Aston University

Some pages of this thesis may have been removed for copyright restrictions.

If you have discovered material in Aston Research Explorer which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our Takedown policy and contact the service immediately (openaccess@aston.ac.uk)

A design environment for deadlock-free concurrent software

Mahmood Ashraf Khan

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

May 1992

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior, written consent.

The University Of Aston in Birmingham

A Design Environment for Deadlock-free Concurrent Software

Mahmood Ashraf Khan

Submitted for the degree of Doctor of Philosophy 1992.

Summary

Using current software engineering technology, the robustness required for safety critical software is not assurable. However, different approaches are possible which can help to assure software robustness to some extent. For achieving high reliability software, methods should be adopted which avoid introducing faults (fault avoidance); then testing should be carried out to identify any faults which persist (error removal). Finally, techniques should be used which allow any undetected faults to be tolerated (fault tolerance).

The verification of correctness in system design specification and performance analysis of the model, are the basic issues in concurrent systems. In this context, modelling distributed concurrent software is one of the most important activities in the software life cycle, and communication analysis is a primary consideration to achieve reliability and safety. By and large fault avoidance requires human analysis which is error prone; by reducing human involvement in the tedious aspect of modelling and analysis of the software it is hoped that fewer faults will persist into its implementation in the real-time environment.

An Occam language is used for the specification of the systems which supports concurrent programming and it is a language where interprocess interaction takes place by communications. This may lead to deadlock due to communication failure. Proper systematic methods must be adopted in the design of concurrent software for distributed computing systems if the communication structure is to be free of pathologies, such as deadlock. The objective of this thesis is to provide a design environment which ensures that processes are free from deadlock.

A software tool was designed and used to facilitate the production of fault-tolerant software for distributed concurrent systems. Where Occam is used as a design language then state space methods, such as Petri-nets, can be used in analysis and simulation to determine the dynamic behaviour of the software, and to identify structures which may be prone to deadlock so that they may be eliminated from the design before the program is ever run. This design software tool consists of two parts. One takes an Occam program and translate it into a mathematical model (Petri-net), which is used for modelling and analysis of the concurrent software. The second part is the Petri-net simulator that takes the translated program as its input and starts simulation to generate the reachability tree. The tree identifies 'deadlock potential' which the user can explore further.

Finally, the software tool has been applied to a number of Occam programs. Two examples were taken to show how the tool works in the early design phase for fault prevention before the program is ever run.

Keywords: Software reliability, Fault tolerance software, Concurrent Software, Occam, Petri-nets, Deadlock, Fault avoidance.

Acknowledgements

I wish to thank my supervisor Dr. Geoffrey F. Carpenter for his guidance, personal encouragement at every stage of this research and in the preparation with the thesis. Without his patience and advice, this research would not be possible. I would also like to thank my research colleagues in the department of Electronic Engineering and Applied Physics at the University of Aston for their stimulating discussions and suggestions.

I am grateful to my parents, brother and rest of the family for the continuous encouragement throughout my stay in U.K. A special thank you is due to my wife, for being patient during my research and helping me in typing of this thesis.

Finally, I would like to thank the Government of Pakistan, Ministry of Science and Technology, and my father for funding me throughout my stay in United Kingdom.

Table of Contents

List of Figures 7

List of Abbreviations 9

Chapter 1 Software Reliability **10**

1.1 Introduction 10

1.2 Research Objectives 13

1.3 Summary of the Thesis 14

Chapter 2 Software Fault Tolerance **17**

2.0 Introduction 17

2.1 Fault Avoidance 18

2.2 Specification and Design of Software For Concurrent Systems 19

2.2.1 C. S. P. 20

2.2.2 Occam 20

2.2.2.1 Primitive Processes 21

2.2.2.2 Occam Constructs 21

2.2.2.3 Occam Mapping to CSP 22

2.2.2.4 Communication in Occam 22

2.3 Techniques for Identifying Design Faults in Software for Concurrent Systems 23

2.3.1 UCLA/Graphical Model of Behaviour(GMB) 23

2.3.2 Petri-nets 25

2.3.3 Comparing Petri-net with UCLA/GMB model 26

2.4 Faults in Software for Sequential Systems 27

2.5 Faults in Software for Concurrent Systems 28

2.6 Fault Tolerance 28

2.7 Software Fault Tolerance in Sequential Processes 29

2.7.1 N-Version Programming 29

2.7.2 Recovery Blocks 30

2.8 Software Fault Tolerance in Concurrent Processes 31

2.8.1 Atomic Action 32

2.8.2 Conversation 32

2.9 Conclusion 33

Chapter 3	**Petri-nets**	**34**
3.1	Introduction	34
3.2	Definition and Properties	34
3.3	Petri-net as a Dynamic Analysis Tool	36
3.4	Analysis Techniques	36
	3.4.1 Reachability Tree	37
	3.4.2 Matrix Equation	39
3.5	Modelling of Concurrent Systems	41
3.6	Example of Petr-net model of am Occam Program	44
3.7	Conclusion	47
Chapter 4	**Automatic Software Analysis Technique**	**48**
4.1	Introduction	48
4.2	Automatic Detection of Occam Deadlocks	50
4.3	Occam Translator	51
	4.3.1 Scanning Phase	53
	4.3.2 Replacement Phase	53
	4.3.3 Semantic Analysis Phase	54
	4.3.4 Petri-net Generation Phase	55
4.4	Petri-net Simulator	57
	4.4.1 Main Menu	58
	4.4.2 Analysis Menu	59
	4.4.3 Tree Menu	59
4.5	Data Structure	59
4.6	Conclusion	62
Chapter 5	**Lex and Yacc: Implementation Issues**	**63**
5.1	Introduction	63
5.2	The Lex Lexical Analyser Generator	63
	5.2.1 Input Specification	64
	5.2.2 Operation of the Generated Lexical Analyser	66
5.3	The Yacc Syntax Analyser Generator	67
	5.3.1 The Input Specification	68
	5.3.1.1 Declaration Section	68
	5.3.1.2 Rules Section	69
	5.3.1.3 User Routines Section	70
	5.3.2 Operation of the Generated Syntax Analyser	70
	5.3.3 Ambiguity in Yacc	72

	5.3.4 Error Recovery in Yacc	75
5.4	The Use of Lex and Yacc	75
	5.4.1 Occam Language Lex Specification	75
	5.4.2 Occam Language Yacc Specification	78
	5.4.3 Mapping	79
	5.4.4 Input and Output of the Implemented Version	79
5.5	Conclusion	80
Chapter 6	**Occam Translator**	**81**
6.1	Introduction	81
6.2	Translation Overview By Example	82
6.3	Op Translation in Detail	86
	6.3.1 Op Translation Sub-routines	86
	6.3.2 Op Translation Rules	89
	6.3.3 Occam Lex Program	104
6.4	Octool Limitations	104
6.5	Conclusion	105
Chapter 7	**Octool Application**	**106**
7.1	Introduction	106
7.2	Example: An Experimental Vehicle Detection System	106
	7.2.1 Overview	107
	7.2.2 Petri-net Conversion	109
	7.2.3 Petri-net Reduction	109
	7.2.4 Reachability Tree	114
	7.2.5 Results	115
7.3	Example: The Conversation Mechanism	119
	7.3.1 Overview	119
	7.3.2 Petri-net conversion	122
	7.3.3 Reachability Tree	124
	7.3.4 Results	124
7.4	Conclusion	127
Chapter 8	**Conclusion and Future Work**	**128**
8.1	Conclusion	128
8.2	Future Work	129
References		**131**
Appendices		**139**

List of Figures

Fig. 2.1	Occam ALT Construct	24
Fig. 2.2	UCLA/GMB graph for Occam ALT construct	25
Fig. 2.3	Transition firing	25
Fig. 2.4	Petri-net graph for ALT construct	26
Fig. 2.5	Transforming UCLA/GMB graphs into Petri-net	27
Fig. 2.6	N-version Programming	30
Fig. 2.7	Syntactic form of recovery block	31
Fig. 2.8	Conversation Scheme	32
Fig. 3.1	Representation of Petri-nets	35
Fig. 3.2	Reachability tree for Fig. 3.1	37
Fig. 3.3	Cyclic Tree showing Terminal node	39
Fig. 3.4	Petri-net representation	40
Fig. 3.5(a)	Petri-net Structure for Sequential Program	41
Fig. 3.5(b)	Petri-net representation of sequential program	42
Fig. 3.6	Petri-net representing PAR (parallelism)	43
Fig. 3.7	Petri-net representing concurrency and deadlock	43
Fig. 3.8	Petri-net for an example	45
Fig. 4.1(a)	Communicating processes representing Non-concurrency	48
Fig. 4.1(b)	Program representing Concurrency without deadlock	49
Fig. 4.2(a)	Program representing crossed channels	49
Fig. 4.2(b)	Corrected program shown in Figure 4.2(a)	50
Fig. 4.3	Overview of dynamic fault detection tool	51
Fig. 4.4	Overall structure of Occam translator	52
Fig. 4.5	Petri-net structure produces by Pre-processor	56
Fig. 4.6	Occam program showing concurrency	60
Fig. 4.7	Data structure of an Occam Program shown in Fig. 4.6	60
Fig. 4.8	Petri-net of an Occam program shown in Fig. 4.6	61
Fig. 5.1	The Lex Lexical analyser generator	66
Fig. 5.2	Cooperation of Lex and Yacc	68
Fig. 5.3	The Yacc syntax analyser generator	71
Fig. 5.4	General Working of the Parser	71

Fig. 5.5	Parse tree for conditional statement	73
Fig. 5.6	Parse tree showing ambiguity grammar	73
Fig. 6.1	Phases of a Compiler	81
Fig. 6.2(a)	PNG for an Occam program	83
Fig. 6.2(b)	WHILE loop	85
Fig. 6.2(c)	PAR construct	85
Fig. 6.2(d)	Output statement	85
Fig. 6.2(e)	Input statement	85
Fig. 6.3(a)	Fragment 1 representation	87
Fig. 6.3(b)	Fragment 2 representation	88
Fig. 6.3(c)	Matrix representation of a program	89
Fig. 6.4	Outer-most places and transitions	90
Fig. 7.1	Vehicle detection system	107
Fig. 7.2	Tree structure of an Occam program	110
Fig. 7.3	PNG showing vehicle detection system	112
Fig. 7.4	PAR reduction example	113
Fig. 7.5	Reduction example for ALT construct	114
Fig. 7.6	Reachability tree for fig. 7.3	116
Fig. 7.7	PNG showing conversation example	121
Fig. 7.8	Reduction mechanism for array channels	123
Fig. 7.9	Reduction for PAR communicating construct	123

List of Abbreviations

BEG	Beginning of Indentation
BNF	Backus Naur Form
CCM	Composite Change Matrix
CHAN	Channel
CSP	Communicating Sequential Processes
END	End of Indentation
GMB	Graphical Model of Behaviour
ID	Identifier
LEX	Lexical Analyser Generator
OCTOOL	Occam Tool (Translator and Simulator)
Op	Occam Program
PNG	Petri-net Graph
PNS	Petri-net Structure
PR	Production Rule
Sc	Syntactic Constructs
sep	End of Line (EOL)
tp	Translation Mapping
YACC	Yet Another Compiler Compiler (Parser)

Chapter 1
Software Reliability - the Scope of the thesis

1.1 Introduction

Rapid advances in microprocessor technology now make it feasible to provide efficient solutions to a wide spectrum of real world problems. In particular, powerful workstations may offer a distributed computing architecture onto which the user may target his software. Compared with the traditional computing environment, distributed computing architectures have several advantages.

- Concurrent execution of multiple processes means the turn-round time of tasks is reduced.

- Higher availability of processors means tasks can be distributed among processors to balance system load.

- Modularity, i.e, the capacity to 'add-on' additional processors and processes means the system has potential for growth.

- More reliable computation since system reconfiguration can be used to tolerate single processor failures.

Characteristics of distributed concurrent software are dynamic, nondeterministic, and undecidable[BAL 88][BURN 89]. Such software represents the system naturally but due to interprocess communication the software is likely to be complex and possess some problems that are discussed later.

The term concurrent and parallel have similar but distinct meanings. Two entities are said to be executing in parallel if at some instant in time both are actually executing. Entities are described as concurrent if they have the potential for executing in parallel. The design of concurrent software becomes highly sophisticated. In particular, fault detection during early phases of software life cycle, such as the design phase, can reduce the testing effort spend in the implementation or later phases.

The verification of correctness in system design specification and performance analysis of the model, are the basic issues in concurrent systems. In this context, modelling distributed concurrent software is one of the most important activities in the software life cycle, and communication analysis is a primary consideration to achieve reliability and safety. These activities should not be divorced from a consideration of the requirements of the system under fault condition. For example, if an application which computes the solution to the scientific problem fails then it may be reasonable to abort the program as only computer time has been lost. However, in the case of real time system this action may not be acceptable and our ultimate interest is the later case.

Before proceeding, definitions of reliability, failure, error and faults are necessary. Randell et al.[RAND 78] and [BURN 89] define the reliability, failure, error and fault of a system as follow:

- reliability is a measure of the success with which the system conforms to some authoritive specification of its behaviour.

- when the behaviour of a system deviates from that which is specified for it, this is called a failure

- failure results from unexpected problems internal to the system and these problems are called errors.

- a fault is the mechanical or algorithmic cause of an error.

Ideally, the specification of a system including its behaviour under fault conditions[BURN 89], should be complete, consistent, comprehensive and unambiguous. All the fault handling techniques used to improve software reliability are known to have strengths and weaknesses. Fault tolerance[RAND 78][ANDE 81], fault elimination and fault avoidance are all complementary techniques which can be successfully applied to achieve high reliability. All these techniques are applied hierarchically. Fault avoidance techniques attempts to limit the introduction of potentially faulty design during the design of the system. In spite of fault avoidance techniques, faults will inevitably be present in the system after its design. The second stage is fault elimination that detect the faults and then those faults can be removed. In spite of the techniques if fault is still present in the system then fault tolerance techniques are implemented. Before one technique can be reduced or

abandoned in favour of another, there need to be at least a convincing demonstration that the second technique would produce reliability improvements. Different techniques have been implemented and given in the literature that provide fault tolerance to the concurrent systems.

In [REYN 79], Reynolds proposed a new approach to verify concurrent programs, where deadlock potential is detected during the interactive construction of a concurrent program, before the program is ever run. The construction of a program is viewed as a sequence of discrete steps. After each step an analysis tool automatically analyses the partially completed program for deadlock potential. This new approach to program verification is called static incremental deadlock detection and is motivated by the desire to simplify the construction of concurrent programs by detecting errors as early as possible in the development cycle.

A model of a concurrent system is a static mathematical representation that captures certain dynamic properties of the system. The main concern is to use a model that can capture synchronisation properties of a concurrent system. Two synchronisation properties that have received attention in the literature are reachability and deadlock potential. The model can be analysed to decide whether the system will exhibit these properties when it is run. Different kinds of models exhibit different modelling capability and allow different properties to be explored[KELL 76].

If a model of a system captures the property of reachability then it can be decided from the model whether the synchronisation constraints of the system allow the system to reach some particular 'state'. The notation of reachability was first identified for vector addition systems in [KARP 69]. Reachability has been studied extensively in the Petri-net literature. Representative work includes [KARP 69], [LAND 78] and [MAYR 84]. If a model of a system captures the property of deadlock potential, then it can be decided from the model whether the system can reach a state where one or more processes can never make further progress. Deadlock potential was first identified for semaphore programs in [DIJK 68].

Many tools [TAYL 83][MEMM 84][SHAT 88][OBER 82][O'HAL 86] are also given in literature that analyse concurrent programs for the detection of deadlock. In [TAYL 83], an algorithm was proposed for the static detection of the concurrent programs where the program is not executing. A simulator was designed for this analysis and Ada was used as a specification language. It was also suggested, that

these results can be applied to other programming languages. Memmi described in [MEMM 84] a tool, called RAFAEL, which is used as an analysis tool for real-time systems. The tool is used to analyse programs that are written in a specification language L. It translates the program fragments into a net called a fifo net which is an extension of Petri-nets. An analyser was designed to analyse those nets and to produce its reachability tree.

Shatz described in [SHAT 88] a Petri-net framework for automated static analysis of Ada tasking behaviour. In the design of this automatic tool, an Ada program is translated into a Petri-net structure, an analyser is then used to generate the reachability tree. After the generation of its reachability tree any possibility of deadlock occurrence can be detected.

To date, no software tool has been described in the literature for Occam programs, that can analyse the complete program for detection of deadlock before the program is ever run. Therefore, this research considers how to provide the Occam language with such a tool. This can be done by translating an Occam program into a Petri-net structure and then analysing the generated Petri-net structure. The Petri-net simulator can be used to generate the reachability tree which can show the possibility of deadlock occurrence.

1.2 Research Objectives

This research was focused on fault avoidance as a basis for producing fault tolerance. Fault avoidance [ANDE 85][BURN 89] was chosen because it is the first step in the process of both fault elimination and fault tolerance. Fault avoidance is a technique for preventing the introduction of faults during software development. It is reported in the literature [ANDE 81] that this technique is imperfect and that fault tolerance techniques should be incorporated in the software. These techniques guarantee the system to work in normal mode of operation. But in real-time environment, if a fault occurs beyond the scope of these techniques it can be fatal and human life could be at risk.

It should be realized that further work can be done to make the fault avoidance technique less imperfect. By and large fault avoidance requires human analysis which is error prone; by reducing human involvement in the tedious aspects of modelling and analysis of software it is hoped that fewer faults will persist into its implementation in a real-time environment.

In concurrent programs, this can be done in two ways, since automated analysis can ensure that interprocess communication is perfect and confirm that the program terminates normally. In other words, it is necessary to determine whether the concurrent program is free from deadlocks.

The Occam language[INMO 89][BURN 88] supports concurrent programming and is a language where interprocess interaction takes place by communications. This may lead to deadlock due to communication failure. These communication failures are due to non-receipt of messages, for example, if a process fails to reach a communication point due to bad program design. Proper systematic methods must be adopted in the design of concurrent software for a distributed computing system if the communication structure is to be free of pathologies, such as deadlock. Therefore, the objective of this thesis is to ensure that processes do not deadlock [DIJK 65] due to communication failure.

To attain the objective of this thesis, a software tool was designed and used to facilitate the production of fault-tolerant software for distributed concurrent systems. Where Occam is used as a design language then state space methods[PETE 78][MURA 89], such as Petri-nets, can be used in analysis[MOLL 82][SUZU 83] and simulation to determine the dynamic behaviour of the software, and to identify structures which may be prone to deadlock so that they may be eliminated from the design before the program is ever run. This design software tool consists of two parts. One takes an input program and translates it into a mathematical model (Petri-net), which is used for modelling and analysis of the concurrent software. The second part is the Petri-net simulator [ARNO 86] that takes the translated program as its input and starts simulation to generate the reachability tree[PETE 81].

1.3 Summary of Thesis

Chapter 2 of the thesis describes how reliable distributed concurrent systems can be designed. Two approaches are discussed that can help the designer to improve the reliability of their systems. The first is fault prevention that attempts to eliminate faults present in the system before it goes operational. The second approach is fault tolerance that enables a system to continue functioning even in the presence of faults. As Occam is used as a specification language, a brief description of the language and its properties are given. Finally, two techniques for identifying the

design faults in concurrent software are given. These techniques UCLA/GMB and Petri-nets are explained and compared.

As Petri-nets are used in the thesis for modelling and analysis, a formal definition and properties of Petri-nets are given in chapter 3. It is also discussed in this chapter the extent to which Petri-nets can be used as a dynamic analysis tool. Finally, it was also shown in the chapter that Petri-nets can be used to model conversation placement. A complex example was taken for this purpose.

Chapter 4 describes the first of two approaches that were used to design an automatic software analysis tool (Octool) for the detection of deadlock. In this approach the Occam translator is divided into four phases: scanning, replacement, semantic analysis and Petri-net generation phases. The semantic analysis phase uses a Petri-net structures table and a keyword table to generate Petri-net structures (PNS). Every keyword sequence (like PAR, SEQ, !, and ?) generates its corresponding PNS which are then combined at the end to represent the complete input (Occam) program. The Petri-net simulator is the second part of this Octool and uses the Petri-net theory to analyse the PNS.

The translator described in chapter 4 suffered from a number of problems which made it unsuitable for large and complex Occam programs. It was also not resilient to the indentations in the programs. An alternative approach to the design of the translator was required. In the second of the two approaches the translator was designed using Lex (Lexical analyser generator) and Yacc (Yet Another Compiler Compiler), the compiler design utilities available under Unix. Before using these utilities it was found necessary to study thoroughly how they could be used to design a compiler. This study is given in chapter 5. The Lex and Yacc specification for Occam programs is also given in this chapter.

Chapter 6 then considers the design of the Occam translator using Lex and Yacc. Here, the complete Occam language was converted into its BNF (Backus Naur Form). Some special features of Occam required modification For example, to separate one process from another uses indentation. The indentation is translated into two tokens, a BEG as a start of an indentation and END as an end of the indentation. Each context-free specification is taken as a rule and its corresponding action is specified. Whenever the rule is recognised its corresponding action takes place. In the rule more than one option can be given and only the option that is

recognised as appropriate will take place. The limitations that are possessed by this translator are also given in this chapter.

The Octool (the translator and simulator) can be used for the real-time applications. Two real-time applications were chosen and were used by Octool to prove that these are deadlock free are explained in chapter 7. The Occam program was first translated and then was analysed by the simulator to produce the reachability tree. The reachability tree for both the Occam programs are given and careful inspection show that these are deadlock free. Chapter 8 summarises the achievement of the research and draws a number of conclusion about these. Also in this chapter a number of areas for further research are suggested.

Chapter No. 2
Software Fault Tolerance

2.0 Introduction

During the past few years, computing systems have become more and more complex and, in many cases, are composed of several, almost independent, units working in an asynchronous parallel mode. This characterisation can be applied to distributed computing systems as well as to centralised systems.

Today one computer may be composed of several CPU's, memories, I/O processors, pieces of software, and each of these units is almost independent. Several computer networks, connecting tens of computers and serving a wide range of users, have been developed and are operating. This increase in the complexity of computing systems will probably continue in the future.[TANE 84][ZAKS 87][BLAC 87].

Many tasks, such as those found in nuclear power and process control applications,[SCHO 84] are inherently distributed because the computer running these processes work independently but are connected through a network. These tasks also have severe real-time constraints. In real-time computing the correctness of the system depends not only on the logical results of the computation but also on the time at which the results are produced. The severe real-time systems are those where it is necessary that the responses occur within the specified deadline.

Real-time computing plays an important role in society, and it covers a wide spectrum from the very simple to the very complex[STAN 88]. Examples of current real-time computing systems include the control of automobile engines, the control of reactions in nuclear power plant, air traffic systems, monitors for patients in intensive care units of hospitals, systems to deal with the complexity of space flight in aerospace programs, and systems to control defence systems. In these examples of real-time computing systems failures could be disastrous[ZORP 85][LEVE 86][LEVE 83]. Thus, it is necessary to organise the system so that if some of its elements are malfunctioning, the rest of the elements will continue to work correctly. In this case, the total performance of the system may be reduced by a failure (and several users can be affected), but one or few elements will not cause the entire system to collapse. Two approaches that can help designers improve the reliability of their system can be implemented[ANDE 81][ANDE 85][RAND 75].

The first is fault prevention; this attempts to eliminate any possible faults present in the system before it goes operational. The second technique is fault tolerance; this enables a system to continue functioning even in the presence of faults.

2.1 Fault Avoidance

Fault avoidance is the set of techniques that help to limit the introduction of potentially faulty components during the construction of the system. These faults may occur in software or/and in hardware. The hardware faults may be limited by three methods[RAND 78][BURN 89]. First, by the use of the most reliable components within the given cost and performance constraints. Second, by the use of well-defined techniques for the inter-connection of components and the assembly of the subsystem. Finally, packaging the hardware to eliminate the possibility of expected forms of interference.

Fault avoidance techniques for software attempt to prevent the introduction of faults into the software during its development. For example, simplifying and regularizing the requirement's description, design and source code should result in fewer faults. The software components of large, distributed systems are nowadays much more complex. It is very difficult in all the cases to write fault-free programs; however, it is assumed that reducing the complexity, or describing the operations to be performed by a set of regular rules, reduces the number of errors introduced during development. The user's expectations of system functionality limit simplification, but use of a controlled development methodology helps to regularize the software design and source code. Techniques associated with fault avoidance can be used during each phase of software development. They start with requirement analysis techniques, which attempts to produce a specification that adequately represents the user's intent. Design techniques, such as Jackson design[JACK 83], are used to transform all specified requirements into a form that can be readily implemented. The quality of software can be improved by precise specification of requirements and the use of proven design methods. Examples of such techniques are: structured programming such as step-wise refinement[WRIT 71], systematic programming, and program verification[ALI 9O]. Other fault avoidance techniques involve formal or informal mathematical verification or synthesis as the design and code is produced[LING 79].

In spite of fault avoidance techniques, faults will still be present in the system after its construction. These faults are due to design faults in both hardware and software components. Here, a fault elimination approach must be implemented. This

approach to software reliability attempts to reduce the incidence of faults in the software by identifying, isolating and removing faults after they have been introduced. The application of fault elimination is a three-step process: fault detection, which determines that some fault is present; fault isolation, which determines what section of software fails under what conditions; and fault removal, which corrects the isolated fault. Some fault elimination techniques, for example, static data flow analysis, are relatively inexpensive to apply, but are suitable for detecting only a few types of fault (for example, uninitialised and unused variables) and are insensitive to all other types of faults. Other techniques, such as formal verification, can potentially detect all types of faults but are too expensive to apply completely.

The work explained in this thesis concentrates on the fault avoidance technique. An automatic technique for fault avoidance and removal which is implemented to concurrent software is explained in chapter 4 (Automatic analysis technique) and chapter 6 (Occam tool using LEX and YACC).

2.2 Specification and Design of Software For Concurrent Systems

Pascal, FORTRAN and COBOL share the common property of being sequential languages. Programs written in these languages have a single thread of control. They start executing in some state and then proceed, by executing one statement at a time, until the program terminates. Concurrent systems can not be fully described using only sequential concepts; additional constructs must be introduced to describe concurrent systems. Dijkstra[DIJK 68] describes a concurrent program as consisting of a collection of autonomous sequential processes, executing in parallel. Although constructs for concurrent programming vary from one language to another there are three fundamental facilities that are present in software for concurrent systems[ANDR 83]. These allow the expression of concurrent execution through the notation of process; process synchronisation; and inter-process communication. Many concurrent languages exist that are facilitated by these constructs.

Here, in this thesis two approaches to concurrent programming are described. These are CSP and Occam. As with Petri-nets, a system represented in CSP can be analysed to determine its behaviour, and the properties of safety and liveness can be examined, which describe the security and operatability of the system respectively (liveness is directly related to deadlock). Occam is further used as a specification

language in the thesis and is discussed in detail in the following sections. A survey of many applicable languages is provided by Bal et al[BAL 88] and [DAVI 87].

2.2.1 C. S. P.

Hoare proposed a novel approach to concurrent programming which he called 'communicating sequential processes' (CSP)[HOAR 78][HOAR 85]. CSP represented the successful attempt to support all three aspects of process interaction.The CSP unifies synchronisation and information exchange. Processes in CSP synchronise and communicate via input and output actions. To transfer a value from process A to process B, process A initiates an output action and provided that process B is ready to execute the corresponding input action, the exchange can take place. However, if the recipient B process is not ready to execute the required input action, then the transmitting process A waits. The complimentary scenario applies should process B be ready to receive first. Mutual exclusion is automatically incorporated in CSP, because a process can only interact with one other process at a time. In CSP the processes are made up of sub-processes, which are in turn made up of further subprocesses. Processes execution and the exact nature of their relation is defined by a set of operators. For example, if two processes R and S are to run sequentially, this would be represented in CSP notation as

R ; S

In turn process R may consist of two sub-processes, K and L which are running in parallel, and this would have the notation

K || L

thus the whole process can also be written as

(K || L) ; S

The basic principles of CSP have influenced the development of the Occam programming language, which is relatively simple to understand and introduces input, output channels (for receiving and sending data/messages) and concurrency as explicit primitives.

2.2.2 Occam

Occam[INMO 88][BURN 88] is a concurrent programming language which enables an application to be expressed in term of concurrent processes which communicate via channels. The basic unit of Occam programming is a "process"

that performs a set of operations and terminates[FLEM 88]. Occam processes are built hierarchically from three primitive processes, that is, assignment, input and output combined using constructs.

In Occam each primitive process and construct keyword occupies a single line; the structure it introduces is most likely to comprise many lines. The components of the constructs are indented.

2.2.2.1 Primitive processes

Process in Occam are built from three 'primitive' processes or actions: assignment, input and output.

v := e where an expression e is assigned to a variable v.

c ! e output the value of expression e from channel named 'c'

c ? v input the value of variable v from channel named 'c'

2.2.2.2 Occam constructs

The primitive processes are combined to form constructs. A construct is itself a process, and may be used as a component of another construct. Conventional sequential programs can be expressed with variables and assignments, combined in sequential, conditional and repetitive constructs. In the sequential (SEQ) construct, the primitives are executed in sequence one after the other. The conditional (IF) construct is followed by a condition. If the condition is true, the primitives encompassed by the construct will be executed. In the repetitive (WHILE) construct, the primitives encompassed by this construct are executed until the condition is false.

Concurrent programs make use of channels, inputs and outputs, combined using parallel and alternative constructs. The PAR construct indicates that the following processes are to be executed independently of each other. For example, if process A and process B are to initiated in parallel then it is written in the program as

```
PAR
  process A
  process B
```

Parallel processes, therefore, run at the same time and, in general, they run asynchronously. Synchronisation is only necessary when processes need to communicate over a channel. The PAR construct terminates only when all

constituent components have terminated. The alternative construct ALT is similar to the conditional construct except that the choice of selected process now depends on the completion of an input process. The alternative construct chooses one of its components for execution. Each component process has a guard which is an input, with an optional condition. The earliest process which is ready to be executed is chosen to the exclusion of the others; the guard (condition) is executed followed by the guarded process. If more than one condition is satisfied the choice as to which alternative is taken is arbitrary. The replicators are also used in Occam with PAR, SEQ and ALT constructs. These are used to describe a collection of similar processes.

2.2.2.3 Occam mapping to CSP

Occam construct have direct mapping to the domain of CSP.

```
SEQ
    L
    M          maps to ( L ; M ; N )
    N

PAR
    L
    M          maps to ( L || M || N )
    N

ALT
    ch1 ? a
          M    maps to ( ch1 ? a -> M □ ch2 ? b -> N )
    ch2 ? b
          N

IF
   X
     M               maps to ( M < X > N )
   NOT X
     N

WHILE X
     M               maps to ( X * M )
```

2.2.2.4 Communication in Occam

Occam[HOAR 85][INMO 88] supports inter-process communication via the use of channels. A channel is a point to point unidirectional software construct which can be mapped directly onto the transputer hardware links. A process may be connected

to another through channel. The syntax for sending a message x through channel c is c ! x and the process that receives the message is statement c ? x.

As the primitive is synchronous when either process encounters one of these statements it will wait for the other process to execute and a deadlock situation may occur due to communication failure. This property of concurrency makes the Occam language both powerful and versatile. Systems can be represented by software in a more natural way since all the real-time software is inherently concurrent.

2.3 Techniques For Identifying Design Faults in Software For Concurrent Systems

Deterministic software for concurrent systems can be modelled and analysed using state space methods. The software design is represented as a network of states and states transition. This abstraction allows the designer to concentrate on potential structural problems, such as those arising from inter-process interactions. The network can be activated to simulate process execution, allowing the designer to resolve many reachability problems and to investigate the dynamic behaviour of the processes. Mathematical analysis can be applied to investigate the topology of the network and therefore the structure of the represented system.

Different techniques are present in the literature for identifying design faults in software for concurrent systems. The techniques that have been applied to model and analyse software for concurrent software are finite state machine[KAWA 71], Petri-net[REIS 85][PETE 81][DUGA 89], UCLA/GMB[CARP 89], temporal logic[SAGO 90] and structure analysis[DEMA 79]. Although the technique that have been used in this thesis for identifying design faults are based on Petri-nets, a detailed explanation of UCLA/GMB is also given and then the two techniques are compared and it is shown how these techniques can be implemented.

2.3.1 UCLA/Graphical Model of Behaviour(GMB)

The UCLA graphic model of behaviour has been developed from the acyclic directed-graph model introduced by Estrin and Martin.[PETE 81]

This model consists of arcs and vertices, where the vertices represent computations and the arcs represents the flow of control between computations. At each vertex there is either OR (+) or AND (*) logic between the input arcs and similarly OR or AND logic between the output arcs. OR input logic means that if any one of the

input arcs is enabled (holds a token) the vertex can execute. OR output logic indicates that when a vertex completes execution it will enable one of the output arcs. AND output logic indicates that when a vertex completes execution it will enable all the output arcs.

A vertex executes by removing a token from its enabling input arc and placing a token on its output arc as indicated by the output logic. If more than one of a set of OR input arcs is enabled, the vertex chooses randomly between the enabled arcs as to which token to remove. Notice that the logic of each arc-node pair is marked on the graph as either + for OR or * for AND logic.[PETE 81][CARP 89]

GMB has been used for modelling a system and allows the designer to investigate the behaviour of a system in three ways[CARP 89]. These domains are:

• the 'control flow' domain, in which the flow of control within and between processes is modelled explicitly. Process synchronisation can be readily incorporated into the model.
• the data domain, in which the flow of data between the processes identified in the control domain is described.
• the interpretation domain, in which the designer can express the effect a transformation would have on the flow of control in a system.

GMB has also been extensively used in the design of software for sequential and concurrent systems.

```
process 1     process 2     process 3
  ...           ...           ...
chan1 ! a     chan2 ! b     ALT
  ...           ...           chan1 ? x
                                ... process guarded by chan1
                              chan2 ? y
                                ... process guarded by chan2
                              ...
```

Fig 2.1 Occam ALT construct

This tool can be used to model Occam constructs. An example of interprocess communication is shown in figure 2.2 by an ALT construct (program given in figure 2.1).

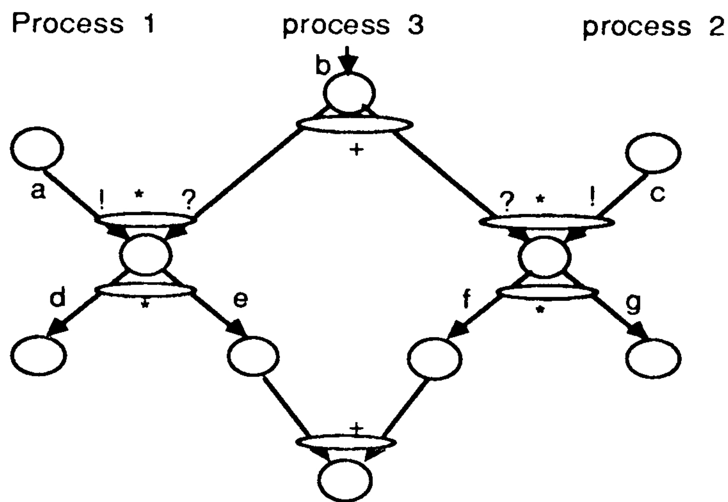


Fig 2.2 UCLA/GMB graph for Occam ALT construct

This construct offers a set of alternative processes, each guarded by a synchronous input or a special SKIP primitive which is always ready for execution.[CARP 88a]

2.3.2 Petri-nets

The use of place-transition nets for modelling and analysing the behaviour of concurrent systems was proposed by C.A. Petri in 1970[REIS 85][PETE 77][BURN 89]. Each place in a net is represented by circle and each transition is represented by a bar '|'. The places and transitions are connected through arcs. A token is used for the execution of the net. The tokens and the places form the state of the modelled system. Rearranging the tokens in the places change the system state. A transition cannot be fired until it has been enabled. It will be enabled when all of the relevant input places have a token.

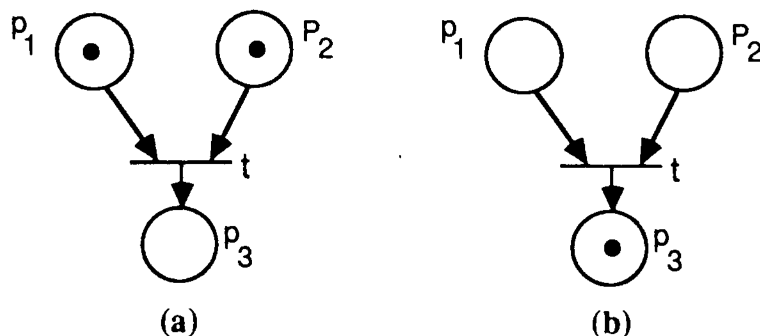


Figure 2.3 Transition firing

Figure 2.3(a) shows a simple Petri-net where transition t is enable and can be fired. Firing a transition will remove all the tokens from the input place(s) and place a

token in all the output places as shown in figure 2.3(b). Thus, by firing a series of transitions it is possible to allow the system to change its state and execute.

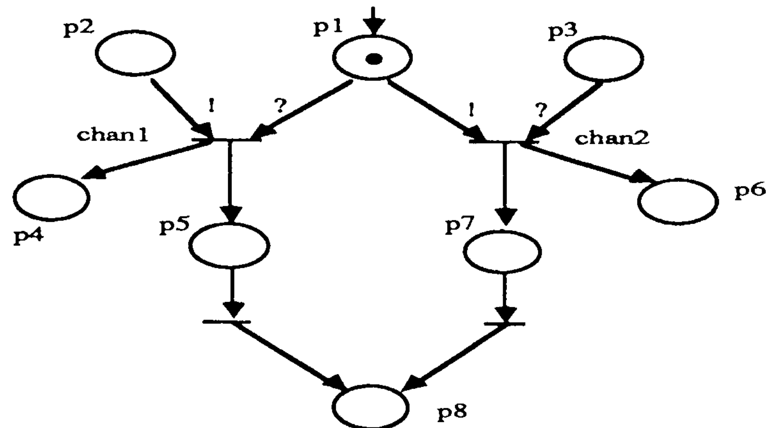


Figure 2.4 Petri-net graph for ALT construct

Petri-nets have been used to model and analyse different Occam constructs[CARP 87], and provide an easy and powerful way of investigating the dynamic properties of Occam constructs which provide interprocess communication and synchronisation. Figure 2.4 shows a Petri-net graph for an Occam ALT construct which is given earlier in figure 2.1.

2.3.3 Comparing Petri-net with UCLA model

A simple correspondence can be established between the Petri-net model and the UCLA model by simply replacing transition bars with vertices (with AND input and AND output logic).

The transformation from a UCLA graph to a Petri-net is also straightforward and every arc in a UCLA graph is represented by a place in a Petri-net. Figure 2.5 shows how the input and output logic for UCLA graphs is represented in an equivalent Petri-net. In a manner similar to the Petri-net model, tokens are used in the UCLA graphic model of behaviour to analyse the flow of control through the graph as the vertices execute.[PETE 81]

The transformation in figure 2.5 shows that any system modelled using these GMB constructs can also be modelled using Petri-nets[PETE 81]. The techniques used to analyse Petri-nets can be used with UCLA/GMB control flow graphs. A reachability tree can be produced and analysed which allows the designer to locate design faults, e.g. deadlock.[CARP 89][CARP 88]

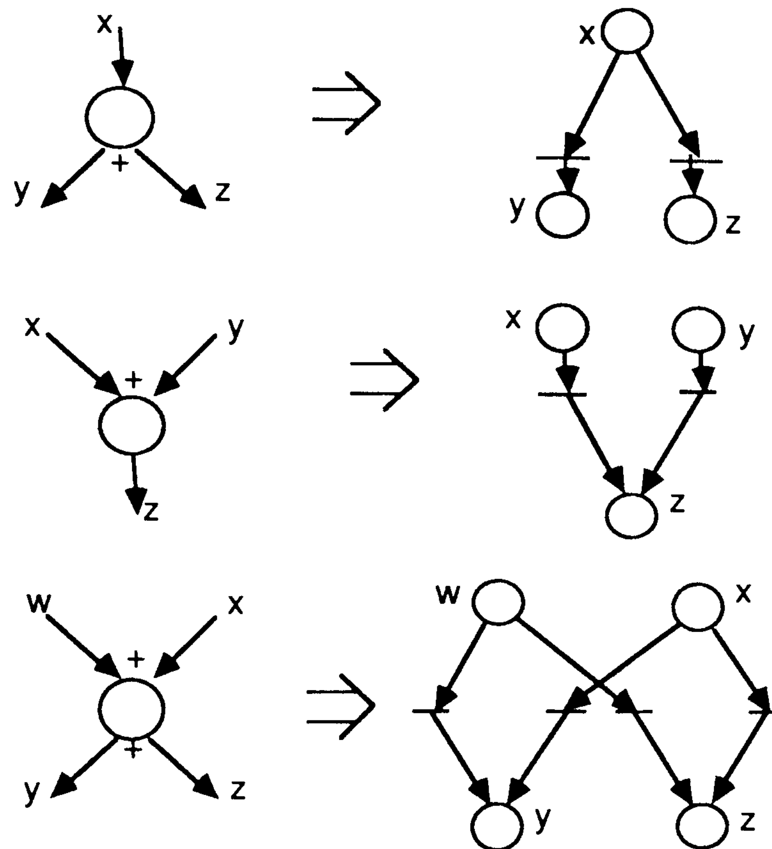


Fig 2.5 Transforming UCLA/GMB graphs into Petri-net

A number of reasons were involved in choosing Petri-nets as a modelling and analysis technique instead of UCLA/ Graphical Model of Behaviour. The main reason for using the Petri-net is that a Petri-net simulator is available; the Occam translator was conceived as a front-end for the simulator, the two tools were giving a complete automatic analysis of Occam programs. Another reason for selecting Petri-nets is that UCLA/GMB is still in the research process while a lot of research has already been done into Petri-net theory. All the work done in this thesis using Petri-net can also be done by using GMB. Extensive work has been done in this field[CARP 89] and it is regarded as motive as Petri-nets.

2.4 Faults in Software For Sequential Systems

Sequential systems are simple to represent and execution is carried out in a sequence in which they are presented. Even with this simplicity, the software for sequential systems may have some faults. Pascal, FORTRAN, COBOL, and 'C' are the common languages used in designing software for sequential systems. All these languages tends to be weak in the facilities they provide for real-time control

and reliability. As a result of these shortcomings it is often necessary to rely on operating system support.

2.5 Faults in Software For Concurrent Systems

Languages which support concurrent processes and interprocess communication allow the designer to write program which incorporate parallelism in a natural way. Inter-process interaction then takes place by inter-process communication. The most common fault in such software is improper inter-process communication, typically leading to a situation of deadlock or incorrect inter-process synchronisation. Different processes run at different times in the concurrent systems, and another issue that also needs attention is process termination. If some process terminates while others are waiting for it to communicate then this situation also leads to the condition of deadlock.

2.6 Fault Tolerance

The fault tolerance approach to software reliability attempts to ensure that the output of a piece of software is correct despite the presence of faults in the software. Fault tolerance [ANDE 81][RAND 75] is the ability of a system to perform according to its specification in the presence of error. The safety of a system can be managed by fault tolerance. The process by which the program can tolerate faults is divided into five steps:

- fault detection
-damage confinement
- damage assessment
- error recovery
- continued operation

The initial point for all fault-tolerance strategies is fault detection. An increase in fault detection results in reliable operation of a system. The second step is damage confinement, which uses static or dynamic features of the program to limit the number of erroneous values in the software states resulting from the faults. After fault detection the damage caused by the fault due to error can be assessed. Damage assessment checks for error propagation throughout the system. It is necessary for the system to adopt strategies for damage assessment in order to try to establish more precisely the extent to which the system state has been damaged. Appropriate

recovery can then be undertaken. Error recovery is the most important technique in the fault tolerance[CAMP 86] transforming an erroneous state of the system into a well-defined and error-free state[ANDE 83][ANDE 81][RAND 75].

Fault detection and error recovery are methods used to eliminate errors, but these errors are merely the symptoms produced by a fault. This fault can cause damage again in the system, if not treated. Therefore, it is necessary to locate the fault, repair the system and, finally, return the system to its normal mode of operation[HECH 79][ANDE 81][RAND 75].

2.7 Software Fault Tolerance In Sequential Processes

Software fault tolerance is of crucial importance in real-time control environments[HECH 76]. Fault tolerance must be based on the provision of useful redundancy. In hardware, this is achieved by duplicating system components and diverse design. The usual method for hardware redundancy is N-modular redundancy[ANDE 81][LALA 85]. In this method, processors or systems are replicated and connected in parallel. The most usual number for N is 3 and is known as triple Modular Redundancy (T.M.R).

In software, however, the redundancy required is not simple replication of program but redundancy of design, because if one faulty module is replaced by another identical module it will produce the error. A general approach for this involves using multiple independently developed and coded "versions" of each module. In the recovery block approach[ANDE 81], the error detection is by means of "acceptance tests", whereas in the N-version programming approach[AVIZ 85], it is via results matching. The recovery block scheme is usually identified with backward recovery.

2.7.1 N-Version Programming

N-version (or multi-version) programming[AVIZ 85][KNIG 91] implies the development of N (where $N \geq 2$) independent versions of the same program (using the same specification). Independent means that the approach to implementation should be different as much as possible and even that different programmers should be used to avoid correlated errors. Replication of segments of the code or the whole program is possible. N-versions of a program are executed in parallel and the required result is obtained by voting on the output of all the units. For example, if

two outputs are the same and one differs, then the correct result is more likely to be same as the two agreeing results. This is known as majority voting. Figure 2.6 illustrate the N-version programming architecture.

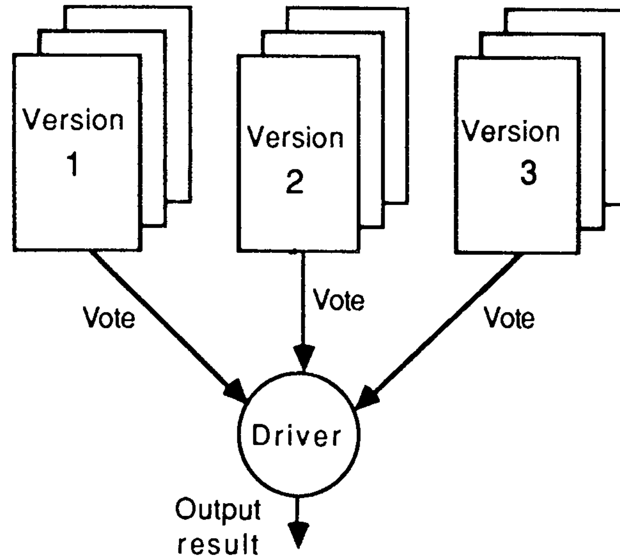


Figure 2.6 N-version Programming

N-version programming is based on the assumptions that the programs can be completely, consistently and unambiguously specified. This assumption may not always be valid. Although N-version programming may have a role in producing reliable software it should be used with care and in conjunction with techniques based on dynamic redundancy[KNIG 86][BRIL 89].

2.7.2 Recovery Block

Recovery blocks[HECH 76][HECH 79] are used in sequential systems to provide backward error recovery. A recovery block consists of a recovery point, an acceptance test, a primary module(block) and a number of secondary modules. Before executing the primary module, the state of computation is saved. This is known as the recovery point. The primary module is then executed and the acceptance test is applied. If the test is passed, the recovery block is exited and the saved recovery point is discarded. Otherwise, the program is rolled back to its recovery point, the saved state is restored and an alternative module is executed. The process continues until the acceptance test become true or the complete computation fails[RAND 75]. The syntactic form[RAND 75] of recovery block is given in figure 2.7

```
Recovery block:     ensure Acceptance test
                    by       Block 1
                    else by  Block 2
                    -------------------
                    else by  Block n
                    else     Error
```

Figure 2.7 Syntactic form of recovery block

This technique does not incur the overheads of N-version programming if faults do not occur but it does suffer from the same design diversity problem. Acceptance test is a critical component in recovery blocks. An acceptance test is a boolean expression involving the initial (i.e. entry time) and final values of the parameter[KANT 87].

Ideally, they should be easy to design, implement and should be complete i.e. only accept outputs that are correct. Complex acceptance tests are less useful than simple acceptance tests because they are more likely to contain design faults[KNIG 91]. Hecht [HECH 79] explains how acceptance tests can be constructed.

2.8 Software Fault Tolerance In Concurrent Processes

In concurrent program structures, a problem can arise in designing error detection and recovery mechanisms where a collection of cooperating asynchronous processes produce the possibility of error propagation through process interaction[KIM 82].

An error in one process can migrate to another process during inter-process communication and lead to an error in the second process. It is therefore necessary to control the error migration and to introduce a co-ordinated recovery in all process involved. Atomic actions[TAYL 86][KANT 85] need to be identified well-known technique called a conversation[RAND 75][KIM 82][TAYL 86] that is been used in fault detection and providing error recovery capabilities to a concurrent software.

2.8.1 Atomic Action

Anderson[ANDE 81] defines atomic action as follow: "The activity of a group of components constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity". Atomic actions provide structuring support for the software of large concurrent systems. This is a method of controlling error propagation by ensuring that the processes within an atomic action do not communicate with processes outside for the duration of the atomic action[ANDE 81][RAND 78][WALK 90].

2.8.2 Conversation

The recovery block scheme for error detection and recovery in sequential process systems cannot be used directly for networking of communicating sequential processes(C.S.P.)[RAND 75]. Arbitrary placement of recovery blocks may lead to the domino effect[RAND 75] where an uncontrolled series of rollback occur (due to error migration by inter-process communication). A useful technique to avoid the domino effect is termed the conversation[RAND 75][KIM 82]. It was developed from the idea of atomic actions and backward error recovery and involves the coordination of recovery for a number of processes. A conversation consists of a recovery line, a test line and two side walls to prevent communication. The recovery line is analogous to the recovery point and consists of a set of recovery points, one for each constituent process and is used during rollback. The test line is a co-ordinated set of acceptance tests for the processes. The acceptance test of a conversation is a set of results produced by the constituent processes, no process is allowed to leave a conversation unless all pass the acceptance test. If any fail then all the processes are rolled back to their recovery point and the conversation is restarted using alternative modules. Figure 2.8 illustrate a typical conversation.

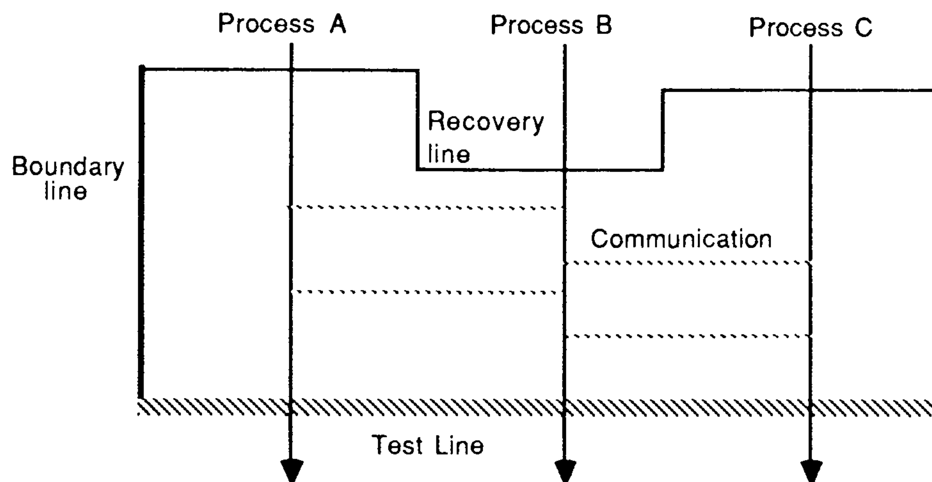


Figure 2.8 Conversation scheme

The implementation of a conversation scheme poses a number of problems. A major problem with implementing a conversation is the identification of atomic actions. Tyrrell and Holding [TYRR 86] proposed a method using process states to solve such problems. This method is simplified by Wu and Fernandez [WU 89] who only consider interprocess communication as being important. This simplified method loses state information and only provides an approximate boundary.

One problem with using such a method is that conversation placement takes place after the system is designed and transformed into a state representation. To overcome the problems associated with conversation Gregery and Knight[GREG 85] have proposed an alternative approach called dialogs and colloquys. The colloquy uses atomic actions and backward error recovery but unlike conversations different processes are allowed after recovery. A colloquy consists of a number of blocks known as dialogs. A dialog indicates the processes that take part in an atomic action and the acceptance test to be used. If a dialog fails its acceptance test then the colloquy can execute another dialog.

2.9 Conclusion

This chapter has built up a number of important ideas that are used in the rest of the thesis. The hierarchical approach to fault tolerance in software is considered and previously implemented techniques for fault tolerance are also given. It has demonstrated that Petri-nets and UCLA/GMB can be used as a notation for state space and space reachability in concurrent systems. In the thesis the main concentration was to elaborate the Petri-net technique which is used later on in the thesis.

Chapter No 3
Petri-nets

3.1 Introduction

Different techniques are used for software reliability. This reliability can be obtained by fault avoidance, fault removal and fault tolerance. All these techniques can be implemented in different stages of software design and implementation. In chapter 2, techniques for identifying design faults in software for concurrent systems are given. These techniques are UCLA/Graphical Model of Behaviour (GMB) and Petri-nets. This chapter investigate the use of Petri-nets in the design of software for distributed systems.

A formal definition, properties and analysis techniques of the Petri-net are given and studied in detail with examples. It is also shown that Petri-nets can be used to model and analyse concurrent software. An Occam program is used to show how Petri-nets can be used successfully in modelling concurrent systems.

Petri nets are capable of modelling a large variety of systems (e.g. computer hardware, computer software, social and biological systems). This modelling technique helps to describe and analyse a structure showing the exact behaviour of the original design.

3.2 Definition and Properties

A Petri-net may be defined[REIS 85][AGER 79] as 6-tuple net, N = (T, P, A, K, W, M), where

T = {t1,t2,t3...tn} is a set of transitions,
P = {p1,p2,p3...pn} is a set of places,
A = (P x T) U (T x P) is the set of directed arcs
K: P → {1, 2, 3,...} is the capacity for each place,
W: A → {1, 2, 3, ...} attaches a weight to each arc of the net,
M: P → {1, 2, 3, ...} is the initial marking,
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

A transition has input and output places representing the preconditions and postcondition of the transition, respectively. If an arc $(p_i, t_j) \in A$, then place pi is an input of transition tj; if $(t_j,p_i) \in A$, then pi is an output of transition tj. In Petri-nets, transitions are equivalent to 'events' and places are equivalent to states for a computer system.

To simulate the dynamic behaviour of a system or the flow of control, tokens are used in a Petri-net. The token flow in a Petri-net models the execution of software. The presence of a certain number of tokens in a place represent the conditions associated with the place. A token distribution over places may be considered as a system state. A marking M0 denotes the initial token distribution of N, called the initial marking of N, where M0(p) denotes the number of tokens in place p at marking M0.

Pictorially, places are represented by circles, transitions by bars and tokens by small dots. For example, figure 3.1 illustrates a Petri-net with five places and five transitions. Place p1 is initially marked with one token so that M0(p1) = 1. The arcs leading into a transition defines its inputs and the arcs leading from a transition defines its outputs.

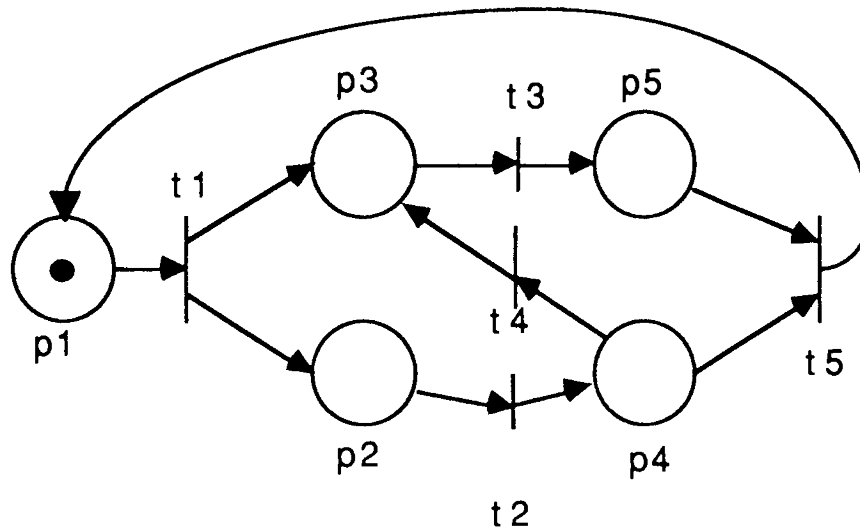


Figure 3.1 Representation of Petri-Nets

A transition t is said to be enabled in a Petri-net with marking M if $M_i > 0$ for all $P_i \in I(t)$ that is if each input place for the transition contains at least one token. When a transition is enabled it will fire only when the corresponding event of the computer system takes place. When an enabled transition fires, it removes a number of tokens from its input places and then depositing into each of its output place one token for each arc from transition to the place. Multiple tokens are produced for multiple output arcs. In figure 3.1, P1 has one token transition t1 is enabled. When t1 fires it removes a token from place P1 and place one token in p2 and p3.

3.3 Petri-net as a dynamic analysis tool

A major strength of Petri-net modelling is its ability to support analysis of many properties associated with concurrent systems[HEIM 78][AGER 78][MURA 89]. The following are the dynamic properties of Petri nets which have been used in analysis of some concurrent systems[JANT 80].

Reachability.

As the enabled transitions of a Petri-net fire, the token distribution (marking) in the net changes accordingly. A marking M_1 is reachable from a marking M_0 in the Petri-net, if there exists a sequence of transition firings that transforms M_0 to M_1.

Boundedness

A Petri-net is bounded if all places in the net are k-safe; i.e. the number of tokens in the places never exceeds k. In this case each place of the Petri-net gets, at most, a finite number k of tokens for every marking reachable from M_0.

Safeness

For a Petri-net that represents a computer system, one of the important properties is safeness. A place in a Petri-net is safe if the number of the tokens in that place never exceeds one; that is, $M(p) \leq 1$ for each place P and every marking M in the net. A Petri-net is safe if all places in the net are safe. By verifying that the Petri-net is safe, it is guaranteed that there will be no increase in number of tokens, regardless of firing sequences chosen.

Liveness

When enough tokens are regenerated in the Petri-net to ensure that each transition in the net always has the potential to fire, the net is said to have the property of liveness. Thus a Petri-net with a live marking guarantees deadlock-free operations, regardless of the firing sequence chosen.

3.4 Analysis techniques

There exists two major techniques for Petri nets analysis: reachability trees and matrix equations. In this section these two analytical techniques have been examined. Both the techniques work differently. In the first method all the reachable markings are generated from the initial given marking and is written as a tree. The second method involves in finding a new marking by Composite Change Matrix (CCM) where two matrices, D+ and D- are used to form CCM. Next follows these two techniques in detail.

3.4.1 Reachability tree

The reachability set is the set of those markings which are potentially reachable from a given initial marking and is represented conventionally as a tree, where the nodes of the tree are markings and the arcs of the tree are transitions which must fire to arrive at the given marking. The tree shows all the markings which could be reached from a given initial marking and the transition sequence which leads to each individual marking. From the formulation of a reachability tree, the behaviour of the Petri-net, and therefore of the modelled system, can be analysed. Exhaustive inspection of the tree can be carried out to determine which transition sequences are live and whether any given marking is reachable.

For the initial marking M_0 in a Petri-net, some transitions may be enabled. Firing each transition enabled in M_0, leads to as many different marking as the number of the enabled transitions. For each new marking resulting from M_0, there may be more enabled transitions. If the above firing of enabled transitions are repeated at each new marking , one can construct a tree representation of marking.

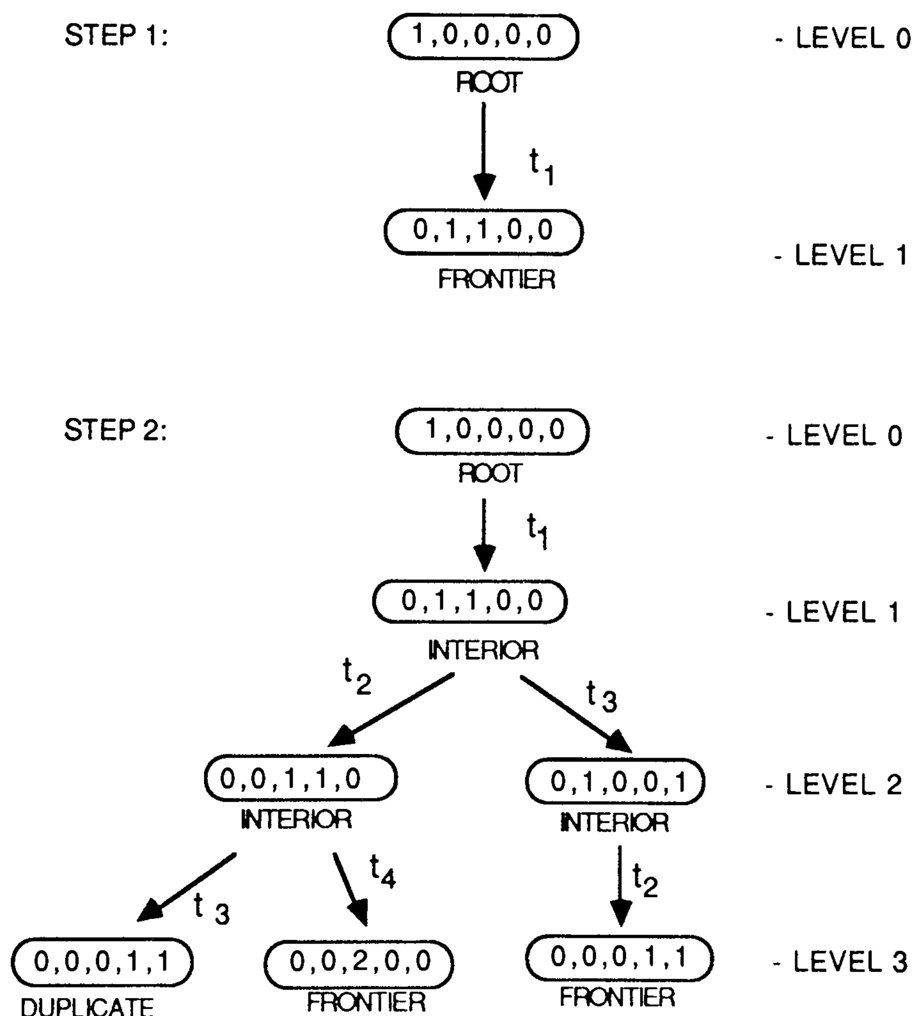


Figure 3.2 Reachability Tree for Fig. 3.1

Here, nodes represent marking generated from M_0 (the root) and its successors, and each arc represents a transition firing showing which marking will result from which as the result of the firing.

Consider, the Petri-net of figure 3.1 with marking (1,0,0,0,0), it was already shown that t1 is enabled and the new marking which results from its firing is (0,1,1,0,0). Further examination shows that t2 and t3 are now enabled. Marking of (0,0,1,1,0) is reached when t2 is fired and marking of (0,1,0,0,1) is reached when t3 is fired. The result of this round of firing is shown in figure 3.2 STEP 2.

Each marking in figure 3.2 is known as a node, and can be classified according to their position within the tree and their contents. Each level of the tree can also be marked to show the depth of attempted firings. Level 0 contains only one node called the ROOT, it is the apex of the tree. The nodes at level one in STEP 1 are classified as FRONTIER, their transition firing has still to be attempted. Before STEP 1 was carried out, the ROOT can also be regarded as a FRONTIER node. Repeating the procedure on the level 1 FRONTIER nodes is shown in figure 3.2 STEP 2. Once the firing have been attempted at a level, the nodes become classified as INTERIOR, except in the case of the ROOT.

At level 3 there are two nodes with the same marking, the tree below each of these will be exactly the same. Only one need to be labeled as FRONTIER, the other could be classified as DUPLICATE and no further firings attempted from that node. Figure 3.3 shows the continuation of the tree, the node with the marking (0,0,0,0,2) at level 5 has no further transitions enabled and is classified as TERMINAL. Therefore, starting from the ROOT node a transition firing sequence of t1, t3, t2, t4, t3 results in a TERMINAL node, this is known as a deadlocking sequence because the TERMINAL node at this point represents a deadlock.

If the above procedure is repeated over and over, every reachable marking will eventually be produced. However, the resulting reachability tree might well be infinite. Every marking in the reachability set will be produced and so for any Petri-net with an infinite reachability set, the corresponding tree would be infinite. The tree represents all possible sequences of transition firing. Every path in the tree corresponds to a legal transition sequence.

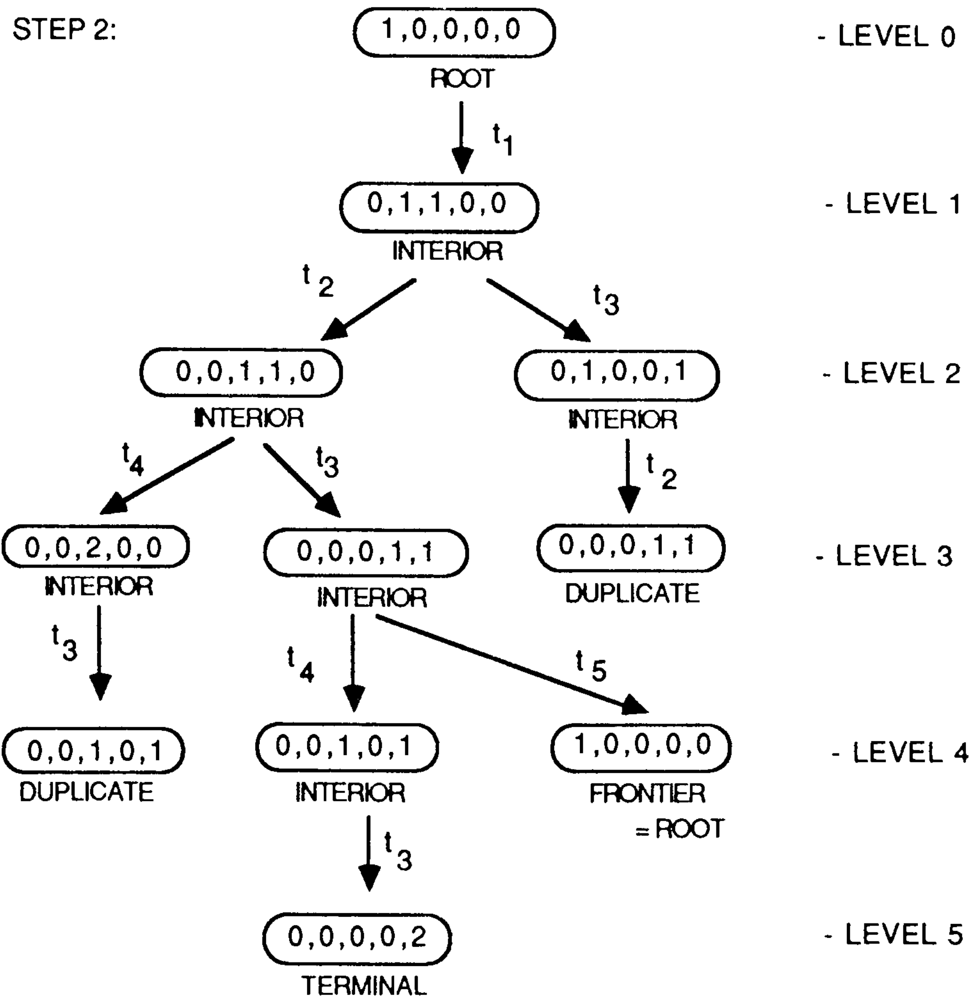


Figure 3.3 Cyclic Tree showing terminal node

Methods exist to limit the tree to a finite size [PETE 81], but this usually involves a loss of marking information.

3.4.2 Matrix equation

An alternative specification of Petri-net [PETE 81] uses a 4-tuple form, C = (T, P, D- , D+). Relating to the previous definition N=(T, P, A, M) here A is represented by D- and D+ matrices. D- and D+, are to represent input and output functions respectively which describe the connectivity between transitions and places. D- has m rows (one for each transition in the Petri-net) and n columns (one for each place) and defines the inputs to each transition; D+ has a similar form and defines the output from a transition. It can be mathematically written as:

$$D^-[j, i] = \#(p_i, I(t_j))$$

$$D^+[j, i] = \#(p_i, O(t_j))$$

This form makes it easy to compute the next state marking, M', from the current marking, M. Consider a transition tj. If ej is a unit vector of length m which is zero

everywhere except in the j'th component, then transition tj is enabled in a marking M if M ≥ ej.D-, and should tj fire, then the new marking is given by:

$$M' = M + ej.(D+ - D-).$$
$$= M + ej.D$$

where D is known as the composite change matrix (CCM).

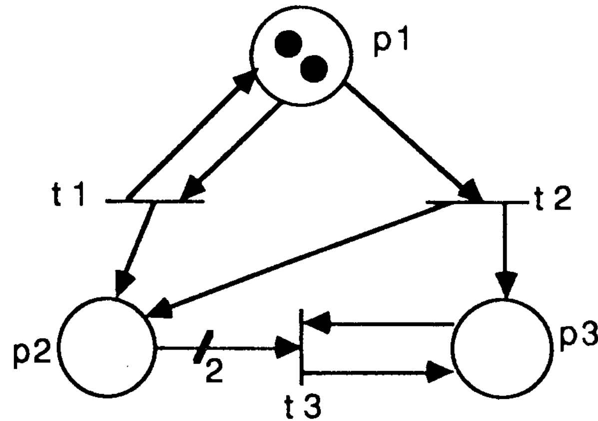


Figure 3.4 Petri-net representation

The initial marking in figure 3.4 is M = (2, 0, 0). After firing transition t1, t2, and t3 the markings are:

(2, 0, 0)
↓ t1
(2, 1, 0)
↓ t2
(1, 2, 1)
↓ t3
(1, 0, 1)

Other firing sequences can be considered. Firing t2, t1, t3 will produce the same result while in other sequences all the transitions cannot be fired successfully. Now for figure 3.4 the input and output matrices are:

$$D^{-} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 2 & 1 \end{bmatrix} \quad \text{and} \quad D+ = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

The CCM is:

$$D = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 1 & 1 \\ 0 & -2 & 0 \end{bmatrix}$$

with the initial marking in figure 3.4, M = (2, 0, 0) and if transition t1 is considered then

$$M' = M + ej . D$$
$$M' = (2, 0, 0) + (1, 0, 0) . \begin{bmatrix} 0 & 1 & 0 \\ -1 & 1 & 1 \\ 0 & -2 & 0 \end{bmatrix}$$

= (2, 0, 0) + (0, 1, 0)
= (2, 1, 0)

The result obtained here is the same as was obtained earlier. Now considering transition t3, the new marking is:

M' = (2, 0, 0) + (0, 0, 1) . D
= (2, 0, 0) + (0, -2, 0)
= (2, -2, 0)

which indicates that t3 is disabled which contradicts the results obtained earlier. Because of the information loss when calculating the CCM, especially in the case of arc loops between adjacent places and transitions this method was found to be insufficient in determining whether a transition was enabled or not. Therefore the matrix equation method was not used for analysis in the Petri-net simulator. The Petri-net simulator uses the reachability tree method for the analysis and is discussed later.

3.5 Modelling of concurrent systems

A wide variety of systems have been modelled by Petri nets. Figure 3.5(a) shows different fragments of a sequential program and its Petri-net representation . This figure shows how easily the structured programming constructs such as IF-THEN-ELSE and WHILE loop can be modelled by the Petri-net.

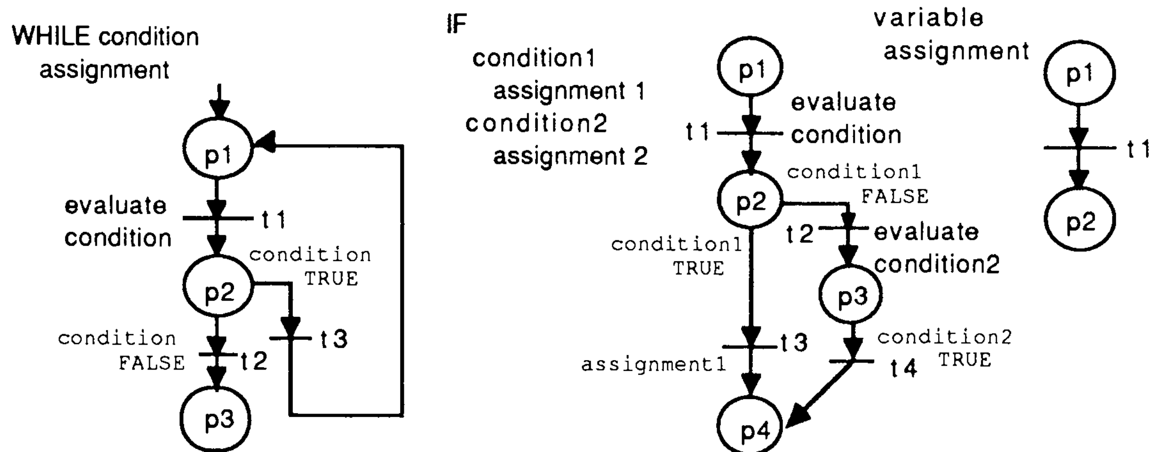


Figure 3.5(a) Petri-net structure for Sequential program

In the WHILE loop transition t1 evaluate the condition; if the condition is TRUE t2 is executed and if it is FALSE t3 is executed. Similarly, in IF-THEN-ELSE figure the condition is evaluated at t1 and either condition1 or condition is executed where condition1 is an inverse of condition2.

As the program fragments can be combined to make a complete sequential program, the Petri-net representations shown in figure 3.5(a) can also be combined to represent the Petri-net for a complete sequential program. This complete Petri-net representation is shown in figure 3.5(b).

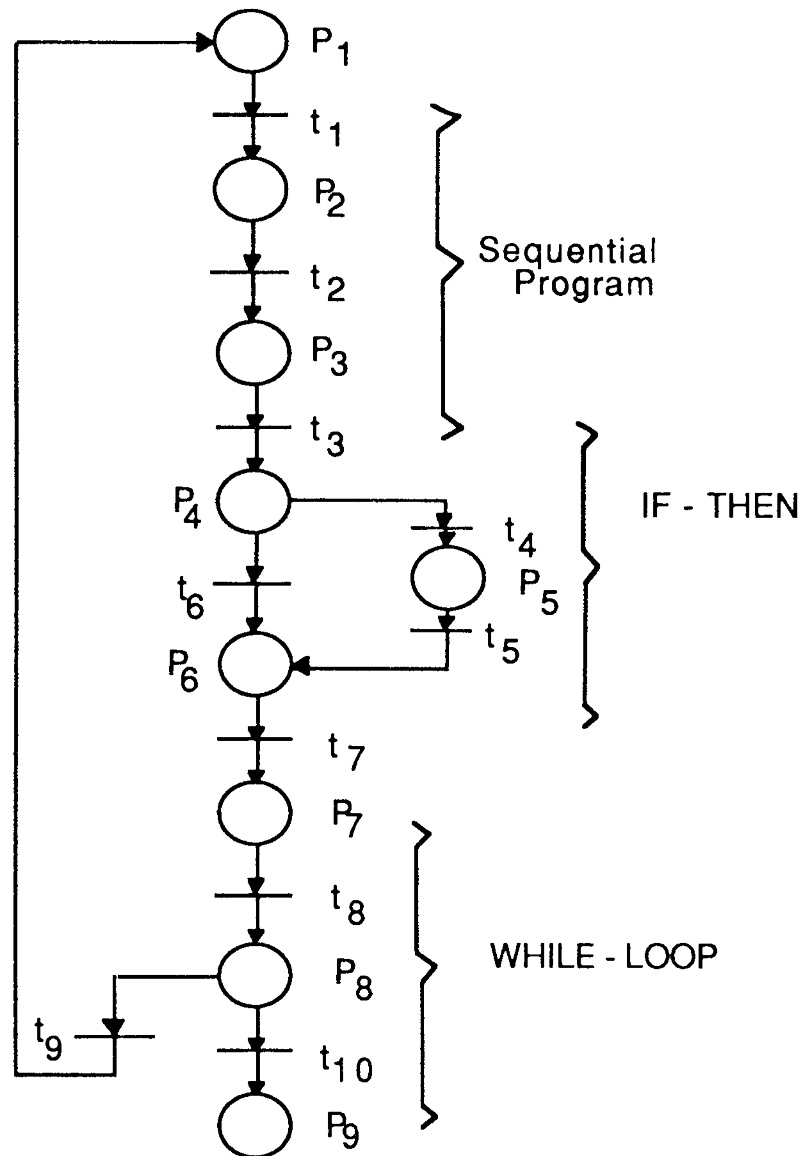


Figure 3.5(b) Petri-net representation of sequential program

Each transition in the figure 3.5(b) has exactly one incoming and one outgoing arc. The subclass of Petri nets with this property is known as a state machine. All sequential programs or their flow of control can be modelled by state machine[MURA 89].

The transition t4 and t6 in 3.5(b) are said to be in conflict, since when place p4 has a token, either t4 or t6 can fire but not both. A conflict is a fundamental concept of Petri-nets and represents a decision from the viewpoint of modelling. A state machine can represent decisions but not concurrency[MURA 83]. The concurrency execution constructs such as PAR can be modelled by the Petri-net shown in figure

3.6. The two transitions t2 and t3 are said to be concurrent since they are causally independent, that is, one transition may fire before or after or in parallel with the other.

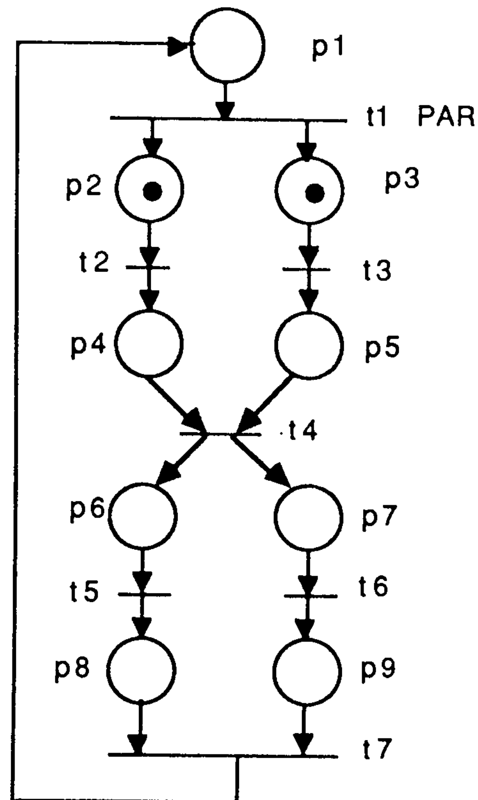


Figure 3.6 Petri-net representing PAR (concurrency)

In figure 3.1, firing transition t1, removes a token from place p1 and puts a token on places p2 and p3, one on each. At this stage, t2 and t3 are both enabled and can fire concurrently since they do not share any input place. After t2 and t3 fires, places p4 and p5 each contain one token.

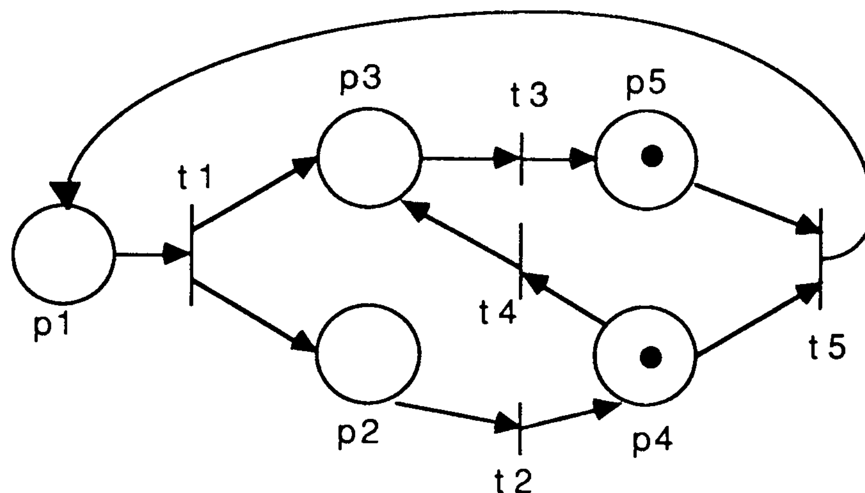


Figure 3.7 Petri-net representing concurrency

Both t4 and t5 are enabled, but the firing of either disables the other. This situation is shown in figure 3.7. If t5 is fired then the net is live but if t4 fires then only t3 is enabled and by firing t3 the net deadlocks. In such a case, the decision as to which one fires is completely arbitrary. This ability to represent both concurrency and deadlock makes Petri-nets very powerful.

3.6 Example of a Petri-net model of an Occam Program

Consider the example of an experimental vehicle control system suitable for use on railways[CARP 89]. The software involves monitoring vehicle movement to and from a protected section of track. A trackside-mounted subsystem is required to monitor access to a restricted section of track, termed the 'protected zone'. There is a single entry-and-exit point to that zone, through which all traffic must pass. The subsystem is required to compute the number of wheels in the zone, to obtain a measure of speed at which the vehicle enters or leaves the zone, and to pass this information immediately to a remote controller. Two sensors are used for speed determination.

To make the example simple and more understandable the speed process is not considered. The five processes that communicate with each other are:

- two processes one for each sensor(process 'sensor') operating in parallel
- a process that counts the number of wheels (process 'counter')
- a process which simulates the action of the remote controller (process 'remotecontroller')
- a process that receive the data messages (process 'remotereceiver')
- a process that handles input/output processes (process 'io')

This vehicle control system can be translated into a Petri-net. The Occam program code for this example is given in appendix C. The complete Petri-net for this example is shown in figure 3.8 and is partitioned into five functional processes which correspond to the actual processes for the vehicle control system.

The functional process boundaries (sensor, counter, remotecontroller, remotereceiver and io) associated with the distributed system can be mapped onto Petri-net models of these systems.

The transition and state transition of the Petri-net can therefore be associated with specific processes and assigned a process-identifier attribute:

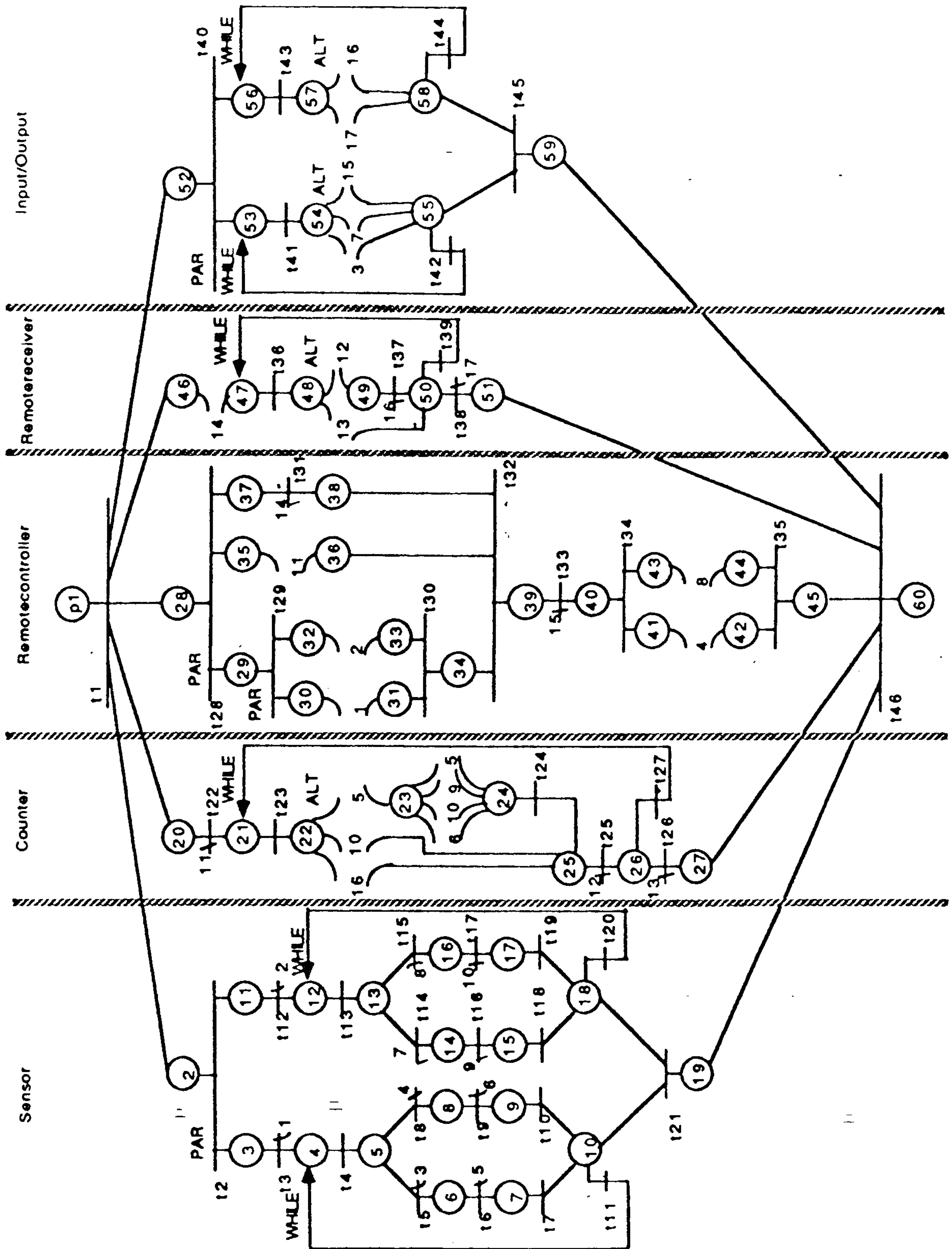


Figure 3.8 PNG for an Occam Program

PROC i = { ti , pi }.
where ti= {ta .. tn } and pi = {pa .. pn+1}.

Using the Petri-net graph each state and transition can be assigned to a process-identifier attribute.

PROC sensor = t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t 15 t16 t17 t18 t19 t20 t21 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15 p16 p17 p18 p19

PROC counter = t6 t9 t16 t17 t22 t23 24 t25 t26 t27 p20 p21 p22 p23 p24 p25 p26 p27

PROC remotecontroller = t1 t3 t8 t12 t17 t22 t28 t29 t30 t31 t32 t33 t34 t35 t46 p1 p28 p29 p30 p31 p32 p33 p34 p35 p36 p37 p38 p39 p40 p41 p42 p43 p44 p45 p60

PROC remotereceiver = t25 t26 t31 t36 t37 t38 t39 p46 p47 p48 p49 p50 p51

PROC io = t5 t14 t33 t37 t38 t40 t41 t42 t43 t44 t45 p52 p53 p54 p55 p56 p57 p58 p59

To complete the execution of the Petri-net graph by using tokens it is necessary to go through different ALT constructs by using the WHILE loop. All the processes communicate within the boundaries and only with each other. Transition t1 and t46 are placed in the remotecontroller process because these transitions are responsible for initialising and terminating all the processes. In the figure, the communicating transitions are the ones that are crossed '\'. The number that is written alongside that transition corresponds to the same number in some other process at the position of the other half of the communication.

An Occam program is used in this example to show how Petri-nets can be used successfully in modelling concurrent systems. The complete system is shown in chapter 7 and is analysed by generating the reachability tree

3.7 Conclusion

Petri-nets are capable of modelling and analysing software for distributed systems. Its representation makes the Petri-net simple and transparent to the users. Each step can be seen and the behaviour of the system can be seen. The dynamic properties of the systems can also be analysed by its dynamic properties. A Petri-net offers two analysis techniques, reachability tree analysis and matrix equation analysis. The reachability tree is used later on in the thesis for software distributed systems.

An example is also given where Petri-net is used to model an Occam program that represents the vehicle detection system. Petri-nets are used as the modelling and analysis technique in the automatic deadlock detection tool which is described in next chapter.

Chapter No 4
Automatic Software Analysis Technique

4.1 Introduction

The Occam programming language's ability to support concurrent programming provides a convenient medium for programmers to design and implement both concurrent and distributed software [NEWP 86].

The Occam model is based on the idea of the process as a unit of programmed activity: a process has a starting point, after which it proceeds and may eventually terminate. Concurrency simply involves several processes proceeding at the same time. The Occam model of concurrency is applicable equally to processes running on separate processors and to processes running within a single processor[INMO 84][POUN 85]. A classic problem in concurrent systems is 'deadlock'. In a distributed system environment, deadlock occurs when a process is waiting for a particular event that will not occur . Then it is in the state of deadlock. Different conditions have been explained in the literature under which deadlock exists in distributed systems[DEIT 84][BURN 88].

The main problem in the Occam model of concurrency is deadlock. These situations can be demonstrated by means of two simple examples.[FLEM 88]

```
PROC communication()
CHAN OF INT chan:
INT x,y:
SEQ
   SEQ
      chan ? x
      ... process 1
   SEQ
      chan ! y
      ... process 2
:
```

Fig 4.1(a) Communicating processes representing Non-concurrency

In the example shown in figure 4.1(a), two sequential processes are operating sequentially. The first SEQ process is programmed to input a value on 'chan', while, the same channel(chan) is programmed to output 'y' in the second SEQ

process.This is the condition of deadlock, since execution of the first SEQ process cannot proceed until transmission over 'chan' is successfully completed.

```
PROC communication()
CHAN OF INT chan:
INT x,y:
PAR
  SEQ
    chan ? x
    ... process 1
  SEQ
    chan ! y
    ... process 2
:
```

Figure 4.1(b) Program representing Concurrency without deadlock

Successful completion is only possible if both these SEQ processes execute in parallel (PAR) and is shown in figure 4.1(b).

```
PROC demonstrate()
CHAN OF INT chan1,chan2:
PAR
  --- process 1
  INT x:
  SEQ
    chan1 ? x
    chan2 ! 2
  --- process 2
  INT y:
  SEQ
    chan2 ? y
    chan1 ! 3
:
```

Figure 4.2(a) Program representing crossed channels

Figure 4.2(a) shows two processes waiting for input data. Since each process is waiting for the other to output, neither can be successful. Reordering of process 2 is needed to resolve the deadlock problem.

```
PROC demonstrate()
CHAN OF INT chan1,chan2:
PAR
  --- process 1
  INT x:
  SEQ
    chan1 ? x
    chan2 ! 2
  --- process 2
```

```
INT y:
SEQ
   chan1 ! 3
   chan2 ? y
:
```

Figure 4.2(b) Reordered program shown in fig. 4.2(a)

A number of strategies have been suggested[ANDE 81] to prevent, detect and recover deadlocks in the design of robust software. Petri nets and many other methods (like state space modelling using Petri nets and GMB/UCLA) can be used to simulate sequential and concurrent software[CARP 88b]. This is done by using a front-end translator to translate an Occam program into a mathematical model like Petri-nets and then by using a simulator to generate the reachability tree. The careful study of the reachability tree would show the presence of any deadlock. A considerable amount of time is then needed to detect and remove Occam deadlocks arising in software due to improper interprocess communication and synchronisation. The automatic detection of any potential deadlock present in an Occam program saves the user/programmer time.

4.2 Automatic detection of Occam deadlocks

For small processes manual analysis is normally exploited; the designer is required to inspect his Occam design and to construct, from templates, the Petri-net representing the flow in the software. For all but the most elementary software, analysis requires computerised tools since the net rapidly becomes complex and the number of reachable markings can be very large.

A software tool has been designed which has the capability to detect Occam deadlocks. The tool simply called octool (Occam tool) has two phases in the analysis:

I Occam translator : In which it translates an Occam program into a Petri-net structure

II Petri-net simulator: In which it analyses a Petri-net structure produced by the

These two parts of octool give an automatic detection technique for Occam deadlock using Petri nets. This deadlock analysis technique starts with the translation of an

Occam program into a Petri-net structure and uses structural and dynamic analysis on this specific Petri-net.

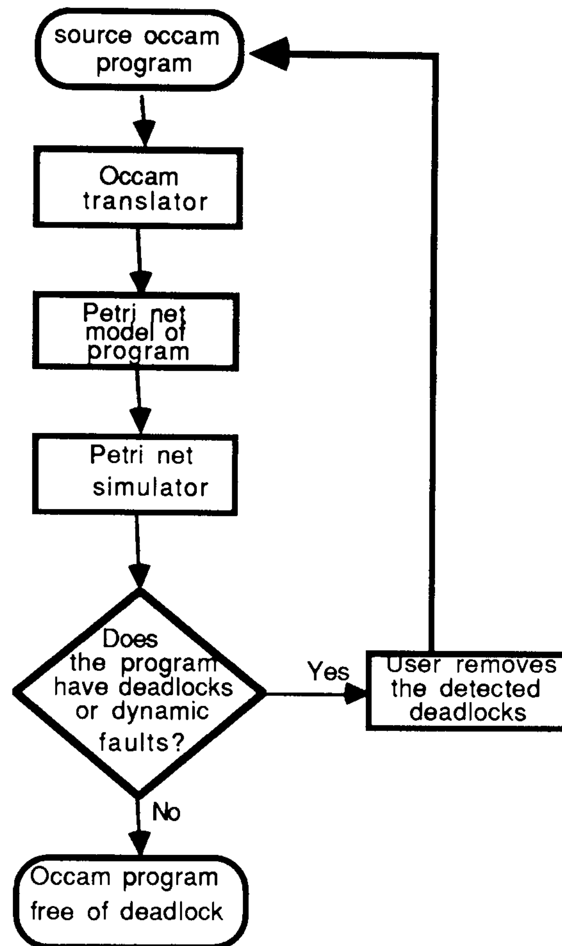


Fig4.3 Overview of dynamic fault detection tool.

The overview of the dynamic fault detection tool is explained through flowcharts in figure 4.3.

4.3 Occam translator

A software tool[KHAN 90] has been developed which will take an Occam source program and automatically translate it into its Petri-net in a form suitable for the simulator. The tool takes as its input a valid Occam program, using any of the legitimate Occam constructs; the program must compile without error using a standard Inmos compiler. There are a number of minor restrictions: the program may include program folds, but separately compiled and filed folds are not allowed. Furthermore, channel protocols are not modelled. If any of these properties are present in an Occam program, the translator will report an error.

Although an Occam translator is a piece of software with a single task, that task is large and complex. For practical manageability it is essential to break-down the processing of software into sub-tasks called 'phases'. Figure 4.4 shows the overall structure of an Occam translator.

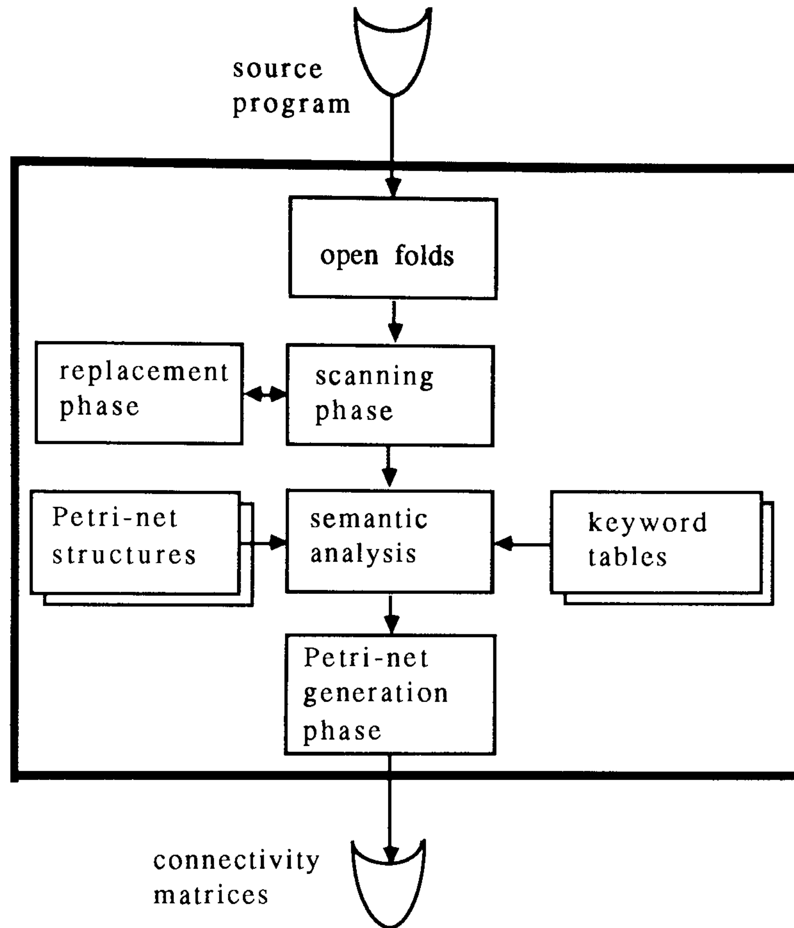


Figure 4.4 Overall structure of Occam translator

This software tool can be divided into four phases:

- scanning phase
- replacement phase
- semantic analysis phase
- Petri-net generation phase

Each of these phases are now explained in detail.

4.3.1 Scanning Phase

At the most fundamental level, an Occam program or source text (which forms an input to the Occam translator) is nothing more than a sequence of characters. The scanning phase searches for the keywords (like CHAN and PROC) for the identification of channels and procedures used in that source text. As mentioned earlier (in chapter 2), the folds in Occam program hide a section of the program from view (i.e. folded away) and that section can be summarised by a single line comment prefixed by ... .

Before the scanning of the program starts, the source program is passed through an 'open fold' procedure where all the folds inside the Occam program are opened for scanning to give a flat, pure Occam, text file. The file is then 'scanned' to determine the location of major keywords, such as the declaration and use of named processes (procedures), and interprocess channels. The function of this phase is explained by the pseudo-code is as follow:

```
begin
      read in the text
      for each line in the text
      begin
            if fold
            then open fold
      end;
end.
```

4.3.2 Replacement phase

Having determined and checked the general structure of the source program, an Occam translator must now look more closely at all the keywords (e.g. PROC and CHAN OF INT, CHAN OF BYTE or CHAN) which are used in the source program. This phase is called the replacement phase because of its involvement in the following two steps:

(i) each channel in the source program recognised is replaced by predefined channel names in the pre-processor (i.e. chan1, chan2,...etc.).

(ii) another file is created and the source program stored away after it has been processed by the replacement phase.

Actions such as these, which involve modification to the structure of substantial program components, are known as global optimisations (in compiler

terminology)[FARM 85][AHO 85]. The same function is also explained in the form of a program which explains in detail the overall procedure of the replacement phase.

```
Procedure replace (expected keyword : keyword type);
 begin
      if  scan.keyword = expected.keyword then
          begin
               replace.keyword
               scan.next keyword
          end;
     else  source.error  /*  pre-define error routines  */
 end;
```

4.3.3 Semantic Analysis Phase

This phase now looks for all the procedures and the replaced channels (which have been replaced during the replacement phase) and involves :

a) verifying correct usage.
b) determining associated semantic requirements

Two tables are used for the semantic analysis. The various program structures are identified by their keywords (like SEQ, PAR, ALT, :=, ?, !, IF and WHILE) and tabulated so that the appropriate Petri-net sub-structures can be correctly sequenced and interrelated. As each occurrence of a keyword is encountered, so a record is kept of the keyword's position in the overall program, together with any associated semantic requirements. A limited check on correct usage is carried out, primarily on named processes and interprocess channel, but the analysis assumes that the overall Occam program is valid and syntactically correct.

A second table contains the different Petri-net templates (in connectivity matrix form) which correspond to the various keywords. The final phase of semantic analysis systematically matches each keyword table against the Petri-net fragment defined by its connectivity matrices. The subroutines (where Petri-net templates are stored) can be called by this semantic analysis phase any time when the sequence of keywords match.

```
idrec = record
        begin
            name :        SEQ or [SEQ, :=, ?, !]
```

```
            attributes : call subroutine
        end;
```

The association of each keyword (and array of keywords) with its attributes must involve the maintenance of some record. The pseudo-code for this phase are:

```
begin
    read in the text
    for each line in the text
      begin
         if keyword
           then store in array
      end;
     for i = 1 to 100
       begin
          compare array with table[i] of keywords
           if array equals table[i]
              then use Petri-net structure given by result[i]
       end;
end.
```

This pseudo-code describes that a number of tables (say, 100 tables) with different matching of keywords and their results(Petri-net structures) is predefined in this phase. This part of the translator simply compares each keyword table with the keyword in the program. If a match is found, it displays the result, otherwise, the search continues in the next table and so on.

4.3.4 Petri-net generation phase

The final phase sees the Petri-net fragments properly connected to form a complete Petri-net defined in term of connectivity matrices.

This procedure ensures the proper interconnection of processes according to their interprocess communication channels. The generated matrices are stored in a data file (.dat) which is the object program of the Occam translator (pre-processor). The matrices may then be input directly to the simulator so that Petri-net reachability analysis can proceed.

The table in figure 4.5 shows a few Occam program models (source program for the pre-processor), each Petri-net graph and its Petri-net structure (produced by the pre-processor). The organisation of the Petri-net structure in memory is given in section 4.5.

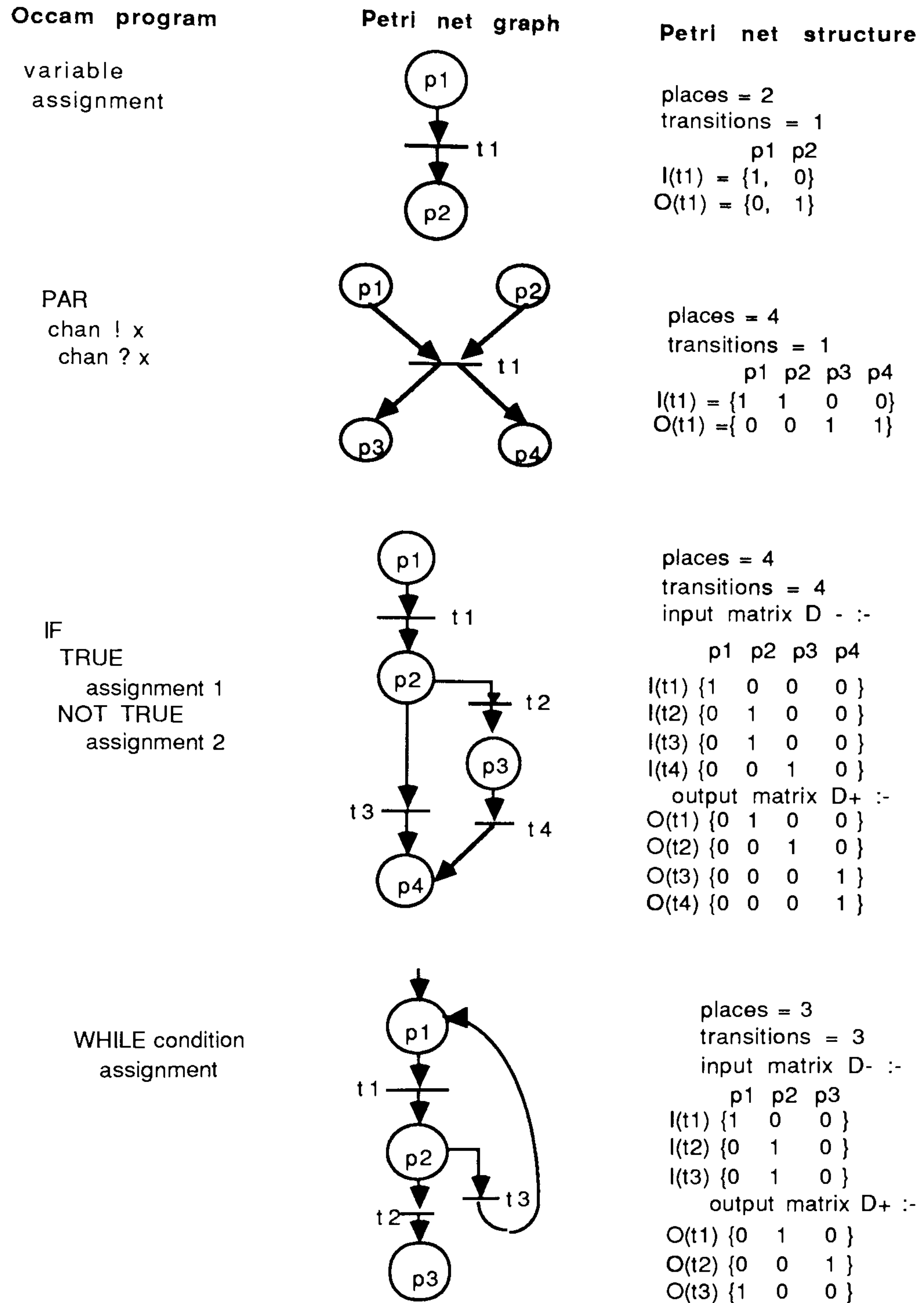


Figure 4.5 Petri-net structures produced by pre-processor

Figure 4.5 shows the Petri-net structure of two matrices, i.e. input matrix and output matrix, produced by translating an Occam program. The input matrix is one which shows the number of arcs from place(s) to transition(s). The output matrix represent the number of arcs from transition(s) to place(s).

The translator works with a number of limitations. Firstly, it can only recognise a small number of programs and is not able to read complex programs because it depends on the keyword tables. For example , if the keyword table has 100 different keyword sequences, then the translator can only recognise those 100 programs. Secondly, the translator was not found to be able to recognise some of the Occam constructs other than SEQ, PAR, ALT, ?, !, :=, WHILE and IF. Finally, the translator was not comprehensive enough to check the syntax of Occam program. So, it was found necessary to compile an Occam program on the Inmos compiler before translating that program into a Petri-net structure. Therefore, another approach is used for the design of the translator which is given in chapter 6.

4.4 Petri-net simulator

A number of in-house simulators have been developed, based on the conventional formulation of the net, on its matrix description, as well as on more innovative descriptions [CARP 89].

One simulator, implemented in 'C' uses Petri-net theory to analyse a concurrent asynchronous system. This simulator was initially designed by Arnold[ARNO 86]. Due to many limitations of this simulator, the first task undertaken in the development of the OCTOOL environment as reported in this thesis was a redesign of this simulator. My research showed that the main limitation of the simulator occured in:

(a) analysing large Petri-net graphs
(b) generating tree structure
(c) fault tolerance (was not found resilient)

It is based upon defining a Petri-net in the matrix definitional form, (P, T, D-, D+), and carrying out systematic transition firing. Firing can either be interrogated step-by-step, or another option allows the generation of a marking tree with subsequent transition sequence searching.

To avoid information loss during the marking tree generation, the only node classification used was the recognition of terminal and duplicate nodes. This limits the tree's size. Nevertheless, this led to dynamic memory shortages with Petri nets which have infinite reachability trees. This could be overcome partially by overlaying components of the menu driven package, as each option is required.

Therefore, this increases the amount of dynamic memory available.The Petri-net simulator program consists of three menus:

1. main menu	Petri-net file creation, modification and display.
2. analysis menu	Firing sequence interrogation.
3. tree menu	tree generation and display, firing sequence determination.

4.4.1 Main menu

This menu has five options and are given below.

- First option is used to enter Petri-net file name for subsequent creation or analysis.

- The second option is used to create Petri-net file. If the file already exists the user is asked to confirm its overwriting.

- Displaying Petri-net file is the third option. This option display the Petri-net in the specified data file.

- Analyse Petri-net file option is also available under this menu. Here control is then passed to the analysis menu.

- The final option is to exit to the system.

Initially the simulator was designed to analyse 25 places and transitions. Now this limit has been increased and the simulator can analyse up to a limit of 200 places and transitions. This extension makes the simulator capable of analysing a large and complex Petri-net structure modelled for real-time systems.

It was observed that at the time of creation of a Petri-net file, a small mistake made by the user in defining the position of the places and transitions will compel the user to start again. To avoid this, the Petri- net creation procedure was changed into one which is very flexible for the user. The simulator was also enhanced by the modification option. This option can modify the existing Petri-net structure.

4.4.2 Analysis menu

This menu offers 10 options. The first option asks the user for the description of the marking. The user may switch between no marking, default marking from the specified data file or user entered markings. The transition firing which is the second option may be switched by the user between :

SINGLE - single stepped firing, the user enters the transitions to be fired one by one.

FILE - the transition are taken from the specified transition file and execute one after another.

Other options ask the user for marking, the sequence of transition firing, execution of firing sequence and display the firing sequence. The menu allows the user to enter in the main menu or in the tree menu by different options.

4.4.3 Tree menu

Three options are available in this menu, which enable the user to generate and display the marking tree. In the option to generate a marking tree, the user is prompted to specify the required size limit for the tree:

1. generate tree to a particular level.
2. generate tree until system runs out of dynamic memory allocation.

It is suggested that the user should use the first option, which generates the tree to a specific level. The other option, in a few cases, keeps on generating a marking tree until the program is aborted by the user.

4.5 Data structures

All the files created by the user are held on the system as *.pns in type, short for Petri-net structure.

```
PAR
 --- process 1
   SEQ
    ....
   chan ! x
    ....
 --- process 2
   SEQ
```

```
....
chan ? y
....
```

Figure 4.6 Occam program showing Concurrency

The data structure for an Occam program given in figure 4.6 is shown in figure 4.7. The Petri-net graph of an Occam program is shown in figure 4.8[CARP 87].

```
10 7                                  ----- row  1
0 0 0 0 0 0 0 0 0 0 0                 -----      2
0 1 0 0 0 0 0 0 0 0 0                 -----      3
0 0 1 0 0 0 0 0 0 0 0                 -----      4
0 0 0 1 0 0 0 0 0 0 0                 -----      5
0 0 0 0 1 1 0 0 0 0 0                 -----      6
0 0 0 0 0 0 1 0 0 0 0                 -----      7
0 0 0 0 0 0 0 1 0 0 0                 -----      8
0 0 0 0 0 0 0 0 1 1 0                 -----      9
0 0 1 1 0 0 0 0 0 0 0                 -----      10
0 0 0 0 1 0 0 0 0 0 0                 -----      11
0 0 0 0 0 1 0 0 0 0 0                 -----      12
0 0 0 0 0 0 1 1 0 0 0                 -----      13
0 0 0 0 0 0 0 0 1 0 0                 -----      14
0 0 0 0 0 0 0 0 0 1 0                 -----      15
0 0 0 0 0 0 0 0 0 0 1                 -----      16
```

Figure 4.7 Data structure of an Occam program

The first row gives the number of places and the number of transitions respectively. The second row vector indicates the default marking:

$$(u_0, u_1, u_2, \ldots u_{10})$$

u_0 (row1, column1) is used as a flag to indicate the presence of a default marking and therefore, in rest of the column a leading zero was added as a first element of a row. Rows 3-9 are the D^- matrix, rows 10-16 are the D^+ matrix.

As an example, row 3 represents the input function I(t1). The second digit is the number of inputs from place 1 to transition 1, the third the number from place 2 to transition 1 etc.

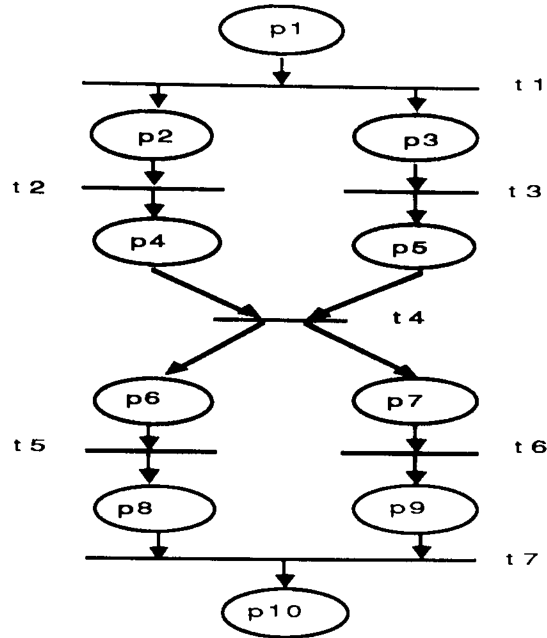


Figure 4.8 Petri-net graph of an Occam program

Expanding the above data file into the format as displayed by the '4' option of the main menu (see section 4.4.1), we have:

No of places = 10

No of transitions = 7

I(t1)	1	0	0	0	0	0	0	0	0	0
I(t2)	0	1	0	0	0	0	0	0	0	0
I(t3)	0	0	1	0	0	0	0	0	0	0
I(t4)	0	0	0	1	1	0	0	0	0	0
I(t5)	0	0	0	0	0	1	0	0	0	0
I(t6)	0	0	0	0	0	0	1	0	0	0
I(t7)	0	0	0	0	0	0	0	1	1	0

D^- matrix
(input matrix)

O(t1)	0	1	1	0	0	0	0	0	0	0
O(t2)	0	0	0	1	0	0	0	0	0	0
O(t3)	0	0	0	0	1	0	0	0	0	0
O(t4)	0	0	0	0	0	1	1	0	0	0
O(t5)	0	0	0	0	0	0	0	1	0	0
O(t6)	0	0	0	0	0	0	0	0	1	0
O(t7)	0	0	0	0	0	0	0	0	0	1

D^+ matrix
(output matrix)

	p1 p2 p3 p4 p5 p6 p7 p8 p9 p10	
u =	0 0 0 0 0 0 0 0 0 0	(marking)

The data file resides in memory as a 2- dimensional array because this array type occupies less memory and the retrieval is fast.

4.6 Conclusion

Software is currently the most expensive part of the system development and in the real-time world it is the most critical part of a system. Nuclear power stations, military command systems and navigation are a few examples where the safety and reliability of the system are critical. At present, different techniques have been implemented in specifying fault-tolerant software for a system. A minor mistake in the real-time software can be fatal. One would like to be able to take a specification and test it for its accuracy and robustness in solving the original problem.

The Occam programming model is designed for concurrent systems. A classical problem which might affect a concurrent system is deadlock. Petri-nets are used as a tool to detect deadlocks. The analysis of large and complex Petri-net structure is difficult. Also, a number of problems can be faced in translating an Occam program into a Petri-net. An easy automatic method of verifying Occam software as being deadlock-free has been developed which helps to avoid critical situations through malfunctioning. Also, the time taken to detect errors in specifications (deadlock) can be saved.

A translator designed for translating an Occam program into Petri-net structures has been given. It takes an Occam program and automatically translates it into its Petri-net structure(PNS). A number of limitations were reported and because of all those limitations, this tool was not found flexible. Therefore, further research was carried out to make the tool more flexible. Hence, Lex and Yacc (compiler design utilities under Unix) were used to design Occam translator. The next chapter gives a detail working of Lex and Yacc and then the use of these utilities to generate an Occam translator are explained in chapter 6.

Chapter No. 5
Lex and Yacc: Implementation Issues

5.1 Introduction

To overcome the limitations mentioned in chapter 4, two utilities under UNIX are used to design a more comprehensive translator that is flexible enough to translate most of the formats of an Occam program into its Petri-net structure. These most commonly used utilities for the automatic generation of compilers are Lex, a lexical analyser generator and Yacc, a syntax analyser generator (or parser generator). Lex and Yacc are available as software tools on the UNIX system, and have been used to help implement hundreds of compilers[AHO 85]. Since a part of this thesis is based on these utilities, the following sections present an account of how they function and interact in the production of an Occam translator.

The software tools utilized in the implementation are discussed and a brief report is given on the implemented version of mapping. Before work on the Occam translator (which provides a tool for automatic translation of an Occam program into Petri-net structure) is given, the lex and yacc specification for a small but illustrative example is discussed. In this way, the ease of use of lex and yacc and to target a number of improvements were gauged. Finally, the input and output formats for the implemented version to the Occam translator are discussed.

5.2 The Lex lexical analyser generator

Lex (a lexical analyser generator) [LESK 75] is a lexical scanner for a user-defined programming language, which is used to segment input (a source program) in preparation for parsing routine. It is intended as a tool which accepts a high-level specification based on regular expressions, and generates a C program to recognize instances of these regular expressions appearing on the input stream. Lex reads an input stream, copies it to an output stream, and partitions the input into strings which match the regular expressions. As a string is recognised, the corresponding action is executed. Hence it is useful for performing editor-like transformations on its input, or for "tokenizing" the input for use by a language syntax analyser[AHO 77]. For this reason it represents a complementary tool to yacc, and indeed was designed with this in mind.

5.2.1 Input Specification

The format of a lex specification is divided into three sections separated by "%%".

The general format is thus:

```
{definition}
%%
{rules}
%%
{user Subroutines}
```

where the first and third sections are optional and are often omitted. The definition section allows the user to give names which are to be associated with commonly used regular expression patterns. For example, since alphabetic characters are permitted in the identifier and keywords of a programming language(Occam in this case), the user may wish to define letter as:

```
letter          [a-zA-Z]
```

User can then use the name "letter" in the rule section in place of its corresponding regular expression. The definition section also allows the inclusion of arbitrary C code fragments placed between the delimiters "%{" and "%}", which are simply copied unchanged into the file containing the generated lexical analyser. The rule section is structured as a sequence of regular expression/action pair. Hence a typical rule will appear thus:

```
integer        { printf("found keyword INT \n"); }
```

Every time the character string integer is encountered in the input, the message "found keyword INT" is displayed. The form of the regular expressions is very similar to those used in QED(KERN 72). Whenever the characters in the input stream match with one of the regular expressions, the corresponding action is executed. Character strings not matching any of the defined regular expressions are copied to the output (since in a compiler application, we must ensure that all possible input, including error text, will be matched). Lex can allow the ambiguous specification, when more than one expression can match the current input. The following rules are applied to resolve the ambiguity:

(1) The longest match is preferred.

(2) Among rules which match the same number of characters, the rule given first is preferred.

suppose

```
integer                 {/* keyword action */}
[a-z][a-z]*             {/* identifier action */}
```

If the input symbol found is `integers` then the identifier action is performed because it matches 8 characters, however if the input found is `integer` both rules match 7 characters and the keyword rule is performed, because it was given first.

In order to implement this matching process, the lexical analyser may need to read a significant number of characters ahead. However when a string is finally matched, the input is appropriately backed-up, so that this lookahead process remains invisible to the user.

The action part of each rule is written by the user as a C code fragment which may include calls to his own function or to functions contained in the lex library. Typically, when lex is used in conjunction with the yacc syntax analyser generator, these actions will include a statement which returns an integer value identifying the group of characters consumed from the input stream(i.e. the language token). For example, when matching an identifier the user would use a rule such as:

```
[a-zA-Z][a-zA-Z]*        return (IDENTIFIER);
```

In many applications the user does not purely require the identifying integer value returned in the manner described above, but also the actual characters in the input which matched the regular expression. For this purpose, lex provides an external character array called `yytext`, which is over-written every time a regular expression is matched. The following example illustrates a rule which echo the character string matched:

```
[a-z]+              printf("%s", yytext);
```

Lex does not automatically evaluate the string which it has matched (e.g. finding the integer value of a string of digits). Such an operation must be written by the user as a C code fragment in the associated action, but lex does provide an external variable `yylval` through which this value can be communicated to the syntax analyser. If

lex is used in conjunction with yacc then yylval should be declared in the generated syntax analyser as a union of all types appropriate to the grammar symbols appearing in the yacc rules.

In certain cases, the user may wish to override lex's method of choosing the longest match from the input stream. The special action reject is provided for this purpose, and calls the lex function yyreject(). Essentially, when lex matches a rule whose action contains REJECT, it passes on to the next alternative match.This is particularly useful when definitions of the items being matched overlap; for example, we will need to distinguish between identifiers and keywords of the language. A lex specification to achieve this would include:

```
        [a-zA-Z][a-zA-Z]*             {/*if yytext isn't in the
                                      keyword table then reject,
                                      this and go and try the
                                      identifier rule */ }
        [a-zA-Z][a-zA-ZO-9]*          { /*identifier action */
}
```

5.2.2 Operation of the generated lexical analyser

The lex specification,as described above, is converted into a C program held in a file lex.yy.c, see Figure 5.1, with a main function called yylex(). This is an interpreter which is driven by a number of generated tables, and acts as a deterministic finite state automaton. The tables are a representation of a transition diagram for the specified language. This results in the generated lexical analyser being quite fast even for a large collection of rules.

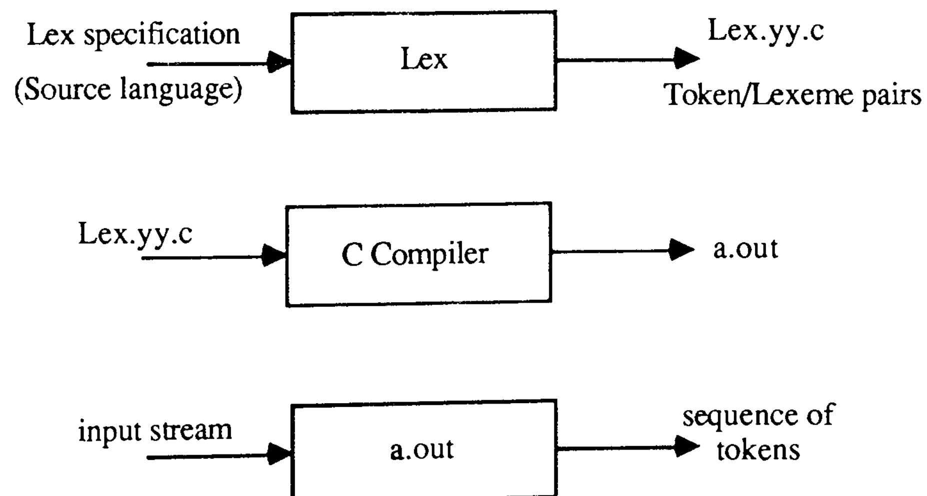


Figure 5.1 The lex lexical analyser generator

In fact, the time taken to partition an input stream is proportional only to that input stream's length, regardless of the complexity of the lex rules, as long as the amount of rescanning required is not excessive, a condition satisfied for typical programming languages. The only overhead of a complex and large lex specification is the increased size of the code produced to implement the lexical analyser.

5.3 The Yacc syntax analyser generator

Yacc[JOHN 78], which is an acronym for "Yet Another Compiler-Compiler", was developed by Johnson(1975) at the AT&T Laboratories and runs mainly on Unix based systems. It is written in portable C and accepts LALR(1) grammars with disambiguating rules. In LALR(1) grammar, one token of lookahead is required to resolve conflicts in the parser. The L means that input is read from left to right. The R means that a rightmost derivation is used to construct the parser tree, i.e., the rightmost nonterminal is always expanded. The LA in LALR(1) stands for 'Look Ahead', that is, information about certain look ahead character is actually built into LALR(1) parser table.

Yacc is used to check whether a proper combination of the lexical token satisfies the syntactic constraints of the user-defined (Occam) language. A yacc user specifies the structure of his input with their associated actions, language fragments to be invoked when each such structure is recognised. The yacc output is generated from such a specification. The parser receives lexical tokens from the Lex program, recognises syntactic constructs of the user-defined programming language, and performs corresponding actions.

Also Yacc is not strictly a compiler-compiler since it does not generate code to perform semantic analysis and code generation, however, it does allow the user to manipulate a semantic stack (a stack of attributes associated with the grammar symbols used), and to associate semantic actions with grammar rules. The syntax analyser generated by yacc is intended to be used in conjunction with a lexical analyser, which can be either hand-coded or produced by a Lex. Figure 5.2 shows an outline of the syntax analyser's operation.

ASTON UNIVERSITY
LIBRARY AND
INFORMATION SERVICES

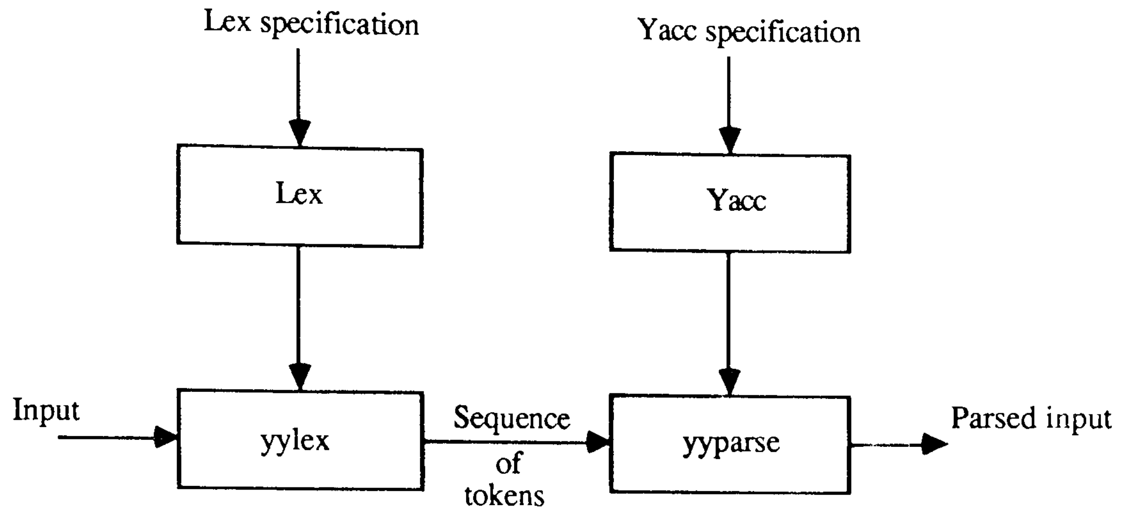


Figure 5.2 Cooperation of lex and yacc

Since the parsing algorithm used in yacc is LALR(1), its power is limited by this method with respect to the class of grammars of which it can generate a syntax analyser. However, it does allow ambiguous grammars to be used and provides a mechanism for resolving conflicts in such grammars.

5.3.1 The Input Specification

Input to yacc is separated into three parts: the declaration, rules and user routines section, where the separation is donated by "%%" in a similar fashion to 'lex' input. Thus it has the following structure:

```
{declaration}
%%
{rules}
%%
{user routines}
```

Each of these sections are explained in more detail.

5.3.1.1 Declaration Section

The declaration section has two optional parts. The first of these, delimited by %{ and %}, contains ordinary C declarations which are used to declare variables and

structures used by the user-written routines to deal with semantic actions. This part can also contain C compiler directives such as #include and #define, for example:

```
%{
#include <stdio.h>
#include <ctype.h>
#define NULL 0
...
struct process_channel
{
 char *channel_name;
 int icntr, ocntr;
 struct process_channel *next;
};
%}
```

All lines enclosed by %{ and %} are copied to the parser; therefore, they must be in a correct C syntax.

The second, and most important part contains declaration of the token returned by the lexical analyser. The declaration of tokens may have the following form:

```
%token      CHAN
%token      PAR
```

The CHAN and PAR are declared here as a token. Tokens declared in this section can then be used in the second and third parts of the yacc specification.

5.3.1.2 Rules Section

The translation rule section contains the context-free grammar of a language, expressed in a BNF notation, augmented with user-defined actions. The left-hand side of each rule is a nonterminal of the grammar; the right-hand side is a sequence of zero or more alternatives separated by a bar "|". The alternatives consist of both terminal and other nonterminals. Each rule consists of a grammar rule and the associated semantic action. For example:

```
<left side> -> <alt 1> | <alt 2> | .... | <alt n>
```

would be written in yacc as

```
<left side> :     <alt 1>     { semantic action 1 }

            |     <alt 2>     { semantic action 2 }
            .....
```

```
        |       <alt n>       { semantic action n }
        ;
```

A quoted single character on the right-hand side is taken to be a terminal symbol, and unquoted strings of letters and digits not declared to be tokens are assumed to be nonterminals.

The user can insert actions to be performed when a rule has been recognized by the syntax analyser. Such actions are written as a C code fragment delimited by "{" and "}", and can be placed anywhere in the right-hand side. The symbols $$ represent the attribute value of the nonterminal appearing on the left-hand side of the rule; the symbol $i (where i is an integer) represents the value of the i^{th} grammar symbol (terminal or nonterminal) of the right-hand side. Normally the semantic action will compute the value of $$ in the term of some function of the $i's. If no action is specified, the default is to evaluate $$ as the value of the first grammar symbol, i.e. $1. For example, the operation of a simple desk calculator in yacc would be written as:

```
expr : expr'+'term  { $$ = $1 + $3; /*expr = expr + term*/ }

     | term         { $$ = $1; /* expr = term */ }
     ;
```

5.3.1.3 User routines section

The actions specified by the user to be performed during parsing may need to be more than a few statements in length. So, yacc provides a section in which the user can declare his own C functions. These can then be called from their associated translation rules. A lexical analyzer by the name `yylex()` must be provided, unless this has been provided by an external reference. The generated syntax analyser calls this function to scan the input and return token values corresponding to those listed in the declaration section. Those token values are returned to the syntax analyser by `yylex()` via the yacc defined variable `yylval`. `yylex()` can either be hand-written or generated by a tool such as lex.

5.3.2 Operation of the generated syntax analyser (Parser)

The specification as described above is transformed by yacc into a C program called `y.tab.c`, whose main procedure is called `yyparse()`, see figure 5.3.

`yyparse()` operates as a finite state automaton - a stack machine. The schematic form of an LR parser is shown in figure 5.4 [AHO 85]. It consists of an input, an

output, a large stack, a driver program (transition matrix), and a parsing table that has two parts, i.e. action and goto.

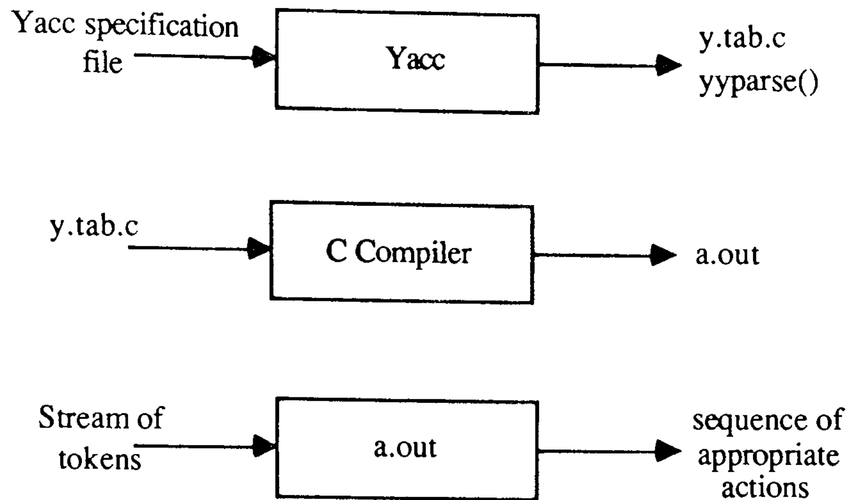


Figure 5.3 The yacc syntax analyser generator

The driver program is the same for all LR parsers: only the user defined parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $S_0a_0S_1a_1...S_ma_m$, where S_m is on the top of the stack. Each a_i is a grammar symbol and each S_i is a symbol called a state. Each state symbol summarises the information contained in the stack machine below it, and the combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision.

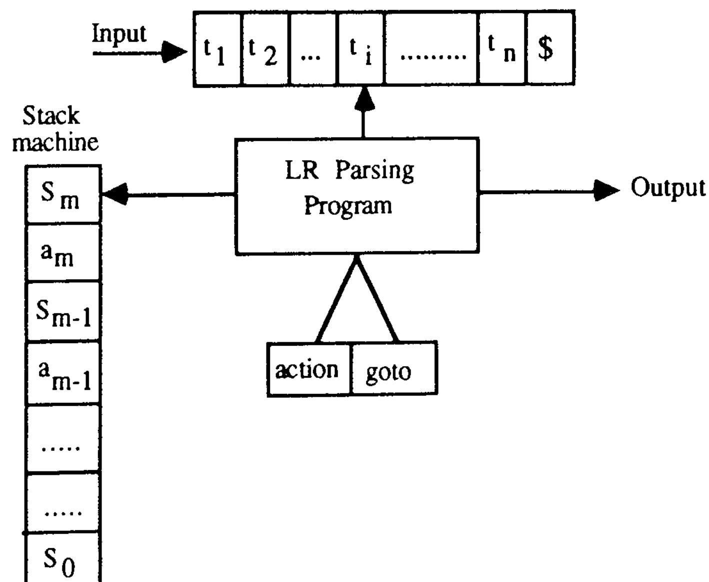


Figure 5.4 General working of the parser

The parsing table consists of two parts, a parsing action and a goto function. The parsing program in the LR parser determines Sm, the state currently on the top of the stack, and t_i, the current input symbol. It then consults action [S_m, t_i], the parsing action table entry for state S_m and input t_i, which can have one of the four values: shift, reduce, accept and error. The function goto takes one state. goto function is a operation for a new state and that new state is pushed on to the stack machine.
As mentioned, the finite state automaton used has four possible actions: shift, reduce, error and accept.

Shift
A shift action is performed when the next token is valid in the current state. A new state is then pushed onto the stack and becomes the current state.

Reduce
When the syntax analyser has successfully matched the entire right-hand side of the rule, a reduce action is taken. When reducing, the syntax analyser will pop the number of states corresponding to the number of grammar symbols on the right-hand side of the translation rule; the current state is then the one remaining on the top of the stack.

Error
If the input cannot be matched against any translation rule then the syntax analyser performs an error action. If the user has not supplied an appropriate error translation rule, the syntax analyser simply prints an error message.

Accept
This operation happens only once. When the syntax analyser reaches the end marker of the grammar it enters the accept state, and returns an indication that its parsed input was a valid sentence of the given language.

5.3.3 Ambiguity in Yacc

Yacc allows the use of ambiguous grammars[JOHN 78][SCHR 85]. In order to resolve ambiguity, the user may provide disambiguating rules, but if these are not supplied yacc takes pre-defined default action. A trace file y.output, which is produced by yacc when called with its "-v" option can be examined to verify that the ambiguities have been resolved as the user intended. This file contains a list of the states entered by the syntax analyser during its operation. Two different types of

conflicts can occur in LR parsers; shift-reduce and reduce-reduce. Shift-reduce conflicts occur when the parser has to decide between shifting, or reducing by a translation rule.

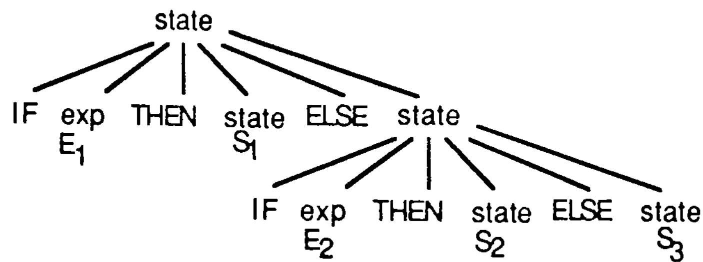


Figure 5.5 Parse tree for conditional statement

Reduce-reduce conflicts occur when the parser has to decide which of several translation rules to reduce. A shift-reduce conflict occurs with IF statement in the programming language. The conflict occurs if the IF statement has an optional ELSE part, so that the user may have a dangling ELSE. A typical grammar for such statement would be:

```
state  ->  if exp then state
         | if exp then state else state  ---   5.1
         | Other
```

According to this grammar, the compound statement is

$$\text{if } E_1 \text{ then } S_1 \text{ else if } E_2 \text{ then } S_2 \text{ else } S_3$$

expands into the parse tree of figure 5.5 and no problems were found. However, the grammar (5.1) is ambiguous. The compound statement

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$$

has two parse trees shown in figure 5.6

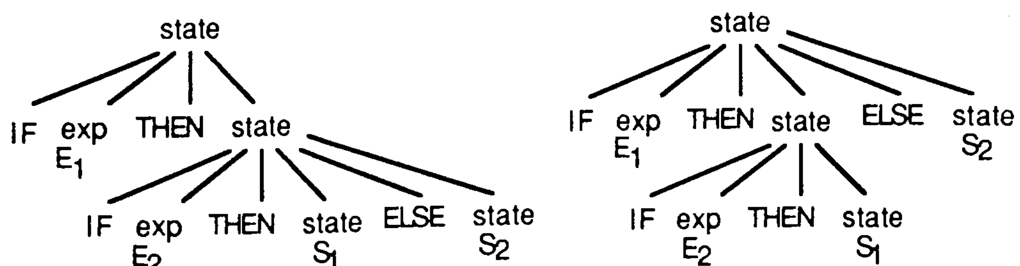


Figure 5.6 Parse tree showing ambiguous grammar

To overcome this ambiguous situation the idea is that a statement appearing between a then and an else must be matched, i.e., it must not end with an unmatched then followed by any statement for else would then be forced to match this unmatched then. The user may use the grammar as:

```
State -->               match_state
          |             unmatch_state
          ;
match_state -->         if exp then match_state else
                        match_state
          |             other
unmatch_state -->       if exp then state
          |             if exp then match_state else
                        unmatch_state
```

This grammar generates the same set of strings as 5.1

A shift/reduce conflict is resolved in favour of shift by default(a decision of Yacc, which resolves the problems such as the "dangling-else"); a reduce/reduce conflict is resolved in favour of the rule appearing earliest in the specification.

If the user wishes to provide his own rule to resolve an ambiguity, this is done by specifying the precedence and associativity of the operators in a table in the declaration section. Associativity is indicated by %left (left-associative), %right (right-associative), or %nonassoc (non-associative). Precedence is given by listing the operators in ascending order of priority. Thus most shift/reduce conflicts are resolved by giving precedence and associativity not only to each symbol but also indirectly to each rule involved in a conflict. In a situation where an operator can be either unary or binary (e.g. '-'), the user can enforce a particular precedence by appending the following "tag" to a rule:

```
%prec <terminal>
```

where the terminal's precedence and associativity have been given in the declaration section.

5.3.4 Error recovery in Yacc

In yacc, error recovery can be performed using a form of error rule. In this method, the user augments the grammar with error rules which are of the form:

```
S --> error T
```

Where S is a nonterminal and T a sequence of grammar symbols (both terminals and nonterminals). The symbol error is a reserved word, and yacc treats a rule containing it like any other rule. When an error is detected the syntax analyser behaves as if it had just seen the special symbol error immediately before the token which caused the error. The syntax analyser then looks for the nearest rule for which the error symbols a valid token and resumes processing at this rule.

5.4 The use of lex and yacc

In this section the implementation of Occam program translation into a Petri-net structure using lex and yacc are described. First, the lex specification is presented, followed by the yacc specification of Occam language syntax. Finally, the input and output of the implemented version are described.

5.4.1 Occam language lex specification

Occam language has the following lexemes:

(1) integer literal: an integer is a string of digits representing an integer in the range 0 to maximum.

(2) boolean literal: a boolean literal is given by the strings TRUE or FALSE, which are keywords.

(3) identifier: an identifier is denoted by an arbitrary length string of letters and digits, beginning with a letter.

(4) keywords:there are a number of keywords which are reserved, and consists of a string of letters.

(5) operators/delimiters: Occam language has a number of logical, arithmetic operators and various delimiters.

Comments in Occam language begins with "--" and continue until the end of the current line of the program. These can be dealt with during lexical analysis, by simply consuming characters without returning a value to the syntax analyser.

These lexemes using the following lex definitions and rules can be specified:

<u>integer literals</u>

```
digit           [0-9]
int             ({digit})({digit})*
{int}           { return(INT); }
```

<u>Identifiers</u>

```
letter          [A-Za-z]
identifier      ({letter})(({letter})|({digit}))*
{identifier}    { return(IDENTIFIER); }
```

<u>Keywords</u>

```
"CHAN"          { return(CHAN); }
"PAR"           { return(PAR); }
"IF"            { return(IF); }
"WHILE"         { return(WHILE); }
...etc for all keywords of Occam language
```

<u>Operators and delimiters</u>

```
"="             { return(ASSIGN); }
"+"             { return(PLUS); }
...etc for all operators and delimiters of Occam language
```

<u>Comments</u>

```
"--"{any_NL}*[\n]       { /* do nothing */ }
```

Assembled valid lex specification obtained is:

```
%{
#include "y.tab.h"
#include <ctype.h>
%}
any_NL          [^\n]
digit           [0-9]
letter          [a-zA-Z]
int             ({digit})({digit})*
identifier      ({letter})(({letter})|({digit}))*
%%
...
"="             { return(ASSIGN); }
"+"             { return(PLUS); }
...
"CHAN"          { return(CHAN); }
"PAR"           { return(PAR); }
"IF"            { return(IF); }
"WHILE"         { return(WHILE); }
...
{identifier}    { return(IDENTIFIER); }
{int}           { return(INT); }
"__"{any_NL}*[\n] { /* it is a comment */ }
...
%%
```

lex specification for Occam language

Another very important aspect of the Occam programming language which is specified in the lex specification is 'indentation '. The Lex program reads indentation(two white spaces), and assigns a BEG token for every start of indentation, and END token for every end of the indentation.

As can be seen from the given Lex specification, it can be quite cumbersome to specify the keywords of a language explicitly in the lex rules. This form of specification also results in a large number of cases in the main C switch statement generated by lex. In many programming languages the patterns for identifiers and

keywords have much in common; often a neater and more efficient solution is to determine their inter-relation, and use a function to distinguish between them, by searching a table of keywords.

5.4.2 Occam language Yacc specification

Yacc only allows rules to be given using BNF(Backus Naur Form). The context-free specification for the Occam language is given below as[BURN 88]:

```
main_program      =    sep process | process
process           =    specification | action | construct
specification     =    declaration | definition
declaration       =    type ID ':'
type              =    CHAN OF protocol |TIMER | BYTE | INT
protocol          =    ANY | ID | INT | BOOL
definition        =    PROC ID '(")' sep BEG process END ':'
action            =    assignment | input | output
assignment        =    ID  ':"='  expr
input             =    ID  '?' expr
output            =    ID  '!' expr
construct         =    sequence | parallel | conditional | alternation | loop
sequence          =    SEQ sep BEG process END
parallel          =    PAR sep BEG process END
conditional       =    IF sep BEG choice END
choice            =    boolean sep BEG process END
alternation       =    ALT sep BEG alternative END
alternative       =    guard sep BEG process END
guard             =    boolean '&' input | input | boolean '&' SKIP
boolean           =    expr
loop              =    WHILE expr sep BEG process END
sep               =    EOL | sep EOL
```

For complete BNF code for the Occam language the reader is advised to see appendix B.

5.4.3 Mapping

The Lex input for Occam programs consists of regular expressions. The format of these regular expressions is given in section 5.4.1. The parser (yacc output), which performs the Occam program translation into Petri-net structure consists of the main routine (Occam syntactic rules and their actions) and few subordinate subroutines. The Lex program and the parser have been compiled and loaded into the file translator. Thus, the user can use

```
translator <input file> output file
```

in order to get a Occam program stored in the file input translated into its corresponding Petri-net structure stored in the file output.

5.4.4 Input and output of the implemented version

The main purpose of translating Occam programs into Petri-net structures is to check the correctness of communication patterns between the processes of an Occam program and to report potential communication anomalies. In that respect, most communication syntactic constructs of an Occam program, such as input, output primitives and, sequential, parallel, and alternative constructs are preserved in Occam syntax. However, syntactic constructs like function, processor, round, workspace and some other constraints are not important (from a communication point of view) for the translation, and are therefore omitted.

As said earlier, the input to the implemented version is a valid Occam program, using any of the legitimate Occam constructs; the program must compile without error using the standard Inmos compiler. There are a number of minor restrictions: the program may include program folds, but separately compiled and filed folds are not allowed.

The output of the implemented version for a source (Occam) program consists of the formal description of a generated Petri-net structure in the format suitable for the Petri-net simulator. That format in the form of a data structure stored in a file contains the information about the number of place(s) of the Petri-net, the number of transition(s) of the Petri-net, input matrix (no. of arcs from place(s) to transition(s)), output matrix (no. of arcs from transition(s) to place(s)), and the initial marking of the Petri-net. The complete format of this data structure is already explained in chapter 4 (see section 4.5).

5.5 Conclusion

In this chapter a detailed account of the use of Lex and Yacc in the construction of compilers is given, since the system described in this thesis is largely based on these two utilities. It is also described in this chapter how Lex and Yacc specification can be constructed for a description of a simple programming language, namely Occam. It is also shown informally how a language implementor proceeds from a reference manual grammar to a Lex and Yacc specification. Although this process is straightforward for an experienced language implementor, it certainly requires effort and is error prone. A significant amount of training is required to be able to use them effectively. All the Occam language grammar is used in the specification of Lex and Yacc, but the action for some of their rules were ignored because it was beyond the aim of the thesis.

The aim is therefore to allow the user to input a reference manual grammar for a particular programming language (in this case Occam), and to generate code (Petri-net structure) for the front-end of the simulator for the Petri-net structure. In the next chapter the design and implementation of such a system is described.

Chapter No. 6
Occam Program Translator

6.1 Introduction

In this chapter the design of the translation system using Lex and Yacc is discussed. It is shown how implementation considerations affect the design, and how translation mapping t_p is introduced. t_p maps every Occam syntactic construct into its corresponding Petri-net Structure (PNS) and defines the linkage between these structures. This translation system is used for Occam 2 version but can be modified for other versions of Occam language.

It is desirable for automated translation of an Occam program into its PNS to be part of a Occam software development environment. The translation could be invoked as a compiler option. This translation process should fit into one of the phases of the compiler design[AHO 77][AHO 85]. A general compiler consists of a set of phases, with the first phase taking a source program as input and the last phase producing machine code as output. Figure 6.1 shows a compiler structured as five phases and two routines for error handling and table management. For more details on compiler construction the interested reader is referred to [AHO 85].

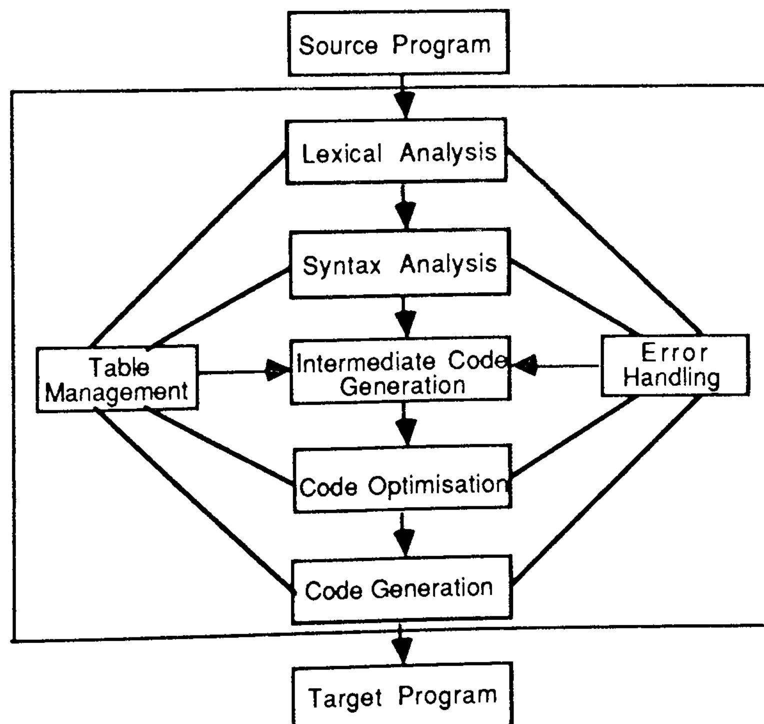


Figure 6.1: Phases of a Compiler

The Occam program translation is being considered as part of the intermediate code generation phase of a Occam compiler. Thus, if a programmer specifies the PNS translation option, then PNS is generated along with machine language code.
The Occam syntactic constructs directly affect the PNs translation process. The complete set of Occam syntactic constructs forms the Occam grammar, which is context-free. An efficient parser for such a grammar is an LALR(1) parser (see chapter 5 for details), a bottom-up parser. This parser scans its input string from left to right. Each Occam syntactic construct S_c is reduced when its right-hand-side is recognised from the currently scanned part of the input. When Sc is reduced, the PNS translation action associated with it takes place along with other code generation action of the compiler. This results in the generation of a part of the PNS associated with the S_c; control is then passed to S_c's parent construct. The process continues until the root of the parser tree is reached. The input Occam program is then either recognised or rejected. If it is recognised, the PNs corresponding to the program has been successfully constructed.
The parts of the net that are generated and linked together in PNS preserves the static behaviour of the input program.

6.2 Translation Overview by Example

Consider the following fragments from an Occam program.

```
Statement   program
0           CHAN OF INT channel:
1           WHILE TRUE
2             PAR
3               INT x:
4               SEQ
5                 channel ? x
6               INT y:
7               SEQ
8                 channel ! y
9           :
```

The translation program maps the program fragments into the Petri-net Graph (PNG) which is illustrated in figure 6.2(a). This PNG represent the proper behaviour of the Occam Program (Op). Places p_1 and p_{12} represents a WHILE statement. It is surrounded by t_1, t_{10} and t_{11} which are also a part of the WHILE representation.

The PAR statement is represented by p_2, p_{11}, t_2 and t_8. Places p_3, p_4, p_9, p_{10} and transitions t_3, t_4, t_6 and t_7 represents the assignment statements inside the PAR statement. The output statement(8) 'channel !' is represented by p_5, p_7 and t_5 while

p_6, p_8 and t_5 represents the input statement(5) 'channel ?'. T_5 is repeated in both input and output statements because it is the communication transition and both input and output statements execute in parallel through t_5. Therefore t_5 is called as communication channel.

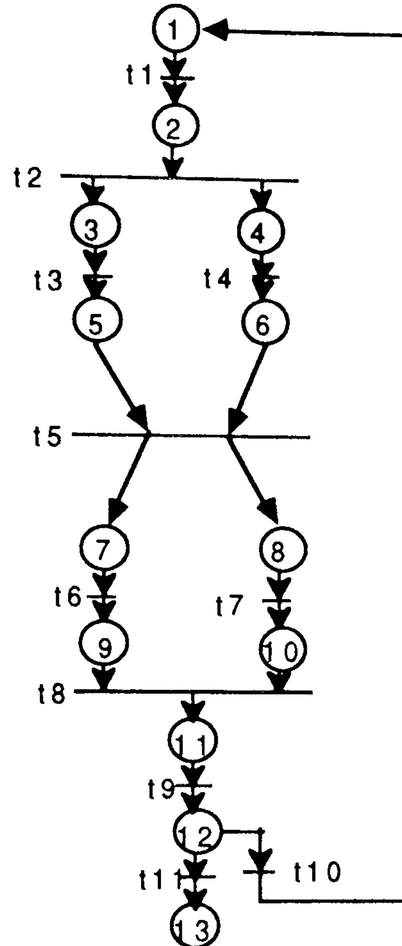


Figure 6.2 (a) PNG for the Occam program

Place p_1 is initially marked. Firing t_1, p_2 becomes marked. Now, t_2 is enabled and by firing t_2, places p_3 and p_4 both become marked. This is the condition of PAR. Now, the system is executing in parallel and both transition t_3 and t_4 are enabled. By firing these two transitions places p_5 and p_6 will become marked. Places p_5 and p_7 are communicating with places p_6 and p_8 through channel t_5. For t_5 to fire, it is necessary for p_5 and p_6 to hold tokens. This marking represents the situation when input statement(5) is trying to communicate with an output statement(8). When these statements are ready to communicate t_5 fires and removes tokens from places p_5 and p_6, and places them in place p_7 and p_8. When the communication between statement(5) and statement(8) terminates, places p_7 and p_8 ensure that t_8 does not fire until after t_6 and t_7 fire.

In order to simplify the discussion of Occam translation this particular program will be used as an example. The intuitive syntactic rules whose actions generate the different parts of the PNS are presented here. For illustration, the PNG version of the PNS will be given. The rules used here are the simplified version of the actual production rules applied to the Occam program. Thus, an intuitive syntactic rule closely resembles the format of its actual counterpart. The intuitive syntactic rules for the Occam program fragments follow next in the order they occur.

(1) Rule 'declaration: type namelist':'

concerns type declarations and is used to handle statement 0 CHAN OF INT channel:

This rule will pass control to another rule where 'type' and 'namelist' are defined and these are defined as follow:

```
"type :        CHAN OF protocols
        |      INT
        |      BYTE" etc and
"protocols :   ANY
        |      ID" etc
```

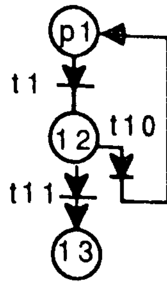
The 'type' recognises the type of channel defined. When it recognises the specific type, it records this information and move towards 'namelist'.

```
"namelist : ID
          | namelist comma ID"
```

Here it recognises the name of the channel defined and stores it as ID. If the namelist has more than one entry, separated by commas, each declared channel is recognised and stored away for further usage in the translation procedure.

(2) Rule "loop: WHILE expr sep BEG process END"

accepts statement(1) to statement(9) WHILE TRUE etc. Here statement 2 to 8 are accepted by the non-terminal 'process'.The rule generates p_1, p_{12}, p_{13}, t_1, t_{10} and t_{11}. Rule(3) then performs the linkage between its PNS and the outermost places and transitions(if they exist). The PNG generated by this rule is shown in figure 6.2(b).



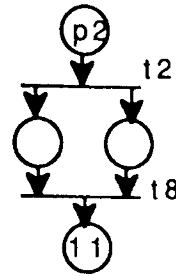
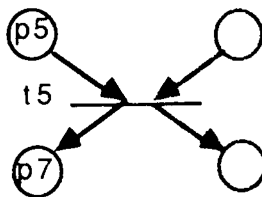


Fig. 6.2(b) WHILE loop Fig. 6.2(c) PAR construct

(3) Rule "parallel : PAR sep BEG process END"
accepts statement(2) and the remaining statements from 3 to 8 are accepted by the non-terminal 'process'. 'sep' represents the End Of Line(EOL) and BEG and END are used as start and end of an indentation respectively. This rule generates the PNG shown in figure 6.2(c). p_2, p_{11}, t_2 and t_8 are generated by this rule(3). It first counts the number of sub-processes in the first indentation and then creates the appropriate number of places. Rule(3) then performs the linkage between its PNS and the one created by rule(2).

(4) Rule "sequence: SEQ sep BEG process END"
accepts statement 4, 5 and statement 7, 8. This rule works with rule(5) and rule(6). In this specific example it does not generate any places or transitions.

(5) Rule " output: chan '!' outlist"
accepts statement(8). This statement generates half of the communication procedure and generates p_5, p_7, and t_5 shown in figure 6.2(d). The 'chan' is defined as an 'element' that invokes 'ID'. This 'ID' is then matched with the ID's that are stored earlier during statement(0) in rule(1).



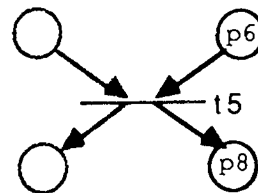


Figure 6.2(d) Output Statement Figure 6.2(e) Input statement

(6) Rule " input: chan '?' inlist"
accepts the 'channel ? y' statement(5). This statement completes the communication procedure and generates p_6, p_8. T_5 is already generated in rule(5), so here the linkage between the two generated PNS fragments is performed. The PNG

generated by this rule is shown in figure 6.2(e). As the communication procedure is complete now, the linkage of the fragments generated by rule(4), (5) and rule(6) with the main program (i.e. WHILE and PAR) is performed.

The end of the O_p (Occam program) is recognised by a ':', which is present in the first column of statement(9). The translation of O_p at this stage is complete and the generated PNG is shown in figure 6.2(a), but the translator will produce PNS instead of PNG as an output file which is discussed later in this chapter.

6.3 Op Translation in Detail

A PNS is generated by an action associated with some O_p syntactic construct. This generated PNS is an intermediate language. The O_p translation mechanism can be seen as a mapping that maps its domains of all the Occam programs into its range of the PNS. Formally, mapping is the set of actions associated with it . Such a rule is just an informal description of what that particular O_p action performs. It is called a mapping rule. The set of all mapping rules describes the complete O_p translation. Thus, in order to give a comprehensive description of Occam programs, O_p translation mapping, and PNS, all the syntactic constructs in the Occam grammar with their corresponding mapping rules are presented.

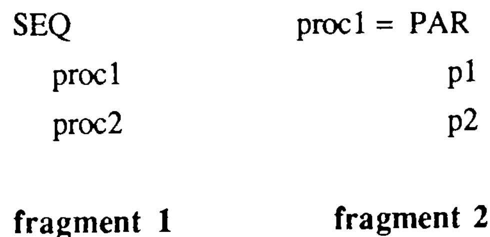
The PNGs can be reduced to its most reduced form without changing the behaviour of the system which it represents. The Petri-net reduction was considered by [DADD 76],[[BERT 76] and [BERT 80]. Any reduction in the complexity of these systems will decrease the sequence of operations and make analysis more treatable and help to verify a complex system easily[DADD 76]. Details for the Petri-net reduction with examples are given in section 7.2.3.

6.3.1 Op Translation Sub-routines

Since the applications mentioned in this thesis are specifically concerned with possible communication patterns in Occam programs, the original Occam grammar is targeted to this need. For example, arithmetic and some other features are completely ignored. The syntax of the program is checked by the translator because of the indentation. The indentation plays an important role in the Occam programming language and it can change the meaning of the program, sometimes, leading to unintended deadlock. So, indentation is checked by the translation rules. If there is anything wrong in indentation the program will not be recognised and no output will be generated by the translator.

The Occam yacc parser works with Lex programs. The parser receives lexical tokens from the Lex program, recognises syntactic constructs of the user defined language, and performs corresponding actions. For Yacc program, the language needs to be specified in the form of syntax grammar. The Occam grammar consists of sixty-one production rules. Some rules express only the Occam syntax constraints and do not take part in a Petri-net construction. Such rules do not have any mapping rules associated with them. Since every mapping rule is a result of one of the Occam syntactic constructs, a formal description of every syntactic construct and its associated mapping rule, if it exists, is given. The Occam rules are presented in a way that for any Occam parse tree, an N-level rule is discussed before any (N-1)-level counterparts. The root Occam rule of the parse tree is thus discussed last. This presentation method makes the discussion compatible with a bottom-up design, since in bottom-up design (see section 6.1) any production rule is reduced only when all the production rules on its right hand side (its children productions) have been reduced.

A few sub-routines are utilised in the mapping. The sub-routines mixmat, loadmat, writemat are needed by mapping rules. Subroutine mixmat plays an important role in the translation leading to the generation of a Petri-net structure. Now, the overall working of this subroutine can be understood by a simple example:



fragment 1 **fragment 2**

Here, fragment 1 can be translated as:

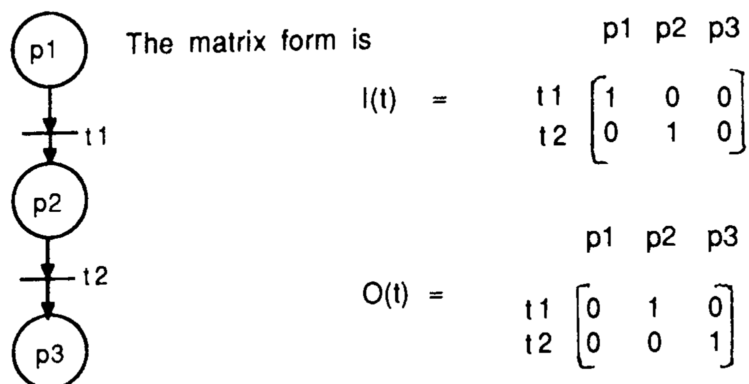


Fig. 6.3(a) fragment 1 representation

The translation of fragment 2 is as:

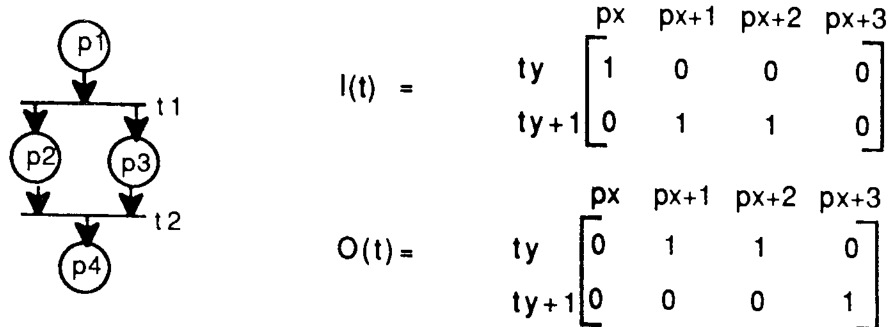


Fig. 6.3(b) fragment 2 representation

The subroutine mixmat() link the two matrices in such a way that the number of rows and columns are increased. Each time this subroutine is called, insert matrix is linked to the base matrix at some pre-defined location i.e. (x, y) where x is a column and y is a row.

During insertion procedure, the base matrix is always increased by a row of zero elements. This way the overwriting of an actual base matrix elements can be avoided. For example, base matrix

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad \text{is converted into} \quad \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

The location of this zero row depends on the insertion points (x, y). If the insertion points are in third row then the base matrix will be like

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

This process is repeated again and again until the root of the input program is reached. The resulting matrices grow and the number of each row and column increases after every execution of mixmat() procedure.

In this case the insertion point is (2, 2) i.e., x=2, y=2. The complete program is obtained by combining fragment 1 and fragment 2.

```
SEQ
  PAR
    p1
    p2
```

proc2

Now the base matrix will become as:

$$I(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \boxed{0} & 0 \\ 0 & 1 & 0 \end{bmatrix} \qquad O(t) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & \boxed{0} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The insertion point in both I(t) and O(t) matrices are boxed i.e., (2, 2). The resulting matrices are:

I(t) =

	p1	p2	p3	p4	p5	p6
t1	1	0	0	0	0	0
t2	0	1	0	0	0	0
t3	0	0	1	1	0	0
t4	0	0	0	0	1	0

O(t) =

	p1	p2	p3	p4	p5	p6
t1	0	1	0	0	0	0
t2	0	0	1	1	0	0
t3	0	0	0	0	1	0
t4	0	0	0	0	0	1

Figure 6.3(c) Matrix representation of the program

As the Occam translator reads an Occam program and translates that Occam program into its Petri-net structure in the form of a matrix, this subroutine is used to link different submatrices and generate one big matrix which represents a complete input (Occam) program. Diagrammatically,

Mixmat() --> base [a][b] ⊕ insert[i][j] => result[a+i+1][b+j]

Where base is the base Matrix and insert is the insertion matrix giving mapping rules to describe the complete Op translation. The ⊕ symbol represents the mixing of the two matrices in the manner described earlier.

The loadmat() subroutine, simply loads a file where already created matrices reside. The mixmat() procedure is executed on that file and each time it generates a new resulting matrix. These matrices are then written to the file by another subroutine that is called writemat(). This writemat() subroutine writes the matrices to the file in a form shown in figure 6.3(c).

6.3.2 Op Translation rules

The Occam rules with their associated mapping rules are now presented. PRxx denotes the current Occam production rule, where xx is its sequence number (for example, PR48 denotes forty eighth production rule in the Occam grammar). The

currently considered mapping rule will be denoted by Ryy, where yy is the sequence number of its production. Thus, for example R48 is the mapping rule associated with production rule PR48. This technique will help us to keep the link between a production rule and its mapping rule. The name of terminals are capitalised. The names of non-terminals are given in lower case.

There are seventy-two (72) production rules in total that are given in the appendix B but here in the translator only sixty-one (61) rules are used for the translation procedure. As only those Occam constructs are translated that are involved in the communication between the processes, therefore, the remaining rules were omitted for simplicity. For every production rule there can be more than one mapping rule, in which case any mapping rule can execute at one time. A two column approach has been adopted to explain production rules. One column shows the rules and the other column shows its associated actions.

PR1: program: sep process or program: process	**R1**: This rule represents the complete structure of the input program. Here 'process' is divided into many subprocesses each defined by its production rule. This rule produces two places and two transitions. These are the outer-most places and transitions that are needed for every program conversion and is shown in figure 6.4.

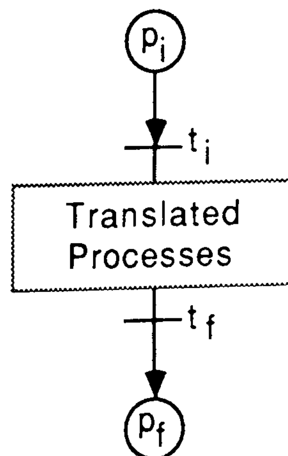


Figure 6.4 Outer-most places and transitions

PR2: process: action sep
or SKIP sep
or STOP sep
or CASE selector sep
or CASE selector sep
BEG selectlist END
or construct
or instance
or specification
or specification sep process
or error sep

All the non-terminals in PR2 are defined in the following production rules until the terminal state is reached.

PR3: action: assignment
or input
or output

R3: The three actions in Occam are assignment, input statement and output statement. The structure of the process is based on these actions. All the actions are explained later on in the following rules.

PR4:selector: expr

/* expression */

PR5: selectlist: select
or selectlist select

PR6:select: expr sep BEG process END
or ELSE sep BEG process END

PR7: instance: ID '('actualist')' sep
or ID '('')' sep

PR8: actualist: actual
or actualist comma actual

PR9: actual: element
or expr

PR10: construct: sequence
or parallel
condition
or alternation
or loop

R10: The constructs define all the or possible ways operations can be combined in a language. In Occam, these operations are the actions: operate in sequence, in parallel, under specified condition, in alternation and in loop. All these constructs are explained in the following rules.

PR11: sequence: SEQ sep
BEG proclist END

or SEQ replic sep BEG process END
or SEQ sep

R11: This production rule when recognised produces two places and two transitions. When considering net reduction this production rule can sometimes be skipped, but not in all cases. Therefore, the mapping rule is executed and performs the following routines.

seq_stat() subroutine
and mixmat() subroutine.

```
{
 perform seq_stat() subroutine
 perform mixmat() subroutine
}
```

The new created matrices are overwritten in the old file which is then ready again to be used by another production rule.

PR12: parallel: PAR sep parproc5

R12: This production rule as stated earlier in rule (3) in section 6.2 finds

out the number of processes executing in parallel. If for example, the number of processes executing in parallel are four then it will invoke the action for that corresponding production rule and the matrices will be generated. Here, this rule recognises the number of processes executing under PAR construct. This rule recognises five processes that are executing in parallel. The following subroutines are executing:

```
loadmat();
par5_stat();
mixmat();
```

The PAR construct can execute a maximum of five processes here. This is done because the very complex example shown in chapter 7 has only five processes executing in parallel. Further extension is possible to any number of processes by simply adding another rule and its action into this set of rules.

or PAR sep parproc4

{
This rule recognises four processes that are executing in parallel and generates the corresponding actions. These actions invoke the following subroutines:

```
loadmat();
par4_stat();
mixmat();
```

}

or PAR sep parproc3

{
This rule recognises three processes that are executing in parallel and generates the corresponding actions.

These actions invoke the following subroutines:

loadmat();
par3_stat();
mixmat();

}

or PAR sep parproc2

{

This rule recognises two processes that are executing in parallel and generates the corresponding actions. These actions invoke the following subroutines:

loadmat();
par2_stat();
mixmat();

}

The minimum number of processes which execute under PAR are two.

or PAR sep parproc
or PAR replic sep BEG process END
or PAR sep

PR13: parproc5: parproc3 sep parproc2

R13: This production rule and the following four rules are used in the parallel production rule to define the structure of the PAR constructs. This rule is only recognised when the number of processes executing in parallel are five. The following three rules are recognised when the number of process executing in parallel are four, three and two respectively.

PR14: parproc4: parproc3 sep parproc

PR15: parproc3: parproc2 sep parproc

PR16: parproc2: parproc sep
BEG proclist END

PR17: parproc: BEG proclist END

PR18: proclist: process
or proclist process

PR19: conditional: IF sep
BEG choicelist END

R19: This rule produces matrices that represent four places and four transitions. The mapping rule executes and loads a file that already exists. The following routines are then performed.

loadmat();
if_stat() routine
mixmat() routine

The output file is then stored away for further modification (if any).

or IF replic sep BEG choice END
or IF sep

PR20: choicelist: choice
or choicelist choice

PR21: choice: boolean sep
BEG process END
or specification sep choice
or conditional

PR22: alternation: ALT sep
BEG alternative_4 END

R22: This production rule when recognised produces the matrices which represent eight places and ten transitions. This rule in fact represents the ALT condition when out of four guarded processes one is to be executed. The matrices generated represent those four guards. Every guard is then followed by a process or an action which is considered in other corresponding PRs. The following two options of this rule do the same work except that these rules are recognised only when the number of guards under ALT construct are three and two respectively. The subroutines that executes are as follow:

```
loadmat()       /* load the matrices
                   from the file */
alt4_stat()     /* load matrices that
                   represent four ALT
                   conditions */
mixmat()        /* mix the two loaded
                   matrices and write
                   new matrices to file */
```

or ALT sep BEG alternative_3 END

```
{
  loadmat();
  alt3_stat();  /* load matrices that
                   represent three ALT
                   conditions */
  mixmat();
}
```

or ALT sep BEG alternative_2 END

```
{
  loadmat();
```

	alt2_stat() /* load the matrices for two guards under ALT */
	mixmat();
	}

or ALT replic sep BEG alternative END	
or ALT sep	
PR23: replic: ID '=' base FOR count	
PR24: base: expr	
PR25: count: expr	
PR26: alternative_4: alternative_3 sep alternative	/* recognises 4 ALT constructs */
PR27: alternative_3: alternative_2 sep alternative	/* 3 ALT constructs are recognised */
PR28: alternative_2: alternative sep guard sep BEG process END	
PR29: alternative: guard sep BEG process END or specification sep alternative or alternation	
PR30: guard: boolean '&' input or input or boolean '&' SKIP	
PR31: boolean: expr	
PR32: loop: WHILE expr sep BEG process END	

R32: This rule when recognised generates three places and three transitions. After this generation of places and transitions by this rule the linkage is performed between the WHILE loop itself and the process which is executing in the WHILE loop. A new PNS is formed and then the linkage between this new PNS and the main PNS is performed. The following subroutines are executed in the order given below.

loadmat() subroutine
while_state() subroutine
mixmat() subroutine

The newly formed PNS is then written to the file which is then ready for any further modification.

PR33: sep: EOL

or sep EOL

R33: This rule always recognises the token 'sep' which represent the end of line.

PR34: comma: ',' EOL

or ','

R34: This rule recognises ','.

PR35: semicolon: ';' EOL

or ';'

R35: This rule recognises ';'.

PR36: specification: declaration

or definition

PR37: declaration: type namelist ':'

R37: This production rule reads all the declarations in the program. The important declaration from the point of view of this research is the channel declaration because it plays a vital role in the communication. In this rule all the declared channels are scanned and stored away for further use in 'input' and 'output' statements. Every channel name is stored with a different identifier(ID).

PR38: namelist: ID
or namelist comma ID

PR39:definition: PROC ID '('fparmlist')'
sep BEG process END ':'

R39: This rule defines the complete structure of an Occam program and makes sure that proper indentation has been implemented. ID takes the name of the process while the 'process' can be any number of processes consisting of constructs and actions. The other options do the same work except the definition of the program is different.

This rule looks similar to PR1 but here it only recognises the syntax of one complete process. The PR1 production rule recognises the complete Occam program which can be composed of more than one process.

or PROC ID '('')' sep
BEG process END ':'

PR40: tag: ID

PR41: protocol: ANY

or ID

or INT

or BOOL

/* These are different possible type of channels */

PR42: fparmlist: fparm

or fparmlist comma fparm

PR43: fparm: type ID

or VAL type ID

PR44: assignment: varlist ':' '=' explist

R44: This rule when recognised produces the matrices for two places and one transition. These generated places and transition do not play any part in the communication directly; therefore these can be omitted in the reduced procedure. In some cases, when two construct like WHILE and IF or IF and ALT comes one after the other and need to be represented separately then this production rule is used. The rule performs the following routines:

loadmat() /*load matrices from file*/

assign_stat() /* load matrices for this rule */

mixmat() /* mixes the two loaded matrices and write these to the file */

PR45: explist: expr

or explist comma expr

PR46: varlist: var

or varlist comma var

/* variable or a list of variables*/

PR47: input: chan '?' inlist

R47: This is one of the two production rules that is responsible for the communication between the channels. A 'chan' is the name of the channel by which signal or data is received. As mentioned in rule (6), this rule produces half of the channel communication. It generates the matrices for this input statement and store it in the memory register for further usage. A special token is assigned each time for different channel names. The following subroutines are executed in the given order:

```
loadmat();
input_stat();
mixmat();
```

PR48: inlist: var
or var ':":' expr
or inlist semicolon expr

PR49: output: chan '!' outlist

R49: This production rule completes the second part of the channel communication. The working of this production rule is mentioned in rule (5). When recognised this rule looks for the pre-produced matrices that are stored in the memory registers and linkage is performed between the two channel communication statements. The sequence of appearance of input

and output statements depends on the specification of the program. The linkage of these rules is performed when both input and output statements are recognised and mixed together. The newly produced matrices by mixing the matrices for input and output statements are then linked to the main program by the mixmat() subroutine.

or chan '!' tag
or chan '!' tag semicolon outlist

PR50: tag: ID /* identifier */

PR51: outlist: expr /* expression that is used by channel */
or expr ':":' expr
or outlist semicolon expr

PR52: var: element

PR53: chan: element

PR54: element: ID /* identifier */
or element '['subscript']'
or '['element FROM subscript TO subscript ']'

PR55: subscript: expr

PR56: expr: monop operand
or operand dyop operand
or monop sep operand
or operand dyop sep operand
or operand
or MOSTPOS type

or MOSTNEG type

PR57: monop: '_'
or NOT
or SIZE
or '~'

PR58: dyop: COMPOP
or '='
or SHIFTOP
or '+'
or '*'
or LOGOP
or BOOLOP
or '/'
or '\\'

/* different operations are defined these are compare, shift, logical and boolean */

The above production rule shows all the logical operations.

PR59: operand: literal
or '(' expr ')'
or '[' explist ']'
or ID '(' explist ')'
or ID '('')'

PR60: literal: NUMBER
or BOOL
or RNUMBER
or CHCON
or STR
or NUMBER '(' type ')'
or RNUMBER '(' type ')'
or CHCON '(' type ')'

/* data conversion type */

PR61: type: CHAN OF protocol
or PORT OF type
or TIMER

/* full range of primitive types */

or BYTE

or INT

or INT16

or INT32

or INT64

or REAL32

or REAL64

The above list contains all the production rules; only those rules that are used in the translator were explained in detail.

6.3.3 Occam LEX program

The Occam Lex program works with the Yacc program to identify the terminal tokens. The complete working of the Lex in conjunction with Yacc is given in chapter 5 (section 5.3). Here in this section a brief description of the Lex working program is given. In this program the end of Line (EOL), comma (,), semicolon (;), logical operators are defined.

As mentioned earlier, Occam uses indentation to separate the processes of the Occam program from one another. The Lex program reads the indentation by two white spaces and translate it into a token. If it reads forward two white spaces it returns a BEG token and if it reads backwards two white spaces it returns an END token. The complete Lex specification for the Occam language is given in appendix B.

6.4 Octool Limitations

Every tool and every piece of software has constraints under which it works. The Octool has also limitations beyond which the tool does not work. The Occam language used here for translation should be used in a restricted manner. This section identifies the restrictions that are imposed on the standard Occam 2 language.

1) The input program must not contains folds and file folds. The program must contain one main procedure and all other processes should work under that procedure. Subprocedures are not to be included in the main program.

2) All the channel declarations should be at the start of the program because while scanning the main program the translator assign tokens to every channel name.

3) All the channels should be straightforwardly defined without any array channels.

4) The PRI PAR type of statements are also not allowed to be used in the restricted Occam 2.

The PAR rules are restricted to five parallel processes and can to extended to more than five papallel processes as explained earlier. Another approach could be where the number of processes are 'n'. The number 'n' is first to be evaluated and then a rule appropriate to the evaluated 'n' can be invoked.

All these limitations are minor and do not effect the communication constructs. Thus, concurrent systems using the basic constructs that are allowed in the restricted Occam can be modelled. These can then be analysed using the Petri-net simulator. The Occam BNF code given in appendix C can be extended and the translator modified to eliminate restrictions 3 and 4; this can be done as a future work as explained in chapter 8.

6.5 Conclusion

The translator presented in this chapter was the second approach in the translator design given in this thesis. Lex and Yacc utilities were successfully used in the design of the translator. During the scanning of the input program the production rules when recognised cause an appropriate action to be taken. The Petri-net structure was generated step by step but the whole procedure is invisible to the user/programmer.

A number of limitations of the translator were reported in this chapter. Therefore, the Occam program that is used for translation and simulation should be specified in a restricted manner, so that it can be completely translated (into a Petri-net Structure) and simulated (by the simulator). Two complex Occam programs were taken; these were converted into restricted Occam, and then were translated for simulation to generate the reachability tree. These two examples with their Petri-nets and reachability tree are given in the next chapter in detail as an application of the Octool.

Chapter No. 7
Octool Application

7.1 Introduction

A distributed system comprises processes executing in parallel which are interconnected by a communication network. Such a system relies on the integrity of inter-process communications along communication links so that individual processes can cooperate in observation, control and the provision of the correct overall system function[CARP 88a]. Octool (Occam translator and Petri-net simulator) can be used for fault avoidance in the software for distributed systems and has been explained in detail in the previous chapters. Its use was illustrated using small programs.

Almost, every tool has its application in the area for which it is designed. Here, in this chapter, two complex examples have been considered in detail as an application of an octool. As a first example an experimental vehicle detection system has been chosen to show that software for this system is deadlock-free and as a second example, the conversation mechanism based software was chosen to show that it is also deadlock-free. By using the Occam translator and Petri-net simulation, it has been shown that these examples (programs) are deadlock free. Occam programs were taken and were completely translated in a Petri-net graph. An execution path was chosen to exercise fully the Petri-net. After execution the simulator can generate the reachability tree where by examining the terminal nodes (which are marked as 'terminal') the occurrence of deadlock (if any) can be found.

7.2 Example: An experimental vehicle detection system

The complexity of this example made it challenging as an application for Octool. The complete system is described later on in the next section but here only those properties of the system are discussed which are important from the application point of view. The vehicle detection system is both safety critical and time critical; therefore a number of properties for this system need to analysed.

1) It is communication intensive; therefore the communication between the processes should never go wrong and should always work.
2) It cannot be tested 'live'; therefore the program must be analysed before it is ever run.
3) The programmer needs to know that the program will never deadlock under any set of vehicle movements.

To analyse the program for the above mentioned properties Occam tool can be used. The Occam program can first be translated into Petri-net. The translated Petri-net can then be input to the simulator to test design pathologies, which only become noticeable when processes interact. The reachablity set is generated which is represented as a tree. The number of marking in the reachablity set is finite, and inspection of the reachability tree allows the designer to determine the properties of the graph and to infer the dynamic behaviour of this system. It is therefore possible to determine which transition sequences are live and whether any given marking is reachable.

The number of reachable markings can be very large. To keep the analysis manageable, it is often necessary to apply reduction techniques to reduce the complexity of the graph. In this research the work of [BERT 79][JANT 80] has been considered and from it a number of useful reductions can be derived and applied to the examples. This technique is also applied to the system and was found very useful to reduce the complexity of the net without losing the information of the system behaviour.

7.2.1 Overview

This example [CARP 89] shows the design of an experimental vehicle control system suitable for use on railways, with particular emphasis on the software involved in monitoring vehicle movement to and from the protected section of track as shown in figure 7.1.

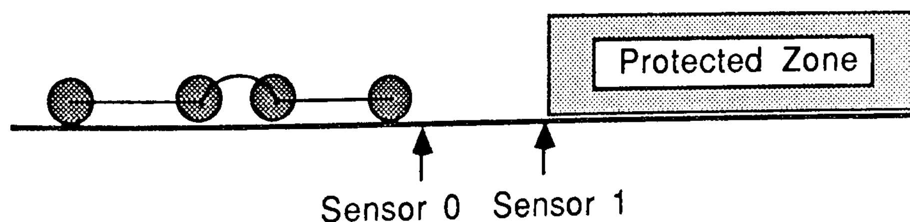


Figure 7.1 Vehicle detection system

A trackside-mounted process is required to monitor access to the restricted section of the track, called the 'protected zone'. There is a single entry and exit point to that zone, through which all the traffic must pass. The process is required to compute the numbers of wheels in the zone, to obtain a measure of the speed at which the vehicle enters or leaves the zone and to pass information instantly to a remote controller. The remote controller is responsible for process initialisation and termination. Wheel-detecting sensors are provided which produce a signal

whenever a wheel passes over them and which hold that signal until read. Two sensors are used for speed determination, positioned as shown in figure 7.1.

The distance between the two sensors is arranged to be smaller than the smallest inter-axle spacing of a vehicle or train of vehicle. The section of track to the right of sensor 1 is deemed the protected zone. For the purposes of this exercise the subsystem will compute the time which elapses between a wheel passing over sensor 0 and sensor 1 and send this to the remote controller as a measure of the speed of the train. The system must therefore be capable of handling four counting rules:

- a wheel traverses completely from left to right, entering the zone.
- a wheel traverses completely from left to right, leaving the zone.
- a wheel enters from the left, halts between sensor 0 and 1, then moves to the left, this wheel deemed as never entering the zone.
- a wheel enters from the right, halts between sensor 1 and 0 then moves to the right; this wheel is deemed as never leaving the zone.

A functionally distributed solution is effective to represent this system and Occam is used to show the inherent parallelism of the system. Event detection and signal conditioning for each sensor is an independent activity and could be carried out in parallel as functionally distinct activity; likewise the counting and time calculations are independent and could be performed in parallel. The design comprises a set of largely autonomous processes, synchronised and sequenced through communication channels. The system is communication intensive for which it is necessary to use the Petri-net methods to demonstrate freedom from dynamic faults. Under steady-state conditions, the design relies upon six separate, concurrently executing processes:

- two processes, one for each sensor for wheel detection. These processes are 'sensor[0]' and 'sensor[1]'.

- a process that counts the number of wheels (process 'counter')

- a process that detects the speed of the vehicle (process 'speed')

- a process which simulates the action of the remote controller by issuing initiation and termination messages (process 'remotecontroller')

- a process which handles the receipt of data messages (process 'remotereceiver')

All these processes have to be initiated properly, executed in the steady state conditions, and then be terminated properly. The steady state condition is the normal state and so is used to define the major software structures.

7.2.2 Petri-net Conversion

The complete Occam program can be translated into a tree structure and is shown in figure 7.2. This tree represent an Occam program by its control flow. The data flow is not present in the tree and therefore, the tree shows only the channels and their actions. The tree shows 1-9 levels and each level represents a new level of indentation. It can also be seen in the tree that if all the channel names are replaced by transitions and all the lines by places, it will be equivalent to the Petri-net graph shown in figure 7.3. A further development in this approach is possible by making the conversion from program to tree automated. In this way it is easier to understand the program and the processes that communicate with each other can be easily recognised.

The vehicle control system is represented in the form of the Petri-net shown in figure 7.3. Initially, after translating the Occam program into the Petri-net the net was very big consisting of 195 places and 149 transitions. It is very difficult to analyse such a large Petri-net because of the large number of places and transitions. The difficulty of the analysis task grows combinationally with the number of elements in the net It was therefore decided to reduce the Petri-net by the methods described in the next section.

7.2.3 Petri-net Reduction

Petri-net reduction is used to transform a net into an equivalent net, which is normally smaller. The transformation is performed by the reduction rules that preserve certain properties such as liveness, reachability, and boundedness. Therefore, the user knows for sure that the original net and the reduced net are equivalent with respect to these properties. Figure 7.4 and figure 7.5 show two examples which are a part of the Petri-net shown in figure 7.3. In first example, three processes are executing in parallel and these processes are not communicating with any other process. The Occam program is shown as

```
Level 1 | Level 2 | Level 3 | Level 4 | Level 5 | Level 6 | Level 7 | Level 8 | Level 9

PAR
  PAR
    SEQ
      sensorinit[0] ? (1)
      readsensor :=
      WHILE
        ALT
          sensorinput[0] ? (4)
            PAR
              counterchan[0] ! (6)
              SEQ
                time ?
                speedchan[0] ! (7)
          haltsensor[0] ? (5)
            PAR
              readsensor :=
              haltspeed[0] ! (8)
              haltcounter[0] ! (9)
    SEQ
      sensorinit[1] ? (2)
      readsensor :=
      WHILE
        ALT
          sensorinput[1] ? (10)
            PAR
              counterchan[1] ! (12)
              SEQ
                time ?
                speedchan[1] ! (13)
          haltsensor[1] ? (11)
            PAR
              readsensor :=
              haltspeed[1] ! (14)
              haltcounter[1] ! (15)
  SEQ
    counterinit ? (22)
    PAR
      counting[0] :=
      counting[1] :=
      count :=
    WHILE
      SEQ
        ALT
          haltcounter[0] ? (9)
            counting[0] :=
          haltcounter[1] ? (15)
            counting[1] :=
          counterchan[0] ? (6)
            ALT
              haltcounter[0] ? (9)
                counting[0] :=
              haltcounter[1] ? (15)
                counting[1] :=
              counterchan[1] ? (12)
                count:=
              counterchan[0] ? (6)
                SKIP
          counterchan[1]? (12)
            ALT
              haltcounter[0] ? (9)
                counting[0] :=
              haltcounter[1] ? (15)
                counting[1] :=
              counterchan[0] ? (6)
                count:=
              counterchan[1] ? (12)
                SKIP
        counteroutput ! (23)
    haltcounteroutput ! (24)
  SEQ
    speedinit ? (25)
    PAR
      readspeed[0] :=
      readspeed[1] :=
      timelapse :=
    WHILE
      SEQ
        ALT
          haltspeed[0] ? (8)
            readspeed[0] :=
          haltspeed[1] ? (14)
            readspeed[1] :=
          speedchan[0] ? (7)
            ALT
              haltspeed[0] ? (8)
                readspeed[0] :=
              haltspeed[1] ? (14)
                readspeed[1] :=
              speedchan[0] ? (13)
                timelapse:=
              speedchan[1] ? (7)
                timelapse:=
          speedchan[1] ?
            ALT
              haltspeed[0] ? (8)
                readspeed[0] :=
              haltspeed[1] ? (14)
                readspeed[1] :=
              speedchan[0] ? (7)
                timelapse:=
              speedchan[1] ? (13)
                timelapse:=
        speedoutput ! (26)
    haltspeedoutput ! (27)
  *
```

(Continued to next page)

Figure 7.2 Tree structure for an Occam program

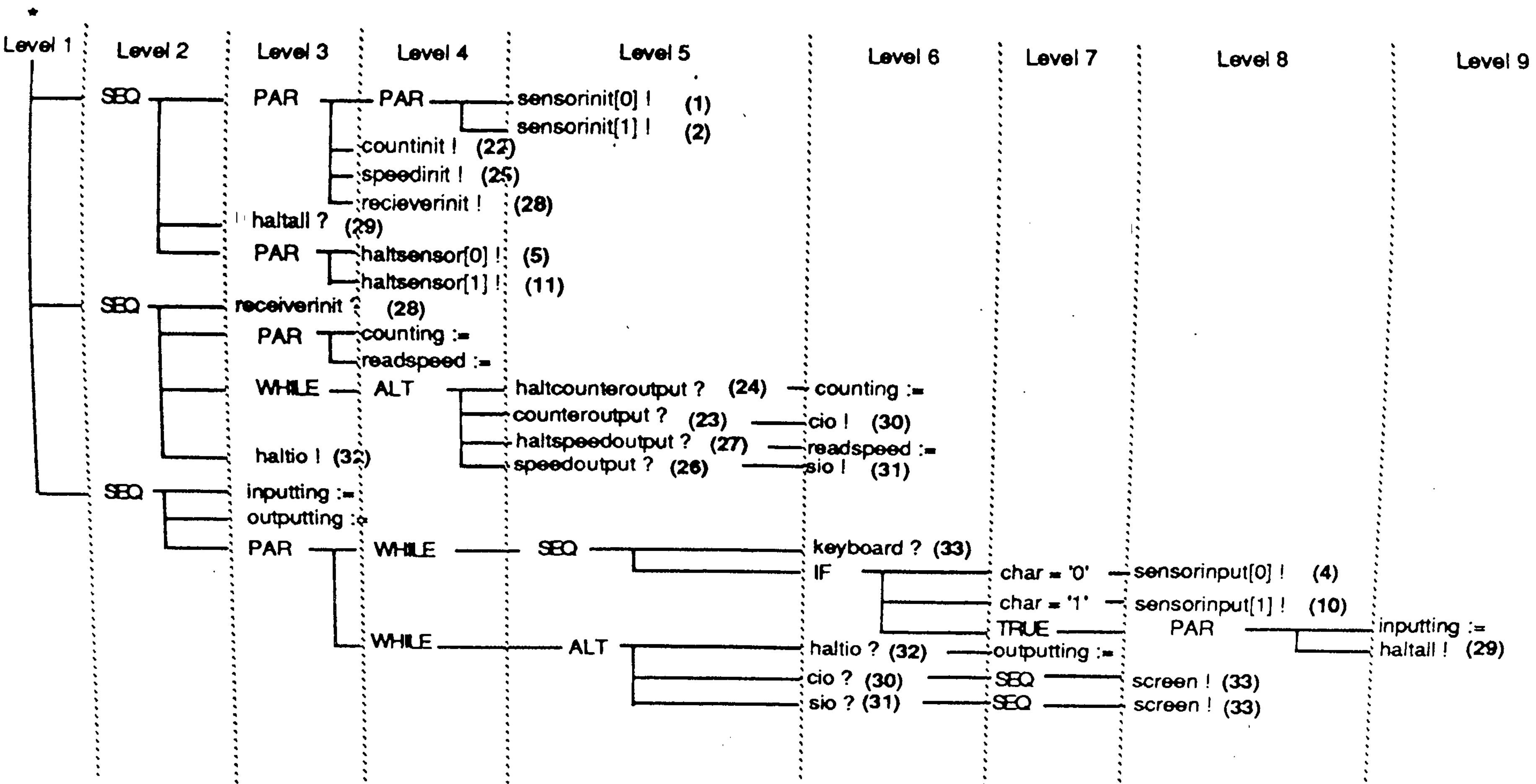


Figure 7.2 (continued) Tree structure

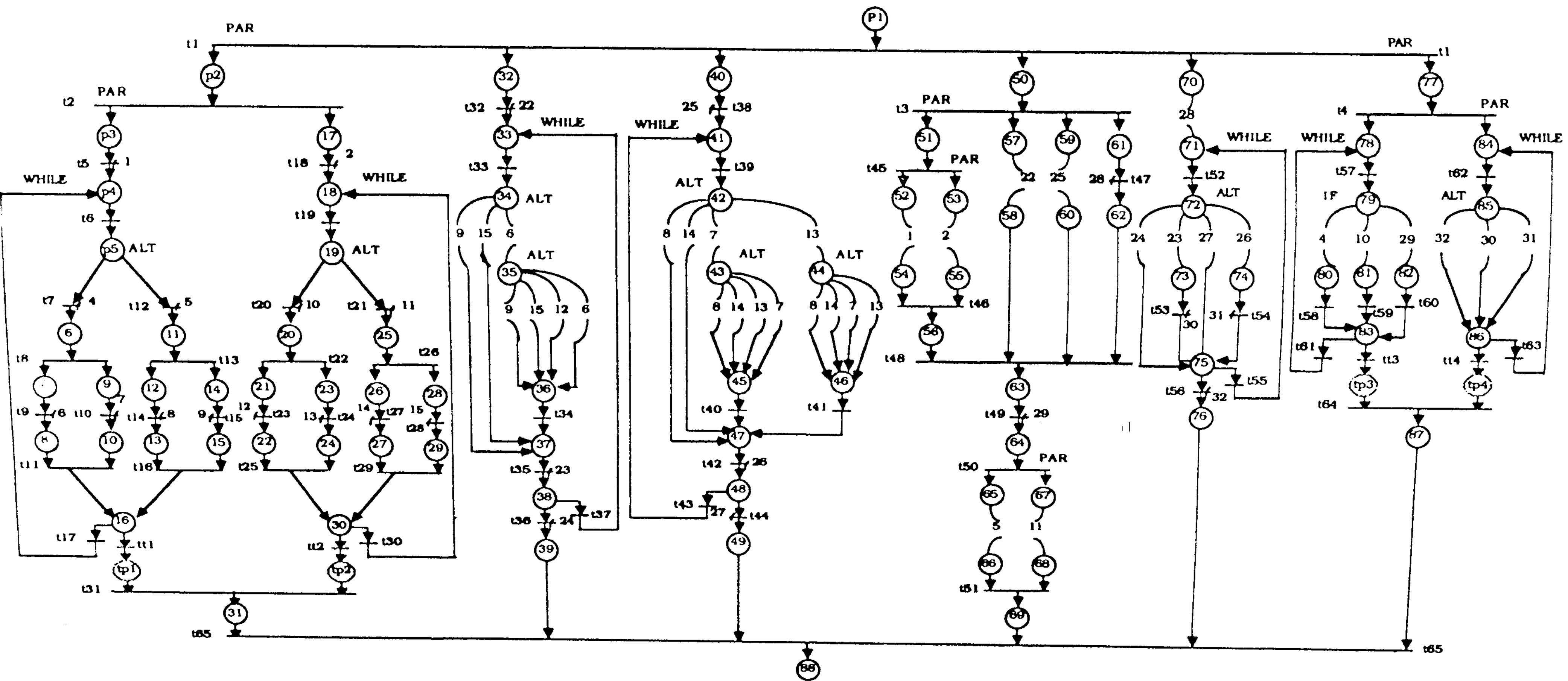


Fig. 7.3 PNG showing vehicle detection system

```
PAR
    counting[0] := TRUE
    counting[1] := TRUE
    count := 0
```

and the Petri-net representation for the above program fragment is:

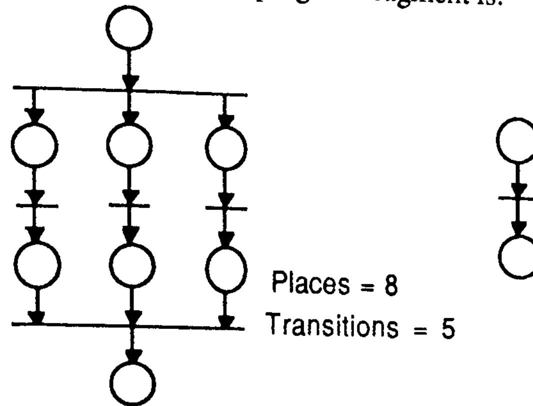


Figure 7.4 PAR reduction example

It is clear that the program fragment contains no input action or output action. It may be reduced to two places and one transition. Figure 7.4 shows the complete Petri-net representation and the reduced form of the given Occam program fragment.

Another program fragment used in the vehicle control system program is given below. This fragment uses a nested ALT construct.

```
ALT
    haltcounter[0] ? any
      counting[0] := FALSE
    haltcounter[1] ? any
      counting[1] := FALSE
    counterchan[0] ? any
      ALT
        haltcounter[0] ? any
          counting[0] := FALSE
        haltcounter[1] ? any
          counting[1] := FALSE
        counterchan[1] ? any
          count := count + 1
        counterchan[0] ? any
          SKIP
    counterchan[1] ? any
      ALT
        haltcounter[0] ? any
          counting[0] := FALSE
        haltcounter[1] ? any
          counting[1] := FALSE
        counterchan[1] ? any
          count := count - 1
        counterchan[1] ? any
          SKIP
```

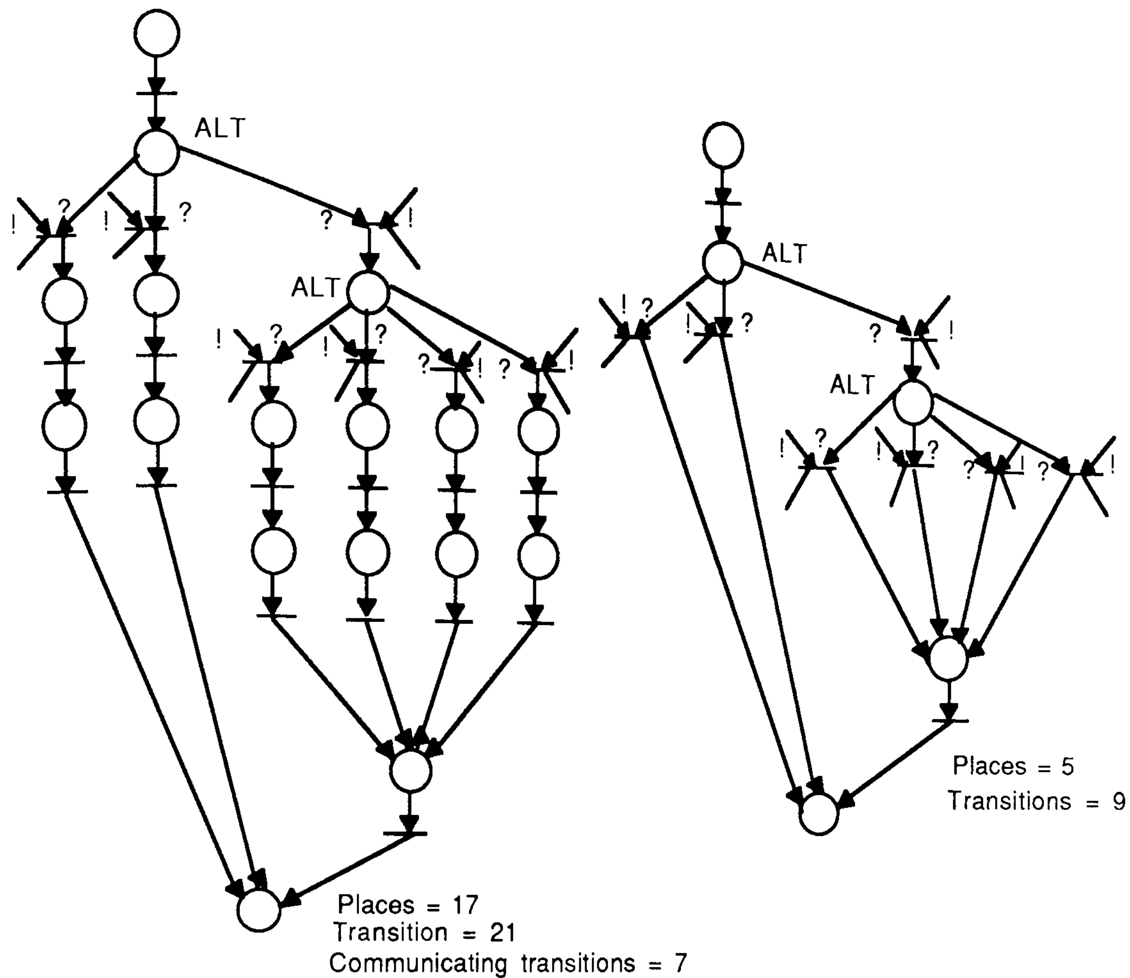


Figure 7.5 Reduction example for ALT construct

The Petri-net of this program fragment is given in figure 7.5. The interprocess communication cannot be reduced; however the subsequent place-transition sequences can be reduced. The reduced Petri-net is also given in the same figure. Translating the program fragment into the Petri-net gives 17 places and 21 transitions from which 7 transitions are communicating transitions. After reduction the Petri-net is reduced to 5 places and 9 transitions.

7.2.4 Reachability Tree

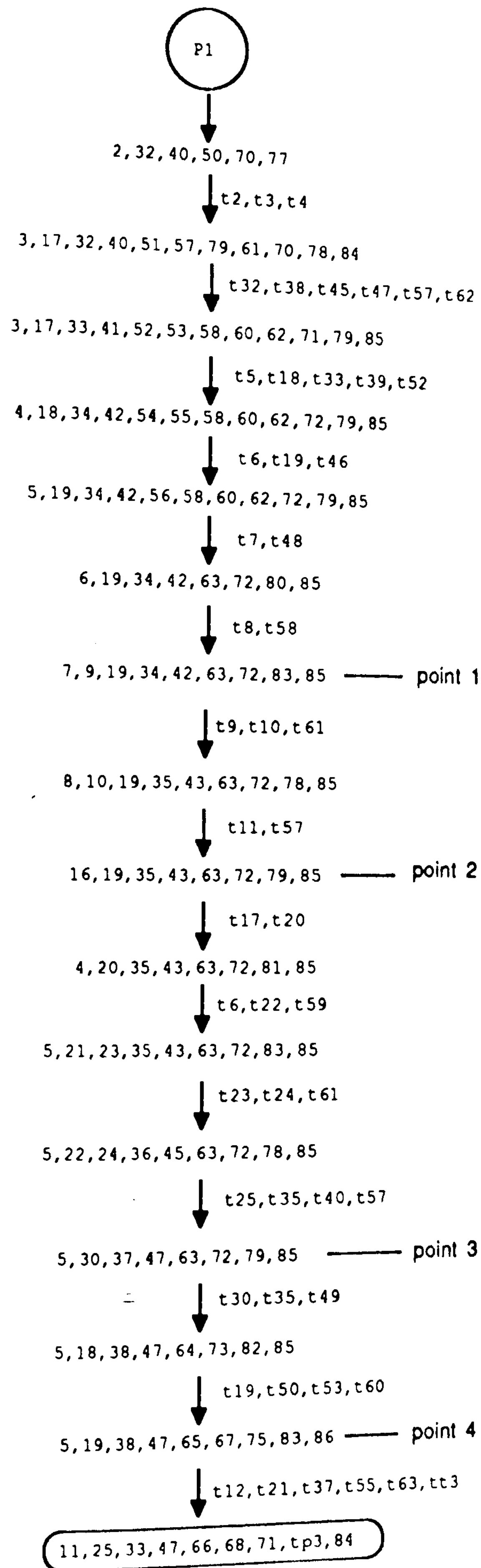
The reachability tree shows those markings which are potentially reachable from the initial marking. From the formulation of the reachability tree, the behaviour of the Petri-net, and therefore of the modelled system can be analysed. Figure 7.6 is the reachability tree for the Petri-net shown in figure 7.3. This reachability tree ends up in three paths where one path ends up in duplicate sequence, the second in deadlock while the third is the successful path of execution.

In the tree, at point 5, t_{17} is enabled which represents the WHILE loop in the program. In the Occam program this loop will terminate when the condition is not true. In the Petri-net, the firing of this transition will lead to a duplication of the tree for every repetition of the WHILE. Therefore, a temporary transition(tt) and temporary place(tp) are introduced after a WHILE loop. If this place and transition are introduced permanently then at point 2 the transition tt_1 is enabled and on firing will lead to deadlock. This is all because the boolean condition cannot be interpreted in the WHILE. Similarly if at point 4, t_{61} is enabled, firing this transition will lead to the duplication of the tree; temporary transition tt_3 and temporary place tp_3 are introduced and tt_3 is fired to explore the subsequent behaviour.

Transition tt1, tt2, tt3, and tt4 in figure 7.3 are enabled at point 2, point 3, point 1, point 4 respectively. If these transitions are fired deadlock is inevitable unless all loops exit after a matched number of interprocess communications. The boolean logic on the WHILE condition ensures this does occur, but this cannot be captured by a Petri-net. In the reachability tree, a few transitions are shown with positive (+) sign which means that these transitions fire one after the other. For simplicity it is shown this way to make the tree short. A default (one) token was used to execute the Petri-net all the way from place p1 to place p88. By following the successful path of execution, it was found subject to the arguments on the WHILE boolean conditions that the Petri-net is live.

7.2.5 Results

The complete Occam program for the vehicle control system is given in appendix A. The tree shown in figure 7.2 also shows the same Occam program but the control flow is only considered. The Occam translator was used to translate this Occam program into its Petri-net structure (the Petri-net graph is given). The overall translation of the program into its Petri-net is very big; therefore, reduction techniques were used to reduce it to a minimum level. The Petri-net simulator was used to generate the reachability tree given in figure 7.6. The reachability tree identifies 'deadlock-potential' which the user can explore further.



Continued....

Figure 7.6 Reachability Tree for Fig. 7.3

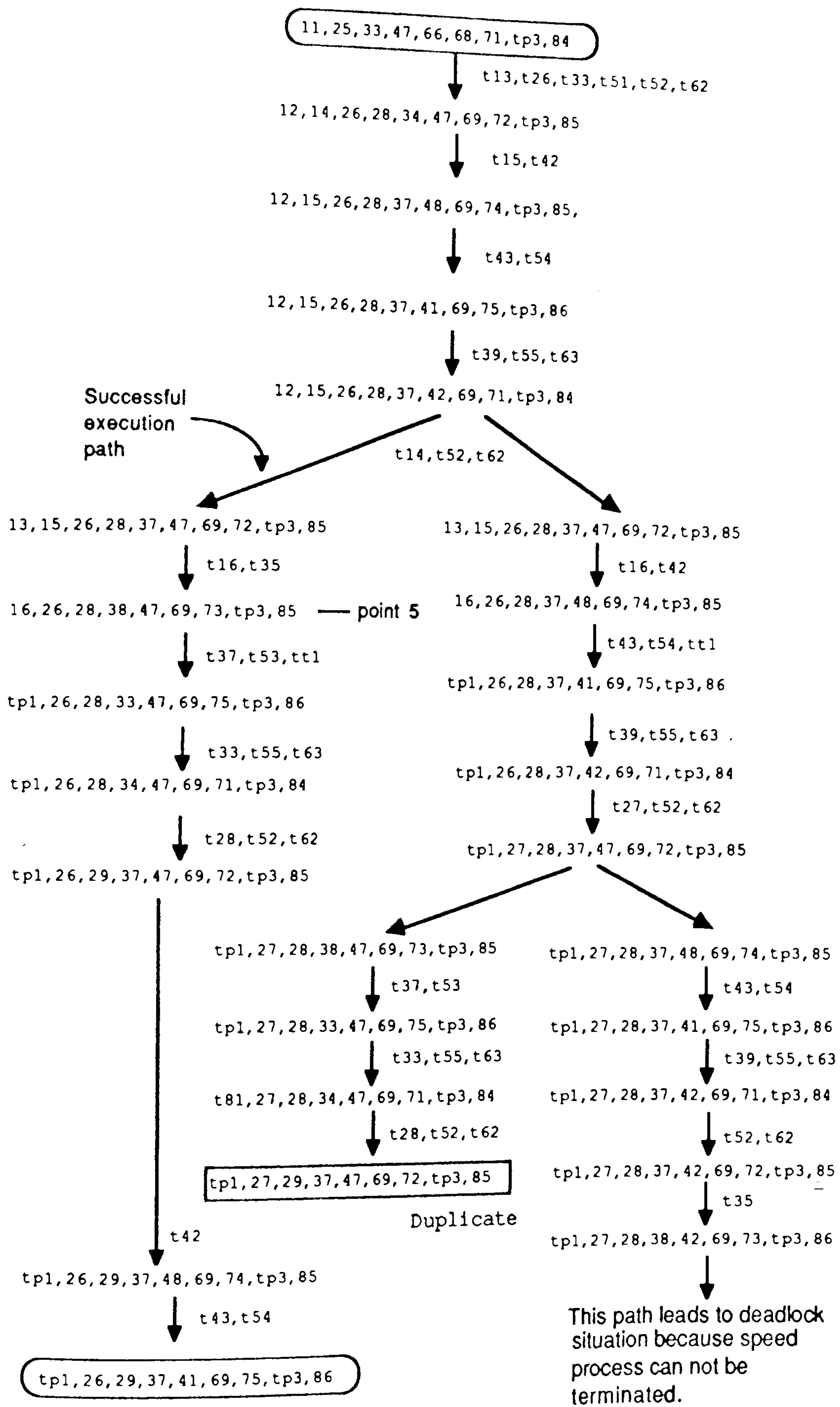


Figure 7.6 Reachability tree (continued)

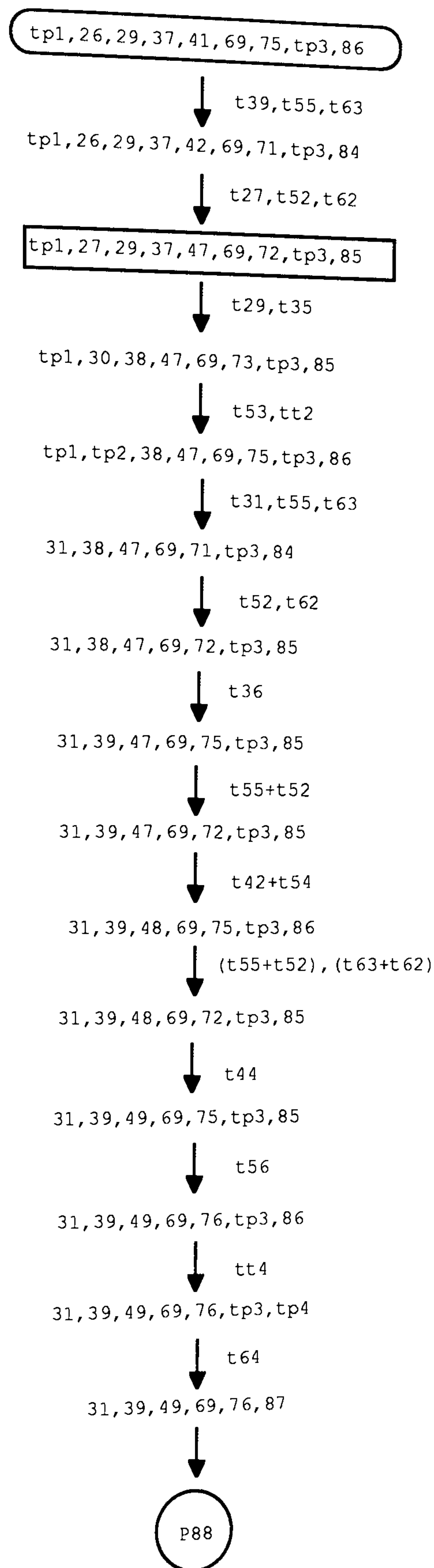


Figure 7.6 Reachability tree (continued)

7.3 Example: The Conversation Mechanism

This example outlines a method of introducing conversation-based fault tolerance into concurrent software. It is understood that all the processes involved in a conversation mechanism are communication intensive. Therefore, for a conversation the processes should never deadlock otherwise the mechanism is not fault tolerant. Occam tool (Octool) has been used on the conversation software for fault prevention (fault avoidance and fault removal) reasons before the program is ever run as a fault tolerant system. The tool has been used to identify any deadlock potential and eliminate it.

The reduction technique was also been implemented and was found very sucessful to reduce the size of the net. An example was also given to show that the reduction of the net is possible without the loss of information of the system behaviour.

7.3.1 Overview

The conversation mechanism[RAND 75] provides a basic recovery mechanism for distributed systems, This example considers the implementation of such a fault tolerant system[CARP 88b]. In particular, it describes how Petri-nets can be used to model the complete fault tolerant system. This model is then analysed and simulated to ensure that the conversation, comprising the recovery line, the acceptance test and error detection system, and the recovery mechanism, operates properly and provides a proper fault tolerant framework for the system.

In the proposed conversation mechanism, three processes exist. P is the preferred process, Q the alternative process, and DEFAULT the second alternative process which is assumed to always produce correct results (or alternatively can be thought of as a proven process that will place the system into a safe state, i.e. a fail-safe process). The design problem is to embed these processes within a fault tolerance framework which comprises a coordinated set of recovery points, a coordinated set of acceptance tests, an alternative process 'Q' and a 'default' process. The process P is divided into three sub-processes, p1, p2 and p3 that all execute in parallel. The primary process (comprising p1, p2, and p3) and the alternative processes ('Q' and 'DEFAULT') are initiated in parallel.

```
SEQ
  establish recovery point
  PAR
    P
    Q
    DEFAULT
```

```
where: P= PAR
          p1
          p2
          p3
```

The structure of the acceptance test requires careful design. In general, the acceptance test can be performed on two phases. First, as each individual process (p1, p2 and p3) produces partial results, the results may be tested on a local basis. If any local test indicates a fault, then the whole set of processes must be decreed faulty. However, local tests are not sufficient for an acceptance test of the whole conversation, because the processes p1, p2 and p3 must cooperate within the conversation and are not independent. Therefore, the local tests must be followed by a second phase of tests which assess the complete set of results in a global sense. This requirment can be achieved by a centralised process.

The implementation of the conversation can be simplified if additional processes are introduced to buffer the partial results generated by the processes, before these are passed to the first (local) phase of the acceptance tests. Thus, for the conversation comprising p1, p2 and p3, process p1 passes its results to a local result-buffer, p1_buffer. On completion, p1 would initiate its local acceptance test, atp1, and terminate since it will play no further part in the conversation.

The buffers that are introduced in the 'Q' and 'DEFAULT' processes are q_conversation_buffer and default_buffer respectively.

The overall structure is illustrated below; superfluous PARs and SEQs are retained for clarity.

```
PAR
  PAR          ------  primary conversation
    PAR
      p1
      p1_buffer
      atp1
    PAR
      p2
      p2_buffer
      atp2
    PAR
      p3
      p3_buffer
      atp3
    p_conversation_buffer
    atp_global
  PAR          ------  alternative conversation
    q
    q_conversation_buffer
    atq
  PAR          ------  default conversation
```

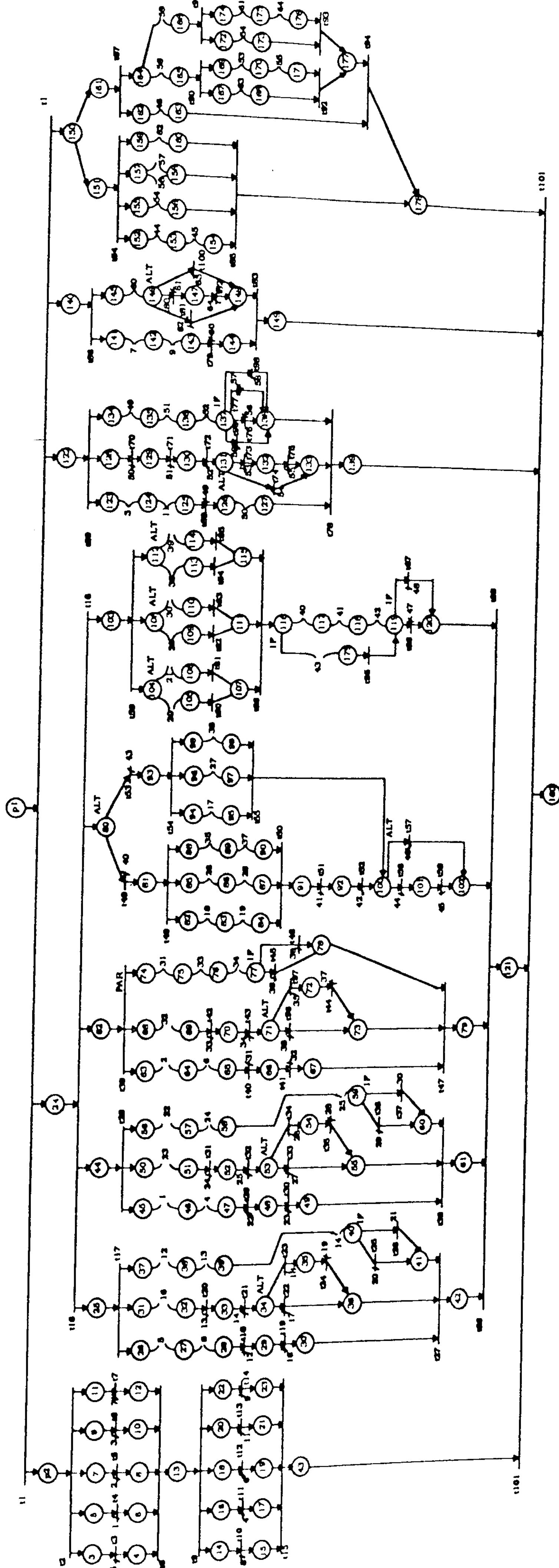


Figure 7.7 PNG showing conversation example

```
    default
    default_buffer
  SEQ
    ensure_arbitrator
```

This is the complete structure of the main process which initiates all the processes described in the Occam program. The complete conversation example is not described here because it is beyond the aim of the thesis. Readers interested in this conversation mechanism are referred to [CARP 88b].

7.3.2 Petri-net Conversion

Each process can be modelled as a Petri-net using the front-end Occam translator. The Petri-net graph shown in figure 7.7 represent all component processes when assembled together. This graph is composed of 180 places and 101 transitions. At first sight the resulting global structure is complex, but it is composed of readily understood substructures and overall comprehension is not lost. The Petri-net shown in figure 7.7 is the reduced version of the actual Petri-net graph. The actual graph consists of about 600 places and 300 transitions. The reduction technique implemented here is described by two examples. In the first example shown in figure 7.8 twenty eight places and 10 transitions are reduced to 4 places and one transition.

In Occam, the program fragment

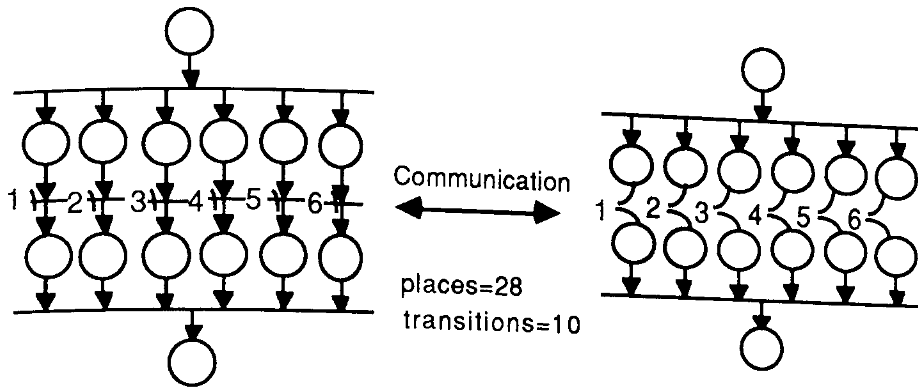
```
PAR i = 0 FOR 6
  init.to.q[i]  !  init[i]
PAR j = 0 FOR 6
  init.to.def[j]  !  init[j]
```

of one process is communicating with the program fragments of other processes which are as follow:

```
PAR i = 0 FOR 6
  init.to.q[i]  ?  q[i]

PAR i = 0 FOR 6
  init.to.def[i]  ?  default[i]
```

All other statements like these can be changed to a single channel because of the symmetry in channels and then can be represented in the reduced form of Petri-net shown in figure 7.8.



is reduced to the following where further reduction is also shown

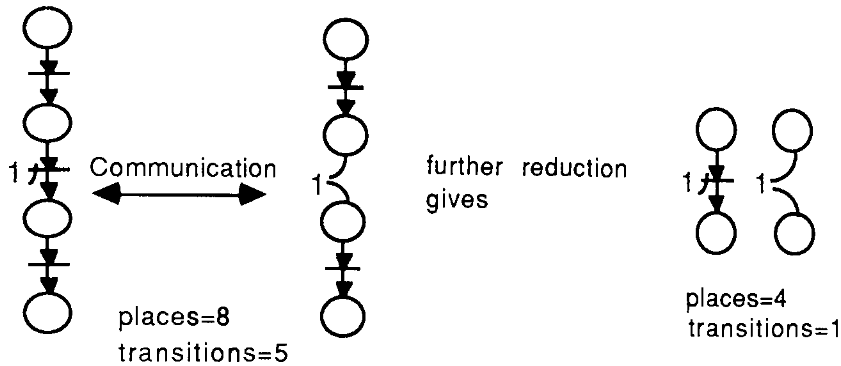
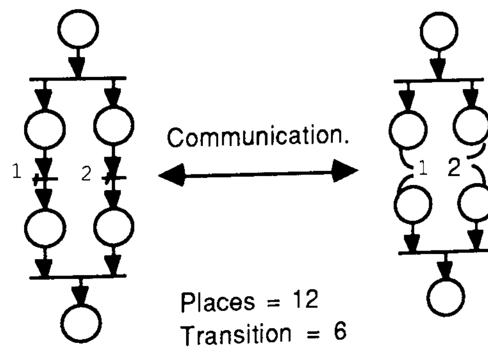


Figure 7.8 Reducing mechanism for array channels

In the second example places and transitions can be reduced in the same way as in the first example. The Occam program fragment for the Petri-net shown in figure 7.9 is also using symmetrical channels that can also be reduced. This way the complete Petri-net is reduced to its most reduced form.



is reduced to

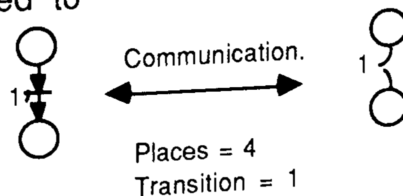


Figure 7.9 Reduction for PAR communicating construct

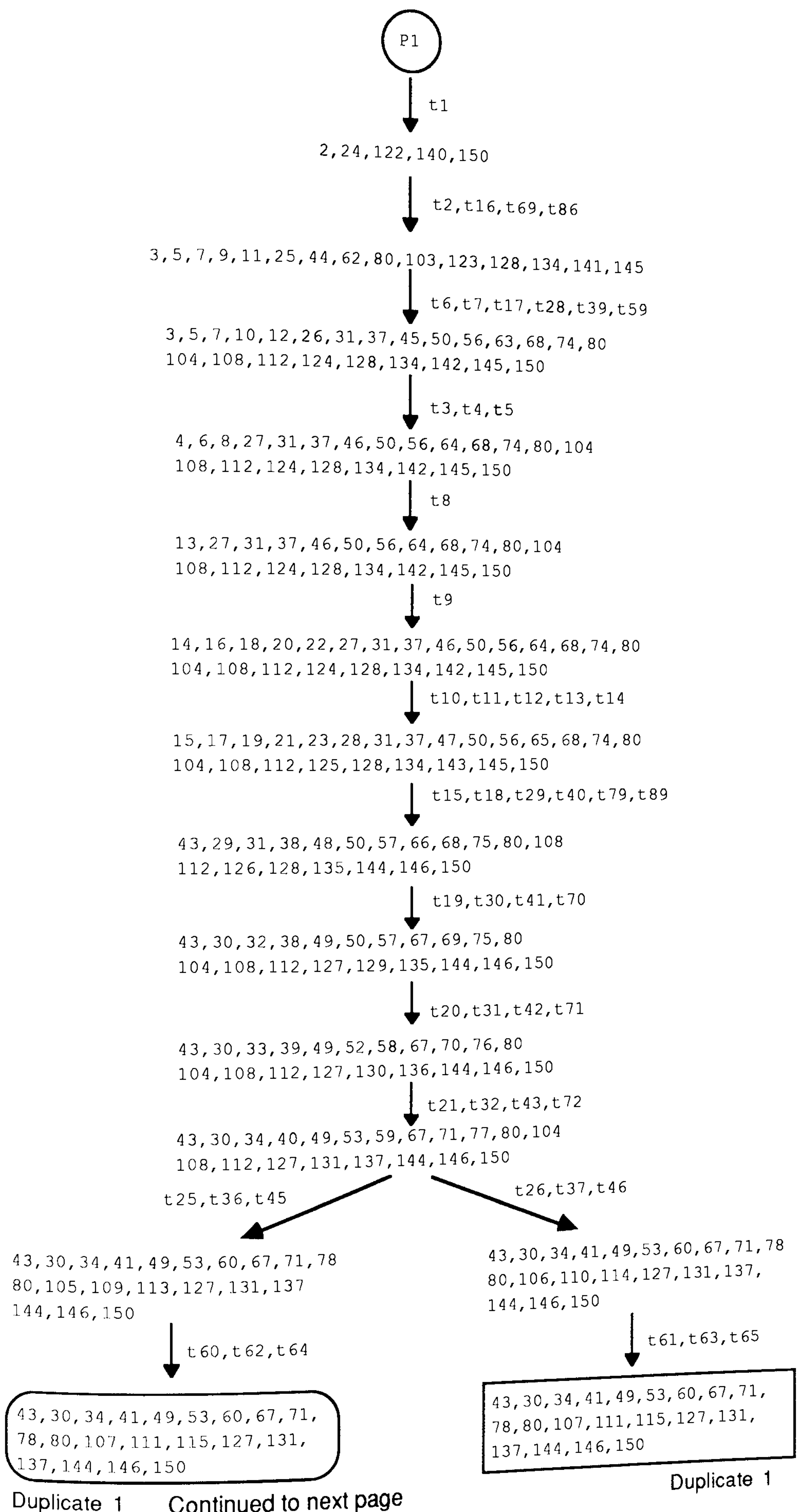
7.3.3 Reachability Tree

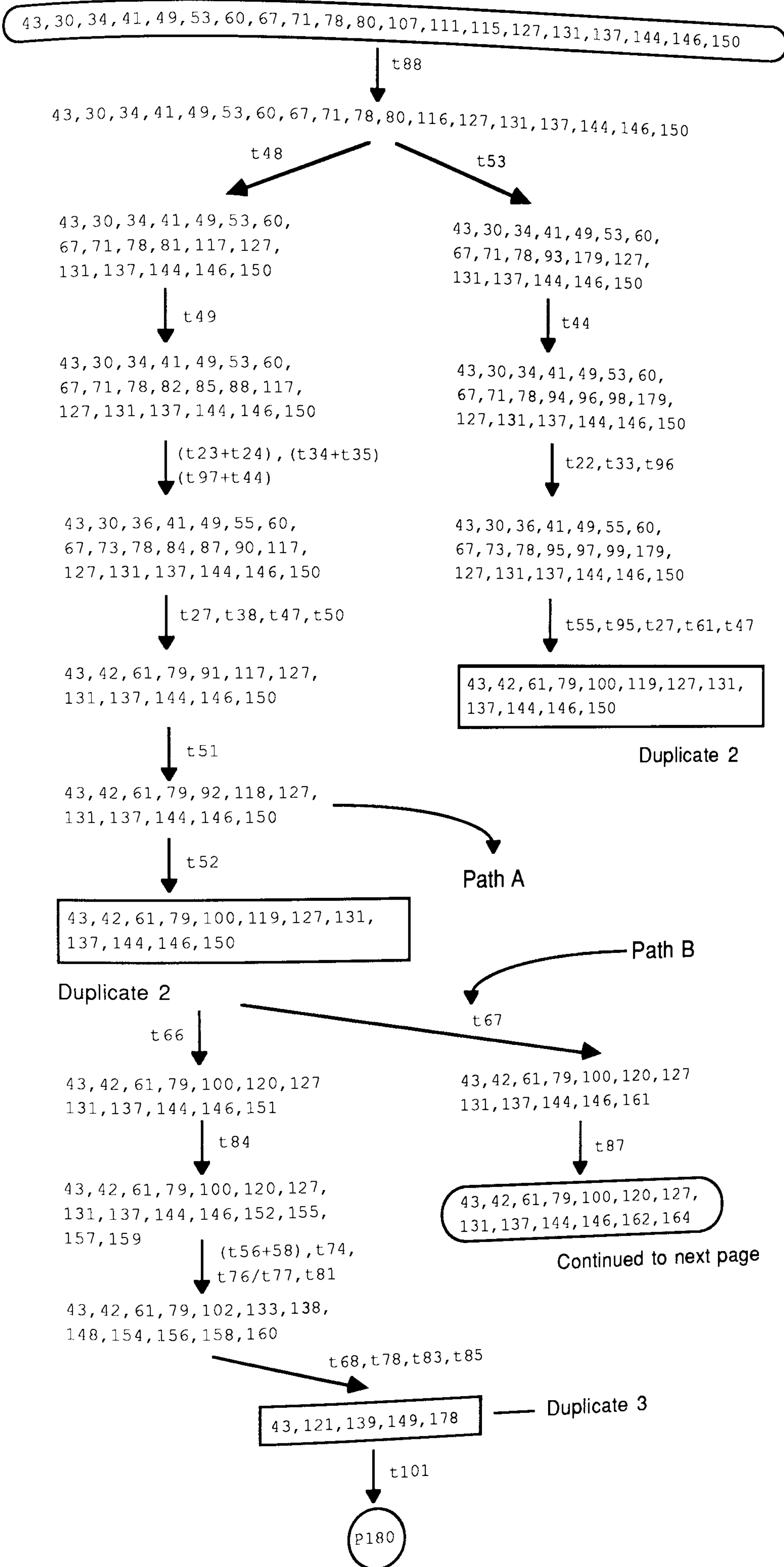
The reachability tree for Petri-net shown in figure 7.7 is given next. In the given tree all the possible paths of execution have been explored. In fact, all the paths lead to the successful execution of the system. For paths which duplicate the tree, only one path was chosen for further exploration. Path A and path B both lead to a successful execution but path A is shorter than path B; choosing this path saves time as well as the number of steps for the complete execution of the system.

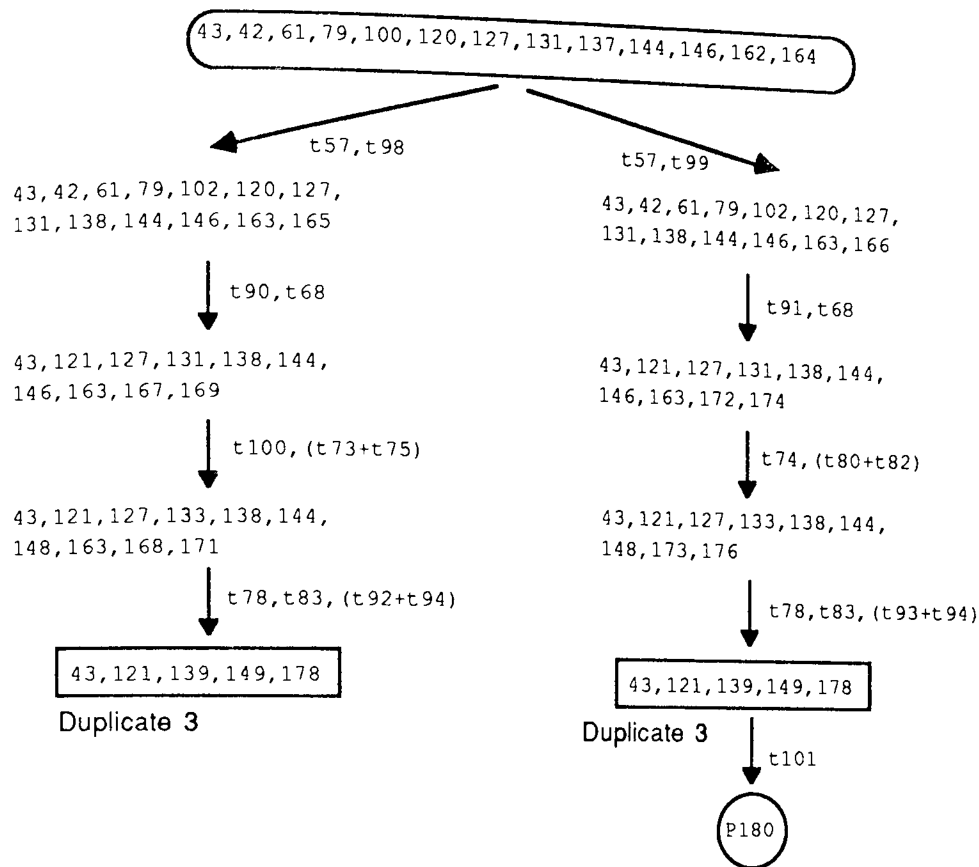
7.3.4 Results

Simulation and analysis has been carried out on the global structure, and its dynamic behaviour has been studied. During analysis, topological graph reduction technique has been implemented and was found useful to reduce the size of the net. In particular it has been possible to concentrate on the synchronising effect of interprocess communication.

The component processes of the conversation described in the example have been shown to be deadlock-free.


P1
t1
2,24,122,140,150
t2,t16,t69,t86
3,5,7,9,11,25,44,62,80,103,123,128,134,141,145
t6,t7,t17,t28,t39,t59
3,5,7,10,12,26,31,37,45,50,56,63,68,74,80
104,108,112,124,128,134,142,145,150
t3,t4,t5
4,6,8,27,31,37,46,50,56,64,68,74,80,104
108,112,124,128,134,142,145,150
t8
13,27,31,37,46,50,56,64,68,74,80,104
108,112,124,128,134,142,145,150
t9
14,16,18,20,22,27,31,37,46,50,56,64,68,74,80
104,108,112,124,128,134,142,145,150
t10,t11,t12,t13,t14
15,17,19,21,23,28,31,37,47,50,56,65,68,74,80
104,108,112,125,128,134,143,145,150
t15,t18,t29,t40,t79,t89
43,29,31,38,48,50,57,66,68,75,80,108
112,126,128,135,144,146,150
t19,t30,t41,t70
43,30,32,38,49,50,57,67,69,75,80
104,108,112,127,129,135,144,146,150
t20,t31,t42,t71
43,30,33,39,49,52,58,67,70,76,80
104,108,112,127,130,136,144,146,150
t21,t32,t43,t72
43,30,34,40,49,53,59,67,71,77,80,104
108,112,127,131,137,144,146,150
t25,t36,t45
t26,t37,t46
43,30,34,41,49,53,60,67,71,78
80,105,109,113,127,131,137
144,146,150
43,30,34,41,49,53,60,67,71,78
80,106,110,114,127,131,137,
144,146,150
t60,t62,t64
t61,t63,t65
43,30,34,41,49,53,60,67,71,
78,80,107,111,115,127,131,
137,144,146,150
43,30,34,41,49,53,60,67,71,
78,80,107,111,115,127,131,
137,144,146,150
Duplicate 1
Continued to next page
Duplicate 1


43,30,34,41,49,53,60,67,71,78,80,107,111,115,127,131,137,144,146,150
t88
43,30,34,41,49,53,60,67,71,78,80,116,127,131,137,144,146,150
t48
t53
43,30,34,41,49,53,60,
67,71,78,81,117,127,
131,137,144,146,150
t49
43,30,34,41,49,53,60,
67,71,78,82,85,88,117,
127,131,137,144,146,150
(t23+t24),(t34+t35)
(t97+t44)
43,30,36,41,49,55,60,
67,73,78,84,87,90,117,
127,131,137,144,146,150
t27,t38,t47,t50
43,42,61,79,91,117,127,
131,137,144,146,150
t51
43,42,61,79,92,118,127,
131,137,144,146,150
t52
43,42,61,79,100,119,127,131,
137,144,146,150
Duplicate 2
43,30,34,41,49,53,60,
67,71,78,93,179,127,
131,137,144,146,150
t44
43,30,34,41,49,53,60,
67,71,78,94,96,98,179,
127,131,137,144,146,150
t22,t33,t96
43,30,36,41,49,55,60,
67,73,78,95,97,99,179,
127,131,137,144,146,150
t55,t95,t27,t61,t47
43,42,61,79,100,119,127,131,
137,144,146,150
Duplicate 2
Path A
Path B
t66
t67
43,42,61,79,100,120,127
131,137,144,146,151
t84
43,42,61,79,100,120,127,
131,137,144,146,152,155,
157,159
(t56+58),t74,
t76/t77,t81
43,42,61,79,102,133,138,
148,154,156,158,160
t68,t78,t83,t85
43,121,139,149,178
Duplicate 3
t101
P180
43,42,61,79,100,120,127
131,137,144,146,161
t87
43,42,61,79,100,120,127,
131,137,144,146,162,164
Continued to next page



7.4 Conclusion

Arguably, the major problem involved in the design and implementation of software for distributed processes is deadlock. Two examples were taken to determine the effectiveness of Octool in analysing dynamic behaviour. For each example, the dynamic behaviour was analysed by assembling all the component processes together. In both cases, the complete translation leads to a Petri-net which is very complex and big. Therefore, a reduction technique was implemented to decrease the complexity of these Petri-nets. The way the reduction technique was implemented has been explained. Simulation and analysis has been carried out on the complete Petri-nets, and the dynamic behaviour has been studied. During analysis, the main concentration was on the synchronising effect of interprocess communication. Finally, implementations of these processes have been analysed using Petri-nets and have been shown to be deadlock free. This clearly shows the usefulness of Octool in analysing sets of concurrent processes for deadlock potential, especially since there is no possibility of running such software in the real-world if dormant faults still exist.

Chapter 8
Conclusion and Future Work

8.1 Conclusion

Using current software engineering technology, the robustness required for safety critical software is not assurable. However, different approaches are possible which can help to assure software robustness to some extent. For achieving high reliability software, methods should be adopted which avoid introducing faults (fault avoidance); then testing should be carried out to identify any faults which persist (error removal). Finally, techniques should be used which allow any undetected faults to be tolerated (fault tolerance).

This thesis has been concerned with the design of fault tolerant concurrent software for distributed computing systems consisting of sets of communicating sequential processes, which can only communicate with each other by message passing. An investigation was carried out into models of concurrent programs inspired by a desire to find accurate and efficient techniques for detecting deadlock potential in concurrent programs before the programs are ever run. Different Fault tolerance techniques can then be used to make the software reliable. Techniques that provide fault tolerance are N-version programming, recovery blocks and conversation mechanism. Fault avoidance is of permanent impotence in producing fault tolerant systems which really are fault tolerant.

A state-transition method was used to model and analyse concurrent systems. Since the Occam language is sufficient to represent concurrent systems, the state-transition model of Occam should be sufficient to model a concurrent system. A concurrent system described in the form of an Occam program specification can be translated into a Petri-net model. All the sequential and concurrent constructs present in Occam can be modelled successfully. It is then possible for the Petri-net to provide information about the concurrent system such as the possibility of process deadlock.

A systematic design of OCTOOL (Occam translator and the simulator) was proposed and implemented in chapter 4. This implementation uses two different approaches. One of the two approaches for designing Occam translator is explained in chapter 4. This tool was designed in 'C' language and is divided into two parts;

an Occam translator and the Petri-net simulator. The simulator uses the Petri-net theory and its execution rules to fire the enabled transitions and generate the reachability tree for analysis.

In the first approach to the translator (pre-processor) design two set of tables were used; one is the keyword table where all the possible keyword sequences were stored, and the second is the Petri-net structure table where all the subsequent matrices are stored. Whenever a keyword sequence is recognised, the subsequent Petri-net structure is invoked and passed to the generation phase of the translator. Research showed that limitations meant that it could not be used for large and complex Occam programs. Another approach was adopted and is discussed in chapter 6.

The design of this Occam translator uses two UNIX utilities; the Lex and Yacc. The Lex (Lexical analyser generator) and Yacc (the syntax analyser generator) are the commonly used utilities under UNIX for the generation of compiler and are described in chapter 5. It was also described in this chapter how Lex and Yacc specification can be constructed for a simple programming language, namely Occam. The design of an Occam translator using Lex and Yacc is presented. Production rules and its associated actions were designed. Whenever the production rule is recognised its appropriate action takes place. The Petri-net structure was generated step by step but the whole procedure is invisible to the user. All the Occam language grammar was used in the specification of Lex and Yacc, but some actions for their rules were deliberately ignored because it was beyond the aim of the thesis.

8.2 Future Work

This thesis considered two approaches to the translation of an Occam program into its Petri-net structure. An automatic deadlock detection tool is described in chapter 4 which explains the first of the two translation techniques. The Petri-net simulator which works with the translator was also described in this chapter. The second approach in designing the translator was shown in chapter 6 and this is sufficient to model the communication aspects of Occam program but this translator can not cope with every type of Occam program. Further work could be done in producing action for those production rules given in chapter 6 which do not have any action. All the production rule already designed are given in chapter 6 and appendix B. The following are the areas for further research.

1) The tool discussed in this thesis is not completely automated. An Occam program is taken as an input and automatically translated into PNS. The generated PNS is then taken by the Petri-net simulator as an input. The Petri-net simulation needs step by step information from the user. The execution path for the Petri-net is very important, is not automatic and should be chosen carefully. Different possible paths could be generated by computer first and then each one could be tried for the successful execution to the final transition. The reachability tree could then be generated for that path for further inspection.

2) After the generation of the reachability tree, the user needs to inspect the tree for any possible deadlock occurrence. This deadlock can by detected by finding out an unexpected 'terminal' node. A further research can be done in providing a back-end analyser to the reachability tree. The analyser takes the reachability tree as an input and performs searches for specific states of interest (i.e. terminal nodes). The output results would include information on the potential points of communication and synchronisation. The output should also identify potential program deadlock and termination states.

3) Another approach that can be used for automatic deadlock detection is by using artificial intelligence in which the tool is again divided into three phases. Three type of database rules can be made, one for each phase.

4) The tool described in this thesis takes an Occam program and analyses it for any possible deadlock before the program is ever run. Further work could be done to provide an option in the compiler that can produce Petri-net structure before the compiler produce its machine code.

The approach could also be used on Occam-3 program as long as the basic communication constructs remain the same. Other modification can be made in this tool to make it feasible to recognise and analyse any version of Occam language. The designer only needs to modify appropriate rules.

The approach stresses the dynamic analysis capability of the Petri-net approach in analysing the properties of a concurrent system. It is strongly recommended that all concurrent systems, especially where safety must be guaranteed be subject to Petri-net analysis since dynamic properties by their very nature can only arise when system is operational and cannot be detected by static, off-line testing.

References

[AGER 78] T. Agerwala, "Some applications of Petri-nets", Proc. National electronic confer., Vol. 32, 1978, pp155-160.

[AGER 79] T. Agerwala, "Putting Petri-nets to work", IEEE computer, Vol. 12, no.12, 1979, pp85-94.

[AHO 77] A. V. Aho and J. D. Ullman, "Principle of compiler design", Addison-Wesley, 1977.

[AHO 85] A. V. Aho, " Compiler: principles, techniques and tools", Addison-Wesley Pub. co. 1985.

[ALI 9O] J. Ali and M. Ali, "The use of executable assertions for error detection and damage assessment.", Journal of System Software, Vol. 12, 1990, pp 15-37.

[ANDE 81] T. Anderson, and P. A. Lee, "Fault tolerance, Principles and practice", Prentice-Hall, Inc. 1981.

[ANDE 83] T. Anderson, and J. C. Knight, "A frame work for software fault tolerance in real-time systems", IEEE trans. on software engg., Vol. SE-9, no. 3, 1983, pp355-364.

[ANDE 85] T. Anderson, P. T. Barrett, D. N. Halliwell, and M. R. Moulding, "Software fault tolerance: An evolution", IEEE trans. on software engg., Vol. SE-11, No. 12, 1985, pp1502-1510.

[ANDR 83] G.R. Andrew and F. Schneider, "Concepts and notations for concurrent programming", ACM Computing Surveys, No. 15(1), 1983, pp3-44.

[ARNO 86] D. W. Arnold, " Petri-net analysis techniques", M. Sc. report, Deptt of elect. and electronic engg., 1986.

[AVIZ 85] A. Avizienis, " The N-version approach to fault tolerant software", IEEE trans. on software engg., Vol. SE-11, No. 12, 1985, pp1491-1501.

[BAL 88] H. Bal, J. Steiner, A.S. Tanenbaum, "Programming languages for distributed computing system", ACM Computing surveys, Vol.21, No.3, 1988, pp261-231.

[BERT 76] G. Berthelot, and G. Roucairol, "Reduction of Petri-nets", Proc. of the symposium on MFCS-76, Lecture notes in computer science, Springer-Berlin, Vol. 45, 1976, pp202-209.

[BERT 79] G. Berthelot, G. Roucairol and R. Valk, "Reduction of nets and parallel programs", Proc. advanced course in general theory of processes and systems, Hamburg, Germany, 1979 in BRAUER, W:(ed), Lecture Notes in computer science, 84, (Springer Verlay, 1980), pp277-290.

[BLAC 87] Black, Uyless, " Computer networks- protocols, standards and interfaces", Prentice-hall, Inc., 1987.

[BRIL 89] S.S. Brilliant, J. C. Knight, N.G. Leveson, "The consistent comparison problem in N-version software", IEEE trans. software engg. Vol.SE-15, No.11, 1989, pp1481-1485.

[BURN 88] A. Burns, "Programming in occam 2", Addison-Wesley publishing company, Inc. 1988.

[BURN 89] A. Burns and A, Wellings, "Real-Time systems and their programming languages", Addison-Wesley Publishing Co. 1989.

[CAMP 86] R.H. Campbell, " Error recovery in asynchronous systems", IEEE Trans. on Software Engg., Vol. SE-12, No. 8, 1986, pp811-826.

[CARP 87] G. F. Carpenter, "The use of occam and Petri-nets in the simulation of logic structures for the control of loosely coupled distributed system", Proc. of UKSC conference on computer simulation, 1987, pp30-35.

[CARP 88a] G. F. Carpenter, "State space modelling in the design of robust software for distributed systems: a case study ", Microprocessing and Microprogramming, Vol. 24 (1988) pp793-800.

[CARP 88b] G. F. Carpenter, D. J. Holding and A. M. Tyrrell, "The design and simulation of software fault tolerant mechanism for application in distributed processing systems", Microprocessing and Microprogramming, No 22, 1988, pp175-185.

[CARP 89] G. F. Carpenter, and A. M. Tyrrell, "The use of GMB in the design of robust software for distributed systems.", Software Engg Journal, Sept 1989, pp268-282.

[DADD 76] L. Dadda, "The synthesis of Petri-net for controlling purposes and the reduction of their complexity", Proc. of EUROMICRO conference, North Holland Publ., 1976, pp251-259.

[DAVI 87] A. C. Davies, "Features of high-level languages for microprocessors", Microprocessors and Microsystems, Vol. 11, No. 2, March 1987, pp77-87.

[DEIT 84] H. M. Deitel, "An introduction to operating systems", Addison-Wesley publishing co, inc. 1984.

[DEMA 79] T. De Marco, "Structure Analysis and system specification", Prentice-Hall, Englewood cliffs, NewJersey, 1979.

[DIJK 65] E. Dijkstra, "Solution of a problem in concurrent program control", Comm. ACM, Vol. 8, No. 9, Sept. 1965, pp569

[DIJK 68] E. Dijkstra, "Co-operating Sequential processes", Programming Languages, F. Genveys(Ed.), Academic Press, 1968.

[DUGA 89] J. B. Dugan, "Stochastic Petri-net analysis of a replicated file system", IEEE transaction on software engg., Vol. 15, no. 4, 1989, pp394-401.

[FARM 85] N. Farmer, " Compiler physiology for beginners", Bromley: Chartwell-Bratt, 1985.

[FLEM 88] P. J. Fleming, "Parallel processing in control: the transputer and other architecture", IEE computing series, No. 38, 1988.

[GREG 85] S. T. Gregory and J.C. Knight, "A new linguistic approach to background error recovery", Proc. 15th Int. Symp. on Fault Tolerant Computing, 1985, pp404-409.

[HECH 76] H. Hecht, " Fault tolerant software for real-time applications", Computer Surveys, Vol. 8, no. 4, 1976, pp391-407.

[HECH 79] H. Hecht, " Fault tolerant software", IEEE trans. on reliability, Vol. R-28, no. 3, 1979, pp227-232.

[HEIM 78] W. L. Heimerdinger, " A Petri-net approach to system level fault tolerance analysis", Proc. National electronic conf. 1978, pp161-165.

[HOAR 78] C.A.R. Hoare, "Communicating Sequential Processes.", Comm. ACM, Vol. 21, No. 8, 1978, pp 666-667.

[HOAR 85] C.A.R. Hoare, "Communicating Sequential Processes (CSP)", Prentice-Hall, 1985.

[INMO 84] Inmos Ltd, "Occam programming manual" , Prentice-Hall, 1984.

[INMO 88] Inmos Ltd, "Occam 2 Reference manual", Prentice-Hall, 1988.

[JACK 83] M. A. Jackson, "System development", Prentice-Hall, 1983.

[JANT 80] M. Jantzen, and R. Valk, "Formal properties of place/transition nets", Lecture notes in computer science, Berlin: Springer-verlag, 1980, pp165-212.

[JOHN 75] S.C. Johnson, "YACC- Yet Another Compiler Compiler", Computing Science Technical Report No. 32, Bell laboratories Murray Hill, N.J. 1975.

[KANT 85] K. Kant, and A. Silberschatz, " Error propagation and recovery in concurrent environment", The computer journal, Vol. 28, No. 5, 1985, pp466-473.

[KANT 87] K. Kant, "Software Fault tolerance in real-time systems", Information Sciences, Vol. 43(3), 1987, pp255-282.

[KAVI 85] K. M. Kavi, et al, "Isomorphisms between Petri-net and data flow graphs", Tech. report. CSE. tr-85-002, UNIVER. Arlington in Texas, Deptt. computer sci. 1985.

[KAWA 71] H. Kawashima, "Functional specification of call processing by state transition diagram", IEEE transaction on communications, Vol. COM-19, No. 5, Oct 1971, pp581-587.

[KELL 76] R. M. Keller, "Formal verification of Parallel programs", Communication of the ACM, Vol. 19(7), July 1976, pp371-384.

[KERN 72] B. W. Kernighan, D.M. Ritchie and K.L. Thompson, "QED Text editor", Computing Science Technical report no. 5, Bell laboratories, Murray Hill N.J. 1972.

[KHAN 90] M. A. Khan and G. F. Carpenter, "The Generation of Petri-net models of Occam Program", UKSC Conference on Computer Simulation 1990, Brighton, pp198-201.

[KIM 82] K. H. Kim, "Approaches to mechanisation of the conversation scheme based on monitors", IEEE trans. on software Engg. Vol. SE-8, No. 3, May 1982, pp189-197.

[KNIG 86] J. C. Knight, N. G. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming", IEEE trans. software engg. Vol.SE-12, No.1, 1986, pp96-109.

[KNIG 91] J. C. Knight, and P. E. Ammann, "Design Fault Tolerance", Reliability Engineering and System Safety, Vol. 32, 1991, PP 25-49.

[LALA 85] P. K. Lala, " Fault tolerant and fault testable hardware design", Prentice-Hall, Inc. 1985.

[LAND 78] L. Landweber and E. Robertson, "Properties of Conflict-free and persistent Petri-nets", Journal of the ACM, Vol. 25, No. 3, July 1978, pp352-364.

[LESK 75] M. Lesk and E. Schmidt, "Lex- A Lexical Analyser Generator", Technical Report, Bell laboratories, Murray Hill, N. J. 1975.

[LEVE 83] N. G. Leveson and P. R. Harvey, "Analysing software safety", IEEE trans. on software engg., Vol. SE-9, No. 5, 1983, pp569-579.

[LEVE 86] N. G. Leveson, "Software safety: why, what and how", A.C.M computing surveys, Vol. 18, No. 2, 1986, pp125-163.

[LEVE 87] N. G. Leveson, and J. L. Stolzy, " Safety analysis using Petri-nets", IEEE trans. on software engg., Vol. SE-13, No. 3, 1987, pp386-397.

[LING 79] R.C.Linger, H.D. Mills and B.I. Witt, "structured programming: Theory and practice", Addison-Wesley, 1979, pp147-212.

[MAYR 84] E. W. Mayr, "An Algorithm for the General Petri-net reachability problem", SIAM Journal on Computing, Vol. 13(3), Aug. 1984, pp441-460.

[MEMM 84] G. Memmi and P. Behm, "RAFAEL: a real-time system analysis tool", Conf. on Software Engineering: Practice and Experience, June 1984, pp1-7

[MOLL 82] M. K. Molloy, "Performance analysis using stochastic Petri-nets"; IEEE transaction on computer, sept. 1982.

[MURA 83] T. Murata, "Modeling and Analysis of Concurrent Systems", Handbook of software Engineering, C. R. Vick and C. V. Ramamoorthy (eds.), Chapter 3, Van Nostrand Reinhold, New York, 1983.

[MURA 89] T. Murata, "Petri-nets: Properties, analysis and applications", Proc. IEEE, Vol. 77, No.4, 1989, pp 541-580

[NEWP 86] J. R. Newport, "An introduction to occam and the development of parallel software", Software Engineering Journ, Jul 1986, pp165-169.

[O'HAL 86] D. R. O'Hallaron and P. F. Reynolds Jr., "A Generalised deadlock predicate", Information Processing Letters, 1986.

[OBER 82] Ron Obermarck, "Distributed deadlock detection algorithm", ACM Trans. on Database systems, Vol. 7, No. 2, June 1982, pp187-206.

[PETE 77] J. L. Peterson, "Petri-nets", A.C.M. computing Surveys, Vol. 9, No. 3, 1977.

[PETE 78] J. L. Peterson, "An introduction to Petri-nets", Proc. National electronic confer., Vol. 32, 1978, pp144-148.

[PETE 81] J. L. Peterson, "Petri-net theory and modelling of system", prentice hall, 1981.

[POUN 85] Dick Pountain, "A tutorial introduction to OCCAM programming ", 1985.

[RAND 75] B. R. Randell, " System structure for software fault tolerance", IEEE trans. on software engg., Vol. SE-1, No.2, 1975, pp220-232.

[RAND 78] B. Randell, P.A.Lee, P.A. Treleaven, "Reliability issue in computing system design", ACM Computing Surveys, Vol. 10, No. 2, 1978, pp123-165.

[REIS 85] W. Reisig, "Petri-nets: An Introduction" IATCS Monographs on theoretical computer science; Spring-Verlag, 1985.

[REYN 79] P. F. Reynolds, Jr., "Parallel processing structures: Languages, Schedules, and Performance results", Ph.D. Thesis, University of Texas, Austin, 1979.

[SAGO 89] J. S. Sagoo and D. J. Holding, "The specification of hard real-time systems using timed and temporal Petri-nets", Microprocessing and Microprogramming, Vol. 30, No. 1-5, 1990, pp389-396.

[SCHO 84] J. D. Schoeffler, "Distributed computer systems for industrial process control", Computer, VOL 17, No. 2. Feb 1984, pp 11-19.

[SCHR 85] A.T. Schreiner and H.G. Friedman, Jr, "Introduction to Compiler construction with Unix", Prentice-Hall, Inc. 1985.

[SHAT 88] S. M. Shatz and W. K. Cheng, "A Petri-net Framework for Automated Static Analysis of Tasking Behaviour", The Journal of Systems and Software, Vol. 8, 1988, pp343-359.

[STAN 88] J. A. Stankovic, "A serious problem for Next-generation systems", IEEE Computer. Vol. 21, No. 10. Oct 1988, pp 10-19.

[SUZU 83] J. Suzuki and T. Murata, "A method for Stepwise refinement and abstraction of Petri-net", Journal of computer and system science, Vol. 27, No. 1 August 1983, pp51-76.

[TANE 84] A. S. Tanenbaum, " Structure computer organization", Prentice-hall, Inc., 1984.

[TAYL 83] R. Taylor, "A general Purpose Algorithm for Analysing Concurrent Programs", Communication of the ACM, Vol. 26, No. 5, May 1983, pp362-376.

[TAYL 86] D. J. Taylor," Concurrency and forward recovery in atomic actions", IEEE Trans. on Software Engg. Vol. SE-12, No. 1, 1986, pp69-78.

[TYRR 86] A. M. Tyrrell, and D. J. Holding, " Design of reliable software in distributed systems using conversation scheme", IEEE trans. on software Engg., Vol. SE-10, No. 9, 1986, pp921-927.

[WALK 90] D. Walker and G. S. Hura, "A Tutorial on fault tolerance issue with applications in Distributed Systems", Microelectron. Reliab., Vol. 30, No. 6, 1990, pp1029-1037.

[WIRT 71] N. Wirth, "program development by step-wise refinement", Communication of the ACM, Vol. 14, No. 4, April 1971, pp 221- 227.

[WU 89] J. Wu, E. B. Fernandez, "A simplification of a conversation design scheme using Petri-nets", IEEE Trans. software Engg., Vol.15, No. 5, 1989, pp658-660.

[ZAKS 87] R. Zaks, and A. Wolfe, " From chip to systems- An introduction to microcomputer", Sybex, Inc., 1987.

[ZORP 85] G. Zorpette, "Computer that are never down", IEEE spectrum, 1985, pp46-54.

Occam Lex Program

```
/*
 *      OCCAM2 lexical analysis routine
 */

#include <stdio.h>
#include <ctype.h>
#include "lex2.h"

#define        MAXLINE    256

#define        TRUE 1
#define        FALSE      0

/***********************************************************/
/* reserved word list - ordered for binary chomp */

static struct reserv { char * word; int tok, len; } rlist[] = {
                "AFTER",        AFTER,      5,
                "ALT",          ALT,  3,
                "AND",          BOOLOP,     3,
                "ANY",          ANY,  3,
                "AT",           AT,   2,
                "BYTE",         BYTE,       4,
                "CASE",         CASE,       4,
                "CHAN",         CHAN,       4,
                "DEF",          DEF,  3,
                "ELSE",         ELSE,       4,
                "FALSE",        BOOL,       5,
                "FOR",          FOR,  3,
                "FROM",         FROM,       4,
                "FUNCTION",           FUNCTION, 8,
                "IF",           IF,   2,
                "INT",                INT,  3,
                "INT16",        INT16,      5,
                "INT32",        INT32,      5,
                "INT64",        INT64,      5,
                "IS",           IS,   2,
                "MOSTNEG",            MOSTNEG,7,
                "MOSTPOS",            MOSTPOS,7,
                "NOT",          NOT,  3,
                "NOW",          NOW,        3,
                "OR",           BOOLOP,     2,
                "OF",           OF,   2,
                "PAR",          PAR,  3,
                "PLACE",        PLACE,      5,
                "PLACED",       PLACED,     6,
                "PORT",         PORT,       4,
                "PRI",                PRI,  3,
                "PROC",         PROC,       4,
                "PROCESSOR",    PROCESSOR,        9,
```

```
        "PROTOCOL",             PROTOCOL, 8,
        "ROUND",            ROUND,      5,
        "REAL",             REAL,       4,
        "REAL32",           REAL32,     6,
        "REAL64",           REAL64,     6,
        "RESULT",           RESULT,     6,
        "RETYPES",          RETYPES,    7,
        "SEQ",              SEQ,  3,
        "SIZE",             SIZE, 4,
        "SKIP",             SKIP, 4,
        "STOP",             STOP,       4,
        "TABLE",            TABLE,      5,
        "TIMER",            TIMER,      5,
        "TO",               TO,   2,
        "TRUE",             BOOL,       4,
        "TRUNC",            TRUNC,      5,
        "VALUE",            VALUE,      5,
        "VAL",              VAL,  3,
        "VALOF",            VALOF,      5,
        "VAR",              VAR,  3,
        "WHILE",            WHILE,      5,
        0,                  0,    0

    };

/************************************************************/

static  char    line[MAXLINE];      /* where we store the input, line as a time */

char    yytext[MAXLINE];    /* where we store text associated with token */

int     yylineno=1,         /* line number of input */
        yylen;              /* amount of text stored */

static  int     llen,           /* how much in line */
                curind,                 /* current indentation */
                indent=0;       /* this lines indent */
                ldebug = TRUE,          /* set to TRUE for debug */
                index;          /* where we are in the line */

/* state we are in: either start - get new input, decide what next
                    ind - processing indentation
                    rest - processing some occam stmt
                    eof - tidy up processing
*/

static  enum    lexstate { Start, Ind, Rest, Eof } state = Start;

/****************************************************************/

yylex()
/* this function returns the next token (defined by lex.h), a character
value or 0 for end of input. The tokens are defined by standard input
*/
{
        int     tok = -1,       /* token to return - init to impossible value */
                sind = index;   /* start of input being processed */
```

```
/* go round and round until token to return */
        while ( tok < 0  ) {

/* decide by state */
        switch (state) {

                case Start: {
/*get some more line */
                        if ( fgets( line, MAXLINE-1, stdin ) == NULL ) {
                                state = Eof;
                                break;

                        } else if ( (llen=strlen(line)) >= MAXLINE-1 ) {
                                fprintf( stderr,
                                        "line <%s> longer than %d\n",
                                        line, MAXLINE-1 );
                                exit( 1 );
                        }/*if*/

                        index = 0;
                        sind = 0;
                        indent = 0;


/* if blank line OR has just comment skip, otherwise got to appropriate state */

                        if ( m_nulline() ) {
                                /* do nowt */

                        } else if ( line[0]==' ' && line[1]==' ' ) {
                                state = Ind;

                        } else {
                                state = Rest;

                        }/*if*/

                break;}/*Start*/

                case Ind: {
/* work out indentation */
                        if ( line[index]==' ' && line[index+1]==' ' ) {
                                indent++;
                                index+=2;
                                sind+=2;
                        } else {
                                state = Rest;

                        }/*if*/

                break;}/*Ind*/

                case Rest: {
/* do we have some indentation to adjust for ... */
                        if ( curind > indent ) {
                                curind--;
                                tok = END;
                                break;
```

```
                    } else if ( curind < indent ) {
                            curind++;
                            tok = BEG;
                            break;

                    }/*if*/

/* process ch as appropriate */
                    switch ( line[index] ) {

/* space ignored */
                            case ' ': {
                                    sind++;
                                    index++;
                            break;}

/* eol change state again */
                            case '\n': {
                                    yylineno++;
                                    index++;
                                    state = Start;
                                    tok = EOL;
                            break;}

/* - a comment OR just itself */
                            case '-': {
                                    if ( line[index+1] == '-' ) {
                                            index = llen+1;
                                            state = Start;
                                            tok = EOL;

                                    } else {
                                            tok = line[index++];

                                    }/*if*/
                            break;}

                            case '<': {
                                    if ( line[index+1] == '<' ) {
                                            index+=2;
                                            tok = SHIFTOP;

                                    } else {
                                            if ( line[index+1] == '=' ||
                                                    line[index+1] == '>' ) {
                                                    index++;
                                            }/*if*/
                                            index++;
                                            tok = COMPOP;
                                    }/*if*/
                            break;}

                            case '>': {
                                    if ( line[index+1] == '>' ) {
                                            index+=2;
                                            tok = SHIFTOP;
```

```
        } else if ( line[index+1] == '<' ) {
                index+=2;
                tok = LOGOP;

        } else {
                if ( line[index+1] == '=' ) {
                        index++;
                }/*if*/
                index++;
                tok = COMPOP;
        }/*if*/

break;}

case '/': {
        if ( line[index+1] == '\\' ) {
                index+=2;
                tok = LOGOP;

        } else {
                tok = line[index++];

        }/*if*/
break;}

case '\\': {
        if ( line[index+1] == '/' ) {
                index+=2;
                tok = LOGOP;

        } else {
                tok = line[index++];

        }/*if*/
break;}

case '#': {
        if ( isxdigit( line[index+1] ) ) {
/* gobble up hex digits */
                index++;
                while ( isxdigit(line[index]) ){
                        index++;
                }/*while*/

                tok = NUMBER;

        } else {
                tok = line[index++];

        }/*if*/

break;}

case '\'': {
        if ( line[index+1] != '*'
            && line[index+2] == '\'' ) {

                index+=3;
```

```
                        tok = CHCON;

                } else if ( line[index+1] == '*'
                    && line[index+2] != '#'
                    && line[index+3] == '\'' ) {

                        index+=4;
                        tok = CHCON;

                } else if ( line[index+1] == '*'
                    && line[index+2] == '#'
                    && isxdigit( line[index+3] )
                    && isxdigit( line[index+4] )
                    && line[index+5] == '\'' ) {

                        index+=6;
                        tok = CHCON;

                } else {
                        tok = line[index++];

                }/*if*/

        break;}


        case '"': {
                int     lindex=index+1;

                while ( line[lindex] != '"'
                    && lindex <= llen ) {
                        lindex++;
                }/*while*/

                if ( line[lindex] == '"' ) {
                        index = lindex+1;
                        tok = STR;

                } else {
                        tok = line[index++];

                }/*if*/

        break;}

/* pass back to yacc & let it handle  - if not digit or alpha */
        default: {
                if ( isdigit( line[index] ) ) {
/* gobble up digits */
                        index++;
                        while ( isdigit(line[index]) ){
                                index++;
                        }/*while*/

                        tok = NUMBER;
                        break;

                } else if ( isalpha( line[index] ) ) {
```

```
                                        int     i, wlen = 1;
                                        index++;
/* gobble up associated chs */
                                        while ( isalpha( line[index] )
                                                || isdigit( line[index])
                                                || line[index] == '.' ){
                                                wlen++;
                                                index++;
                                        }/*while*/

/* now check against reserved word list */
                                        for ( i=0;
                                            rlist[i].word != NULL;
                                                i++ ) {

                                                if ( rlist[i].len
                                                        != wlen ) {
                                                        continue;
                                                }/*if*/

                                                if ( strncmp(
                                                  &line[index-wlen],
                                                  rlist[i].word,
                                                  wlen ) == 0 ) {

                                                 tok = rlist[i].tok;
                                                 break;
                                                }/*if*/
                                        }/*for*/

/* not a reserved word */
                                        if ( tok < 0 ) {
                                                tok = ID;
                                        }/*if*/
                                        break;

                                }/*if*/

                                tok = line[index++];

                        break; }/*default*/

                }/*switch*/

        break; }/*Rest*/

        case Eof: {
/* do we have some indentation to adjust for ... */
                if ( curind > 0 ) {
                        curind--;
                        tok = END;
                } else {
                        tok = 0;
                }/*if*/


        break; }/*Eof*/
```

```
	}/*switch*/

	}/*while*/

/* return whats required after setting yytext etc */
	if ( index > sind ) {
		int	i;
		yylen = index - sind;

		for ( i = 0; i < yylen; i++ ) {
			yytext[i] = line[sind+i];
		}/*for*/

		yytext[yylen] = '\0';

	} else {
		yylen = 0;
		yytext[0] = '\0';

	}/*if*/

/* debug report */
	if ( ldebug ) {
		fprintf( stderr, "yylex: token %d <%s>\n", tok, yytext );
	}/*if*/

	return( tok );

}/*yylex*/

/***********************************************************/

m_nulline()
/* return true if a null line */
{

	int	lindex=index;	/* local index */

/*  spaces */
	while ( line[lindex] == ' ' ) {
		lindex++;
	}/*while*/

/* any comment ? */
	if ( line[lindex] == '-' && line[lindex+1] == '-' ) {
		yylineno++;
		return( TRUE );

/*  end of the line */
	} else if ( line[lindex]== '\n' ) {
		yylineno++;
		return( TRUE );

	}/*if*/

	return( FALSE );
```

```
}/*m_nulline*/

/* end occam2lex.c */
```

Appendix B

Occam Production Rules for Yacc

PR1: program: sep process
or program: process

PR2: process: action sep
or SKIP sep
or STOP sep
or CASE selector sep
or CASE selector sep BEG selectlist END
or construct
or instance
or specification
or specification sep process
or caseinput
or allocation sep process
or error sep

PR3: action: assignment
or input
or output

PR4: selector: expr /* expression */

PR5: selectlist: select
or selectlist select

PR6: select: expr sep BEG process END
or ELSE sep BEG process END

PR7: instance: ID '('actualist')' sep
or ID '('')' sep

PR8: actualist: actual
or actualist comma actual

PR9: actual: element
or expr

PR10: allocation: PLACE ID AT expr ':'

PR11: construct: sequence
or parallel
or condition
or alternation
or loop

PR12: sequence: SEQ sep BEG proclist END
or SEQ replic sep BEG process END
or SEQ sep

PR13: parallel: PAR sep parproc5
or PAR sep parproc4
or PAR sep parproc3
or PAR sep parproc2
or PAR replic sep BEG process END
or PAR sep
or PRI PAR sep BEG proclist END
or PRI PAR replic sep BEG process END
or PRI PAR sep
or PLACED PAR sep BEG placelist END
or PLACED PAR replic sep BEG placement END
or PLACED PAR sep

PR14: parproc5: parproc3 sep parproc2

PR15: parproc4: parproc3 sep parproc

PR16: parproc3: parproc2 sep parproc

PR17: parproc2: parproc sep BEG proclist END

PR18: parproc: BEG proclist END

PR19: proclist: process
or proclist process

PR20: placelist: placement
or placelist placement

PR21: placement: PROCESSOR expr sep BEG process END

PR22: conditional: IF sep BEG choicelist END
or IF replic sep BEG choice END
or IF sep

PR23: choicelist: choice
or choicelist choice

PR24: choice: boolean sep BEG process END
or specification sep choice
or conditional

PR25: alternation: ALT sep BEG alternative_4 END
or ALT sep BEG alternative_3 END
or ALT sep BEG alternative_2 END
or ALT replic sep BEG alternative END
or ALT sep
or PRI ALT sep BEG alternativelist END
or PRI ALT replic sep BEG alternative END
or PRI ALT sep

PR26: replic: ID '=' base FOR count

PR27: base: expr

PR28: count: expr

PR29: alternative_4: alternative_3 sep alternative

PR30: alternative_3: alternative_2 sep alternative

PR31: alternative_2: alternative sep guard sep BEG process END

PR32: alternative: guard sep BEG process END
or specification sep alternative
or alternation

PR33: guard: boolean '&' input
or input
or boolean '&' SKIP

PR34: boolean: expr

PR35: loop: WHILE expr sep BEG process END

PR36: sep: EOL
or sep EOL

PR37: comma: ',' EOL
or ','

PR38: semicolon: ';' EOL
or ';'

PR39: specification: declaration
or definition

PR40: declaration: type namelist ':'

PR41: namelist: ID
or namelist comma ID

PR42: definition: PROTOCOL ID sep BEG CASE sep END ':'
or PROTOCOL ID sep BEG CASE sep BEG tagprotolist END END
':'

or PROC ID '('fparmlist')' sep BEG process END ':'

or PROC ID '('')' sep BEG process END ':'

or typelist FUNCTION ID '(' fparmlist ')' sep BEG valof END':'

or typelist FUNCTION ID '(' ')' sep BEG valof END':'

or typelist FUNCTION ID '(' fparmlist ')' IS ':'

or typelist FUNCTION ID '(' ')' IS ':'

PR43: tagprotolist: tagproto

or tagprotolist sep tagproto

PR44: tagproto: tag

or tag semicolon protocol

PR45: tag: ID

PR46: protocol: ANY

or ID

or INT

or BOOL

PR47: typelist: type

or typelist comma type

PR48: fparmlist: fparm

or fparmlist comma fparm

PR49: fparm: type ID

or VAL type ID

PR50: assignment: varlist ':' '=' explist

PR51: explist: expr

or explist comma expr

PR52: verlist: var

or varlist comma var

PR53: input: chan '?' inlist

or chan '?' CASE taggedlist

PR54: taggedlist: tag
or tag semicolon inlist

PR55: inlist: var
or var ':'':' expr
or inlist semicolon expr

PR56: caseinput: chan '?' CASE sep
or chan '?' CASE sep BEG variantlist END

PR57: variantlist: variant
or variantlist sep variant

PR58: variant: taggedlist sep BEG process END
or specification sep variant

PR59: output: chan '!' outlist
or chan '!' tag
or chan '!' tag semicolon outlist

PR60: outlist: expr
or expr ':'':' expr
or outlist semicolon expr

PR61: var: element

PR62: chan: element

PR63: element: ID
or element '['subscript']'
or '['element FROM subscript TO subscript ']'

PR64: subscript: expr

PR65: expr: monop operand
or operand dyop operand

or monop sep operand
or operand dyop sep operand
or operand
or conversation
or MOSTPOS type
or MOSTNEG type

PR66: monop: '_'
or NOT
or SIZE
or '~'

PR67: dyop: COMPOP
or '='
or SHIFTOP
or '+'
or '*'
or LOGOP
or BOOLOP
or '/'
or '\\'

PR68: conversation: type operand
or type ROUND operand
or type TRUNC operand

PR69: operand: literal
or '(' expr ')'
or '[' explist ']'
or '(' valof sep ')'
or ID '(' explist ')'
or ID '('')'

PR70: literal: NUMBER
or BOOL
or RNUMBER
or CHCON
or STR

or NUMBER '(' type ')'
or RNUMBER '(' type ')'
or CHCON '(' type ')'

PR71: valof: VALOF sep BEG process RESULT explist sep END
or specification sep valof

PR72: type: CHAN OF protocol
or PORT OF type
or TIMER
or BYTE
or INT
or INT16
or INT32
or INT64
or REAL32
or REAL64

Appendix C

Program 1

Conversation Mechanism Program Code

```
PROC Conversation (CHAN OF ANY keyboard, screen)

BOOL any1:
[6] INT save, init:

CHAN OF BOOL init.p1, init.p2, init.p3, init.q, init.def:
CHAN OF BOOL to.arbitrator.use.p, to.arbitrator.not.use.p:
CHAN OF BOOL to.arbitrator.use.q, to.arbitrator.not.use.q:
CHAN OF BOOL p.global.request, p.global.abort:
CHAN OF BOOL q.results.request, q.results.abort:
CHAN OF BOOL def.results.request, def.results.abort:

CHAN OF INT init.to.p1.first, init.to.p1.second:
CHAN OF INT init.to.p2.first, init.to.p2.second:
CHAN OF INT init.to.p3.first, init.to.p3.second:

[6] CHAN OF INT init.to.q, init.to.def, to.atpglobal.p, to.arbitrator.p:
[6] CHAN OF INT to.q.res.b, to.atq.q, to.arb.q, to.def.res.b, to.arb.def:

SEQ
                        -- This is process set vars. and est. rec. pt.
 SEQ
  PAR
   init[0] := 467
   init[1] := 129
   init[2] := 284
   init[3] := 672
   init[4] := 512
   init[5] := 499
   any1 := TRUE
  PAR i = 0 FOR 6
   save[i] := init[i]
  SEQ
   SEQ
    newline(screen)
    newline(screen)
    newline(screen)
    write.full.string (screen, "          ")
    write.full.string (screen, "C O N V E R S A T I O N    ")
    write.full.string (screen, "M E C H A N I S M")
    newline(screen)
    write.full.string (screen, "          ")
    write.full.string (screen, "---------------------------")
    write.full.string (screen, "-----------------")
    newline(screen)
    newline(screen)
    newline(screen)
```

```
    write.full.string (screen, "                    ")
    newline(screen)
    newline(screen)
    newline(screen)
    newline(screen)

   SEQ
    write.full.string (screen, "  The initial values are :    ")
    write.int (screen,init[0],0)
    write.full.string (screen, "    ")
    write.int (screen,init[1],0)
    write.full.string (screen, "    ")
    write.int (screen,init[2],0)
    write.full.string (screen, "    ")
    write.int (screen,init[3],0)
    write.full.string (screen, "    ")
    write.int (screen,init[4],0)
    write.full.string (screen, "    ")
    write.int (screen,init[5],0)
    newline(screen)
    newline(screen)
    newline(screen)

PAR
               -- This is process init. and pass initial data ...
 SEQ
  PAR
   init.p1 ! any1
   init.p2 ! any1
   init.p3 ! any1
   init.q ! any1
   init.def ! any1
  PAR
   PAR
    init.to.p1.first ! init[0]
    init.to.p1.second ! init[1]
   PAR
    init.to.p2.first ! init[2]
    init.to.p2.second ! init[3]
   PAR
    init.to.p3.first ! init[4]
    init.to.p3.second ! init[5]
   PAR i = 0 FOR 6
    init.to.q[i] ! init[i]
   PAR j = 0 FOR 6
    init.to.def[j] ! init[j]


 CHAN OF BOOL to.p1.buffer.request, to.p1.buffer.abort:
 CHAN OF BOOL to.p2.buffer.request, to.p2.buffer.abort:
 CHAN OF BOOL to.p3.buffer.request, to.p3.buffer.abort:
 CHAN OF BOOL obtain.data, not.obtain.data, atpglobal.request:

 [4] CHAN OF BOOL to.atpglobal.accept.p, to.atpglobal.not.accept.p:

 CHAN OF INT to.cb.p1.lowest.value, to.cb.p1.highest.value:
 CHAN OF INT to.cb.p2.lowest.value, to.cb.p2.highest.value:
 CHAN OF INT to.cb.p3.lowest.value, to.cb.p3.highest.value:
```

```
PAR

 CHAN OF BOOL init.atp1, atp1.request:
 CHAN OF INT p1.lowest.value, p1.highest.value:
 CHAN OF INT to.atp1.lowest.value, to.atp1.highest.value:
 PAR
                                    -- This is process p1
  INT temp1:
  [2] INT p1:
  SEQ
   SEQ
    init.p1 ? any1
   PAR
    init.to.p1.first ? p1[0]
    init.to.p1.second ? p1[1]
   SEQ
    init.atp1 ! any1
   IF
    p1[0] > p1[1]
     SEQ
      temp1 := p1[0]
      p1[0] := p1[1]
      p1[1] := temp1
    TRUE
     SKIP
   PAR
    p1.lowest.value ! p1[0]
    p1.highest.value ! p1[1]

                               -- This is process p1.buffer
  [2] INT p1b:
  SEQ
   PAR
    p1.lowest.value ? p1b[0]
    p1.highest.value ? p1b[1]
   atp1.request ? any1
   PAR
    to.atp1.lowest.value ! p1b[0]
    to.atp1.highest.value ! p1b[1]
   ALT
    to.p1.buffer.request ? any1
     PAR
      to.cb.p1.lowest.value ! p1b[0]
      to.cb.p1.highest.value ! p1b[1]
    to.p1.buffer.abort ? any1
     SKIP

                                   -- This is process atp1
  [2] INT atp1:
  BOOL acceptable:
  SEQ
   SEQ
    init.atp1 ? any1
   atp1.request ! any1
   PAR
    to.atp1.lowest.value ? atp1[0]
    to.atp1.highest.value ? atp1[1]
```

```
SEQ
  IF
    atp1[0] <= atp1[1]
      acceptable := TRUE
    NOT (atp1[0] <= atp1[1])
      acceptable := FALSE
IF
  acceptable
    to.atpglobal.accept.p[1] ! any1
  NOT acceptable
    to.atpglobal.not.accept.p[1] ! any1


CHAN OF BOOL init.atp2, atp2.request:
CHAN OF INT p2.lowest.value, p2.highest.value:
CHAN OF INT to.atp2.lowest.value, to.atp2.highest.value:
PAR
                                      -- This is process p2
  INT temp2:
  [2] INT p2:
  SEQ
    SEQ
      init.p2 ? any1
    PAR
      init.to.p2.first ? p2[0]
      init.to.p2.second ? p2[1]
    SEQ
      init.atp2 ! any1
    IF
      p2[0] > p2[1]
        SEQ
          temp2 := p2[0]
          p2[0] := p2[1]
          p2[1] := temp2
      TRUE
        SKIP
    PAR
      p2.lowest.value ! p2[0]
      p2.highest.value ! p2[1]

                                  -- This is process p2.buffer
  [2] INT p2b:
  SEQ
    PAR
      p2.lowest.value ? p2b[0]
      p2.highest.value ? p2b[1]
    atp2.request ? any1
    PAR
      to.atp2.lowest.value ! p2b[0]
      to.atp2.highest.value ! p2b[1]
    ALT
      to.p2.buffer.request ? any1
        PAR
          to.cb.p2.lowest.value ! p2b[0]
          to.cb.p2.highest.value ! p2b[1]
      to.p2.buffer.abort ? any1
        SKIP
```

```
                              -- This is process atp2
[2] INT atp2:
BOOL acceptable:
SEQ
  SEQ
    init.atp2 ? any1
  atp2.request ! any1
  PAR
    to.atp2.lowest.value ? atp2[0]
    to.atp2.highest.value ? atp2[1]
  SEQ
    IF
      atp2[0] <= atp2[1]
        acceptable := TRUE
      NOT (atp2[0] <= atp2[1])
        acceptable := FALSE
  IF
    acceptable
      to.atpglobal.accept.p[2] ! any1
    NOT acceptable
      to.atpglobal.not.accept.p[2] ! any1


CHAN OF BOOL init.atp3, atp3.request:
CHAN OF INT p3.lowest.value, p3.highest.value:
CHAN OF INT to.atp3.lowest.value, to.atp3.highest.value:
PAR
                              -- This is process p3
  INT temp3:
  [2] INT p3:
  SEQ
    SEQ
      init.p3 ? any1
    PAR
      init.to.p3.first ? p3[0]
      init.to.p3.second ? p3[1]
    SEQ
      init.atp3 ! any1
    IF
      p3[0] > p3[1]
        SEQ
          temp3 := p3[0]
          p3[0] := p3[1]
          p3[1] := temp3
      TRUE
        SKIP
    PAR
      p3.lowest.value ! p3[0]
      p3.highest.value ! p3[1]

                              -- This is process p3.buffer
  [2] INT p3b:
  SEQ
    PAR
      p3.lowest.value ? p3b[0]
      p3.highest.value ? p3b[1]
    atp3.request ? any1
    PAR
```

```
      to.atp3.lowest.value ! p3b[0]
      to.atp3.highest.value ! p3b[1]
    ALT
      to.p3.buffer.request ? any1
        PAR
          to.cb.p3.lowest.value ! p3b[0]
          to.cb.p3.highest.value ! p3b[1]
      to.p3.buffer.abort ? any1
        SKIP

                                   -- This is process atp3
  [2] INT atp3:
  BOOL acceptable:
  SEQ
    SEQ
      init.atp3 ? any1
    atp3.request ! any1
    PAR
      to.atp3.lowest.value ? atp3[0]
      to.atp3.highest.value ? atp3[1]
    SEQ
      IF
        atp3[0] <= atp3[1]
          acceptable := TRUE
        NOT (atp3[0] <= atp3[1])
          acceptable := FALSE
    IF
      acceptable
        to.atpglobal.accept.p[3] ! any1
      NOT acceptable
        to.atpglobal.not.accept.p[3] ! any1


                        -- This is process p.conversation.buffer
[6] INT p:
SEQ
  ALT
    obtain.data ? any1             -- Command from atpglobal
      SEQ
        PAR
          SEQ
            to.p1.buffer.request ! any1
            PAR
              to.cb.p1.lowest.value ? p[0]
              to.cb.p1.highest.value ? p[1]
          SEQ
            to.p2.buffer.request ! any1
            PAR
              to.cb.p2.lowest.value ? p[2]
              to.cb.p2.highest.value ? p[3]
          SEQ
            to.p3.buffer.request ! any1
            PAR
              to.cb.p3.lowest.value ? p[4]
              to.cb.p3.highest.value ? p[5]
      atpglobal.request ? any1
      PAR i = 0 FOR 6
        to.atpglobal.p[i] ! p[i]
```

```
   not.obtain.data ? any1
    PAR
      to.p1.buffer.abort ! any1
      to.p2.buffer.abort ! any1
      to.p3.buffer.abort ! any1
 ALT
   p.global.request ? any1
    PAR i = 0 FOR 6
      to.arbitrator.p[i] ! p[i]
   p.global.abort ? any1
    SKIP

                          -- This is process atp.global
[6] INT atpg:
[4] BOOL acceptable:
BOOL local.acceptable, global.acceptable, atp1, atp2:
SEQ
 global.acceptable := FALSE
 PAR i = 1 FOR 3                    -- Input local acceptance test
  ALT                            -- results from atp1, atp2, atp3
   to.atpglobal.accept.p[i] ? any1
    acceptable[i] := TRUE
   to.atpglobal.not.accept.p[i] ? any1
    acceptable[i] := FALSE
 local.acceptable := acceptable[1] AND acceptable[2] AND acceptable[3]
 IF
  local.acceptable
   SEQ
    obtain.data ! any1          -- Instruct p.conv.buf to get data
    atpglobal.request ! any1
                    -- get results from p.conversation.buffer
    PAR i = 0 FOR 6
     to.atpglobal.p[i] ? atpg[i]

                     -- compute acceptability of p results
    SEQ
     atp1 := (atpg[1]<atpg[0]) OR (atpg[2]<atpg[1])
     atp2 := (atpg[3]<atpg[2]) OR (atpg[4]<atpg[3]) OR (atpg[5]<atpg[4])
     IF
      (atp1) OR (atp2)
       global.acceptable := FALSE
      TRUE
       global.acceptable := TRUE

  NOT local.acceptable
   SEQ
    global.acceptable := FALSE
    not.obtain.data ! any1
 IF
  global.acceptable
   to.arbitrator.use.p ! any1
  NOT global.acceptable
   to.arbitrator.not.use.p ! any1
```

```
CHAN OF BOOL init.atq, atq.request:
PAR
                                        -- This is process q
 [6] INT q:
 SEQ
  SEQ
   init.q ? any1
  PAR i = 0 FOR 6
   init.to.q[i] ? q[i]
  SEQ
   init.atq ! any1

  BOOL notcompleted:
  INT index, lastindex, k, temp:
  SEQ
   index := 1
   lastindex := 5
   notcompleted := TRUE
   WHILE ((index <= 5) AND notcompleted)
    SEQ
     notcompleted := FALSE
     k := 5
     WHILE (k <> (index -1))
      SEQ
       IF
        q[k-1] > q[k]
         SEQ
          temp := q[k-1]
          q[k-1] := q[k]
          q[k] := temp
          notcompleted := TRUE
          lastindex := k
        TRUE
         SKIP
       k := k-1
     IF
      notcompleted
       index := lastindex +1
      TRUE
       SKIP
 PAR i = 0 FOR 6
   to.q.res.b[i] ! q[i]

                                -- This is process q.results.buffer
[6] INT qb:
SEQ
 PAR i = 0 FOR 6
   to.q.res.b[i] ? qb[i]
 atq.request ? any1
 PAR i = 0 FOR 6
  to.atq.q[i] ! qb[i]
 ALT
  q.results.request ? any1
   PAR i = 0 FOR 6
    to.arb.q[i] ! qb[i]
  q.results.abort ? any1
   SKIP
```

```
                                        -- This is process atq
BOOL atq1, atq2, acceptable.q:
[6] INT atq:
SEQ
  SEQ
    init.atq ? any1
  atq.request ! any1
  PAR i = 0 FOR 6
    to.atq.q[i] ? atq[i]
                        -- test q results for acceptability
  SEQ
    atq1 := (atq[1]<atq[0]) OR (atq[2]<atq[1])
    atq2 := (atq[3]<atq[2]) OR (atq[4]<atq[3]) OR (atq[5]<atq[4])
    IF
      (atq1) OR (atq2)
        acceptable.q := FALSE
      TRUE
        acceptable.q := TRUE

  IF
    acceptable.q
      to.arbitrator.use.q ! any1
    NOT acceptable.q
      to.arbitrator.not.use.q ! any1


PAR
                                        -- This is process default
  [6] INT default:
  SEQ
    SEQ
      init.def ? any1
    PAR i = 0 FOR 6
      init.to.def[i] ? default[i]
    INT index, pos, temp, k:
    SEQ
      index := 0
      WHILE (index <> 5)
        SEQ
          pos := index
          k := index +1
          WHILE (k <> 6)
            SEQ
              IF
                default[k] < default[pos]
                  pos := k
                TRUE
                  SKIP
              k := k +1
          IF
            pos <> index
              SEQ
                temp := default[pos]
                default[pos] := default[index]
                default[index] := temp
            TRUE
              SKIP
```

```
      index := index +1

    PAR i = 0 FOR 6
      to.def.res.b[i] ! default[i]

                      -- This is process default.results.buffer
  [6] INT defb:
  SEQ
    PAR i = 0 FOR 6
      to.def.res.b[i] ? defb[i]
    ALT
      def.results.request ? any1
        PAR i = 0 FOR 6
          to.arb.def[i] ! defb[i]
      def.results.abort ? any1
        SKIP


SEQ
                              -- This is process Arbitrator
  [6] INT final:              --
  INT char:                   --
  SEQ                     -- arbitrate according to ENSURE notation
    ALT                       --
      to.arbitrator.use.p ? any1    -- atpglobal -> acceptable
        PAR                   -- use p results, abort q and default
          SEQ
            p.global.request ! any1
            PAR i = 0 FOR 6
              to.arbitrator.p[i] ? final[i]
            SEQ
              write.full.string (screen, "  The final values are :    ")
              write.int (screen, final[0],0)
              write.full.string (screen, "    ")
              write.int (screen, final[1],0)
              write.full.string (screen, "    ")
              write.int (screen, final[2],0)
              write.full.string (screen, "    ")
              write.int (screen, final[3],0)
              write.full.string (screen, "    ")
              write.int (screen, final[4],0)
              write.full.string (screen, "    ")
              write.int (screen, final[5],0)
              newline(screen)
              newline(screen)
              newline(screen)
              newline(screen)
              write.full.string (screen, "              ")
              write.full.string (screen, "The    P    results were used.")
              newline(screen)
              newline(screen)
              newline(screen)
              newline(screen)

          q.results.abort ! any1
          BOOL dummy:
          SEQ
            ALT
```

```
    to.arbitrator.use.q ? dummy
      SKIP
    to.arbitrator.not.use.q ? dummy
      SKIP
  def.results.abort ! any1
to.arbitrator.not.use.p ? any1
 PAR
  p.global.abort ! any1         -- abort p.conversation.buffer
  ALT
   to.arbitrator.use.q ? any1       -- atq -> acceptable
    PAR
     SEQ
      q.results.request ! any1
      PAR i = 0 FOR 6
       to.arb.q[i] ? final[i]
      SEQ
       write.full.string (screen, " The final values are : ")
       write.int (screen, final[0],0)
       write.full.string (screen, "    ")
       write.int (screen, final[1],0)
       write.full.string (screen, "    ")
       write.int (screen, final[2],0)
       write.full.string (screen, "    ")
       write.int (screen, final[3],0)
       write.full.string (screen, "    ")
       write.int (screen, final[4],0)
       write.full.string (screen, "    ")
       write.int (screen, final[5],0)
       newline(screen)
       newline(screen)
       newline(screen)
       newline(screen)
       write.full.string (screen, "                    ")
       write.full.string (screen, "The Q results were used.")
       newline(screen)
       newline(screen)
       newline(screen)
       newline(screen)

     def.results.abort ! any1
   to.arbitrator.not.use.q ? any1     -- atq -> unacceptable
    PAR
     SEQ
      def.results.request ! any1
      PAR i = 0 FOR 6
       to.arb.def[i] ? final[i]
      SEQ
       write.full.string (screen, " The final values are : ")
       write.int (screen, final[0],0)
       write.full.string (screen, "    ")
       write.int (screen, final[1],0)
       write.full.string (screen, "    ")
       write.int (screen, final[2],0)
       write.full.string (screen, "    ")
       write.int (screen, final[3],0)
       write.full.string (screen, "    ")
       write.int (screen, final[4],0)
       write.full.string (screen, "    ")
```

```
          write.int (screen, final[5],0)
          newline(screen)
          newline(screen)
          newline(screen)
          newline(screen)
          write.full.string (screen, "                    ")
          write.full.string (screen, "The DEFAULT results were used.")
          newline(screen)
          newline(screen)
          newline(screen)
          newline(screen)

        q.results.abort ! any1       -- Abort q.results.buffer
SEQ
  write.full.string (screen, "                    ")
  write.full.string (screen, "Press any key to return to TDS2 ")
  read.char (keyboard, char)

:
```

Program 2

Occam Program Code for Vehicle Detection System

```
-- PROGRAM speed_trap

... occam.utilities.fold

VAL  invalid IS 0:

-- channel declarations

[2] CHAN OF INT sensorinput, counterchan, speedchan:
CHAN OF INT counteroutput, speedoutput:
[2] CHAN OF INT sensorinit:
CHAN OF INT counterinit, speedinit, receiverinit:
[2] CHAN OF INT haltsensor, haltspeed, haltcounter:
CHAN OF INT haltspeedoutput, haltcountoutput:

-- process declarations

PROC sensor(VAL INT i)
  TIMER TIME :
  INT value:
  INT any:
  BOOL readsensors:
  SEQ
    sensorinit[i] ? any
    readsensors := TRUE
    WHILE readsensors
      ALT
        sensorinput[i] ? any
          PAR
            counterchan[i] ! any
            SEQ
              TIME ? value
              speedchan[i] ! value
        haltsensor[i] ? any
          PAR
            readsensors := FALSE
            haltspeed[i] ! any
            haltcounter[i] ! any
:


PROC speed ()
  [2] INT value:
  [2] BOOL readspeed:
  INT timelapse:
  INT any:
  SEQ
    speedinit ? any
```

```
    PAR
      readspeed[0] := TRUE
      readspeed[1] := TRUE
      timelapse := invalid
    WHILE readspeed[0] OR readspeed[1]
      SEQ
        ALT
          haltspeed[0] ? any
            readspeed[0] := FALSE
          haltspeed[1] ? any
            readspeed[1] := FALSE
          speedchan[0] ? value[0]
            ALT
              haltspeed[0] ? any
                readspeed[0] := FALSE
              haltspeed[1] ? any
                readspeed[1] := FALSE
              speedchan[1] ? value[1]
                timelapse := (value[1]-value[0])
              speedchan[0] ? value[0]
                timelapse := invalid
          speedchan[1] ? value[1]
            ALT
              haltspeed[0] ? any
                readspeed[0] := FALSE
              haltspeed[1] ? any
                readspeed[1] := FALSE
              speedchan[0] ? value[0]
                timelapse := (value[1]-value[0])
              speedchan[1] ? value[1]
                timelapse := invalid
        speedoutput ! timelapse
    haltspeedoutput ! any
:



-- counter
PROC counter()
  [2] BOOL counting:
  INT count:
  INT any:
  SEQ
    counterinit ? any
    PAR
      counting[0] := TRUE
      counting[1] := TRUE
      count := 0
    WHILE counting[0] OR counting[1]
      SEQ
        ALT
          haltcounter[0] ? any
            counting[0] := FALSE
          haltcounter[1] ? any
            counting[1] := FALSE
          counterchan[0] ? any
            ALT
              haltcounter[0] ? any
```

```
            counting[0] := FALSE
          haltcounter[1] ? any
            counting[1] := FALSE
          counterchan[1] ? any
            count := count + 1
          counterchan[0] ? any
            SKIP
      counterchan[1] ? any
        ALT
          haltcounter[0] ? any
            counting[0] := FALSE
          haltcounter[1] ? any
            counting[1] := FALSE
          counterchan[0] ? any
            count := count - 1
          counterchan[1] ? any
            SKIP
    counteroutput ! count
  haltcountoutput ! any
:


PROC remotecontroller()
  INT any:
  SEQ
    PAR
      PAR i = 0 FOR 2
        sensorinit[i] ! any
      counterinit ! any
      speedinit ! any
      receiverinit ! any
    -- await terminating condition
    PAR i = 0 FOR 2
      haltsensor[i]  ! any
:



PROC remotereceiver()
  INT count, timelapse:
  INT any:
  BOOL counting, readspeed:
  SEQ
    receiverinit ? any
    PAR
      counting := TRUE
      readspeed := TRUE
    WHILE counting OR readspeed
      ALT
        haltcountoutput ? any
          counting := FALSE
        counteroutput ? count
          -- act on count
        haltspeedoutput ? any
          readspeed := FALSE
        speedoutput ? timelapse
          -- act on timelapse
:
```

```
-- main program section
PAR
  PAR i = 0 FOR 2
    sensor(i)
  counter()
  speed()
  remotecontroller()
  remotereceiver()
```