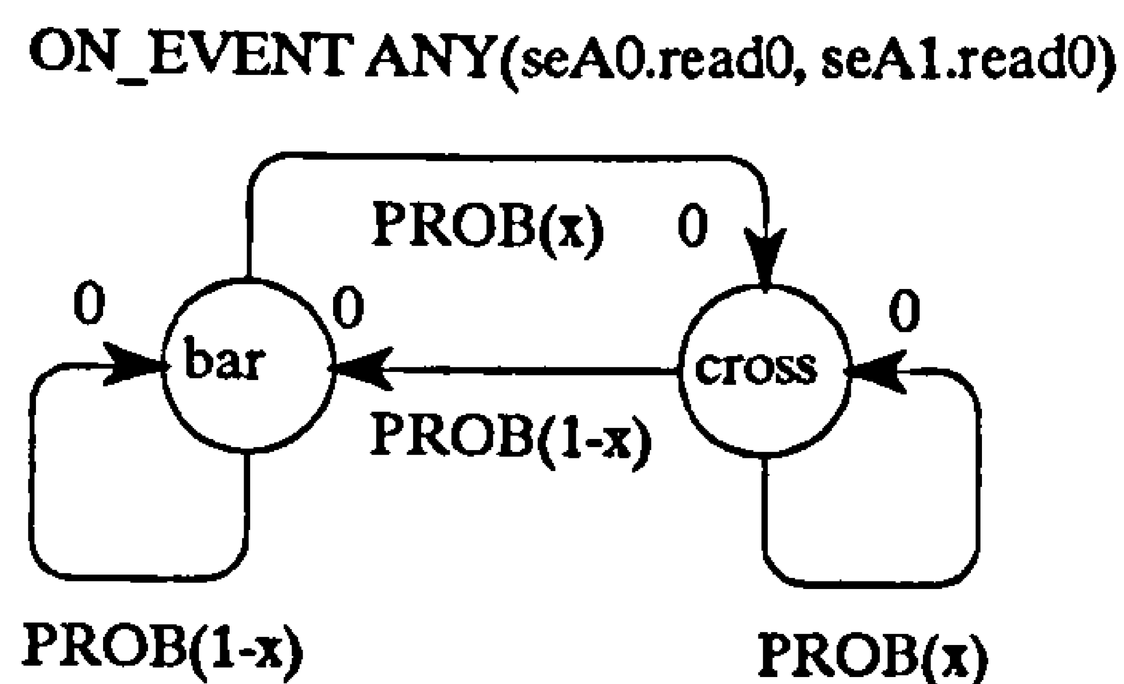


Figure 4.4 State diagram of cross-bar component

For a balanced routing all probabilities will be 0.5. By changing these probabilities the route can be altered. In the listing 4.3 of the BEHAVIOUR statement for an upper branch component the probabilities are set so that there is a bias for the *bar* position. This means there is more traffic arriving at the upper input destined for the upper output than for the lower output. The probabilities for the lower branch are all set to 0.5, hence traffic arriving at the lower input will have equal likelihood of being destined for either output. By adopting this approach, which allows flexibility in the balance of traffic, we can investigate Bruneeli and Wittevongel's [23] finding that queuing deteriorates in output buffered SEs as correlation in the routing gets higher.

```
BEHAVIOUR be_dest_in0 {
   ON_EVENT ANY(sea0.read0, sea1.read0) {
      0 cross -> bar PROB(0.65);
      0 cross -> cross PROB(0.35);
      0 bar -> cross PROB(0.35);
      0 bar -> bar PROB(0.65);
   }
}
BEHAVIOUR be_dest_in1 {
   ON_EVENT ANY(sea0.read1, sea1.read1) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}
```

Listing 4.3 BEHAVIOUR statement for cell routing

Note that the component changes state each time the switching element has read a cell from the proceeding buffer.

4.5 The Switching Elements

In many mathematical models a general expression is derived which expresses the output conditions dependant upon the input and it is not possible to monitor the internal performance of the switch. For many applications this type of method is appropriate as loss probability is a comprehensive enough measure of performance [24]. In this model however we wish to monitor the behaviour of various queues within the interconnection network and switching elements are therefore modelled individually. Each switching element is represented by four components. This seems verbose on first inspection but when examined it allows for simplicity. Two components are required for the two queues. It would be possible to represent the two queues by two counters in one component but by using one counter each in separate components it allows the queues to function in parallel without state transitions being delayed. This reflects the operation of the hardware design.

Initially each queue and its operation was modelled by one component. This is restrictive as checks for read and write operations had to be made sequentially. The final implementation uses two components. The first is used to monitor whether a queue may read from a proceeding queue during each time slot and the second handles the actual updating of the queue. The state diagram of the first component is shown in figure 4.5.

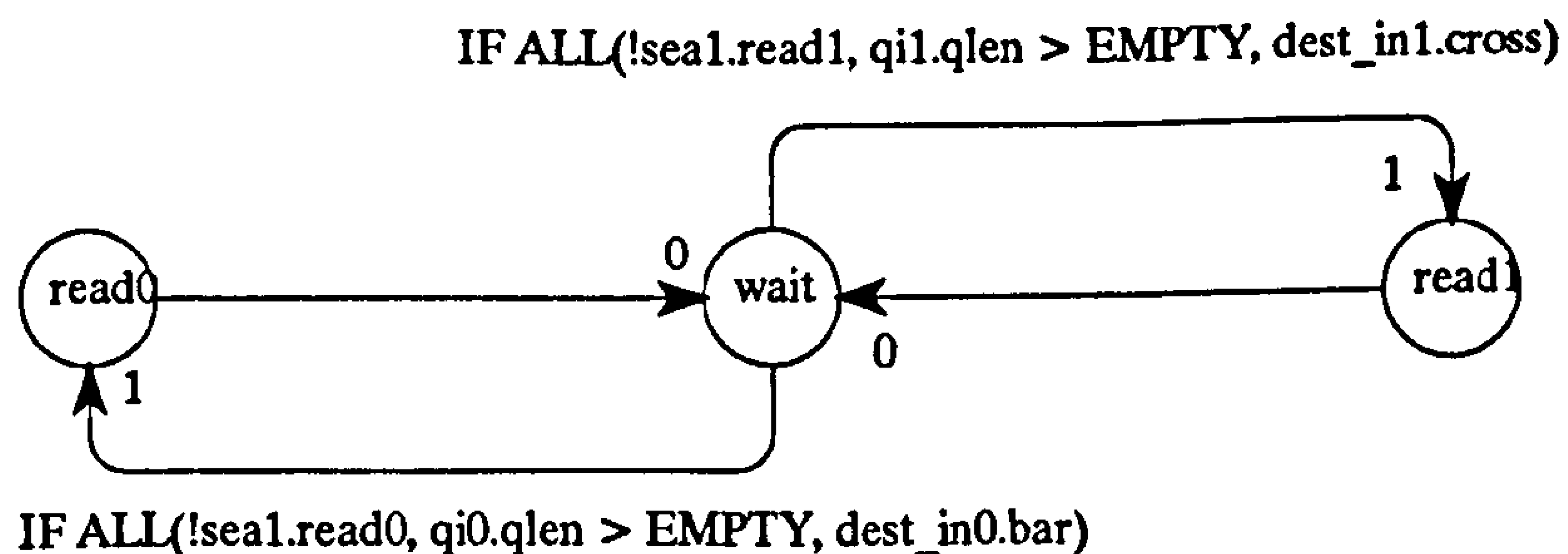


Figure 4.5 State diagram of SE queue reading component

From the state diagram it can be seen that the queue can read from either the proceeding 0 (upper) or 1 (lower) queue. The conditions for reading the proceeding 0 buffer are that the other queue in the SE is not currently

reading from it, the queue is not empty and the cross-bar is in the *bar* position. The conditions for reading from the proceeding lower queue are similar but the cross-bar must be in the *cross* position. The position of the cross-bar is determined by the cross-bar component discussed in section 7.4.4 which selects cell routing. Note that only one cell may be read in one time slot, following the operational procedure proposed by Jenq [17]. The behaviour statement for this component is shown in listing 7.4.

```
BEHAVIOUR be_sea0 {
        IF_ON ALL(!sea1.read0, qi0.qlen > EMPTY, dest_in0.bar) {
                1 wait -> read0; }
        IF_ON ALL(!sea1.read1, qi1.qlen > EMPTY, dest_in1.cross) {
                1 wait -> read1; }
        0 read0 -> wait;
        0 read1 -> wait;
}
```

Listing 4.4   BEHAVIOUR statement for queue reading component

The component will firstly check to see if there is a cell in the upper preceding buffer and if it is destined for the upper queue. If so it will read it, if not it will check the lower buffer. There is no read operation during the time slot if there are no cells available or if the queue is blocked by the complimentary queue reading from the required preceding buffer. By making the SE timeslots faster than the networks timeslots (say a speed-up factor of two) it would be possible for each SE queue to read from the same preceding buffer in the same SE timeslot [25]. The model could be simply changed to encompass this feature by changing the timing on the transitions. Speed-up can also be accomplished at switch level [26] but is limited by the network speed.

The component that models the updating of the queues has the same state diagram as that for the input controllers shown in figure 4.3 and the behaviour is identical. The corresponding behaviour statement is given in listing 4.5.

```
BEHAVIOUR be_sea0_q {
        IF_ON ANY(sea0.read0, sea0.read1) {
                0 wait -> inc; }
        IF sea0_q.qlen > FULL {
                0 inc -> drop; }
        IF_ON ANY(seb0.read0, seb1.read0) {
                0 wait -> dec; }
        ON_EVENT ALL(!sea0.read0, !sea0.read1) {
                0 inc -> wait;
                0 drop -> wait; }
        ON_EVENT ALL(!seb0.read0, !seb1.read0) {
                0 dec -> wait; }
}
```

Listing 4.5   BEHAVIOUR statement for queue updating component.

Note that all the transition timings are again 0. This allows the transitions to be wholly determined by the queue reading components and facilitates the possibility of a cell being read into and read from the same queue within one time slot.

4.6 The Output Controllers

The output controllers present a new challenge within themselves. What is required is a suitable buffer on each output port with consideration of both capacity allocation and overflow. This will be largely dependant upon the network to which the switch is connected [27]. Our aim is to concentrate on the switches behaviour and thus we assume infinite capacity queues in the output controllers. This assumption is equivalent to

assuming that the output network is available to read one cell per time slot. The space diagram for the output controllers is shown in figure 4.6.

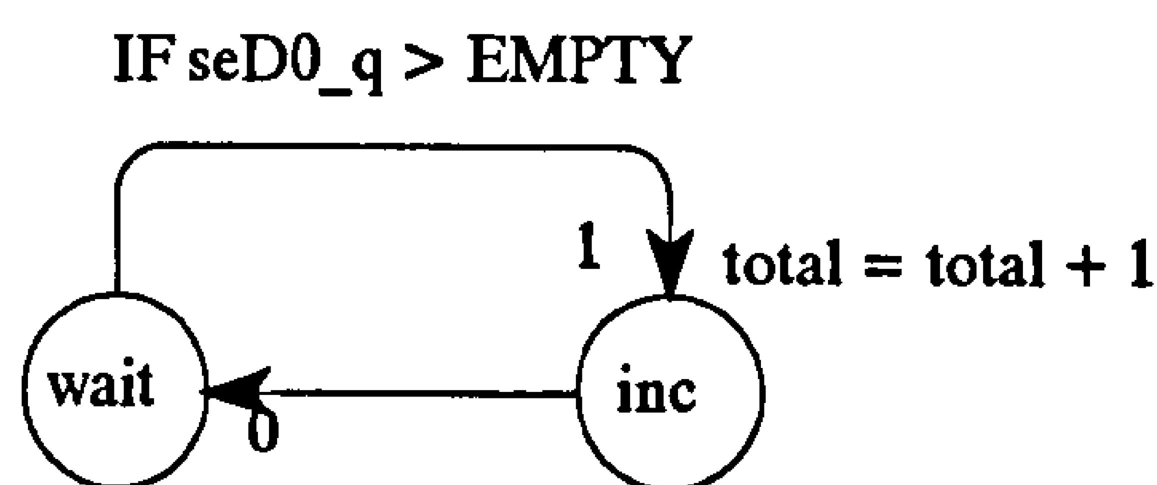


Figure 4.6 Space diagram of Output Controllers

The output controllers will read a cell from the final (4th) level switching elements each time slot if there is a cell to read. There is one output controller dedicated to each output port and hence there will be no blocking in these components. Counters have been associated to each output so that the total number of cells leaving the switch may be monitored. The behaviour statement is given in Listing 4.6.

```
BEHAVIOUR be_qo0 {
        IF_ON sed0_q.qlen > EMPTY {
                1 wait -> inc; }
        0 inc -> wait;
}
```

Listing 4.6 BEHAVIOUR statement of Output Controllers

4.7 Model Validation

With a model of this size and complexity it is necessary to analyse its behaviour to ensure that it reflects correctly the operation of the system being modelled. The post-processor *viz* is a suitable tool for this. Full validation required two steps, the first being to examine a textual event trace of a simulation of the model to ensure components behave as expected, and the second being to run a short simulation examine the resulting counter values.

For the first step a short simulation (100 time slots) was run. The textual event trace for this simulation was obtained using *viz*. Each type of component was considered in turn. Every transition was examined for each component type to ensure the firing and timing corresponded to that which was expected. This step highlighted the timing problems that were discussed in section 4.3 and thus proved a valuable technique.

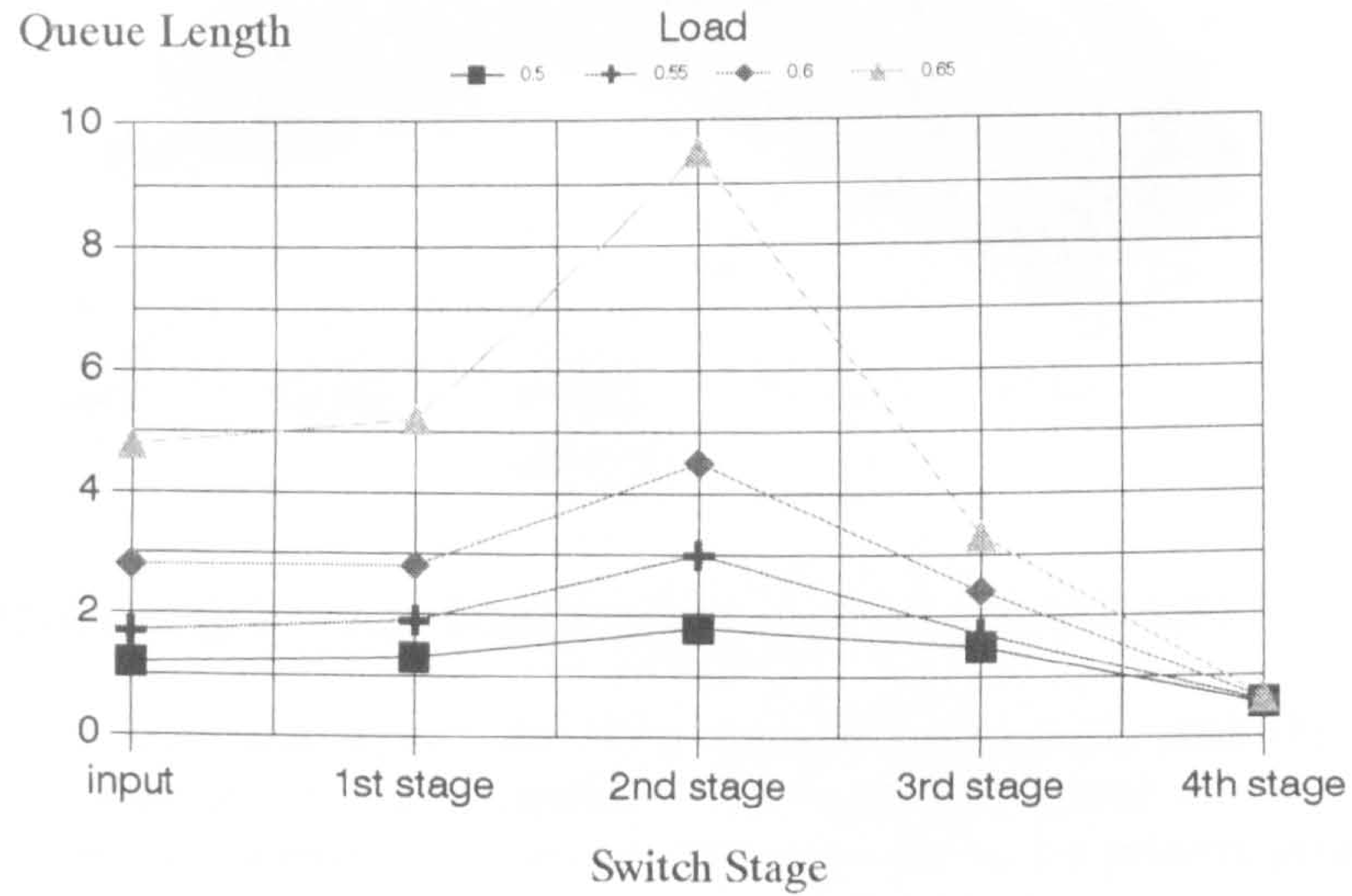
For the second step all the counter values were noted at the end of simulation from the previous event trace. These values are given in table 4.1. The validating technique is as follows. For each pair of components, eg sea0 and sea1, the sum of the *total* values minus the sum of the *qlen* values should be equal to the sum of the *total* values for the following pair of components. By performing this check for each pair of components correct counter operation can be confidently determined.

Cnt. Check	Components											
Counters	qi0	qi1	sea0	sea1	seb0	seb1	sec0	sec1	sed0	sed1	qo1	qo0
total	65	56	57	61	50	59	61	47	50	57	50	56
qlen	2	1	1	8	0	1	1	0	0	1	50	56

Table 4.1 Validation of counter operation

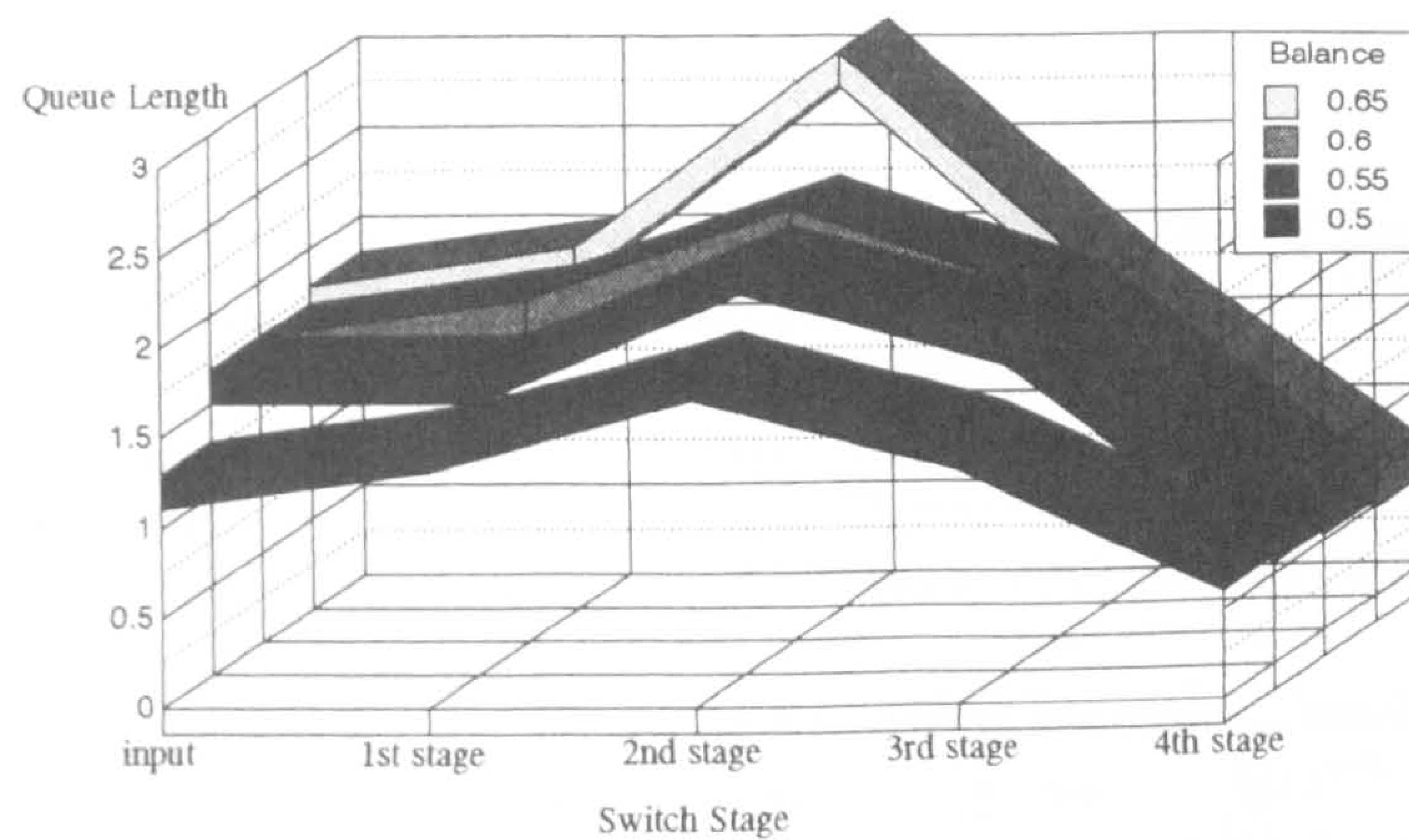
4.8 Simulation Results

The model of the ATM switch was run on the I_SIM software to investigate the mean queue length at each stage dependant upon both the load presented to the switch and the routing balance. For each case the simulator was run for 10 trials of 10000 time slots each. Traffic loading was varied from 0.5 to 0.8 in steps of 0.05. For each of these runs the routing balance was varied from 0.5 (balanced) to 0.65/0.35 in steps of 0.05. A sample of the outcomes are plotted in graphs 4.1-4.3.

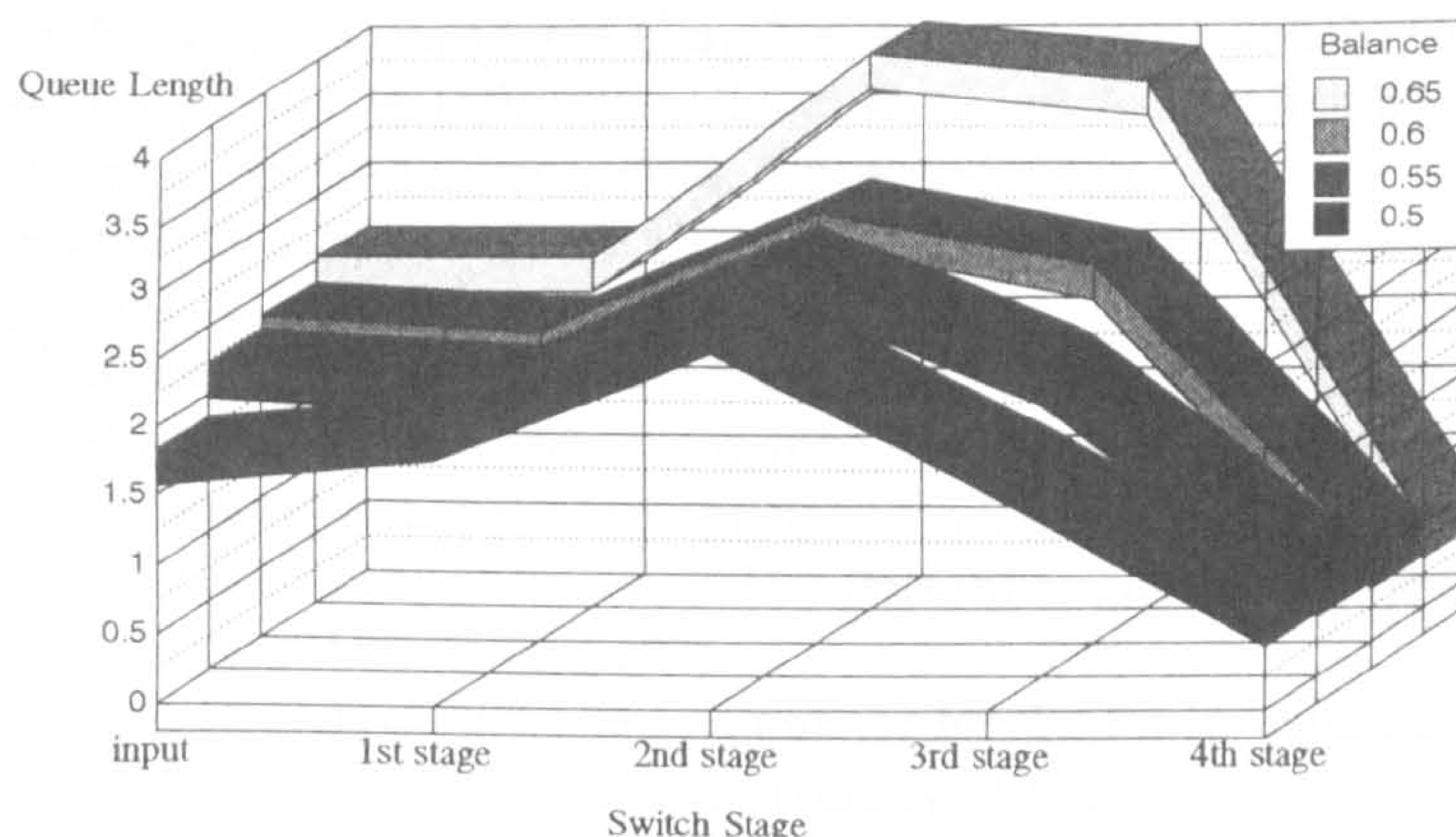


Graph 4.1 Queue length for varied balanced loads

It is interesting to note the behaviour of the second stage queue and that the ratio of this queue length to the others increases with load. As expected all queue lengths increase with load save for the 4th stage which is modelled as feeding an infinite capacity network.



Graph 4.2 Queue lengths for unbalanced upper path for a load of 0.50



Graph 4.3 Queue lengths for unbalanced lower path for a load of 0.50

The upper two graphs again show the importance of the second stage queue, with the lower queue being slightly longer due to the imbalance of traffic routing. The third stage queue on the lower path shows a marked increase over its upper counterpart. This we can assume is due to the priority given to the upper path causing the lower cross-bar, cross path, to be subject to head of line blocking.

5.0 Conclusions

The ATM switch example has demonstrated the ICE language's ability to model complex interacting components with a low level of abstraction and a manageable descriptive state space. The I_SIM software is capable of generating appropriate statistics on each component, state and counter. The generic nature of the language provides flexibility for the modelling of all types of systems. The approach is limited by the lack of static analysis capability which stems from the descriptive power of the language that takes it beyond solution by analytic or numerical techniques.

References

[1] Sun Z, Cosmas J, Cuthbert L G, "Simulation Studies of Multiplexing and Demultiplexing Performance in ATM Switch Fabrics" 10th UK Teletraphic Symposium : Performance engineering in telecommunications networks, pp21/1-5, IEE, April 1993.

[2] Santamaria M L, Puigjaner R; *Banyan ATM switch: grade of service under unbalanced load*, Computer Networks, Architecture and Applications (C-13), Elsevier Science Publishers B.V. pp261-70, 1993.

[3] Tobagi F A; *Fast packet switch architectures for broadband integrated services digital network*, Proc. IEEE, Vol. 78, No. 1, pp133-167, 1990.

[4] Tubtiang A, Kwon H I, Pujolle G : "A method for ATM switches classification", ICCT'92 Proc. of 1992 International Conf. on Communication Technology, Vol. 1, pp12.03.1-5, Sept 1992.

[5] Fan Y, Wang J and Wang C : "Performance Analysis of Banyan Network Based ATM Switches" IEEE International Conf. on Communications, Vol. 3-4, pp, June 1992.

[6] Morris T D, Perros H G : "Performance Modelling of a Multi-Buffered Banyan Switch under Bursty Traffic" In Proc. of Infocom 92, Vol. 1, IEEE, 1992.

[7] Yegani P, "Performance Models for ATM Switching of Mixed Continuous-Bit-Rate and Bursty Traffic with Threshold-Based Discarding" Supercomm 92 Discovering a New World of Communication, Vol.

3, pp354.3.1-7, IEEE, June 1992.

[8] Silva M, "Interleaving Functional and Performance Structural Analysis of Net Models", 14th Conf. on the Application and Theory of Petri Nets, Springer-Verlag, pp16-23, June 1993.

[9] Buchholz P, "Hierarchies in Coloured GSPNs" 14th Conf. on the Application and Theory of Petri Nets, Springer-Verlag, pp106-25, June 1993.

[10] Trivedi K S, Ciardo G, Malhotra M and Garg S : "Dependability and Performability Analysis Using Stochastic Petri Nets", Proc. of 11th International Conf. on Analysis and Optimization of Systems - discrete event systems, pp144-57, June 1994, Springer-Verlag.

[11] Jensen K : "Coloured Petri Nets, volume 1", EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[12] Deng Y, Chang S K, de Figueired J C A and Perkusich A, "Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information Systems", 14th Conf. on the Application and Theory of Petri Nets, Springer-Verlag, pp106-25, June 1993.

[13] Scrase A S, "On RDL and its Application to The Performability of Communication Networks", PhD Thesis, The Robert Gordon University, 1991.

[14] Smith W, "Design and Implementation of a Simulator for the Performance Analysis of Manufacturing Systems", PhD Thesis, The Robert Gordon University 1991.

[15] Miller A J, Wells R and Walker K S, "Simulation Using the Reliability Description Language", Reliability '91, ed R H Matthews, 1991.

[16] Tubtiang A, Kwon H I, Pujolle G; *A Simple ATM Switching Architecture for Broadband-ISDN and its Performance* Modelling and Performance Evaluation of ATM Technology (C-15), Elsevier Science Publishers B.V. (North Holland), 1993, pp361-371.

[17] Jenq Y-C; *Performance Analysis of a Packet Switch Based on Single-Buffered Banyan Network* IEEE Journal on Selected Areas in Communications, Vol. SAC-1 No. 6, December 1983, pp401-8.

[18] Santamaria M L, Puigjaner R; *Analysis of Grade of Service in an ATM Switch* Computer and Information Scences VI, Elsevier Science Publishers BV (NOrth Holland), 1991, pp515-23.

[19] Parr G P, Wright S, Marshall A; *Modelling ATM Switch-Fabric Based on the Knockout Principle* 10th UK TeletrafficSymposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp22/1-22/8.

[20] Kouvatsos D D, Tabet-Aouel N M, Denazis S G; *A Discrete-Time Queuing Model of a Shared Buffer ATM Switch Architecture with Bursty Arrivals* 10th UK TeletrafficSymposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp19/1-19/9.

[21] Del Re E, Fantacci R; *Efficient fast packet switch fabric with shared input buffers* IEE Proceedings-I, Vol. 140, No. 5, October 1993.

[22] Dagiuklas A K, Ghanbari M; *Priority Queuing Disciplines in ATM Switches Carrying Two Layer Video Traffic* 10th UK TeletrafficSymposium : Performance Engineering in Telecommunications Networks, Martlesham Heath, IEE, 14-16 April 1993, pp4/1-4/6.

[23] Bruneel H, Wittenvrongel S; *Analytic performance study of ATM switching elements with on/off sources and correlated routing* Modelling and Performance Evaluation of ATM Technology (C-15), Elsevier Science Publishers B.V. (North Holland), 1993, pp41-59.

[24] Schulzrinne H, Kurose J F, Towsley D F; *Loss Correlation for Queues with Bursty Input Streams* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp308.4.1-6.

[25] Xiong Y, Bruneel H; *Approximate Analytic Performace Study of an ATM Switching Element with Train Arrivals* Supercomm 92, Chicago, June 1992 IEEE, pp354.2.1-7.

[26] Chan D X, Mark J W; *Delay and Loss Control of An Output Buffered Fast Packet Switch Supporting Integrated Services* Supercomm 92, Chicago 14-18 June 1992, IEEE, pp335A.1.1-5.

[27] Chen D X, Mark J W; *A Buffer Managment Scheme for the SCOQ Switch Under Nonuniform Traffic Loading* Infocomm 92, Florence 1992 IEEE, pp1D.4.1-9.

[28] Corr G A, "On ICE and its Application to Performability", PhD Thesis, The Robert Gordon University, 1996.

Appendix A Partial ICE listing of 16x16 Delta-2 Banyan switch model

```
// File    :  asw4a.ice
// Author  :  GAC
// Date    :  31.10.94
// Purpose :  Models four layers of a 16i/p Delta-2 Banyan Switch with
//            o/p buffered SEs. Has both i/p and o/p controllers.

CONSTANT {EMPTY = 0; FULL = 20;}

// Input traffic model

STATE_SET ss_ch_in {
   COUNTERS : total;
   STATES {
      quiet :;
      arrive : {total = total +1};
   }
}
BEHAVIOUR be_ch_in {
   1 quiet -> arrive PROB(0.65);
   1 quiet -> quiet PROB(0.35);
   0 arrive -> quiet;
}
COMPONENT ch_in1, ch_in0 {ss_ch_in; be_ch_in; quiet(total = 0);}

// Traffic routing balance model

STATE_SET ss_dest {
   COUNTERS :;
   STATES {
      cross :;
      bar :;
   }
}
BEHAVIOUR be_dest_in0 {
   ON_EVENT ANY(sea0.read0, sea1.read0) {
      0 cross -> bar PROB(0.5);
      0 cross -> cross PROB(0.5);
      0 bar -> cross PROB(0.5);
      0 bar -> bar PROB(0.5);
   }
}
COMPONENT dest_in0 {ss_dest; be_dest_sec0; bar;}

// Switching element and controllers queue model

STATE_SET ss_q {
   COUNTERS : qlen, drop, total;
   STATES {
      wait :;
      inc : {qlen = qlen + 1, total = total + 1};
      dec : {qlen = qlen - 1};
      drop : {qlen = qlen - 1, drop = drop + 1};
   }
}

// Input controller model

BEHAVIOUR be_qi0 {
   IF_ON ch_in0.arrive {
      0 wait -> inc; }
   IF_ON ANY(sea0.read0, sea1.read0) {
      0 wait -> dec; }
   IF qi0.qlen > FULL {
      0 inc -> drop; }
   ON_EVENT ch_in0.quiet {
      0 inc -> wait; }
   ON_EVENT ALL(!sea0.read0, !sea1.read0) {
      0 dec -> wait;
```

```
            0 drop -> wait; }
}
COMPONENT qi0 {ss_q; be_qi0; wait(qlen = 0, drop = 0);}

// Cross-bar element model

STATE_SET ss_se {
    COUNTERS :;
    STATES {
        wait :;
        read0 :;
        read1 :;
    }
}
BEHAVIOUR be_sea0 {
    IF_ON ALL(!sea1.read0, qi0.qlen > EMPTY, dest_in0.bar) {
        1 wait -> read0; }
    IF_ON ALL(!sea1.read1, qi1.qlen > EMPTY, dest_in1.cross) {
        1 wait -> read1; }
    0 read0 -> wait;
    0 read1 -> wait;
}
COMPONENT sea0 {ss_se; be_sea0; wait; }

BEHAVIOUR be_sea0_q {
    IF_ON ANY(sea0.read0, sea0.read1) {
        0 wait -> inc; }
    IF sea0_q.qlen > FULL {
        0 inc -> drop; }
    IF_ON ANY(seb0.read0, seb1.read0) {
        0 wait -> dec; }
    ON_EVENT ALL(!sea0.read0, !sea0.read1) {
        0 inc -> wait;
        0 drop -> wait; }
    ON_EVENT ALL(!seb0.read0, !seb1.read0) {
        0 dec -> wait; }
}
COMPONENT sea0_q {ss_q; be_sea0_q; wait(qlen = 0, drop = 0);}

// Output controller model

BEHAVIOUR be_qo0 {
            IF_ON sed0_q.qlen > EMPTY {
                        1 wait -> inc; }
            0 inc -> wait;
}
COMPONENT qo0 {ss_q; be_qo0; wait(qlen = 0, total = 0, drop = 0);}

// Simulation Control

SEED(23452);
STOPTIME(10000);
RUN(10);

// Ends
```