1

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

National Library
of Canada

Bibliothèque nationale
du Canada

CANADIAN THESES
ON MICROFICHE

THÈSES CANADIENNES
SUR MICROFICHE

0-315-22185-2

NAME OF AUTHOR/NOM DE L'AUTEUR

ANGUS HAU-CHUNG MA

TITLE OF THESIS/TITRE DE LA THÈSE

A COMPARATIVE STUDY OF SPECIFICATION AND
VERIFICATION TECHNIQUES FOR PARALLEL
PROCESSING SYSTEMS

UNIVERSITY/UNIVERSITÉ

CARLETON UNIVERSITY

DEGREE FOR WHICH THESIS WAS PRESENTED/

GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE

MASTER OF ENGINEERING

YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE GRADE

1984

NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE

DR. M. WOODSIDE

Permission is hereby granted to the NATIONAL LIBRARY OF
CANADA to microfilm this thesis and to lend or sell copies
of the film.

The author reserves other publication rights, and neither the
thesis nor extensive extracts from it may be printed or other-
wise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈ-
QUE NATIONALE DU CANADA de microfilmer cette thèse et
de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la
thèse ni de longs extraits de celle-ci ne doivent être imprimés
ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/DATE

April 1984

SIGNED/SIGNÉ

PERMANENT ADDRESS/RÉSIDENTIE FIXE

701-1144 ROCKINGHAM AVE.

OTTAWA, ONTARIO

CANADA K1M 8L7

A COMPARATIVE STUDY OF SPECIFICATION AND VERIFICATION
TECHNIQUES FOR PARALLEL PROCESSING SYSTEMS

by

© Angus H.C. Ma, B.Sc. (E.E.)
Queen's University, 1980

A Thesis

submitted to the Faculty of Graduate Studies in
partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING

Department of Systems and Computer Engineering
Carleton University

Ottawa, Canada

April, 1984

LICENCE TO CARLETON UNIVERSITY

In the interests of facilitating research by others at this institution and elsewhere, I hereby grant a licence to:

CARLETON UNIVERSITY

to make copies of my thesis:

A Comparative Study of Specification & Verification
Techniques for Parallel Processing Systems

or substantial parts thereof, the copyright which is invested in me, provided that the licence is subject to the following conditions:

Only singled copies shall be made or authorized to be made at any one time, and only in response to a written request from the library of any university or similar institution on its own behalf or on behalf of one of its users.

This licence shall continue for the full term of the copyright, or for so long as may be legally permitted.

The Universal Copyright Notice shall appear on the title page of all copies made under the authority of this licence.

This licence does not permit the sale of authorized copies at a profit, but does permit the collection by the institution or institutions concerned of charges covering actual costs.

All copies under the authority of this licence shall bear a statement to the effect that the copy in question "is being made available in this form by the authority of the copyright owner solely for the purpose of private study and research and may not be copied or reproduced except as permitted by the copyright laws without written authority from the copyright owner."

The foregoing shall in no way preclude the granting by the author of a licence to the National Library of Canada to reproduce the thesis and to lend or to sell copies of the same. For this purpose it shall also be permissible for Carleton University to submit the abovementioned thesis to the National Library of Canada.

Signature of Witness

Apr 25 M.E.
Date Degree

Signature of Student

Systems & Comp. Eng.
Department of Student

The undersigned recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

A COMPARATIVE STUDY OF SPECIFICATION AND VERIFICATION
TECHNIQUES FOR PARALLEL PROCESSING SYSTEMS

submitted by Angus H.C. Ma, in partial fulfillment of the
requirements for the degree of

MASTER OF ENGINEERING

Thesis Supervisor

Chairman, Department
of Systems and Computer Engineering

Carleton University

April, 1984

ABSTRACT

This thesis is a comparative study of ways to specify and to verify the interactions between independent and asynchronous processes in a parallel processing environment. In recent years, as a result of decreasing cost of hardware, distributed processing systems have become increasingly cost effective. In these distributed processing systems, there are usually a large number of processes. The task of programming is difficult because these systems are typically concurrent, asynchronous and non-deterministic. A methodology is therefore needed to help designers specify and verify the operation of these systems.

In this thesis, three distinct approaches to the problem are identified - transition models, programming language models and hybrid models. The three models are examined in terms of their relative strengths and weaknesses. This study indicates that programming language models offer the greatest flexibility for general application.

Beginning with a programming language (CSP) that Hoare has proposed, some extensions are proposed in this thesis to make it more complete. The technique of symbolic execution is generalised so that it can be applied to CSP. In order to test the usefulness of CSP and symbolic execution, a hypothetical telephone switching system is modelled and verified. This exercise shows that symbolic execution, while easy to understand, is powerful enough to handle most of the problems in program verification. It is further concluded that more research in symbolic execution is necessary to investigate new techniques to detect race conditions without causing a "path explosion".

ACKNOWLEDGEMENTS

The author wishes to express his gratitude and respect for the late Professor Sigurdson who acted as Thesis Supervisor before his premature death. Without his support and assistance, this study probably would never have started. The author also wishes to thank Professor Woodside who acted as Thesis Supervisor, for his encouragement and advice throughout this study.

The author also wishes to thank his family and close friends for their patient support.

C O N T E N T S

page number

CHAPTER 1 INTRODUCTION

1.1 Thesis Objective	1.1
1.2 Thesis Organization	1.2

CHAPTER 2 BASIC CONCEPTS

2.1 System Processes (Internal Processes)	2.1
2.2 External Processes	2.3
2.3 Process State	2.3
2.4 System State	2.4
2.5 Process Specification	2.5
2.6 System Specification	2.5
2.7 Parallelism	2.5
2.8 Inter-Process Communications	2.5

CHAPTER 3 SPECIFICATION AND VERIFICATION

3.1 The Hierarchy of Specifications	3.1
3.2 Correctness Criteria of a PPS	3.4
3.3 Verification and the Hierarchy of Specifications	3.6
3.4 Specification Language	3.6
3.5 Commonly Used Specification and Verification Techniques	3.8

CHAPTER 4 STATE TRANSITION MODELS

4.1 Finite-State Machine (FSM) Model	4.1
--	-----

4.2 Verification of a FSM Model	4.3
4.3 Petri Net Model	4.5
4.4 Verification of Petri Net Models	4.7
4.5 Strengths and Weaknesses of Transition Models ..	4.8

CHAPTER 5 HYBRID MODELS

5.1 Enhanced FSM Model	5.1
5.2 Keller's Model	5.2
5.3 A Bounded Buffer: An Example	5.5
5.4 Strengths and Weaknesses of Hybrid Models	5.7

CHAPTER 6 A PROGRAMMING LANGUAGE MODEL - CSP

6.1 Advantages and Disdvantages of a Programming Language Model	6.1
6.2 Communicating Sequential Processes (CSP)	6.2
6.2.1 The Parallel Command	6.3
6.2.2 Inter-Process Communications	6.4
6.3.3 The Guarded Command	6.5
6.2.4 The Alternative Command	6.6
6.2.5 The Repetitive Command	6.7
6.3 Extensions to CSP	6.7
6.3.1 The Exit Statement	6.8
6.3.2 The Else clause	6.9
6.3.3 Constructors and Signals	6.10
6.4 Relationships between CSP and Keller's Model ..	6.11

CHAPTER 7 VERIFICATION OF A PROGRAMMING LANGUAGE MODEL

7.1 Program Proving	7.1
---------------------------	-----

7.2 Proof Systems Based on CSP	7.3
7.3 Basic Concepts of Symbolic Execution	7.6
7.4 I/O Commands	7.8
7.5 Parallelism	7.9
7.6 Non-Determinacy	7.17
7.7 Strengths and Weaknesses of Symbolic Execution in CSP	7.17

CHAPTER 8 MODELLING AND ANALYSIS OF A TELEPHONE SWITCHING SYSTEM

8.1 Basic Concepts in Telephony and Telephone Switching	8.1
8.2 Trunk-to-Trunk Call	8.2
8.3 TSS as a Stored Program Controlled Digital Toll Switch	8.4
8.3.1 Telephone Switching Equipment	8.4
8.3.2 System Architecture of TSS	8.5
8.3.3 Trunk Supervision and Fault Detection	8.8
8.3.4 A Trunk-to-Trunk Call in the Context of TSS	8.9
8.4 Modelling TSS Using CSP	8.10
8.5 Verification of TSS Using Symbolic Execution ..	8.11

CHAPTER 9 CONCLUSIONS

Appendix A: CSP Model of TSS

Appendix B: Verification Procedure of TSS

REFERENCES

$\geq$

Greater Than or Equal to

$\leq$

Less Than or Equal to

*

Multiplication

LIST OF FIGURES

- Figure 2.A An Example of a PPS
- Figure 3.A The Hierarchy of Specifications in the Design of a Communication Protocol
- Figure 4.A An Example of a FSM
- Figure 4.B A Simple Computer Protocol with Two Users and Two Communication Stations
- Figure 4.C FSM Model of the Communication Protocol
- Figure 4.D Global State Transition Diagram for the Communication Protocol
- Figure 4.E An Example of a State Transition Diagram with a Deadlock and a Livelock
- Figure 4.F An Example of a Petri Net
- Figure 4.G The Same Petri Net After Transitions A and B Have Fired
- Figure 4.H Petri Net Model of the Communication Station in Figure 4.B
- Figure 4.I Global State Transition Diagram for the Petri Net in Figure 4.H
- Figure 5.A Enhanced FSM Model for a Bounded Buffer
- Figure 5.B Keller's Model for the Bounded Buffer
- Figure 5.C State Transition Diagram for Keller's Model
- Figure 6.A Keller's Model of I/O Commands
- Figure 6.B Keller's Model of an Alternative Command
- Figure 6.C Keller's Model of a Repetitive Command
- Figure 7.A Verification Tree with Race Conditions Checked
- Figure 7.B Verification Tree with Race Conditions not Checked
- Figure 8.A A Sample Telephone Network
- Figure 8.B A Block Diagram of TSS
- Figure 8.C A Trunk-to-Trunk Call between Trunks i and j
- Figure 8.D TSS within a Telephone Switching Network
- Figure 8.E Timing Diagram of a Trunk-to-Trunk Call
- Figure 8.F Verification Tree for TSS

CHAPTER 1 INTRODUCTION

1.1 Thesis Objective

This thesis is a comparative study of ways to specify and to verify the interactions between independent and asynchronous logical entities in a Parallel Processing System (PPS). Each logical entity corresponds to a physical entity such as a chip, a processor or even a person. However, most of these logical entities that will be considered in this thesis will be software entities, that is, processes.

In recent years, as a result of the decreasing cost of hardware, there has been a proliferation of distributed processing systems. Examples of such systems are computer networks, real-time control systems and distributed data bases. In these distributed processing systems, there are a large number of asynchronous and parallel processes each of which has to be programmed to do precisely the right things at the right time in order to produce the correct results. The task of programming is difficult because these systems are inherently concurrent, asynchronous and non-deterministic. The objective of this thesis is to treat these systems as parallel processing systems and to analyse their behaviour. In general, a PPS consists of a network of communicating processes which cooperate to produce some "desirable" results. The "desirable" results are commonly quantified by a set of assertions which have to hold for the system to be considered correct.

Current research into this problem can generally be found in three different areas - communication protocols, distributed algorithms and software engineering. In a communication protocol, the processes represent the various communication stations (or parties) which interact in order

to achieve the ultimate goal of synchronization and data transfer. Researchers in protocol design are interested in finding ways to represent protocols and to analyse them to reveal potential deadlocks. [SCHW82], [BOCH80], [DAY79] and [SUNS79] contain surveys of research in this area. In distributed algorithms and other resource sharing systems, researchers are interested in analysing systems to ensure that mutual exclusion is achieved and that the scheduling scheme is fair such that nobody will be deprived of a resource permanently. Examples of such research can be found in [KOHL81] and [RICA81]. Finally, there is interest in PPS from the point of view of software engineering. Here, researchers are interested in developing a ~~design~~ methodology for parallel programs in general, that is, how should a parallel program be specified, how should it be written (with what synchronisation primitives) and how should it be analysed in order to achieve correctness. [CHEN83], [OWIC82], [MISR81] and [FLON81] are some examples of verification techniques for general parallel programs. Another objective of this thesis is to survey the techniques used in these different areas and to find a general method that can be applied to all of these problems.

1.2 Thesis Organization

In this thesis, three classes of specification techniques will be examined in detail - transition models, programming language models and hybrid models which combine transition models with programming language models.

Having studied the three classes of specification techniques the author has concluded that for general applications, programming language models offer the greatest flexibility. Beginning with a programming language (CSP) that Hoare proposed in [HOAR78], the author has added some extensions

to it to make it more complete. The author also adapted the technique of symbolic execution to verify CSP models. Finally, the author applied the techniques to a hypothetical telephone switching system. A CSP model was developed for the switching system and was verified using symbolic execution.

To summarize, the author's contributions in this research are as follows.

1. Extensions to CSP.
2. Application of symbolic execution to CSP.
3. Modelling of a telephone switching system using CSP with extensions.
4. Verification of the model using symbolic execution.

The thesis is organized into nine chapters. The next chapter describes the basic components and characteristics of a PPS. Chapter 3 defines the meaning of specification and verification with respect to a PPS. Currently used techniques are surveyed and the three classes of modelling techniques are identified. Chapter 4 introduces two transition models, the finite state machine model and the Petri Net model. Verification techniques for each will be discussed. Chapter 5 will describe two hybrid models based on the two transition models in chapter 4. Chapter 6 introduces CSP as a programming language model. The author's proposed extensions to CSP will also be described. Chapter 7 surveys and compares verification techniques for parallel programs and demonstrates how symbolic execution can be adapted to verify CSP models. Chapter 8 introduces a hypothetical telephone switching system TSS and presents its CSP model. The model will be analysed to verify that the telephone switch is deadlock free and it releases all system resources when a telephone call is terminated. Chapter 9 concludes the thesis by discussing the strengths

and weaknesses of the technique proposed in the thesis.

CHAPTER 2 BASIC CONCEPTS

This chapter discusses the basic components of a PPS. Since this thesis is concerned mostly with software entities, they will hereafter be referred to as processes. A process is also known as a task in some programming languages such as ADA. A PPS consists of a network of communicating processes. These processes are known as system or internal processes and they are designed to co-operate in order to produce some "desirable" results. Most of the concepts discussed in this chapter are taken from [CHEN83], [MISR81] and [KELL76]. The reader is also referred to [CASH80] for a survey of synchronization primitives in various parallel programming languages.

Figure 2.A shows an example of a PPS. It consists of 7 system processes which run on 3 processors. The PPS interacts with 2 external processes.

2.1 System Processes (Internal Processes)

A PPS consists of a set of network of system processes. In this thesis, all system processes communicate exclusively via messages. While in some systems, processes communicate via shared variables instead of messages, it can be shown that interactions through shared variables can be achieved through messaging and vice versa ([CASH80]). However, message-based systems have two distinct advantages. First of all, some systems may not have shared memory that is accessible by more than one system process. In a distributed system for example, the private memory of one processor may be accessible only to the process(es) running on that processor. While it is possible to implement shared memory in a distributed system, message-based

Figure 2.A An Example of a PPS

systems are by far cheaper to implement and hence more common. Secondly, process interactions in a message-based system are more controlled in that a sender names an explicit receiver. In a shared-memory system, a process essentially "broadcasts" its message by writing to common memory and may cause unintentional reactions by any process which happens to be accessing the shared memory.

Thus all systems will be assumed to be message-based. Where this is not the case, the system will be modelled as an equivalent message-based system.

A system process is said to be a slave process if it is totally input driven, that is, in the absence of any input, the process will forever stay in the same state and will not generate any output. For example, a server process is a slave process which remains idle until it receives an order to do work. An autonomous process is one that can undergo state transitions and generate outputs without any input. A fault generator is an autonomous process which under non-determinate conditions, introduces faults to the system.

A terminating process is one which is designed to terminate after a period of operation. An example of such a process may be a maintenance process which is created whenever a fault is detected in the system. It will perform a series of tests to isolate and diagnose the problem and will then terminate. A non-terminating process is one that runs forever. An example is a server process which renders a frequently required service. When it receives a request for service, it performs some actions and then waits for the next request to arrive.

If all the internal processes of a PPS are terminating processes, then the system itself may eventually terminate

when all of its processes have terminated.

2.2 External Processes

External processes are processes which interact with the PPS. In most cases, they are also the users of the PPS. An example of a PPS is a computer system. Each user of the system can be considered an external process. Inputs from the user are received by a command interpreter (a system process) which then transmits the request from the user to other system processes such as a magnetic tape process to mount a tape or a batch process to run a batch job. The replies from these processes are sent back to the command interpreter which then displays the messages to the user.

2.3 Process State

The state of a process at a given time defines what the process is doing at that particular moment. The state of the process implicitly defines what inputs the process is waiting for and what actions it will take when it receives an input.

In [KELL76], Keller defines a process state to be a product of the control state and the data state. In terms of the program which defines the operation of the process, the control state is the instruction pointer whereas the data state is the vector of all the private variables of that process. Keller also defines the operation of a process as a sequence of state transitions each of which takes the form $q \rightarrow q'$ where q is the current state and q' is the next state. The state transition is considered an indivisible operation.

Consider the following example. If we are designing a microprocessor-based telephone set, we might program the telephone process to be in one of the following states - idle, dialling, ringing and talking. In the idle state, that is, when the handset is onhook and the phone is not ringing, the process can accept two inputs, one being a ring signal from the central office and the other being an offhook signal generated when the subscriber picks up the handset. In the case of the first input, the telephone will ring and the telephone process is now in the ringing state. In the case of the second input, the process will react by sending an offhook signal down to the central office and the telephone process will now be in the dialling state.

A process is a finite-state process if it can only be in a finite number of states. It should be noted that the state transition of a slave process can always be attributed to an input (or inputs) from other processes. In contrast, the state transition of an autonomous process may be spontaneous.

2.4 System State

The state of a PPS is the Euclidean product of the states of all the system processes. If there are n system processes in the system, the system state will be represented by the n -tuple

$$(S_1, S_2, S_3, \dots, S_n)$$

where S_i is the process state of the process P_i .

2.5 Process Specification

The process specification defines the behaviour of a process. Given the present state and the present input of a process, the process specification defines the next state of the process and its output.

In chapter 3, the characteristics of a process specification will be discussed in detail.

2.6 System Specification

While the individual process specifications determine the behaviour of the individual processes, the system specification determines the behaviour of the whole system. In a PPS, the system specification is simply the specifications of all the internal processes and the specification of the process topology, that is, the interconnections of all the processes.

2.7 Parallelism

In a PPS, it is assumed that all processes are truly parallel, that is, they can all be running at the same time which will be the case if each process runs on a dedicated processor. In reality, at least some of the processes will share the same processor. However, the sharing of a processor affects only the real-time behaviour but not the logical behaviour of a system.

2.8 Inter-Process Communications

As mentioned before, a PPS is assumed to be message-based. It will further be assumed that basic communication is

reliable, that is, if one process sends a message to another process, the destination process will be able to receive the message without any data corruption. Where the is not the case, the communication medium itself will be modelled as a separate system process which is capable of corrupting or losing messages ([BRAN78]).

CHAPTER 3 SPECIFICATION AND VERIFICATION

This chapter describes the meaning of specification and verification in the context of a PPS. To a large extent, specification and verification of a PPS is a direct generalisation of the concept of program specification and program proving. In this chapter, the concept of a hierarchy of specifications will be introduced. It will be shown that a specification is in fact an implementation of a previous specification. The meaning of verification and the specific verification conditions will be outlined followed by a discussion on the characteristics of a good model. The last section contains a summary of current modelling and verification techniques.

3.1 Hierarchy of Specifications

To understand the concept of design verification, we must first understand the relationships between the specification and the implementation of a design. In this section, the concept of a hierarchy of specifications will be introduced which explains these relationships.

The concept of the hierarchy of specifications is a generalisation of the concept of the equivalence of programs in [BIRM76]. In the paper, Birman and Joyner define program correctness in terms of the equivalence of two programs, the "specification" and the "implementation". In other words, one verifies an implementation by proving that it is equivalent to its specification. This concept can be generalised as follows. By employing a top-down design approach, a hierarchy of specifications will be generated during the design process. Top-down design [WIRT71] refers to the design approach whereby an initial

idea is refined in a stepwise fashion with more details (and hence design decisions) added during each iteration until an implementable design is generated. The hierarchy of specifications is generated when we start with an initial specification and during each iteration of the design, implement the previous specification which then becomes the new specification for the next iteration.

To illustrate the concept, consider the hypothetical design process of a computer. The initial specification of a computer might be a device that can calculate projectiles automatically. After carefully considering the problem of calculating projectiles, the designer comes up with an algorithm which involves the use of some variables and some basic primitives such as FOR, GOTO, +, -, * and / with which one will be able to calculate projectiles automatically. Thus after the first iteration, the specification for a computer has changed from "a device which calculates projectiles automatically" to "a device which can execute a set of predefined primitives such as FOR, GOTO, +, -, *, and / on a set of variables". During the next iteration, the designer comes up with an even lower level implementation of those primitives using instructions such as PUSH, POP, JMP, ADD, SUB, etc. which operate on the stack, memory registers and memory locations. These "machine instructions" then become the new specification for the computer. Finally the designer designs a circuit which implements those instructions using gates, flip-flops and memory.

In [BIRM74] for example, the instruction set of the S-machine implements all the functions required by a high level language. The instruction set itself must then be implemented at the micro-instruction level by microcode. Thus the high-level language, the S-machine and the microcode are three different specifications of a computer

in a descending order of abstraction.

To summarize, during the design process, a sequence of specifications

$S_0, S_1, S_2, \dots, S_n$

are generated with S_0 being the initial specification and S_n being the directly implementable specification. The sequence appears in a descending order of abstraction. S_0 describes the design in the broadest and most abstract terms. After each iteration, more and more implementation details are added until all design issues have been resolved in S_n . Each design is an implementation of the preceding specification which is also the specification for the succeeding design.

For example, in the area of protocol design, it is usually possible to identify a hierarchy of three specifications - protocol intent, protocol specification and protocol implementation ([SCHW82], [BOCH80] and [SUNS79]). The protocol intent, also known as service specification, defines the environment that the protocol must operate in and what the protocol is supposed to achieve. Thus the protocol intent is concerned with issues such as the number of parties involved, whether transmission is half-duplex, full-duplex and whether or not there is a master-slave relationship among the parties. In other words, it describes what services the protocol must deliver to the users. Knowing what conditions and constraints there are, the protocol specification implements the protocol intent by defining the mechanics of a protocol that meets the requirements of the protocol intent. Thus the protocol specification is concerned with the syntax and semantics of a protocol. For example, a protocol specification for HDLC [CARL80] will contain the meaning and usage of commands

LIST OF ABBREVIATIONS AND SYMBOLS

A/D	Analog-to-Digital
CC	Central Control
CO	Central Office
CP	Call Process
CSP	Communicating Sequential Processes
DMS	Digital Multiplex System
D/A	Digital-to-Analog
FG	Fault Generator
FSM	Finite State Machine
NM	Network Module
I	Set of Inputs
I/O	Input/Output
M	Output Function
N	State Transition Function
NP	Network Process
O	Set of Outputs
P	Set of Places
Post	Backward Incidence Function
Pre	Forward Incidence Function
PPS	Parallel Processing System
T	Set of Transitions
TK	Trunk
TM	Trunk Module
TO	Toll Office
TP	Terminal Process
TSS	Telephone Switching System
VC	Verification Condition
X	Set of System States
$\Rightarrow$	Implication
!	Output
?	Input
$:=$	Assignment
$\langle \rangle$	Not Equal

such as Receive Ready (RR), Receive Not Ready (RNR), Reject (REJ), Selective Reject (SREJ) etc. and how data transfer can be accomplished using these commands. The protocol implementation implements the protocol specification by defining a particular algorithm that satisfies the requirements of the protocol specification. Thus the protocol implementation can be a program if the protocol is implemented in software or a chip or circuit if the protocol is implemented in hardware. In the latter case, it is possible to split up the protocol implementation into a high level implementation and a low level implementation. Figure 3.A shows the hierarchy of specifications in protocol design.

3.2 Correctness Criteria of a PPS

Proving that a sequential program is correct generally involves proving two things - program termination and partial correctness. Program termination means that the program will eventually terminate and partial correctness means that whenever the program terminates, a set of assertions which define the intent of the program will be true. The correctness criteria for parallel programs are generalisations of these two conditions ([OWIC82], [FLON81], [APT80], [SUNS79], [MERL79], [KELL76]) and are listed as follows.

1. Partial Correctness ([LAMP80])

For a PPS that is designed to terminate after a period of operation, partial correctness is defined in exactly the same way as in a sequential program. Whenever the PPS terminates, a set of assertions which define the intent of the system must hold. In a non-terminating PPS, the processes are designed to return to their home-states (instead of terminating) after a period of

Figure 3.A The Hierarchy of Specifications in the
Design of a Communication Protocol

4
operation. In this case, partial correctness requires that the set of assertions hold true whenever the system reaches its home-state (which is the state where all the internal processes are in their home-states).

2. Absence of Deadlock ([CHAN79])

Absence of deadlock means that during system operation, the system will never reach a state where no further state transition is possible, that is, all internal processes have been suspended and no external inputs can cause any of the system processes to resume execution. This condition is also known as a global deadlock because in this state, all the system processes are locked up.

3. Absence of Livelock ([OWIC82])

Absence of livelock means that during system operation, the system will never cycle through a set of states indefinitely without making any real progress. In the case of a communication protocol, a livelock arises if two stations keep transmitting the same piece of data to each other over and over again. To prove that a PPS is free from livelock, a function of the system state which measures progress must be shown to be increasing monotonically.

4. Absence of Starvation and Degradation ([FLON81], [RICAB1])

A condition of starvation exists in a system if a system process is permanently deprived of a resource that it needs. A condition of degradation exists if a process is suspended forever. In either case, there is one fewer active process in the system. Since only one process is locked up, this condition is also known as a local deadlock.

3.3 Verification and the Hierarchy of Specifications

During the design process, a hierarchy of specifications is generated where each specification is an implementation of the previous design. To ensure that the final design is correct, every specification at each stage must be independently verified, that is, each specification must be shown to be correct according to the criteria outlined in the previous section.

In [BIRM76], partial correctness is defined to be an equivalence between the specification and the implementation. Similarly, the partial correctness of the specification S_i can be established by showing that S_i implements S_{i-1} .

In section 3.1, three possible specifications were identified in protocol design - the protocol intent, the protocol specification and the protocol implementation. In the literature, protocol verification typically refers to the verification of the protocol specification instead of the protocol implementation. When one verifies X.21, one is in fact trying to verify that the protocol specification is correct with respect to the protocol intent, namely, that data can be transferred reliably and that the protocol will not lead to a system deadlock. Once the protocol specification has been verified, it can then be implemented and the protocol implementation itself must also be verified to ensure that it satisfies the protocol specification.

3.4 Specification Language

The way a specification is expressed and the language in which it is expressed affect very much its usefulness. Just as the syntax and the semantics of a natural language imposes limits on what can be expressed in that language, a

specification language imposes similar limits on a specification. Thus a good specification language must be powerful enough to allow a designer to succinctly express his ideas. Moreover, the designer must also be trained to use the specification language in an effective and efficient manner.

In [BOCH80], [SUNS79] and [CCIT76], the characteristics of a good specification language are discussed and they can be summarized as follows. It should be noted that a specification is also a model of the final implementation. Hence a specification language is also referred to as a modelling language in the literature.

1. Completeness

A good specification language must provide the designer with the necessary tools to describe all pertinent aspects of a design. The description must be complete and precise so that no ambiguities or inconsistencies exist in the specification.

2. Abstraction

A good specification language must be sufficiently abstract so that the reader will not be burdened with details that are not relevant at that point of the design process. In the hierarchy of specifications, each specification is an abstraction of its succeeding design. Furthermore, a specification language must be abstract enough so that a family of solutions can be generated. The more abstract a specification is, the fewer implementation restrictions it imposes on a solution and the larger the family of acceptable solutions will be from which the most suitable one can then be chosen.

3. Ease of Understanding

A good specification language must be easy to learn and easy to understand. One way to achieve this is to make the language modular so that a large model can be broken down models of the major components.

4. Amenity to Formal Analysis

Since every specification has to be ultimately verified to ensure correctness, the ideal specification language should lend itself to verification. A similar statement is also true in programming. If a programmer intends to prove the correctness of his program, he will be tempted to write it in a structured language such as Pascal rather than say, FORTRAN.

3.5 Commonly Used Specification and Verification Techniques

Although computer scientists have always been interested in program proving, the interest in proving parallel programs has multiplied in recent years because of the widespread use of distributed processing. Numerous models and extensions to basic models have been proposed each equipped with unique features that enable them to handle different situations. Surveys of some of these techniques used in protocol design can be found in [SUNS79], [DAY79], [SUNS78] and [BOCH80].

One of the most commonly used techniques for specification is still the narrative form written in a natural language. If carefully written, it has the advantages of being direct and easy to understand. However, with the complexity of modern day parallel processing systems, the narrative form quickly becomes clumsy and convoluted because a natural language lacks the control and data structures necessary to express complex algebraic and logical relationships. Thus in recent years, the interest has been mostly on formal

methods of specification. Nevertheless, the narrative form will still remain as a useful auxiliary specification technique that can be used to supplement formal models.

In [BOCH80], formal models have been classified into three main categories, namely, the state transition models, the programming language models and the hybrid models.

State transition models are based on the observation that in a PPS, each process goes through a sequence of state transitions during its operation and by defining the logic of the state transitions, the behaviour of the process can be specified. Transition models are usually expressed in terms of graphs and hence are also referred to as graphical models. Examples of such methods in protocol design include the finite state machine model [BOCH78], Petri-Nets [MERL79], grammars [HARA73], duologues [ZAFI80] and UCLA graphs [POST76]. Verification of state transition model is usually achieved through a reachability analysis. In [YAU83] and [NELS83], a state transition model (Petri Net) is used on general parallel programs although such use of state transition models are less common.

Programming language models describe the operation of a PPS through the use of a (usually high-level) programming language. A programming language is inherently suited for the specification of variable assignments, conditionals, repetitive actions, numeric and logical relationships. The programming language has an added advantage in that it is very close to the final implementation. This means that the specification can be translated into an implementation very easily. Verification of a programming language model is achieved through logical deduction based on the semantics of the model. By analysing the structure of the program, one can deduce for example, that a certain relationship between some program variables exists whenever

a certain point of the program is reached. CSP [HOAR78] is an example of a programming language model designed for concurrent programming. An example of the use of a programming language to specify a communication protocol can be found in [STEN76].

Hybrid models ([BOCH77] and [KELL76]) attempt to combine the two approaches and thereby maximizing its usefulness and minimizing its limitations. A hybrid model tries to separate a PPS into two aspects, the control aspect which defines the state transitions of the system and the data manipulation aspect which describes the effects of state transitions on state variables. The former is described by a transition model and the latter by a programming language model. The advantage of this approach is that the control aspect can be captured in a concise and easy-to-understand model. It will be seen in chapter 5 that this method can also reduce the number of states necessary to describe the system. Verification of a hybrid model involves a reachability analysis on the control structure of the model to reveal any deadlocks, livelocks or degradations and logical deduction to establish relationships between variables.

In the next four chapters, the three approaches will be described in greater detail but in a slightly different order. Chapter 4 will describe two transition models, namely the finite state machine (FSM) model and the Petri Net model. Chapter 5 will describe the enhanced FSM model and Keller's model. Chapter 6 will describe CSP as a programming language model and chapter 7 will describe verification techniques for CSP.

CHAPTER 4 STATE TRANSITION MODELS

A transition model is one which specifies a PPS in terms of its state transitions. Two such models will be described in this chapter, the finite state machine model and the Petri Net model. Verification techniques for these two models will also be discussed. The strengths and weaknesses of the state transition models will then be discussed.

4.1 Finite-State Machine (FSM) Model

In a PPS, a system process progresses through a number of process states during its operation. If the number of process states is finite, then the process specification can be expressed in terms of a FSM. In a similar way, if the system progresses through a finite number of system states, the system specification can also be expressed in terms of a FSM. FSM models are commonly used in the modelling of computer protocols ([BOCH78], [DANT78]). In protocol design, the FSM that represents a communication station (or party) is the local protocol machine whereas the FSM which represents the state transition of the whole system (which comprises all the parties) is the global protocol machine [DANT78].

A FSM is a 5-tuple represented by

$$\langle X, I, O, N, M \rangle$$

where

X is a finite set of states;

I is a finite set of inputs;

O is a finite set of outputs;

N is a state transition function of $I \times X \rightarrow X$;

M is the output function of $I \times X \rightarrow O$.

Based on the current input and the current state, N gives the next state of the machine and M gives the output.

Figure 4.A shows an example of a FSM. The machine can be in one of three states, x_1 , x_2 and x_3 . There are three possible inputs, i_1 , i_2 and i_3 and two possible outputs, o_1 and o_2 . The state transition graph is a graphical representation of N and M. Since both the next state and the output are functions of the present state and the present input, a FSM is by definition a deterministic machine. In chapter 2, it was mentioned that some system processes may be autonomous, that is, they may go through a state transition without an input. Using a FSM model, an autonomous process is still modelled as a deterministic machine but dummy inputs are supplied to trigger the desired transitions.

Figure 4.B and 4.C illustrate the use of a FSM model to specify the operation of a simple protocol. In this example, the two communication stations in figure 4.B are the internal processes of the PPS. The purpose of the PPS is to provide a communication link between the two external processes (users). The communication stations are symmetrical and the operation of each station is modelled as a FSM as in figure 4.C. Each station can be in one of three possible states, Id (idle state), Wa (wait for acknowledgement) and Co (consume data). Each receives inputs from two sources, the communication station at the other end and the end user at this end and generates outputs to the same two processes. In the idle state for example, the communication station can accept two inputs, DATA from the other communication station or SEND from the user. If it receives DATA from the other station, the data will be passed on to the user and the station goes into the Co state. If on the other hand, it receives SEND from the user, it will generate data to the

$$X = \{x_1, x_2, x_3\}$$

$$I = \{i_1, i_2, i_3\}$$

$$O = \{o_1, o_2\}$$

$$N = \begin{bmatrix} x_1 & x_3 & x_2 \\ x_2 & x_1 & x_1 \\ x_3 & x_1 & x_3 \end{bmatrix}$$

$$M = \begin{bmatrix} o_2 & o_1 & o_2 \\ o_1 & o_1 & o_2 \\ o_1 & o_2 & o_1 \end{bmatrix}$$

Figure 4.A An Example of a FSM

Figure 4.B A Simple Computer Protocol with Two Users and Two Communication Stations

Figure 4.C FSM Model of the Communication Protocol

other station and then go to the Wa state.

The graphical representation in figure 4.C has been simplified by deleting all unproductive inputs. An unproductive input is one which does not cause any state transition or output. For example, if the station is in the idle state and it receives an acknowledgement from the other station, the unexpected input will simply be ignored.

4.2 Verification of a FSM Model

Verification of a FSM model is commonly achieved by a reachability analysis of the global state transition model. The global state transition model is obtained by combining all the local transition models. Figure 4.D shows the global state transition diagram for the simple protocol described in the previous section. The system state is an ordered pair of the local states of the two communication stations. Because the two stations have essentially the same inputs and the same outputs, a state transition must be qualified with a station name to identify the station which is undergoing the transition.

From the global state transition diagram, verification is carried out as follows.

1. Deadlock

A deadlock is present in the system if there exists a state reachable from the initial state for which no state transition is possible. Figure 4.E shows the global state diagram of a system. Starting from the initial state 1, one can end up in state 3 (that is, state 3 is reachable from state 1) for which no further state transition is possible. Hence state 3 is a deadlock.

Figure 4.D Global State Transition Diagram for the Communication Protocol

Figure 4.E An Example of a State Transition Diagram
With a Deadlock and a Livelock

2. Livelock

A livelock is present in the system if there exists a state reachable from the initial state for which a transition is possible but from which the terminal state (or home state) cannot be reached. In figure 4.E, state 7 is reachable from state 1. However the terminal state, namely state 11, is not reachable from state 7. In other words, no sequence of state transitions can cause the system to move from state 7 to state 11. Hence state 7 is an example of a livelock.

3. Degradation

A degradation exists in the system if there exists a state reachable from the initial state for which the process state of at least one process stops progressing in the subsequent states. In other words, if s_{ij} is the process state of process j when the system is in state S_i and s_{ij} is not a terminal state, then a degradation exists if $s_{ij} = s_{kj}$ for all S_k reachable from S_i .

Partial correctness cannot be verified directly from the global state transition diagram. It can be done indirectly if one can associate partial correctness with a certain state or a set of states and can show that the state(s) is always traversed before the system terminates.

Returning to the example of the simple communication protocol in figure 4.C, I_d is the home-state for a communication station. After either side has sent data, the system should eventually return to the system home state where both communication stations are idle. In this system, there is one deadlock state. If both end users decide to send data to each other at the same time, then both communication stations will end up in the W_a state from which no further state tran-

sition is possible. This scenario is confirmed by the global state transition diagram in figure 4.D. (Id, Id) is the home state for the system. The state (Wa, Wa) is reachable from the initial state (Id, Id) but no transition is possible from (Wa, Wa). Hence it is a deadlock.

4.3 Petri Net Model

A Petri Net is another transition model that can be used to model asynchronous processes. While a FSM model emphasizes states and state transitions, a Petri Net model emphasizes events and conditions. In a Petri Net, a state transition is possible whenever a set of conditions are true. For a more detailed discussion of Petri Nets, the reader is referred to [PETE77].

A Petri Net is a directed graph consisting of directed arcs and nodes. There are two types of nodes - place nodes which are represented by circles and transition nodes which are represented by bars. A place node represents a condition whereas a transition node represents an action. A transition node is enabled when all the input places (places with arcs going into the transition) are filled with tokens. A transition can fire only if it is enabled. When it fires, a token is taken from each of its input places and a token is put into each of its output places (places with arcs emerging from the transition).

Figure 4.F shows an example of a Petri Net. Since transition A only has one input place, namely, place 1 which is filled, transition A is enabled. Similarly, transition B is enabled. Figure 4.G shows the same Petri Net after both transitions A and B have fired. Now only transition C is enabled.

In more formal terms, a Petri Net is a 5-tuple represented by

$P = \{1, 2, 3, 4, 5, 6\}$

$T = \{A, B, C, D\}$

Pre =

	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	0	1
4	0	0	1	0
5	0	0	1	0
6	0	0	0	1

Post =

	A	B	C	D
1	0	0	0	1
2	0	0	1	0
3	1	0	0	0
4	1	0	0	0
5	0	1	0	0
6	0	0	1	0

$M_0 = (1, 1, 0, 0, 0, 0)$

Figure 4.F An Example of a Petri Net

$\langle P, T, Pre, Post, M_0 \rangle$

where

P is a set of places;

T is a set of transition;

Pre is a forward incidence function of $P \times T \rightarrow \{0,1\}$

$Post$ is a backward incidence function of $P \times T \rightarrow \{0,1\}$;

M_0 is the initial marking.

The forward incidence function works as follows. If there exists an arc from place p to transition t , then

$$Pre(p,t) = 1.$$

Otherwise, $Pre(p,t) = 0$. Similarly, $Post(p,t) = 1$ if there exists an arc from transition t to place p .

Otherwise, $Post(p,t) = 0$.

At any given moment, the state of the Petri Net can be represented by a mapping function M where $M(p) = n$ if there are n tokens in place p . The initial marking is denoted by the mapping function M_0 .

A transition t is enabled if $M(p) \geq Pre(p,t)$ for all p . When the transition fires, a new marking M' is produced such that

$$M'(p) = M(p) - Pre(p,t) + Post(p,t)$$

Thus starting from the initial marking M_0 , a sequence of markings $M_1, M_2, M_3, \dots$ can be constructed such that M_{i+1} is obtained by the firing of a transition from the previous state of M_i .

Petri Nets have been used extensively to model protocols ([BERT82], [MERL79] and [MERL76]). More recently, Petri

Figure 4.G The Same Petri Net After
Transitions A and B Have Fired

Figure 4.H Petri Net Model of the Communication Station
in Figure 4.B

Figure 4.I Global State Transition Diagram for the Petri Net in Figure 4.H

Nets are also being used to model parallel programs ([YAU83] and [NELS83]).

Figure 4.H shows the Petri Net model of the communication station described in section 4.1. For brevity, the inputs from the user and outputs to the user process have been omitted. The place nodes and transition nodes of the other communication station have been identified with primes. Each place node represents a condition. For example, when node AR is filled with a token, it means that an acknowledgement has been received. The node Wa means that the station is waiting for an acknowledgement. If both nodes are filled, that is, if the station is waiting for an acknowledgement and one has arrived, then transition t4 is enabled. When it fires, the station returns to the idle condition.

4.4 Verification of Petri Net Models

A Petri Net model is verified by reducing it to a global state transition diagram and then analysing it the same way as a FSM model ([PETE77], [MERL79] and [MERL76]). As mentioned earlier, the state of a Petri Net can be represented by a marking M. If we use a number of Petri Nets to model different processes in a PPS, the marking can be generalised to include all places in all Petri Nets in the system. Starting from an initial state M0, we can construct a sequence of markings which represent the sequence of system states as a result of state transitions.

Figure 4.I shows the state transitions derived from the Petri Net of figure 4.H. A system state is denoted by a list of non-empty place nodes. A place p with n tokens is denoted as np. Thus the state labelled (Id, AR', Wa') has one token in each of the listed places and none in the

rest. The global state transition diagram is very similar to the one in figure 4.0 which was generated from a FSM model. The same deadlock is located, namely when both stations have sent data to each other and both are waiting for an acknowledgement.

More recently, other techniques have been developed to analyse properties of Petri Nets. In [BERT82], some of these newer techniques are demonstrated. For example, the reduction method, originally used for UCLA graphs, has been adapted for Petri Nets. It simplifies a Petri Net while at the same time, preserving important properties such as boundedness, safeness, liveness, home-states and linear invariants. Using a reduced net, place-invariants can be asserted from which properties such as freedom from deadlock, liveness and safeness can be deduced.

4.5 Strengths and Weaknesses of Transition Models

Transition models are relatively simple and easy to understand provided that the size of the model is not too large. They are also highly abstract with few or no implementation details. They can be represented very easily in a computer thus making automatic verification possible. There are also major limitations to the usefulness of transition models and they are as follows.

1. One of the major limitations of a transition model is the 'state explosion' problem. In a complex system, the number of states involved can get quite large. The result is a graph which is confusing and hard to understand. The large number of states also makes the analysis of the model very difficult. Since the analysis of a transition model basically boils down to the computation of all possible system states and state

transitions, the problem can very quickly become unmanageable with a large number of states. In a system with m processes which can each be in one of n states, the number of system states that have to be computed will be in the order of n to the power m . Thus analysis of a complex system with a large number of processes may prove too large even with automated methods. So far, application of transition models has been most successful in systems with few processes, namely, computer protocols ([ZAF180]).

2. Another weakness of a transition model is that it does not provide the necessary tools to model adequately program variables which are, in a lot of systems, an essential element in the operation of the system. All program variables that are significant in the operation of the system have to be modelled as separate states with one for each possible value of the variables. For example, in a protocol which uses sequence numbers that go from say 0 to 9, there will be 10 'send' states with one for each possible sequence number. Likewise, the number of 'receive' states will also be 10. Apart from aggravating the state explosion problem discussed above, this representation also makes the verification of partial correctness very difficult. To prove for example, that the window between the 'send' sequence number and the 'receive' sequence number is less than say, 5, one must first compute all the states for which the condition is true and then prove that one of those states is traversed before the system reaches the home state.
3. The modelling of messages is also another weakness of the transition models. In a transition model, a message is only identified by its type with no provisions for its contents. For example, we may have a message called

'data' and another called 'acknowledgement' but there is no means of addressing the contents of these messages.

Thus it will be hard to prove that the contents of a message is preserved after it has passed through a number of process.

CHAPTER 5 HYBRID MODELS

In chapter 4, transition models were introduced and the major limitations were the state explosion problem and the inability to represent program variables. In this chapter, hybrid models will be introduced which enhance transition models with program variables. This not only will reduce the number of states but will also allow numerical and logical relationships between program variables to be modelled. In the first section, a hybrid model will be introduced which combines a FSM model with the use of program variables. Then Keller's model will be described. The example of the bounded buffer will be presented and modelled using both methods. Finally the strengths and weaknesses of hybrid models will be discussed.

5.1 Enhanced FSM Model

As mentioned in the previous chapter, one of the problems associated with a FSM model is its inability to represent program variables. If each possible value of a program variable must be represented as a separate state, then the total number of states in the system will very quickly become unmanageable, making analysis very difficult. Furthermore, using states alone to express relationships between variables is often clumsy and inconvenient. The enhanced FSM model is an attempt to correct these deficiencies by combining the use of a FSM with some programming language constructs.

Many hybrid models based on the FSM model have been proposed to date ([SCHU80], [BOCH77] and [BIRK72]). The enhanced FSM model that will be described here is a slightly modified form of the one proposed by Bochmann in [BOCH77].

In the enhanced FSM model, associated with each local model of a PPS are local program variables. These variables are local in the sense that they can be accessed (written to and read from) only by the local process and not by any other processes in the system. The basic state of a process is the state of its FSM model whereas the total state is determined by its basic state together with the values of its local variables. In a FSM model, a state transition is caused by an input from another process. In the enhanced FSM model, a transition is caused by an input and a condition being true. The condition which must be true before a transition is possible is called an enabling predicate. It is simply an expression involving the local variables which based on their current values, evaluates to either true or false. In the FSM model, a transition causes some output to be generated. In the enhanced FSM model, a transition causes both output to be generated and a transition action to be executed. A transition action is a sequence of actions which modify the local variables.

Typically, a hybrid model is verified in two steps [BOQH77]. First a reachability analysis is performed on the basic FSM to reveal any system deadlocks or livelocks. Then assertions are attached to the various states of the state transition diagram which are verified by inference based on the transition actions.

5.2 Keller's Model

Keller first proposed his model in [KELL76] in the context of parallel programs. In his model, a parallel program has a state which consists of a control state and a data state. In simple terms, the control state tells us where the current program counters (one for each process) are and the data state tells us what values the program variables have at a

given moment.

The transitions of the control states are represented in a graphical form very similar to a Petri Net. The marking of the graph (defined in the same way as a Petri Net) determines the control state of the program. The marking is represented by a mapping function M such that $M(p)$ is the number of tokens in node p . The data state of the program is represented by a vector of program variables x where each component of x is a variable which is assigned a value from a specific domain.

Associated with each transition node is an enabling predicate $P(x)$ and an action function $F(x)$ on the program variable vector x . A transition t is enabled if each of its input places is filled and if $P(x)$ is true. When the transition fires, a token is taken from each of the input places and a token is placed in each of the output places. In addition, the program variables are also modified by $F(x)$ such that the new data state $x' = F(x)$.

The significance of Keller's work lies in his attempt to devise a formal proof system which covers all the verification conditions, namely, partial correctness, termination, deadlock etc. This section will summarize some of concepts in his proof system in [KELL76].

Based on Keller's model, the execution of a parallel program can be described as follows. The program starts from an initial state q_0 (which consists of both the control state and the data state). From q_0 , the next state transition takes the program to q_1 and the next one takes it to q_2 and so on. Thus the execution of the program can be represented by a sequence $q_0, q_1, q_2, \dots$. Depending on the nature of the program, there may or may not be a terminal state.

Keller proposed several principles that can be used to verify a parallel program. Most of these principles are generalisations of the corresponding principles for serial programs.

1. Invariance

Let Q be the set of all possible states of a parallel program. A predicate J on Q is said to be q_0 -invariant if for each q of Q that is reachable from q_0 , $J(q)$ is true. In other words, to say that J is q_0 -invariant means that if the program starts with state q_0 , then during the execution sequence, $J(q)$ will be true after every state transition.

2. Partial Correctness

If a program is supposed to terminate with state q_T , then partial correctness stipulates that $J(q_T)$ must hold where J is the verification condition. Likewise, if instead of a terminal state, the program has a home state, then $J(q_H)$ must be true.

3. Deadlocks and Livelocks

To prove that a program does not end up in either a deadlock or a livelock, Keller proposes to use the invariance of reachability. For a program with a terminal state q_T , we must prove that after every transition, q_T must be reachable from the current state. In other words, the condition " q_T is reachable from the current state" must be invariant. If a program has a home state q_H , then the condition of " q_H is reachable" must be invariant. If the invariance cannot be asserted, then the system has a deadlock or a livelock.

5.3 A Bounded Buffer: An Example

This example is taken from [HOAR78] in which the buffer is modelled using CSP. In this section, the same buffer will be modelled using both the enhanced FSM model and Keller's model. In this example, consider a buffering process which has a buffer BF which holds up to 10 data items. There are two pointers IN which points to the first empty slot in the buffer and OUT which points to the first data item to be consumed. If the buffer is not yet full, the buffering process waits for the producer to generate more data. If the buffer is not empty, the process will also wait for the consumer to accept new data. When the consumer wants more data, it sends a message to the buffering process which returns the next available data item.

Figure 5.A shows the enhanced FSM model for the buffer. There are three states, NORM (buffer is neither full nor empty), FULL (the buffer is full) and EMPTY (the buffer is empty). The possible state transitions are tabulated as follows.

Initial Conditions :
in := 0; out := 0

Transition	Enabling Predicate	Input	Output	Transition Action
t1	$in < out + 9$	data	--	$bf[in \bmod 10] := data$ $in := in + 1$
t2	$in < out + 1$	more	$bf[out \bmod 10]$	$out := out + 1$
t3	$in = out + 9$	data	--	$bf[in \bmod 10] := data$ $in := in + 1$
t4	true	more	$bf[out \bmod 10]$	$out := out + 1$
t5	$in = out + 1$	more	$bf[out \bmod 10]$	$out := out + 1$
t6	true	data	--	$bf[in \bmod 10] := data$ $in := in + 1$

Figure 5A Enhanced FSM Model for a Bounded Buffer

Figure 5.B Keller's Model for the Bounded Buffer

The model can be analysed as follows. First of all, based on the state transitions, one can conclude that the buffering process is deadlock free because from every state, there is a state transition that can carry the process back to the normal state. If the buffer is full, the enabling predicate is true. Therefore, the next message from the consumer will cause the buffer to go back to the normal state. Similarly, if the buffer is empty, the enabling predicate is always true and the next message from the producer will cause it to go back to the normal state. Having proved that the process is deadlock free, we can proceed to prove that the system is partially correct. Since the buffer can only hold 10 data items, it must be proved that the following assertion is true at all times.

$$0 \leq \text{out} \leq \text{in} \leq \text{out} + 10$$

The assertion holds initially with $\text{IN} = 0$ and $\text{OUT} = 0$. It can further be shown that every transition will maintain the validity of the assertion. For example, the action function of t_1 increments IN by 1. However, the transition will take place only if the enabling predicate is true, that is, $\text{IN} < \text{OUT} + 9$.

$$\begin{aligned} &\text{in} < \text{out} + 9 \text{ and } \text{in} := \text{in} + 1 \\ &\Rightarrow \text{in} < \text{out} + 10 \\ &\Rightarrow \text{in} \leq \text{out} + 10 \end{aligned}$$

Therefore, the assertion holds after t_1 has taken place. By repeating the procedure for the other transitions, it can be shown that the assertion always holds.

Figure 5.8 shows the model for the same buffering process using Keller's method. The enabling predicates and the action functions for T_1 and T_2 are as follows.

Figure 5.C State Transition Diagram for Keller's Model

Initial Conditions :
in := 0; out := 0

Transition	Enabling Predicate	Action Function F(I)
T1	in < out+10	b[(in mod 10)] := data in := in + 1
T2	in < out	data := b[(out mod 10)] out := out + 1

Verification of this model is very similar to the previous case using the enhanced FSM model. For partial correctness, it must be shown that the assertion

$$0 \leq \text{out} \leq \text{in} \leq \text{out} + 10$$

is invariant, that is, the assertion is true after every transition. To prove termination, the state transition diagram has been computed and shown in figure 5.C. Again the process is shown to be deadlock free. Note that in (p1), the process will progress to (p1, p2) when the producer has sent another data item. However, if the buffer is full, that is, (IN = OUT + 10), transition T1 cannot fire. The only transition possible is to (p1, p2, p3) as a result of the consumer wanting more data. In that case, transition T2 fires which increments OUT and since the enabling predicate is now true, T1 can fire.

5.4 Strengths and Weaknesses of Hybrid Models

With the inclusion of program variables, hybrid models have alleviated dramatically the state explosion problem and have allowed numerical and logical relationships between program variables to be expressed.

However, even with the addition of program variables, hybrid models still have significant semantic inadequacies which will result in things that the designer would like to implement but which cannot be adequately specified in the model. In particular, the modelling of process communication is still inadequate. Even in Keller's model, the contents of the message cannot be adequately represented. While a message may be distinguishable by its type, that is, an acknowledgement or a request, any data that is present in the message cannot be modelled. In [BETH82], an extension is proposed that attaches attributes to a token in a Petri Net. These attributes can be assigned any value and hence may be used to represent the contents of a message. However, this extension means that verification using traditional methods will become more difficult.

Although Keller's proof system is conceptually appealing, it is quite difficult to apply in practice, especially as it relates to deadlocks and livelocks. In order to prove that deadlocks are absent, we must be able to prove that reachability is invariant which eventually still boils down to computing all possible state transitions for the whole system.

In [BETH82], it was pointed out that Keller's model suffers two disadvantages.

"The first one is that it mixes two semantics, the semantics of Petri nets and the semantic of guarded commands. It follows that the interrelations between these two semantics must be extremely carefully defined if we want to avoid contradictions or underspecification and also to derive rigorous proofs of correctness. The second disadvantage is that there is no clear frontier between those aspects of protocols which must be

represented by Petri nets and the others."

In fact, all hybrid models suffer the same drawbacks to some extent. In [BOCH77] for example, it was shown that the alternating bit protocol can be represented using one, three or six places. What it means is that states and variables can be used interchangeably in a hybrid model. Being in a certain state implies some relationships between variables and vice versa. In the bounded buffer example, being in the state FULL implies implicitly that

$$IN = OUT + 10$$

and being EMPTY implies that $IN = OUT$. The end result is that the separation between the control state and the data state is not as clear cut as one would like. Hence whatever proof system is used for one semantic, it must also take into consideration the existence of the other semantic. This "fuzziness" can cause inconsistencies and contradictions in a model that can sometimes be very hard to detect.

CHAPTER 6 A PROGRAMMING LANGUAGE MODEL - CSP

So far, both the transition models and the hybrid models have proved to be inadequate in the description of important aspects of a general PPS because of significant semantic differences between the specification language and the ultimate implementation. In this chapter, a programming language model CSP will be introduced as a specification language. CSP is a concurrent language first proposed by Hoare in [HOAR78]. The basic concepts of CSP will be briefly described followed by some extensions proposed by the author.

6.1 Advantages and Disadvantages of a Programming Language Model

A programming language model has the following advantages. Variables and parameters which may take on a large (even infinite) number of values can be handled in a very natural way. Furthermore, there is a strong correspondence between the specification language and the implementation language. This close correspondence means that the specification can be implemented much more easily and that the subsequent implementation can be easily verified. Referring to the hierarchy of specifications, if S_{i+1} , an implementation of S_i are both written in the same (or similar) language, it will be relatively easy to prove that S_{i+1} does what S_i does.

One of the disadvantages of a programming language model is that too many implementation details may be included which may obscure the important operational principles of a design. Another disadvantage is that the specification may not be abstract enough and may have unnecessary restrictions on the final implementation. Having specified a design in

a programming language, we may have unintentionally eliminated certain implementations which differ syntactically from the specification. For example, having specified a design in FORTRAN, one would be much more inclined to implement the design in FORTRAN rather than say LISP because of significant differences between the two languages. Therefore, a programming language model must be "lightly loaded", that is, it must be simple and easy to understand containing only basic primitives and little or no implementation details. The language must also be syntactically and semantically close to a family of commonly used implementation languages.

6.2 Communicating Sequential Processes (CSP)

In [HOAR78], Hoare proposed a language for communicating sequential processes. This language has since been referred to as CSP in the literature. CSP is attractive as a specification language because it was designed to contain only what Hoare considered to be the basic primitives for parallel processing systems. Moreover, CSP is syntactically close to a number of modern high level concurrent programming languages. In [WEGN83] and [CASH80] for example, it was shown how some of the language constructs in Ada could be translated into CSP and vice versa. Since the paper first appeared, various verification techniques for CSP have been proposed ([MISR81] and [APT80]) and some extensions to CSP have also been proposed in [SIL883].

In the next section, the basic programming constructs of CSP will be briefly described. The notations in this thesis are slightly different from Hoare's. Instead of using the original short notations such as [...] and *[], the longer but more readable notations - if...fi and do...od - will be used instead.

6.2.1 The Parallel Command

A parallel command has the following form.

```
P1 // P2 // P3 // P4 ... // Pn
```

where P1 to Pn are processes. When this command is executed, all the processes will be started at the same time and this command ends only when all the processes have terminated.

For example,

```
program fred;  
  
    process tom;  
        var x : integer;  
        begin  
            ... {body of tom}  
        end;  
  
    process dick;  
        var x : integer;  
        begin  
            ... {body of dick}  
        end;  
  
    begin {main program body}  
        tom // dick;  
    end.
```

In the sample program FRED, there are two constituent processes TOM and DICK which are activated simultaneously when FRED is executed. The program FRED terminates only when both TOM and DICK have terminated.

As mentioned in chapter 2, variables are local to the processes and processes only communicate via messages. X is declared in both TOM and DICK and each declaration represents a distinct variable.

6.2.2 Inter-Process Communications

Inter-process communications are achieved via input and output commands.

For example,

```
process dick;  
  var y : integer;  
  begin  
    tom ? y           {input command}  
  end;
```

When the input command is executed, process DICK will be suspended until a message containing an integer has been received from process TOM. When the message is received, the variable Y will have the value of the integer in the message.

```
process tom;  
  var x : integer;  
  begin  
    x := 10;  
    dick ! x         {output command}  
  end;
```

When the output command is executed, a message which contains the value of the variable X will be sent to process DICK. Process TOM will be suspended until process

DICK is ready to accept the message, that is, when DICK executes the input command at which time a message transfer occurs and the variable Y in DICK will be assigned the value of the variable X in TOM.

Apart from transmitting a simple variable such as an integer, a constructor can be used to transmit structured variable. A constructor may have several components which are themselves simple variables. For example, WITHDRAWAL may be a constructor which consists of two components, an account number and the amount of money to be withdrawn. Thus the following two I/O commands will result in communication.

```
bank_teller ! withdrawal(123, 100)
```

```
customer ? withdrawal(account, amount)
```

A constructor with no components is a signal.

6.2.3 The Guarded Command

Guarded commands in CSP are taken directly from [DIJK76]. A guarded command consists of two parts, a guard and a command. A guard can be input commands or boolean expressions or both. The command can be selected for execution only if the associated guard is satisfied.

For example,

```
x = 5 -> y := 6
```

In this example, the first part ($x = 5$) is the guard and the second part ($y := 6$) is the command. The second part can be selected for execution only if the first part is

satisfied, that is, if X is equal to 5.

```
var full : boolean;  
    z : integer;  
...  
not full; dick ? z -> mary ! z
```

In this guarded command, the guard is satisfied if the boolean variable FULL is false and a message containing an integer has been received from process DICK.

6.2.4 The Alternative Command

An alternative command consists of a list of guarded commands. When the alternative command is executed, all the guards are checked. If none of the guards are satisfied, the alternative command fails and the process will abort. If one of the guards is satisfied, the associated command will be executed. If more than one guard is satisfied, one of the commands will be selected for execution and the outcome is non-determinate.

For example,

```
if x = 2 -> S1  
|   x = 3 -> S2  
|   x mod 3 = 0 -> S3  
fi
```

In this example, if X is equal to 2, S1 will be executed. If X is equal to 3, either S2 or S3 will be executed. If X is equal to 1, the alternative command will fail causing the process to abort.

6.2.5 The Repetitive Command

The repetitive command is very similar to the alternative command with the exception that the selection of a command to be executed is repetitive until all the guards fail.

For example,

```
do x = 2 -> x := 3  
  || x = 3 -> x := 6  
  || x mod 3 = 0 -> x := 4  
od
```

If X has an initial value of 2 when the repetitive command starts execution, then the following will happen. X will be assigned a value of 3. Two commands can now be selected for execution. If the second one is executed, X will be assigned a value of 6. Now since only the third guard is satisfied, X will be assigned a value of 4. Since all the guards now fail, the repetitive command is completed. If on the other hand, when X was 3, the third command was executed instead of the second one, then X would be assigned 4 and again the repetitive command would be completed.

6.3 Extensions to CSP

Hoare intended CSP to be the framework for a parallel programming language and hence he included only a very small set of language features. In order to make CSP a more complete specification language which is suitable for general application, the author found that it was necessary to add some extensions. In [Sil83], Silberschatz also proposed some extensions to CSP to make it easier to model resource sharing systems. Although these extensions will

not be used in this thesis, they are not incompatible with those proposed here. The author believes that while these extensions add more to the power of CSP, they do not go against Hoare's original intention of making CSP a highly abstract and simple language.

6.3.1 The Exit Statement

In a large program, there are usually cases where an error or an abnormal condition is detected at which point we wish to terminate the execution of a loop. Although it is possible to implement this with boolean flags, the exit statement provides a much simpler solution. The target of an exit statement is a label which is associated with either an alternative command or a repetitive statement. When the exit statement is executed, control is passed to the statement following the alternative or repetitive statement.

For example,

```
label session;
...
do user ? logon_request ->
    session :
    . do user ? command ->
        ...

    . if terminal ? power_down ->
        exit session
    .
...
fi

od;
clean_up;
```


od

In this example, `SESSION` is a label associated with the inner repetitive statement which keeps interpreting commands from the user. When the terminal goes down however, the exit statement passes control to the statement following the repetitive statement, that is, `CLEAN_UP`.

6.3.2 The `ELSE` Clause

In the original CSP, the alternative command will fail and the repetitive command will be skipped if all the guards fail. In guarded commands with complicated conditions, it is often desirable to have a `ELSE` clause which gets executed when all the guards fail.

For example,

```
if x = 2 -> S1
| x = 3 -> S2
| x mod 3 = 0 -> S3
else S4
fi
```

In this alternative command, `S4` will be executed if all the guards fail. This command can also be represented without using the `ELSE` clause albeit in a more verbose form as follows.

```
if x = 2 -> S1
| x = 3 -> S2
| x mod 3 = 0 -> S3
| (x <> 2) and
  (x <> 3) and
  (x mod 3 <> 0) -> S4
```

fi

Note that a repetitive command with an ELSE clause will never terminate.

```
do x = 2 -> S1  
else S2  
od
```

In this example, S1 will be executed if x = 2. Otherwise, S2 will be executed.

6.3.3 Constructors and Signals

In this thesis, constructors and signals will be declared globally. They are assumed to be known by all processes.

For example,

```
constructor  
  withdrawal (account : integer;  
              amount   : integer);
```

A structured message can be sent using a constructor and specifying the components. For example,

```
data_base ! withdrawal(1234, 100);
```

A structured message can be received using a constructor and specifying the variables to which the components are assigned. For example,

```
customer ? withdrawal(account, amount);
```

Unlike the original CSP, signals will be declared

separately for simplicity.

For example,

```
signal acknowledgement;  
  
producer ! acknowledgement;  
  
consumer ? acknowledgement;
```

is equivalent to

```
constructor acknowledgement ();  
  
producer ! acknowledgement ();  
  
consumer ? acknowledgement ();
```

6.4 Relationships between CSP and Keller's Model

The basic language constructs in CSP have equivalent forms in Keller's model. For example, the model in figure 6.A is equivalent to the following I/O statements.

```
process tom;  
  begin  
    dick ! 10;  
  end;  
  
process dick;  
  var i : integer;  
  begin  
    tom ? i;  
  end;
```

Similarly, the following alternative command is equivalent to the model in figure 6.8. Note that if all the guards fail, then the statement will cause the process to abort.

```
if x = 2 -> x := 3
| x = 3 -> dick ! 10; x := 9
| x mod 3 = 0 -> mary ! 9; x := 0
fi
```

The following repetitive command is equivalent to the model in figure 6.C. If all the guards fail, the statement will be skipped and the next statement will be executed.

```
do x = 2 -> x := 3
| x = 3 -> dick ! 10; x := 9
| x mod 3 = 0 -> mary ! 9; x := 0
od
```

It was mentioned in chapter 5 that one of the weaknesses of Keller's model was that the tokens were indistinguishable. In terms of CSP, what this means is that constructors and signals cannot be adequately represented by Keller's model.

Figure 6.A Keller's Model of I/O Commands

Figure 6.B Keller's Model of an Alternative Command

Figure 6.C Keller's Model of a Repetitive Command

CHAPTER 7 VERIFICATION OF A PROGRAMMING LANGUAGE MODEL

In the preceding chapter, CSP was introduced as a programming language model suitable for the specification of parallel processing systems. In this chapter, verification of such a model will be discussed. In the first section, program proving will be introduced and its relationship with system verification examined. In the second section, different verification techniques for CSP will be surveyed and briefly compared. In 7.3, the basic concepts of symbolic execution will be explained. In the subsequent sections, extensions to symbolic execution will be discussed that will enable it to be used to verify CSP models. In the last section, the overall suitability of symbolic execution as a verification technique for parallel processing systems will be evaluated.

7.1 Program Proving

Ever since the first computer programs were written, programmers have been concerned with program correctness. Although program testing can increase a programmer's confidence in the correctness of his program, it alone cannot guarantee absolute correctness (unless all possible input values have been tested).

Program proving refers to the procedure whereby the correctness of a program is deduced based on the structure and semantics of the program. Most of the program proving techniques that are in use today are derivatives of Floyd's original work [FLOY67]. Floyd's technique was later refined by Hoare in [HOAR71]. It is worthwhile at this point to briefly examine Hoare's proof system to get an intuitive understanding of program proving. In his system, there are a number of axioms that define the meaning of the language

constructs used in a particular program. Each axiom takes the following form.

$$\{P\} \ S \ \{Q\}$$

The axiom means that if an assertion P is true before the execution of a statement S , then Q must be true after the execution of S . For example, the semantics of the assignment statement in Pascal can be defined by the following axiom.

$$\{P(a)\} \ x := a \ \{P(x)\}$$

What this means is that if $P(a)$ is true before the assignment statement, then $P(x)$ must be true after the assignment statement has been executed where $P(x)$ is obtained by substituting ' x ' in the expression $P(a)$ wherever ' a ' occurs. Similar axioms can be defined for other language constructs such as the WHILE loop, IF THEN ELSE etc.

To tie the different statements together, there is an axiom for sequential composition as follows.

$$\begin{aligned} & \{P\} \ S1 \ \{Q\} \ \text{and} \ \{Q\} \ S2 \ \{R\} \\ \Rightarrow & \{P\} \ S1;S2 \ \{R\} \end{aligned}$$

What it means is that if we know the semantics of statements $S1$ and $S2$, we will then also know the semantics of executing $S1$ followed by $S2$.

Based on the set of axioms, a program can then be annotated by attaching assertions to the statements in the program. Thus a program may be annotated as follows.

$\{true\}$	$S1;$
$\{A1\}$	$S2;$
$\{A2\}$	$S3;$

.....
 {An-1} Sn;
 {An}

Initially before the first statement of a program has been executed, no assertions can be made about the variables of the program. After each statement has been executed, we can then make an assertion based on the assertion that was deduced prior to the statement and the axiom for the statement. Using this method, one will be able to assert 'An' at the end of statement 'Sn'. Using this proof system, program correctness can then be proved if the design intent of the program can be deduced from the program. Examples of program proving using this method can be found in [GRIE76], [HOAR71] and [HOAR69].

In a sequential program, statements are executed sequentially and hence assertions are developed sequentially as well. In a parallel program, there are parallel statements whose execution is interleaved in a non-deterministic order. Thus the proof system described earlier must be generalised and extended to handle parallelism. Examples of parallel program proving techniques can be found in [OWIC82], [FLON81], [LAMP80], [OWIC76a], [OWIC76b], [HOWA76], [ASHC75], [CLIN73] and [DIJK71]. Unfortunately, most of these techniques are dependent on the particular synchronizing primitives such as monitors and coroutines used in the programming languages. In other words, the proof system used for one language cannot be applied readily to another language.

7.2 Proof Systems Based on CSP

As mentioned in chapter 3, in a PPS, certain properties must be verified, namely absence of deadlock, absence of livelock, absence of starvation and degradation and finally partial

correctness. If we decide to use CSP as the modelling language for PPS, then we must adapt a verification technique for CSP to verify the above conditions. In recent years, different techniques have been proposed specifically for CSP.

The approach proposed by Apt, Frances and Rover in [APT80] can be considered a direct generalisation of the axiomatic approach used for sequential programs. In this approach, an isolated proof is obtained for each process which involves assertions on variables local to the process only. After the isolated proofs have been established, it must then be shown that these proofs cooperate, that is, they do not invalidate each other and that together they imply the correctness of the parallel program as a whole. To prove that the proofs cooperate, all interprocess communications must be examined to ensure that no process communication will invalidate the isolated proofs. This is typically the most difficult task in this proof system. First of all, interprocess communications can take place only if there are two I/O commands that match syntactically. Two I/O commands match syntactically if the two will lead to communication if they are executed in parallel. However, checking every syntactically matching pair is unnecessary because communication may not take place because the pair do not match semantically, that is, they will never be executed in parallel because of the logic of the program. To remedy this problem, one must also prove the existence of a global invariant which can be used to eliminate pairs of I/O commands that match syntactically but not semantically and thereby reducing the total number of pairs to check in order to establish cooperation among proofs. The main disadvantage of this approach is that it does not lend itself to automation. To an unaided human mind, this approach is tedious and requires a high degree of training.

In [CHAN81], Chandy and Misra proposed a proof system based

on traces. A trace is the record of the message flow from one process to another. Instead of putting the emphasis on the effects of a statement on program variables as in conventional program proofs, this method puts the emphasis on the effects of statements on traces. Using this technique, one must formulate the correctness of a program in terms of assertions on traces, that is, what properties the traces of a process (or a network of processes) must have before the program can be considered as correct. These assertions are then deduced from the program based on the effects of each program statement on the relevant traces. Unlike the previous method, the program proof is quite straightforward once the assertions to establish correctness have been identified. However, expressing correctness in terms of assertions on traces is the difficult step. If we wish to make assertions about variables that do not manifest themselves in traces, fictitious processes will have to be introduced in order to reveal these variables.

The third approach is a direct generalisation of the concept of symbolic execution in sequential programs as described in [HANT76]. The application of symbolic execution to parallel programs and to communication protocols has been proposed in [BRAN78]. However, the application of symbolic execution to CSP specifically has not been proposed to the knowledge of the author. In symbolic execution, program variables are assigned symbolic values which are then manipulated during symbolic execution by the program statements in ways identical to true run-time execution. After the whole program has been executed symbolically, the variables will then consist of symbolic expressions which can be used to deduce the correctness of the program. The advantage of this approach is that the manipulation of the symbolic values resembles very closely the run-time execution of the program and every step can thus be interpreted in an intuitive way. Furthermore, the manipulation of symbolic values is very

mechanical and can be automated to a large extent. In fact, such automated tools already exist as described in [BRAN78]. For the remainder of the chapter, the basic concepts as well as the generalisations required to handle parallelism will be described in greater detail.

7.3 Basic Concepts of Symbolic Execution

Symbolic execution has been successfully applied to sequential program ([DANN81], [HANT76] and [KING76]) and some automated tools for symbolic execution already exist ([BRAN78]). This section briefly explains the basic concepts behind symbolic execution in sequential programs.

In symbolic execution, statements in a sequential program are executed in the same order as they do during run-time. A program variable does not have a definite value until it has been assigned one. If a variable is assigned a value which is unknown until run-time (for example, an input variable of a procedure), then the variable will be assigned a symbolic value.

Sometimes, a program may generate different execution paths depending on the run-time values of the variables. In such a case, these different paths will be executed independently with a path condition attached to each path.

For example, the following procedure takes two input variables X and Y, finds the maximum of the two and assigns its value to the output variable Z.

```
procedure maximum
(x      : integer;
 y      : integer;
 var z  : integer);
```

```

begin
  if x > y -> z := x
  [] x <= y -> z := y
fi
end;

```

When we execute this procedure symbolically, the values for X and Y are unknown until run-time. Hence they are assigned two symbolic values \$X and \$Y. Execution of the procedure generates two execution paths. The first path has a path condition of $\$X > \Y , that is, the run-time value of X is greater than that of Y. Following this path, Z will be assigned the value of X, that is \$X. Similarly the other path has a path condition of $\$X \leq \Y and Z will be assigned \$Y. The result of the symbolic execution can be presented, as follows.

```

x = $X
y = $Y
1.
path condition :  $\$X > \$Y$ 
z := x
=> z = $X
2.
path condition :  $\$X \leq \$Y$ 
z := y
=> z = $Y

```

To prove an assertion, one simply has to show that the assertion holds with the symbolic values of the variables in each of the execution paths. In the above example, it is necessary to prove that Z has a value which is the greater of X and Y. In the first path, Z ends up with a symbolic value of \$X, which is the value of X. Since the path condition states that $\$X > \Y , Z is therefore the greater of X and Y, namely, X. Similarly, in the second

path, Z ends up with \$Y but since the path condition states that $\$X \leq \Y , Z therefore is also the greater of X and Y. Thus the assertion holds for both execution paths and the procedure is proved.

To be able to apply symbolic execution to CSP, some new rules must be developed in order to handle process communications, concurrency and non-determinism. These new rules for symbolic execution will be described in the following sections.

7.4 I/O Commands

In CSP, processes communicate via I/O commands. Symbolic execution of an I/O command can be applied as follows. If a process under execution executes an output command, then the process will be suspended from execution. If a process under execution executes an input command, then the process will be suspended if there is no pending message or if there is a message available, the target variable will be assigned the symbolic value of the contents of the message.

For example, assume that process A is under execution and it executes the following instruction

B ! x

where X has a symbolic value of \$X. Since A executes an output command, it will be suspended. Suppose that process B is now under execution and it executes the instruction

A ? y.

Since there is a message available, the target variable Y will be assigned the symbolic value of the contents of the

message, namely, \$X.

7.5 Parallelism

In CSP, there are a number of parallel processes and since we do not make any assumptions about the execution speeds of the processes, they may interleave in their executions in any arbitrary order thus giving rise to a large number of possibilities. One might then conclude that symbolic execution of a parallel program would generate a large number of execution paths. Fortunately, in CSP, the number of execution paths due to parallelism can be greatly reduced because processes do not share any variables and they communicate via messages only. In other words, the execution of each process is independent of one another until inter-process communications occur. This property allows us to "serialize" the executions of processes. In an extreme case where processes do not communicate with each other at all, processes can be executed sequentially in any order and the result would be identical.

To illustrate this idea, consider two parallel processes running concurrently. Assume that the first process executes 3 indivisible instructions A1, A2 and A3 and the second process executes another 3 instructions B1, B2 and B3. If the processes are allowed to share variables, one must consider the following execution sequences each of which might produce different results.

A1 A2 A3 B1 B2 B3
A1 A2 B1 A3 B2 B3
A1 A2 B1 B2 A3 B3
A1 A2 B1 B2 B3 A3
A1 B1 A2 B2 B3 A3

.....

B1 B2 B3 A1 A2 A3

In total, there are $6!/3!*3! = 20$ different paths to be considered. On the other hand, in CSP, different results will arise only when the processes actually communicate. For example, if A3 and B2 are I/O instructions, then the number of distinct execution paths will be reduced to 2 as follows.

A1 A2 A3 B1 B2 B3
B1 B2 B3 A1 A2 A3

The execution sequence of A1 A2 B1 B2 A3 B3 would generate exactly the same result as B1 B2 B3 A1 A2 A3 because in both sequences, B2 is executed before A3 and only the sequencing of those two I/O instructions could alter the execution result. In the context of CSP, some execution sequences may be meaningless because of the unbuffered nature of input and output. For example, if A3 is an output command and B2 is an input command, then B1 B2 B3 A1 A2 A3 becomes meaningless because B2 will not be completed until A3 has been executed. However, in this case, the number of distinct execution paths still remains 2, namely,

A1 A2 A3 B1 B2 B3
B1 B2 A1 A2 A3 B3.

Based on the above observation, we can then apply symbolic execution to CSP in the following way. Whenever N processes are ready to run, one must generate N distinct execution paths. For each execution path, one process is picked to run. That process is allowed to run until it executes an I/O command at which point a selection is made for another process (which may be the same process again) to run.

In section 7.3, it has been shown that an assertion can be proved by substituting the symbolic values into the variables of the assertion. Symbolic execution can also be used to prove other properties such as deadlocks, livelocks and race conditions. A deadlock exists in a system if during symbolic execution, there exists a node for which all processes are blocked and none of them can be selected for execution.

A livelock exists in a system if there exists a node which has the same state as one of its ancestor nodes. In the context of symbolic execution, a node N1 is the ancestor node of N2 if N2 lies on one of the execution paths of N1. For example, node 1.2.2.4 is an ancestor node of 1.2.2.4.5.1.2.3. Two nodes have the same state if all the variables of all the processes have the same values and the same set of processes are ready-to-run. What this means is if after executing a number of steps starting from node N1, one arrives at node N2 which has the same state as N1, then the system has a livelock because there will be an execution path from N2 which will arrive in N3 which has the same state as N2 ad infinitum.

A race condition exists in a system if different execution results are obtained by executing processes in a different order. For example, if at one point during symbolic execution, two processes A and B are both ready-to-run. Two paths are generated with A or B running first in each of the paths. If the two paths end up in two sets of nodes which have different states, then a race condition exists, that is, the behaviour of the system is dependent on the relative execution speeds of the processes.

Consider the following example,

process A;

```

var x : integer;
begin
  if B ? x ->
    x := x * 3;
    B ! x
  | C ? x ->
    x := x + 10;
    C ! x
  fi
end;

```

```

process B;
var y : integer;
begin
  A ! 5;
  A ? y
end;

```

```

process C;
var z : integer;
begin
  A ! 30;
  A ? z
end;

```

The symbolic execution of the two processes is as follows.

ready-to-run : B, C

1.

process : B

A ! 10

--- A unblocked, B blocked

1.1

process : A

B ? x

--- B unblocked

=> x = 10

x := x * 3

=> x = 30

B ! x

--- A blocked

=> B ! 30

1.1.1

process : B

A ? y

--- A terminated, B terminated

=> y = 30

process : C

A ! 30

--- C blocked

1.1.2

process : C

A ! 30

--- C blocked

process : B

A ? y

--- A terminated, B terminated

=> y = 30

1.2

process : C

A ! 30

--- C blocked

1.2.1

process : A

B ? x

--- B unblocked

=> x = 10

x := x * 3

=> x = 30

B ! x

--- A blocked

=> B ! 30

process : B

A ? y

--- A terminated, B terminated

=> y = 30

1.2.2

```

process : A
C ? x          --- C unblocked
=> x = 30
x := x + 10
=> x = 40
C ! x          --- A blocked
=> C ! 40
process : C
A ? z
=> z = 40      --- A terminated, C terminated

```

2.

```

process : C
A ! 30        --- A unblocked, C blocked

```

2.1

```

process : A
C ? x          --- C unblocked
=> x = 30
x := x + 10
=> x = 40
C ! x          --- A blocked
=> C ! 40

```

2.1.1

```

process : C
A ? z          --- A terminated, C terminated
=> z = 40
process : B
A ! 10         --- B blocked

```

2.1.2

```

process : B
A ! 10         --- B blocked
process : C
A ? z          --- A terminated, C terminated

```

=> z = 40

2.2

process : B

B ! 10 --- B blocked

2.2.1

process : A

B ? x --- B unblocked

=> x = 10

x := x * 3

=> x = 30

B ! x --- A blocked

=> B ! 30

process : B

A ? y

=> y = 30 --- A terminated, B terminated

2.2.2

process : A

C ? x ---- C unblocked

=> x = 30

x := x + 10

=> x = 40

C ! x --- A blocked

=> C ! 40

process : C

A ? z

=> z = 40 --- A terminated, C terminated

Figure 7.A shows the execution tree in graphical form. There are 8 leaf nodes, namely, 1.1.1, 1.1.2, 1.2.1, 1.2.2, 2.1.1, 2.1.2, 2.2.1 and 2.2.2. Each of the nodes represents a case where a process (either B or C) is left blocked while the other two processes have terminated successfully. That is, the program will always result in one process in a deadlock.

It can be seen that even with CSP, the number of distinct execution paths is still quite large even with the simple example. By considering all the execution paths, we can reveal all system anomalies due to race conditions. If we ignore the possibility of different execution results due to a different execution sequence, the verification procedure can be greatly simplified as follows.

```
process : B
A ! 10      --- A unblocked, B blocked
process : C
A ! 30      --- C blocked

1.
process : A
B ? x      --- B unblocked
=> x = 10
x := x * 3
=> x = 30
B ! x      --- A blocked
=> B ! 30
process : B
A ? y
=> y = 30   --- A terminated, B terminated

2.
process : A
C ? x      --- C unblocked
=> x = 30
x := x + 10
=> x = 40
C ! x      --- A blocked
=> C ! 40
process : C
A ? z
```

=> z = 40

--- A terminated, C terminated

In this execution tree, process B was arbitrary picked for execution. Then process C was selected and finally process A was picked. It turns out that the program is independent of the execution sequence. The simplified tree reveals the same two deadlocks as in the expanded tree. In this particular example, there are no race conditions, that is, the result is the same regardless of the execution order. The simplified verification tree is shown in figure 7.8. There are only two nodes due to non-determinacy (see below).

7.6 Non-Determinacy

Non-determinacy arises in CSP whenever more than one guard succeed in an alternative command or a repetitive command. Symbolic execution in CSP can be applied as follows. In a case where N guards succeed in an alternative or a repetitive command, execution will proceed in N execution paths with each path selecting one guarded command for execution. In the previous example, non-determinacy arises when both B and C have sent a message to A. In that case, two execution paths will be generated with A accepting a message from either B or C. In figure 7.A, non-determinacy gives rise to 1.2.1, 1.2.2, 2.2.1 and 2.2.2. In figure 7.8, non-determinacy gives rise to 1 and 2.

7.7 Strengths and Weaknesses of Symbolic Execution in CSP

Of the three proof systems for CSP that were introduced in section 7.2, symbolic execution has some very distinct advantages over the other two. It is a proof system that is easy to understand and intuitively appealing to a programmer because of the close resemblance to the actual execution of

the program. Most programmers are used to the idea of "walking-through" their programs which is really an informal form of symbolic execution. During such an exercise, a programmer verifies a program by mentally constructing an execution sequence. In formal symbolic execution, the only part of the proof that requires training and practice is the transformation of correctness to assertions. The programmer must learn to define the correctness of a program in terms of numeric and logical relationships between program variables which must hold during and after program execution.

Another advantage of symbolic execution is the ability to prove a wide range of properties including deadlocks, livelocks and race conditions using the same technique. In other proof systems, these properties all require separate but equally involved proofs.

Finally, because symbolic execution involves mainly the manipulation of symbols according to a set of well-defined rules, the procedure can be automated. With an automated system, the programmer then will only have to annotate the program with the appropriate assertions and the program prover will automatically execute the program symbolically and reveal the validity of the assertions.

The main disadvantage of symbolic execution is the possibility of "path-explosion". In section 7.5, it was seen that in order to reveal the possibility of race conditions, the execution tree must cover all possible execution sequences which can lead to a large number of paths. Although the exclusive use of messages in process communications reduces the number of relevant execution sequences to be considered, the total number of paths is still often very large. However, it is apparent that further research can alleviate this problem by eliminating paths which will not result in race conditions.

For example, all three processes A, B and C are ready to execute the following I/O instructions.

```
Process A :      B ! x
Process B :      C ! y
Process C :      B ? z
```

Regardless of the order of execution of these instructions, the result will always be the same, namely, with process C receiving a message from process B. In this case, different execution paths do not have to be generated because there is no possibility of a race condition. However, a race condition might arise if process C executes an alternative command which accepts inputs from either process A or process B. Rules which can help us detect these cases will greatly simplify the verification procedure.

Figure 7.A Verification Tree with Race Conditions Checked

Figure 7.B Verification Tree with Race Conditions not Checked

2 2
.OF / DE

In this chapter, a telephone switching system will be modelled and analysed using CSP. In section 8.1, some basic concepts in telephony and telephone switching will be introduced. Section 8.2 examines in detail a trunk-to-trunk call and the steps involved in processing such a call. A hypothetical telephone switching system called TSS will be described in 8.3. TSS is based largely on the Northern Telecom DMS switches. The system architecture of TSS will be examined. In section 8.4, the operation of TSS will be modelled using CSP and in 8.5, the model will be analysed using symbolic execution.

8.1 Basic Concepts in Telephony and Telephone Switching

This section introduces some of the basic concepts in telephone switching. For brevity, some definitions will be simplified and the reader is referred to [TALL79] for more detailed and precise definitions of telephony terms.

A telephone switching network consists of a network of telephone switches which reside in telephone offices. Most telephone offices have only one telephone switch in each office. Each subscriber telephone set is connected to a telephone office via a telephone circuit called the subscriber loop (or simply a line). Such an office (where subscriber lines terminate) is called a central office or a local office. Telephone offices are interconnected via telephone circuits called trunks. Each trunk is capable of carrying one telephone call at any given time. An office which only has trunks and no subscriber lines is called a toll office. Figure 8.A shows a simple telephone network with three central offices (CO1, CO2 and CO3) and two toll

Figure 8.A A Sample Telephone Network

offices (T01 and T02). An intraoffice call is a call which involves two subscribers of the same central office. An inter-office call is one whose parties are connected to two different offices. Such a call must therefore be switched through one or more trunks. Typically all intra-office calls are local non-chargeable calls whereas some of the inter-office calls are chargeable.

Referring again to figure 8.A, if subscriber s11 wishes to call subscriber s13, the call can be handled entirely within C01. All that C01 has to do is to set up a connection between the two parties. This is an example of an intra-office call. Because this type of a call involves two lines of an office, it is also called a line-to-line call. However, if s12 wishes to call s33, the call must be switched through the two toll offices, namely, T01 and T02. First, C01 must connect s12 to a trunk to T01, namely, t1. T01 must then connect t1 to t3 and T02 connects t3 to t4. Finally, C03 connects t4 to s33 and now a continuous path is established between s12 and s33. As the call is switched through different offices, the processing required in each case is also different. To C01, the call is a line-to-trunk call (s12 to t1). To T01 and T02, the call is a trunk-to-trunk call and finally, to C03, it is a trunk-to-line call.

In this chapter, the handling of a trunk-to-trunk call will be modelled.

8.2 Trunk-To-Trunk Call

As mentioned in the previous section, a trunk is a circuit between two offices capable of carrying a single call at any given time. For simplicity, only two-way trunks (trunks in which the office at either end of the trunk can originate a call) will be considered. In the example in figure 8.A,

before CO1 can make the connection between s12 and t1, it must first inform the office at the other end of t1, namely T01, that it intends to use the trunk. Furthermore, it must also be able to forward the dialled digits to T01 to enable it to route the call to T02. The transfer of information between the two offices of a trunk is done via trunk signalling. Signalling is commonly achieved by setting the trunk circuit into one of a set of electrical states such as on-hook, off-hook, seizure and wink etc. Apart from these signals which are intended for the office at the other end, an office may also signal directly to a subscriber through the use of audible signals. For example, if T01 finds out that trunk t3 is in use and hence cannot be used for a call between s12 and s33, it will generate a busy tone back to s12.

The processing of a trunk-to-trunk call can be described in terms of a series of discrete events. In figure 8.A, the connection between t1 and t3 in T01 is an example of a trunk-to-trunk call. CO1 first initiates the call to T01 by sending a 'seizure' signal to T01, meaning that CO1 is now requesting for the use of the trunk. If T01 is ready to accept the incoming call, it will acknowledge by sending a 'wink' back to CO1. (A glare occurs when both CO1 and T01 try to seize the trunk at the same time. Both offices will detect the glare condition and both will attempt to use another trunk. In this case, there is no other trunk which can be used and the call will be denied.) CO1 will now start sending the dialled digits to T01. Based on these digits, T01 then decides to route the call to T02 via t3. It then goes through the same protocol of sending a seizure signal to T02 and waiting for a wink. If t3 is seized successfully, T01 will then send the digits to T02. After all the digits have been sent, a physical connection will then be made between t1 and t3. The trunk-to-trunk call is now connected as far as T01 is concerned. After all the connections have been made, the telephone set of s33 will

start ringing. When s33 answers the call by picking up the handset (thus going off-hook), C03 will send an off-hook signal to T02 via t4. Similarly, T02 will send an off-hook signal to T01 which then gets sent to C01. The call now enters the talking state. Thus T01 knows that the s33 has answered the call when it sees an off-hook signal from T02 via t3. A similar sequence of events happen when the call is terminated. If s12 decides to terminate the call by hanging up and thereby going on-hook, C01 will send an on-hook signal to T01 via t1 and T01 sends it to T02 and so on. In this case, T01 knows that the call has been terminated when it detects an on-hook signal from either t1 or t3. When the call has been terminated, the physical connection between t1 and t3 will be taken down.

8.3 TSS as a Stored Program Controlled Digital Toll Switch

8.3.1 Telephone Switching Equipment

Each telephone office has at least one telephone switch which is capable of performing the following functions automatically - signalling, supervision, translation and switching. There are many types of telephone switching equipment. The earlier ones were step-by-step switches and crossbar switches which were controlled mostly by mechanical devices such as relays. The newer switching equipment is stored program controlled which means that the operation of the telephone switch is controlled by programmable computers.

The even newer generation of telephone switches are digital which means that a voice call is being switched in a digital form within the switch.

In the following sections, a model will be developed for a hypothetical digital telephone switch called TSS (Telephone

Switching System).

8.3.2 System Architecture of TSS

TSS is a hypothetical telephone switching system based largely on the Northern Telecom DMS-200 ([PENN82], [BOUR80], [SEVI80] and [PENN80]) with simplifications made to keep the problem of specification and verification a manageable one.

TSS is a digital toll switch which connects to other telephone switches in the network via analog trunks. In other words, the voice being transmitted through a trunk must first go through an analog-to-digital conversion as it enters the switch. The digitized voice samples can then be switched within TSS to the destination trunk where it goes through a digital-to-analog conversion before it is transmitted to another office. Because TSS is a toll switch, all calls are trunk-to-trunk calls and as mentioned previously, we assume all trunks to be two-way trunks.

TSS is a distributed processing system with a central control responsible for the overall co-ordination and supervision of the activities carried out by an array of peripheral processors which in turn are responsible for specialized functions. Figure 8.8 shows a block diagram of the major components of TSS.

Each trunk on a TSS is identified by a unique number from 1 to N and the number is called the trunk index.

Central Control (CC)

The central control is the control centre of the telephone switch. It co-ordinates and supervises the activities in

Figure 8.B A Block Diagram of TSS

Figure 8.C A Trunk-to-Trunk Call between Trunks *i* and *j*

the peripherals (trunk modules and network module). The CC is programmed in a high level concurrent language and the operating system allows a large number of concurrent processes to exist at the same time. Each telephone call is handled by a separate call process.

The call processes are created during system initialization and each call process is dedicated to one trunk. Thus each call process can also be identified by a trunk index. The call process is involved in the handling of a call only if its trunk is the originator. For example, if TRUNK [i] tries to originate a trunk-to-trunk call, that is, if the office to which TRUNK [i] is connected tries to seize it, then CALL_PROCESS [i] will be involved and it will be perform all the necessary processing required for the call for its whole duration.

Trunk Module (TM)

The trunk modules are the peripherals which interface directly with the adjacent telephone offices within the switching network. Up to 30 analog trunks can terminate on a single trunk module. Each trunk module connects to the network module via two digital links. Each link consists of 32 full-duplex channels. Channel 0 is used for messaging only. Channels 1 to 15 and channels 17 to 31 are speech channels which carry digitized voice signals from the 30 analog trunks. Channel 16 is not used. There is a fixed one-to-one mapping between each analog trunk and a speech channel. When the voice signal enters the switch via the trunk, it undergoes an analog-to-digital conversion and the digitized voice samples are then transmitted via one of the channels into the network module where they can be switched. Likewise voice samples from the other party can be switched through to the same channel back to the trunk module where it undergoes a digital-to-analog conversion before it gets

transmitted out on the trunk again.

The trunk module has 30 concurrent terminal processes with each dedicated to one trunk. Thus if trunks i to $i+29$ terminate on a particular trunk module, the terminal processes will be designated as TRUNK_PROCESS [i] to TRUNK_PROCESS [$i+29$].

Each terminal process is responsible for the signalling and supervision of its trunk. It reports any change of state in the trunk to the call process in the CC and sends the appropriate signals down the trunk under the direction of the CC.

Network Module (NM)

The network module is the switching centre of TSS. It consists of two identical planes each of which is capable of switching voice samples from any channel on a link to a TM to another channel on a link to a TM. The planes are duplicated for increased fault tolerance (see below). The software of the network module is relatively simple. It has a single network process which constantly waits for messages from any of the call processes from the CC. A message from the CC tells the network process to either make a new connection or to disconnect an old connection on each plane of the network. Figure 8.C shows an example of a network connection. Trunk i is mapped into channel 21 of TM3 and trunk j is mapped into channel 7 of TM5. When the voice samples from trunk i enter the network, they are directed to channel 7 of TM5 which they undergo a D/A conversion and get transmitted out on trunk j . Likewise, voice samples from trunk j enter the network through channel 7 of TM5 where they get switched to channel 21 of TM3 and eventually to trunk i .

8.3.3 Trunk Supervision and Fault Detection

Before a trunk has been connected via the network to another trunk, all that the terminal process has to do is to report any state changes to the call process in the CC and to send signals down the trunk under the direction of the CC. Once the trunk has been connected, the terminal process must also be able to monitor the state of the trunk to which it is connected. This is done via a supervision bit (or SV bit) tagged on to the voice sample. In figure 8.C for example, once trunk i has been connected to trunk j, `TERMINAL_PROCESS [i]` will convert the state of trunk i to either `SV_ONHOOK` or `SV_OFFHOOK` which gets transmitted together with the voice samples to `TERMINAL_PROCESS [j]` and thereby enabling `TERMINAL_PROCESS [j]` to monitor the state of trunk i.

In addition to monitoring the SV bit, a terminal process must also monitor the quality of speech coming back from the network module. The trunk module is connected to the two planes of a network via two identical links. The incoming voice samples from a trunk are transmitted on both links into the network. Both streams of voice samples are switched and both emerge out of the other end. However each terminal process only extracts the voice samples from one plane of the network at any given time. It monitors the speech samples for parity errors and if an error is detected, it will automatically extract the voice samples from the other plane and report the condition to the call process in the CC. This condition is called an integrity failure. If a fault is detected on the other plane as well, the call will be dropped and this condition is known as an integrity loss. In figure 8.C for example, trunk i enters the network on both planes but only receives voice samples of trunk j from one plane, say, plane 0. If an

Figure 8.D TSS within a Telephone Switching Network

integrity failure occurs on plane 0, the terminal process will switch the network plane and select voice samples from plane 1 instead.

8.3.4 A Trunk-to-Trunk Call in the Context of TSS

Having already described the architecture of TSS, a trunk-to-trunk call will be traced as it progresses through the switch. Figure 8.D shows a TSS in toll office A. One of the trunks which connects office A and office B is trunk i. Assume now that office B has decided to route a call to office A via trunk i. It does it by first sending a seizure signal to office A. Terminal process i in TM1 receives the signal and immediately sends an origination message to call process i in the CC. The call process then sends a message to the terminal process instructing it to start collecting digits. The terminal process then sends a wink down the trunk. Upon receiving the wink, office B will start sending down the digits. When the terminal process has collected all the digits, the digits will be sent back to the call process which translates them into a destination office, say, office C. One of the trunks connecting office A to C may be trunk j in which case a message will be sent by the call process to terminal process j instructing it to seize the trunk. If the trunk is seized successfully, the call process will send a message to the network process asking it to make a connection between trunks i and j. After the connection has been made, two messages will be sent to the two terminal processes informing them that the connection has been made. Terminal process j will then start sending the digits to office C via trunk j. On the other hand, if a glare occurs while the terminal process attempts to seize the trunk, the condition will be reported back to the call process and it will attempt to reroute the call through another trunk such as trunk k.

Figure 8.E Timing Diagram of a Trunk-to-Trunk Call

Once the digits have been sent to office C, both terminal processes will be waiting for the called party to answer. When this happens, terminal process j will report to the call process. After the call has been answered, the terminal processes will then wait for either of the two parties to hang up. When an on-hook condition is detected, the terminal process will send a message to the call process which will in turn terminate the call by instructing the network process to take down the connection.

While the call is in the connected state, both terminal processes will constantly monitor the integrity of the call. If a fault is first detected, the other network plane will be used instead and a report is sent to the call process. If a fault is detected for a second time, the call be dropped.

Figure 8.E shows the sequence of events that happen during a successful trunk-to-trunk call.

8.4 Modelling TSS Using CSP

The operation of TSS can be modelled by specifying the processes involved in a trunk-to-trunk call, namely, the call process, the terminal process and the network process. The model is presented in Appendix A. In addition, a process called FAULT_GENERATOR has also been included. It is a process dedicated to a pair of terminal processes which under non-deterministic conditions generates faults to those two processes. Thus for example FAULT_GENERATOR [i, j] will occasionally generate faults to both terminal processes i and j. For simplicity, it is assumed that a fault is always bi-directional in that once it occurs, both i and j will be affected at the same time.

In chapter 2, it was mentioned that the parallel processing systems in this thesis do not have any global variables. From the model, the reader may be led to believe that variables such as TRUNK_STATE in the call process and CONNECTION_MAP in the network process are global. However, they are really not intended to be shared. They are there only to facilitate the analysis of the model. They are similar in nature to the auxiliary variables described in [OWIC76b] and history variables as in [HOWA76]. One might then ask how the translation of the digits can be achieved without knowing the states of other trunks. Without shared memory, there will have to be an additional process whose sole function is to perform digit translation. Every call process must then inform this process of any change in state of its trunk so that it knows at any given time which trunks are available and which ones are not. The algorithm of digit translation is not essential in the analysis and hence has not been included in the model.

8.5 Verification of TSS Using Symbolic Execution

One condition will be verified using symbolic execution, namely, that during a trunk-to-trunk call, the system is deadlock free and that all resources will be released when the call has been terminated. If the system is deadlock free, then it must be shown that during symbolic execution, the system does not end up in a state where no further progress is possible. To prove that all resources are released when a call has been terminated, one must first express the statement in the form of an assertion. Since a trunk-to-trunk call involves two trunks and a network connection, we must be able to prove that at the end of a call, both trunks involved are idle and that the network connection between the two has been taken down.

Hence if a trunk-to-trunk call involves two trunks m and n, then at the end of the call, it must be shown that the following verification condition (VC) is true.

```
VC : (trunk_state [m] = idle) and  
      (trunk_state [n] = idle) and  
      (connection_map [m, n] = false)
```

Having identified the verification condition, we must then decide where to assert the condition. It was said earlier that the verification condition must be true when the call has been terminated. Since the call process co-ordinates and supervises a call, the most logical place to assert the condition is in the call process. The call process waits for its trunk to originate a call. Once the call is in progress, it interacts with the terminal processes and the network process until the call has been terminated. It then waits for the next call to commence. Thus it becomes obvious that the verification condition must hold while the call process is idle waiting for an origination.

There is still another complication. So far, it has been assumed that the call first gets connected before it is terminated. From the model, it can be seen that under some conditions, the call may not get connected at all, for example, if all trunks are busy. In that particular case, the verification condition is not meaningful any more because there is no trunk to which the originator is connected. Another variable has been included in the model to resolve this problem. The boolean variable CONNECTED is set to true only when the network connection has been properly set up and false otherwise. The verification condition can now be modified as follows.

```
VC : (trunk_state [m] = idle) and
```

((not connected) or
((trunk_state [n] = idle) and
(connection_map [m, n] = false)))

where trunk m is the originator. In this verification condition, the terminator has to be idle only if the call is connected, that is, if CONNECTED is true.

The actual verification procedure is presented in Appendix B. Figure 8.F is a graphical representation of the verification tree.

The symbolic execution of the model was carried out manually using the rules described in chapter 7. To simplify the verification procedure, the possibility of race conditions was ignored. Thus whenever more than one process was ready to run, one was picked arbitrarily. Starting from the initial conditions, the model was executed symbolically. Whenever there were more than one possible outcome, different execution paths would be split up and each branch would be executed independently. For example, after all the dialed digits have been collected at node 2, there are two possible outcomes. If the digits are correct and a route is available, execution will progress to node 2.2. Otherwise, execution will progress to node 2.1. 2.1 is a terminal node because here the call is terminated, that is, TRUNK_PROCESS [m] returns to its home state waiting for a new origination. All the execution paths were traced until they eventually reached a terminal node. Although the symbolic values of the variables have not been included in the verification procedure, their values can be traced by following the paths down the verification tree.

After the verification tree has been constructed, it must then be shown the assertion holds whenever a call has

terminated. That is, the assertion must be tested at each of the terminal nodes. For example, 2.2.1.2.2.1.3.1 is a terminal node. The "history" of the node can be interpreted as follows. Trunk m originated a call (0). The originator completed dialling (2) and trunk n was picked as the route (2.2). However a glare occurred (2.2.1) and trunk p was picked as the alternate route (2.2.1.2). Trunk p was successfully seized (2.2.1.2.2). An integrity failure occurred (2.2.1.2.2.1) before the call was eventually answered (2.2.1.2.2.1.3). However, a failure was detected again and this time, the call was dropped (2.2.1.2.2.1.3.1). Because trunk m was connected to trunk p, the verification condition becomes

```
VC : (trunk_state [m] = idle) and
      ((not connected) or
       ((trunk_state [p] = idle) and
        (connection_map [m, p] = false)))
```

First we must trace the value of TRUNK_STATE [m]. It got set to BUSY in 0 and got set to IDLE in 2.2.1.2.2.1.3.1. Next we must trace the value of CONNECTED. It was set to false in 0 and was later set to true in 2.2.1.2.2. TRUNK_STATE [p] was set to BUSY in 2.2.1.2 and later set to IDLE in 2.2.1.2.2.1.3.1. CONNECTION_MAP [m, p] was set to TRUE in 2.2.1.2.2 and set to FALSE in 2.2.1.2.2.1.3.1. Thus when the assertion point is reached, the variables have the following values.

```
trunk_state [m] = idle
connected = true
trunk_state [p] = idle
connection_map [m, p] = false
```

=> VC

Thus the verification condition has been shown to hold for this node. After repeating the same procedure for every terminal node on the verification tree, the verification condition was found to hold in every case. Thus we conclude that the system is partially correct, namely, that the assertion holds once a terminal node is reached.

Furthermore, by carrying out the symbolic execution, it was found that the system does not contain any deadlocks. That is, during symbolic execution, one did not reach a state which was not terminal but from which no further execution was possible because all processes were blocked. Thus we can conclude that starting from the initial conditions, a terminal node is always reachable and since it was concluded earlier that the assertion holds for every terminal node, it can now be concluded that the system is totally correct. In other words, a trunk-to-trunk call in TSS will not cause the system to lock up and will always release all the resources that it used at the end of the call.

Finally, it should be noted that the above analysis was done without taking into consideration the possibility of race conditions. In fact, the system as modelled does produce race conditions which may result in deadlocks under some conditions. For example, if a call process decides to route a call via trunk i, it will send a message to the terminal process instructing it to attempt to seize the trunk. If before the message reaches the terminal process, the terminal process detects an off-hook condition from trunk i, it will send a message back to the central control reporting an origination and wait for the central control to signal digit collection. Under these circumstances, the system will result in a deadlock with the terminal process waiting for an instruction from the central control to start digit collection while the call process in the central control is waiting for the terminal process to report on the seizure

attempt. The potential deadlock was not discovered during symbolic execution because not all the execution paths due to parallelism were examined. In the real system, that is, DMS, more elaborate exception handling and timeouts are used to prevent race conditions.

Figure 8.F Verification Tree for TSS

Figure 8.F (cont.)

CHAPTER 9 CONCLUSIONS

The objective of this thesis was to study specification and verification techniques for Parallel Processing Systems. In chapter 3, the importance of formal specification and verification was explained. Completeness, abstraction, ease of understanding and amenity to formal analysis were found to be the criteria for a good specification language. The verification conditions for a PPS were examined and the following properties were identified to be conditions that a system must satisfy in order to be "correct" - absence of deadlock, absence of livelock, absence of starvation and degradation and finally partial correctness.

Three classes of specification and verification techniques were examined and compared in chapters 4 to 6. Transition models are highly abstract which offer the designer an abstract and high level view of the operation of a system. However, transition models are inadequate in their representation of messaging as well as program variables. Verification of transition models typically involves the use of reachability analysis which is made difficult because of the state-explosion problem. It also suffers from the inability to verify numerical and logical relationships between variables.

In chapter 5, two hybrid models were studied. Hybrid models attempt to partition a system in two aspects - the control aspect and the data aspect - and to model each separately using two different formalisms, typically a state transition model and a programming language model. With the help of program variables, they manage to achieve a significant reduction in the total number of states, thus alleviating the state-explosion problem encountered in pure transition models. Furthermore, the inclusion of program

variables means that numerical and logical relationships can be adequately represented. However, the representation of message-based communications is still inadequate. There are significant differences between the model of messages and the way messages are implemented in practice. Another difficulty with hybrid models is that it is generally impossible to obtain a clear cut partitioning into the control and the data aspects. Hence hybrid models have a danger that there may be contradictions or underspecification in the model.

CSP was introduced in chapter 6 as a programming language model. Traditionally, there are two objections raised against using a programming language model, the first being that the model limits the choice of the implementation language. Secondly, the model might be overloaded with implementation details. CSP minimises these shortcomings by being a "lightly loaded" language consisting of only the essential language features and few implementation details. Furthermore, CSP is syntactically and semantically similar to many high level structured languages and hence does not compromise the choice of the implementation language. A model in CSP also has the advantage that it can be analysed using parallel program proving techniques. Some extensions were also proposed in chapter 6.

In chapter 7, parallel program proving techniques were examined and symbolic execution was found to be the most intuitive method offering the possibility of an automated program prover and the ability to prove a variety of properties using a unified approach. However, if race conditions are to be included as one of the verification conditions, then symbolic execution may also lead to a path explosion problem similar to the state explosion in the transition models.

In chapter 8, a hypothetical telephone switching system (TSS)

was modelled using CSP and analysed using symbolic execution. Even though TSS was hypothetical, it was based on a real telephone switching system and hence did represent a realistic modelling problem. CSP was found to be adequate in the representation of the complex interactions between the processes involved in a trunk-to-trunk call. The model appeared to be accurate and concise. The subsequent analysis of the model, carried out manually, although long was still manageable, thus demonstrating the feasibility of symbolic execution. The analysis would also be much easier if automated tools were available.

Just as no craftsman could restrict himself to the use of a single tool, no specification language can be expected to be effective for all applications. It was found in this thesis that a properly chosen programming language can be an effective tool in the specification of a wide range of systems. However, that conclusion does not rule out the use of transition or hybrid models for specific problems which may have a very simple messaging mechanism and which may be heavily state oriented with little need for program variables. Some fairly complicated communication protocols such as X.21 and X.25 have been modelled and verified successfully using transition models. CCITT also has in its recommendation Z.101 proposed a basically finite state model called the Specification and Description Language (SDL) to be used in the functional specification and description of stored program controlled telephone exchanges.

In the view of the author, future research on specification and verification techniques should diverge in two directions, the first being the development of a general purpose specification language such as CSP and the second being the development of special purpose specification languages such as SDL. Future Parallel Processing Systems are likely to be complex systems which consist of perhaps hundreds of

processors and thousands of processes. These systems will most likely consist of many functional subsystems such as a switching subsystem, a communication subsystem, a maintenance subsystem and a data base management subsystem. In such a system, a general purpose specification language will be required to define the global interactions between the major subsystems. These interactions must be verified to ensure that they will not lead to a deadlock and that they will eventually lead to some "desirable" results being produced. After the overall organisation has been verified, the individual subsystems must then be specified and analysed using special purpose specification languages each tailored to handle the characteristics of the subsystem being specified.

APPENDIX A: CSP MODEL OF TSS

```

    call_process [m] ? circuits_busy;
    trunk [m] ! busy_tone;
    trunk [m] ? onhook;
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process ? call_terminated;
    trunk_state [m] := idle;
ASSERT VC

```

2.2

```

path condition : j = n <> 0
    { there is an available route trunk [n] }
process : call_process [m]
    trunk_state [n] := busy;
    terminal_process [n] ! seizure;
process : terminal_process [n]
    call_process [m] ? seizure;
    trunk [n] ! offhook;

```

2.2.1

```

path condition : trunk [n] ? glare
    { failed to seize trunk [n] }
process : terminal_process [n]
    trunk [n] ? glare;
    call_process [m] ! glare;
process : call_process [m]
    first_try := true;
    terminal_process [n] ? glare;
{ if first_try -> }
    trunk_state [n] := idle;
    find_alternate_route (j)

```

2.2.1.1

```

path condition : j = 0
    { no alternate route }
process : call_process [m]
    terminal_process [m] ! circuits_busy;
process : terminal_process [m]
    call_process [m] ? circuits_busy;
    trunk [m] ! busy_tone;
    trunk [m] ? onhook;
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process ? call_terminated;
    trunk_state [m] := idle;
ASSERT VC

```

2.2.1.2

```

path condition : j = p <> 0
    { alternate route is trunk [p] }
process : call_process [m]
    trunk_state [p] := busy;
    terminal_process [p] ! seizure;
process : terminal_process [p]

```

process call_process [i];

(This is the call process residing in the central control that supervises all originations from trunk [i]. Thus, whenever this process is actively handling a call, trunk [i] must be the originating trunk and trunk [j] will be the trunk on which the call is routed.)

var

i, j : trunk_index;
first_try : boolean;
connected : false;
da : digit_array;

label call_in_progress;

begin

trunk_state [i] := idle;
connected := false;
do terminal_process [i] ? origination ->
(trunk [i] has just initiated a call by going offhook)
connected := false;
trunk_state [i] := busy;
terminal_process [i] ! collect_digits;
call_in_progress :
do terminal_process [i] ? call_terminated ->
(the caller hung up without dialling the digits)
trunk_state [i] := idle;
exit call_in_progress
|| terminal_process [i] ? digits (da) ->
(the caller completed dialling the digits)
find_route (da, j);
if j = 0 ->
(either the dialled digits were incorrect or there is
no available trunk to route the call.)
terminal_process [i] ! circuits_busy;
terminal_process [i] ? call_terminated;
trunk_state [i] := idle;
exit call_in_progress
|| j <> 0 ->
(a route has been found - try to seize it)
trunk_state [j] := busy;
terminal_process [j] ! seizure;
first_try := true;
do terminal_process [j] ? glare ->
(a glare has occurred)
if not first_try ->
(failed for the second time - give up)
terminal_process [i] ! circuits_busy;
terminal_process [i] ? call_terminated;
trunk_state [i] := idle;
trunk_state [j] := idle;
exit call_in_progress


```

! first_try ->
( failed for the first time - try again )
  trunk_state [j] := idle;
  find_alternate_route (j);
  if j = 0 ->
    ( no alternate route )
    terminal_process [i] ' circuits_busy;
    trunk_state [i] := idle;
    exit call_in_progress
  ! j <> 0 ->
    ( there is an alternate route
      - try to seize it )
    trunk_state [j] := busy;
    terminal_process [j] ' seizure;
    first_try := false
  fi
fi
! terminal_process [j] ? wink ->
( trunk [j] has been seized successfully )
  network_process ' make_connection (j);
  connected := true;
  terminal_process [j] ! digits (da);
  terminal_process [i] ' start_supervision;
  terminal_process [j] ! start_supervision;
  do terminal_process [i] ? integrity_failure ->
    log_integrity_failure
  ! terminal_process [j] ? integrity_failure ->
    log_integrity_failure
  ! terminal_process [i] ? integrity_loss ->
    ( failure occurred twice - drop the call )
    trunk_state [i] := idle;
    trunk_state [j] := idle;
    terminal_process [j] ? integrity_loss;
    network_process ! remove_connection (j);
    exit call_in_progress
  ! terminal_process [j] ? integrity_loss ->
    ( failure occurred twice - drop the call )
    trunk_state [i] := idle;
    trunk_state [j] := idle;
    terminal_process [i] ! integrity_loss;
    network_process ! remove_connection (j);
    exit call_in_progress
  ! terminal_process [i] ! call_terminated ->
    ( the caller has hung up )
    trunk_state [i] := idle;
    trunk_state [j] := idle;
    network_process ! remove_connection (j);
    exit call_in_progress
  ! terminal_process [j] ! answered ->
    ( the called party has answered )
    start_billing (i);
    do terminal_process [i] ? integrity_failure ->
      log_integrity_failure
    ! terminal_process [j] ? integrity_failure ->

```

```

log_integrity_failure
! terminal_process [i] ? integrity_loss ->
  stop_billing (i);
  trunk_state [i] := idle;
  trunk_state [j] := idle;
  terminal_process [j] ! integrity_loss;
  network_process ! remove_connection (j);
  exit call_in_progress
! terminal_process [j] ? integrity_loss ->
  stop_billing (i);
  trunk_state [i] := idle;
  trunk_state [j] := idle;
  terminal_process [i] ? integrity_loss;
  network_process ! remove_connection (j);
  exit call_in_progress
! terminal_process [i] ! call_terminated ->
( the caller has hung up )
  stop_billing (i);
  trunk_state [i] := idle;
  trunk_state [j] := idle;
  network_process ! remove_connection (j);
  exit call_in_progress
! terminal_process [j] ! call_terminated ->
( the called party has hung up )
  stop_billing (i);
  trunk_state [i] := idle;
  trunk_state [j] := idle;
  network_process ! remove_connection (j);
  exit call_in_progress
od
od
fi
od
od
end;

```

process terminal_process (i);

(This terminal process services trunk i under the direction of
a call process in the central control.)

var

i, j : trunk_index;

fail_number : integer;

da : digit_array;

label origination_in_progress;

begin

do trunk (i) ? offhook ->

(trunk (i) is initiating an origination by going offhook)

call_process (i) ! origination;

origination_in_progress :

do call_process (i) ? collect_digits ->

(this office is ready to accept the call

- wink to acknowledge)

trunk (i) ! wink;

do trunk (i) ? onhook ->

(the caller has aborted the call by not hanging up)

call_process (i) ! call_terminated

if trunk (i) ? digits (da) ->

(the caller has completed dialling all the digits)

call_process (i) ! digits (da);

if call_process (i) ? circuits_busy ->

(all circuits are busy - generate busy tone)

trunk (i) ! busy_tone;

trunk (i) ? onhook;

call_process (i) ! call_terminated;

exit origination_in_progress

if call_process (i) ? start_supervision ->

(the call has been connected)

fail_number := 0;

do fault_generator ? failure ->

(an integrity failure has occurred)

if fail_number = 0 ->

switch_network_plane;

fail_number := 1;

call_process (i) ! integrity_failure

if fail_number <> 0 ->

trunk (i) ! onhook;

call_process (i) ! integrity_loss;

exit origination_in_progress

fi

if trunk (i) ? onhook ->

(the caller has hung up)

trunk (i) ! onhook;

terminal_process (j) ! sv_onhook;

```

        call_process [i] ! call_terminated;
        exit origination_in_progress
    } terminal_process [j] ? sv_offhook ->
    { the called party has answered }
        trunk [i] ! offhook;
    } terminal_process [j] ? sv_onhook ->
    { the called party has hung up }
        trunk [i] ! onhook;
        exit origination_in_progress
    od
fi
od
od

} call_process [j] ? seizure ->
{ trunk [j] is now the originator and it is trying to make
a call to trunk [i] }
    trunk [i] ! offhook; { initiate a call to trunk [i] }
    termination_in_progress :
    do trunk [i] ? glare ->
    { a glare has occurred }
        call_process [j] ! glare;
        exit termination_in_progress
    } trunk [i] ? wink ->
    { trunk [i] has acknowledged the seizure }
        call_process [j] ! wink;
        call_process [j] ? digits (da);
        trunk [i] ! digits (da);
        call_process [j] ? start_supervision;
        fail_number := 0;
        do fault_generator ? failure ->
        { an integrity failure has occurred }
            if fail_number = 0 ->
                switch_network_plane;
                fail_number := 1;
                call_process [j] ! integrity_failure
            } fail_number <> 0 ->
                trunk [i] ! onhook;
                call_process [j] ! integrity_loss;
                exit termination_in_progress
        fi
    } trunk [i] ? offhook ->
    { the called party has answered }
        terminal_process [j] ! sv_offhook;
        call_process [j] ! answered;
    } trunk [i] ? onhook ->
    { the called party has hung up }
        trunk [i] ! onhook;
        terminal_process [j] := sv_onhook;
        call_process [j] ! call_terminated;
        exit termination_in_progress
    } terminal_process [j] ? sv_onhook ->
    { the originator has hung up }
        trunk [i] ! onhook;

```

exit termination_in_progress

od

od

od

end;

```
process fault_generator [i, j];
```

(The fault generator is an autonomous process which may under indeterminate conditions generate integrity failures.)

```
var  
  i, j      : trunk_index;
```

```
begin  
  do true ->  
    ( do nothing - that is, no error )  
  | true ->  
    ( generate failure messages to trunk [i] and trunk [j] )  
    terminal_process [i] ! failure;  
    terminal_process [j] ! failure  
  od  
end;
```

```
process network_process;
```

(The network process is responsible for making and removing speech path connections. CONNECTION_MAP is a map of all the connections that have been set up. If CONNECTION_MAP [i, j] is true, then there is a connection between the two trunks and false otherwise.)

```
var  
  i      : trunk_index;  
  destination : trunk_index;
```

```
begin  
  do call_process [i] ? make_connection (destination) ->  
    connection_map [i, destination] := true  
  | call_process [i] ? remove_connection (destination) ->  
    connection_map [i, destination] := false  
  od  
end;
```

(main body of the program)

```
var  
  i, j      : trunk_index;
```

```
begin  
  ( start up all the call process, terminal processes, fault generators  
    and the network process )  
  
  for i over trunk_index : call_process [i] //
```

```
for i over trunk_index : terminal_process (i) //  
for i over trunk_index :  
for j over trunk_index : fault_generator (i, j) //  
network_process;
```

end.

3

APPENDIX B: VERIFICATION PROCEDURE OF TSS

~

(PROLOG

In the following verification tree, the originator is given a symbolic value of m. If there is a route available for the call, the destination trunk is given a symbolic value of n. If this trunk fails (that is, if a glare is encountered), an alternate route will be chosen. If there is one available, the alternate route will be given a symbolic value of p.

)

0.
process : terminal_process [m]
 trunk [m] ? offhook;
 call_process [m] ! origination;
process : call_process [m]
 terminal_process [m] ? origination;
 trunk_state [m] := busy;
 connected := false;
 terminal_process [m] ! collect_digits;
process : terminal_process [m]
 call_process [m] ? collect_digits;
 trunk [m] ! wink

1.
path condition : trunk [m] ? onhook
 { caller hangs up without completing dialling }
process : terminal_process [m]
 trunk [m] ? onhook;
 call_process [m] ! call_terminated;
process : call_process [m]
 terminal_process [m] ? call_terminated;
 trunk_state [m] := idle;
 ASSERT VC

2.
path condition : trunk [m] ? digits (da)
 { caller completed dialling }
process : terminal_process [m]
 trunk [m] ? digits (da);
 call_process [m] ! digits (da);
process : call_process [m]
 terminal_process [m] ? digits (da);
 find_route (da, j);

2.1
path condition : j = 0
 { invalid digits or no available route }
process : call_process [m]
 terminal_process [m] ! circuits_busy
process : terminal_process [m]

```

    call_process [m] ? circuits_busy;
    trunk [m] ! busy_tone;
    trunk [m] ? onhook;
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process ? call_terminated;
    trunk_state [m] := idle;
ASSERT VC

```

2.2

```

path condition : j = n <> 0
    { there is an available route trunk [n] }
process : call_process [m]
    trunk_state [n] := busy;
    terminal_process [n] ! seizure;
process : terminal_process [n]
    call_process [m] ? seizure;
    trunk [n] ! offhook;

```

2.2.1

```

path condition : trunk [n] ? glare
    { failed to seize trunk [n] }
process : terminal_process [n]
    trunk [n] ? glare;
    call_process [m] ! glare;
process : call_process [m]
    first_try := true;
    terminal_process [n] ? glare;
    { if first_try -> }
    trunk_state [n] := idle;
    find_alternate_route (j)

```

2.2.1.1

```

path condition : j = 0
    { no alternate route }
process : call_process [m]
    terminal_process [m] ! circuits_busy;
process : terminal_process [m]
    call_process [m] ? circuits_busy;
    trunk [m] ! busy_tone;
    trunk [m] ? onhook;
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process ? call_terminated;
    trunk_state [m] := idle;
ASSERT VC

```

2.2.1.2

```

path condition : j = p <> 0
    { alternate route is trunk [p] }
process : call_process [m]
    trunk_state [p] := busy;
    terminal_process [p] ! seizure;
process : terminal_process [p]

```

```

call_process [m] ? seizure;
trunk [p] ! offhook;

```

2.2.1.2.1

```

path condition : trunk [p] ? glare
  { fails to seize trunk [p] }
process : terminal_process [p]
  trunk [p] ? glare;
  call_process [m] ! glare;
process : call_process [m]
  first_try := false;
  terminal_process [n] ? glare;
  { if not first_try -> }
  terminal_process [m] ! circuits_busy;
process : terminal_process [m]
  call_process [m] ? circuits_busy;
  trunk [m] ! busy_tone;
  trunk [m] ? onhook;
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process ? call_terminated;
  trunk_state [m] := idle;
  trunk_state [p] := idle;
ASSERT VC

```

2.2.1.2.2

```

path condition : trunk [p] ? wink
  { seizes trunk [p] successfully }
process : terminal_process [p]
  trunk [p] ? wink;
  call_process [m] ! wink;
process : call_process [m]
  terminal_process [p] ? wink;
  network_process ! make_connection (p);
process : network_process
  call_process [m] ? make_connection (destination);
  connection_map [i, destination] := true;
  { => connection_map [m, p] = true }
process : call_process [m]
  connected := true;
  terminal_process [p] ! digits (da);
process : terminal_process [p]
  call_process [m] ? digits (da);
  trunk [p] ! digits (da);
process : call_process [m]
  terminal_process [m] ! start_supervision;
process : terminal_process [m]
  call_process [m] ? start_supervision;
  fail_number := 0;
process : call_process [m]
  terminal_process [p] ! start_supervision;
process : terminal_process [p]
  call_process [m] ? start_supervision;
  fail_number := 0;

```

2.2.1.2.2.1

```

path condition : ( integrity failure )
process : fault_generator [m, p]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, p] ? failure;
{   if fail_number = 0 ->   }
    switch_network_plane;
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [m] ? integrity_failure;
    log_integrity_failure;
process : fault_generator [m, p]
    terminal_process [p] ! failure;
process : terminal_process [p]
    fault_generator [m, p] ? failure;
{   if fail_number = 0 ->   }
    switch_network_plane;
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [p] ? integrity_failure;
    log_integrity_failure;

```

2.2.1.2.2.1.1

```

path condition : ( integrity failure )
process : fault_generator [m, p]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, p] ? failure;
{   if fail_number <> 0 ->   }
    trunk [m] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [m] ? integrity_loss;
process : fault_generator [m, p]
    terminal_process [p] ! failure;
process : terminal_process [p]
    fault_generator [m, p] ? failure;
{   if fail_number <> 0 ->   }
    trunk [p] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [p] ? integrity_loss;
    trunk_state [m] := idle;
    trunk_state [p] := idle;
    network_process ! remove_connection (p);
process : network_process
    call_process [m] ? remove_connection (destination);
{   => destination = p }
    connection_map [i, destination] := false;
{   => connection_map [m, p] = false   }

```

```

process : call_process [m]
    ASSERT VC

2.2.1.2.2.1.2
path condition : trunk [m] ? onhook
    ( caller hangs up before the call has been answered )
process : terminal_process [m]
    trunk [m] ? onhook;
    terminal_process [p] ! sv_onhook;
process : terminal_process [p]
    terminal_process [m] ? sv_onhook;
    trunk [p] ! onhook;
process : terminal_process [m]
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process [m] ? call_terminated;
    trunk_state [m] := idle;
    trunk_state [p] := idle;
    network_process ! remove_connection (p);
process : network_process
    call_process [m] ? remove_connection (destination);
    { => destination = p }
    connection_map [i, destination] := false;
    { => connection_map [m, p] = false }
process : call_process [m]
    ASSERT VC

```

```

2.2.1.2.2.1.3
path condition : trunk [p] ? offhook
    ( called party answers )
process : terminal_process [p]
    trunk [p] ? offhook;
    terminal_process [m] ! sv_offhook;
process : terminal_process [m]
    terminal_process [p] ? sv_offhook;
    trunk [m] ! offhook;
process : terminal_process [p]
    call_process [m] ! answered;
process : call_process [m]
    terminal_process [p] ? answered;
    start_billing (m);

```

```

2.2.1.2.2.1.3.1
path condition : ( integrity failure )
process : fault_generator [m, p]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, p] ? failure;
    { if fail_number <> 0 -> }
    trunk [m] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [m] ? integrity_loss;
process : fault_generator [m, p]

```

```

        terminal_process [p] ! failure;
process : terminal_process [p]
    fault_generator [m, p] ? failure;
    ( if fail_number <> 0 -> )
        trunk [p] ! onhook;
        call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [p] ? integrity_loss;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [p] := idle;
    network_process ! remove_connection (p);
process : network_process
    call_process [m] ? remove_connection (destination);
    ( => destination = p )
        connection_map [i, destination] := false;
    ( => connection_map [m, p] = false )
process : call_process [m]
    ASSERT VC

```

2.2.1.2.2.1.3.2

```

path condition : trunk [m] ? onhook
    ( caller terminates the call )
process : terminal_process [m]
    trunk [m] ? onhook;
    terminal_process [p] ! sv_onhook;
process : terminal_process [p]
    terminal_process [m] ? sv_onhook;
    trunk [p] ! onhook;
process : terminal_process [m]
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process [m] ? call_terminated;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [p] := idle;
    network_process ! remove_connection (p);
process : network_process
    call_process [m] ? remove_connection (destination);
    ( => destination = p )
        connection_map [i, destination] := false;
    ( => connection_map [m, p] = false )
process : call_process [m]
    ASSERT VC

```

2.2.1.2.2.1.3.3

```

path condition : trunk [p] ? onhook
    ( called party terminates the call )
process : terminal_process [p]
    trunk [p] ? onhook;
    terminal_process [m] ! sv_onhook;
process : terminal_process [m]
    terminal_process [p] ? sv_onhook;
    trunk [m] ! onhook;

```

```

process : terminal_process [p]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [p] ? call_terminated;
  stop_billing (m);
  trunk_state [m] := idle;
  trunk_state [p] := idle;
  network_process ! remove_connection (p);
process : network_process
  call_process [m] ? remove_connection (destination);
  { => destination = p }
  connection_map [i, destination] := false;
  { => connection_map [m, p] = false }
process : call_process [m]
  ASSERT VC

```

2.2.1.2.2.2

```

path condition : trunk [m] ? onhook
  { caller hangs up before the call has been answered }
process : terminal_process [m]
  trunk [m] ? onhook;
  terminal_process [p] ! sv_onhook;
process : terminal_process [p]
  terminal_process [m] ? sv_onhook;
  trunk [p] ! onhook;
process : terminal_process [m]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [m] ? call_terminated;
  trunk_state [m] := idle;
  trunk_state [p] := idle;
  network_process ! remove_connection (p);
process : network_process
  call_process [m] ? remove_connection (destination);
  { => destination = p }
  connection_map [i, destination] := false;
  { => connection_map [m, p] = false }
process : call_process [m]
  ASSERT VC

```

2.2.1.2.2.3

```

path condition : trunk [p] ? offhook
  { called party answers }
process : terminal_process [p]
  trunk [p] ? offhook;
  terminal_process [m] ! sv_offhook;
process : terminal_process [m]
  terminal_process [p] ? sv_offhook;
  trunk [m] ! offhook;
process : terminal_process [p]
  call_process [m] ! answered;
process : call_process [m]
  terminal_process [p] ? answered;
  start_billing (m);

```

2.2.1.2.3.1

```

path condition : { integrity failure }
process : fault_generator [m, p]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, p] ? failure;
{ if fail_number = 0 -> }
    switch_network_plane; \
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [m] ? integrity_failure;
    log_integrity_failure;
process : fault_generator [m, p]
    terminal_process [p] ! failure;
process : terminal_process [p]
    fault_generator [m, p] ? failure;
{ if fail_number = 0 -> }
    switch_network_plane;
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [p] ? integrity_failure;
    log_integrity_failure;

```

2.2.1.2.2.3.1.1

```

path condition : { integrity loss }
process : fault_generator [m, p]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, p] ? failure;
{ if fail_number <> 0 -> }
    trunk [m] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [m] ? integrity_loss;
process : fault_generator [m, p]
    terminal_process [p] ! failure;
process : terminal_process [p]
    fault_generator [m, p] ? failure;
{ if fail_number <> 0 -> }
    trunk [p] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [p] ? integrity_loss;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [p] := idle;
    network_process ! remove_connection (p);
process : network_process
    call_process [m] ? remove_connection (destination);
{ => destination = p }
    connection_map [i, destination] := false;

```



```

{   => connection_map [m, p] = false   }
process : call_process [m]
    ASSERT VC

```

2.2.1.2.2.3.1.2

```

path condition : trunk [m] ? onhook,
    { caller terminates the call }
process : terminal_process [m]
    trunk [m] ? onhook;
    terminal_process [p] ! sv_onhook;
process : terminal_process [p]
    terminal_process [m] ? sv_onhook;
    trunk [p] ! onhook;
process : terminal_process [m]
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process [m] ? call_terminated;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [p] := idle;
    network_process ! remove_connection (p);
process : network_process
    call_process [m] ? remove_connection (destination);
    {   => destination = p }
    connection_map [i, destination] := false;
    {   => connection_map [m, p] = false   }
process : call_process [m]
    ASSERT VC

```

2.2.1.2.2.3.1.3

```

path condition : trunk [p] ? onhook
    { called party terminates the call }
process : terminal_process [p]
    trunk [p] ? onhook;
    terminal_process [m] ! sv_onhook;
process : terminal_process [m]
    terminal_process [p] ? sv_onhook;
    trunk [m] ! onhook;
process : terminal_process [p]
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process [p] ? call_terminated;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [p] := idle;
    network_process ! remove_connection (p);
process : network_process
    call_process [m] ? remove_connection (destination);
    {   => destination = p }
    connection_map [i, destination] := false;
    {   => connection_map [m, p] = false   }
process : call_process [m]
    ASSERT VC

```

2.2.1.2.2.3.2

```

path condition : trunk [m] ? onhook
  { caller terminates the call }
process : terminal_process [m]
  trunk [m] ? onhook;
  terminal_process [p] ! sv_onhook;
process : terminal_process [p]
  terminal_process [m] ? sv_onhook;
  trunk [p] ! onhook;
process : terminal_process [m]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [m] ? call_terminated;
  stop_billing (m);
  trunk_state [m] := idle;
  trunk_state [p] := idle;
  network_process ! remove_connection (p);
process : network_process
  call_process [m] ? remove_connection (destination);
  { => destination = p }
  connection_map [i, destination] := false;
  { => connection_map [m, p] = false }
process : call_process [m]
  ASSERT VC

```

2.2.1.2.2.3.3

```

path condition : trunk [p] ? onhook
  { called party terminates the call }
process : terminal_process [p]
  trunk [p] ? onhook;
  terminal_process [m] ! sv_onhook;
process : terminal_process [m]
  terminal_process [p] ? sv_onhook;
  trunk [m] ! onhook;
process : terminal_process [p]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [p] ? call_terminated;
  stop_billing (m);
  trunk_state [m] := idle;
  trunk_state [p] := idle;
  network_process ! remove_connection (p);
process : network_process
  call_process [m] ? remove_connection (destination);
  { => destination = p }
  connection_map [i, destination] := false;
  { => connection_map [m, p] = false }
process : call_process [m]
  ASSERT VC

```

2.2.2

```

path condition : trunk [n] ? wink
  { seizes trunk [n] successfully }
process : terminal_process [n]

```

```

    trunk [n] ? wink;
    call_process [m] ! wink;
process : call_process [m]
    terminal_process [n] ? wink;
    network_process ! make_connection (n);
process : network_process
    call_process [m] ? make_connection (destination);
    connection_map [i, destination] := true;
    { => connection_map [m, n] = true }
process : call_process [m]
    connected := true;
    terminal_process [n] ! digits (da);
process : terminal_process [n]
    call_process [m] ? digits (da);
    trunk [n] ! digits (da);
process : call_process [m]
    terminal_process [m] ! start_supervision;
process : terminal_process [m]
    call_process [m] ? start_supervision;
    fail_number := 0;
process : call_process [m]
    terminal_process [n] ! start_supervision;
process : terminal_process [n]
    call_process [m] ? start_supervision;
    fail_number := 0;

```

2.2.2.1

```

path condition : { integrity_failure }
process : fault_generator [m, n]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, n] ? failure;
    { if fail_number = 0 -> }
    switch_network_plane;
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [m] ? integrity_failure;
    log_integrity_failure;
process : fault_generator [m, n]
    terminal_process [n] ! failure;
process : terminal_process [n]
    fault_generator [m, n] ? failure;
    { if fail_number = 0 -> }
    switch_network_plane;
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [n] ? integrity_failure;
    log_integrity_failure;

```

2.2.2.1.1

```

path condition : { integrity_failure }
process : fault_generator [m, n]

```

```

    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, n] ? failure;
{   if fail_number <> 0 ->   }
    trunk [m] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [m] ? integrity_loss;
process : fault_generator [m, n]
    terminal_process [n] ! failure;
process : terminal_process [n]
    fault_generator [m, n] ? failure;
{   if fail_number <> 0 ->   }
    trunk [n] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [n] ? integrity_loss;
    trunk_state [m] := idle;
    trunk_state [n] := idle;
    network_process ! remove_connection (n);
process : network_process
    call_process [m] ? remove_connection (destination);
{   => destination = n }
    connection_map [i, destination] := false;
{   => connection_map [m, n] = false   }
process : call_process [m]
    ASSERT VC

```

2.2.2.1.2

```

path condition : trunk [m] ? onhook
{ caller hangs up before the call has been answered }
process : terminal_process [m]
    trunk [m] ? onhook;
    terminal_process [n] ! sv_onhook;
process : terminal_process [n]
    terminal_process [m] ? sv_onhook;
    trunk [n] ! onhook;
process : terminal_process [m]
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process [m] ? call_terminated;
    trunk_state [m] := idle;
    trunk_state [n] := idle;
    network_process ! remove_connection (n);
process : network_process
    call_process [m] ? remove_connection (destination);
{   => destination = n }
    connection_map [i, destination] := false;
{   => connection_map [m, n] = false   }
process : call_process [m]
    ASSERT VC

```

2.2.2.1.3

```

path condition : trunk [n] ? offhook

```

```

    { called party answers }
process : terminal_process [n]
    trunk [n] ? offhook;
    terminal_process [m] ! sv_offhook;
process : terminal_process [m]
    terminal_process [n] ? sv_offhook;
    trunk [m] ! offhook;
process : terminal_process [n]
    call_process [m] ! answered;
process : call_process [m]
    terminal_process [n] ? answered;
    start_billing (m);

```

2.2.2.1.3.1

```

path condition : { integrity failure }
process : fault_generator [m, n]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, n] ? failure;
    { if fail_number <> 0 -> }
    trunk [m] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [m] ? integrity_loss;
process : fault_generator [m, n]
    terminal_process [n] ! failure;
process : terminal_process [n]
    fault_generator [m, n] ? failure;
    { if fail_number <> 0 -> }
    trunk [n] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [n] ? integrity_loss;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [n] := idle;
    network_process ! remove_connection (n);
process : network_process
    call_process [m] ? remove_connection (destination);
    { => destination = n }
    connection_map [i, destination] := false;
    { => connection_map [m, n] = false }
process : call_process [m]
    ASSERT VC

```

2.2.2.1.3.2

```

path condition : trunk [m] ? onhook
    { caller terminates the call }
process : terminal_process [m]
    trunk [m] ? onhook;
    terminal_process [n] ! sv_onhook;
process : terminal_process [n]
    terminal_process [m] ? sv_onhook;
    trunk [n] ! onhook;

```

```

process : terminal_process [m]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [m] ? call_terminated;
  stop_billing (m);
  trunk_state [m] := idle;
  trunk_state [n] := idle;
  network_process ! remove_connection (n);
process : network_process
  call_process [m] ? remove_connection (destination);
  ( => destination = n )
  connection_map [i, destination] := false;
  ( => connection_map [m, n] = false )
process : call_process [m]
  ASSERT VC

```

2.2.2.1.3.3

```

path condition : trunk [n] ? onhook
  ( called party terminates the call )
process : terminal_process [n]
  trunk [n] ? onhook;
  terminal_process [m] ! sv_onhook;
process : terminal_process [m]
  terminal_process [n] ? sv_onhook;
  trunk [m] ! onhook;
process : terminal_process [n]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [n] ? call_terminated;
  stop_billing (m);
  trunk_state [m] := idle;
  trunk_state [n] := idle;
  network_process ! remove_connection (n);
process : network_process
  call_process [m] ? remove_connection (destination);
  ( => destination = n )
  connection_map [i, destination] := false;
  ( => connection_map [m, n] = false )
process : call_process [m]
  ASSERT VC

```

2.2.2.2

```

path condition : trunk [m] ? onhook
  ( caller hangs up before the call has been answered )
process : terminal_process [m]
  trunk [m] ? onhook;
  terminal_process [n] ! sv_onhook;
process : terminal_process [n]
  terminal_process [m] ? sv_onhook;
  trunk [n] ! onhook;
process : terminal_process [m]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [m] ? call_terminated;

```

```

        trunk_state [m] := idle;
        trunk_state [n] := idle;
        network_process ! remove_connection (n);
process : network_process
    call_process [m] ? remove_connection (destination);
{   => destination = n }
    connection_map [i, destination] := false;
{   => connection_map [m, n] = false   }
process : call_process [m]
    ASSERT VC

```

2.2.2.3

```

path condition : trunk [n] ? offhook
{   called party answers   }
process : terminal_process [n]
    trunk [n] ? offhook;
    terminal_process [m] ! sv_offhook;
process : terminal_process [m]
    terminal_process [n] ? sv_offhook;
    trunk [m] ! offhook;
process : terminal_process [n]
    call_process [m] ! answered;
process : call_process [m]
    terminal_process [n] ? answered;
    start_billing (m);

```

2.2.2.3.1

```

path condition : { integrity failure }
process : fault_generator [m, n]
    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, n] ? failure;
{   if fail_number = 0 ->   }
    switch_network_plane;
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [m] ? integrity_failure;
    log_integrity_failure;
process : fault_generator [m, n]
    terminal_process [n] ! failure;
process : terminal_process [n]
    fault_generator [m, n] ? failure;
{   if fail_number = 0 ->   }
    switch_network_plane;
    fail_number := 1;
    call_process [m] ! integrity_failure;
process : call_process [m]
    terminal_process [n] ? integrity_failure;
    log_integrity_failure;

```

2.2.2.3.1.1

```

path condition : { integrity loss }
process : fault_generator [m, n]

```

```

    terminal_process [m] ! failure;
process : terminal_process [m]
    fault_generator [m, n] ? failure;
( if fail_number <> 0 -> )
    trunk [m] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [m] ? integrity_loss;
process : fault_generator [m, n]
    terminal_process [n] ! failure;
process : terminal_process [n]
    fault_generator [m, n] ? failure;
( if fail_number <> 0 -> )
    trunk [n] ! onhook;
    call_process [m] ! integrity_loss;
process : call_process [m]
    terminal_process [n] ? integrity_loss;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [n] := idle;
    network_process ! remove_connection (n);
process : network_process
    call_process [m] ? remove_connection (destination);
( => destination = n )
    connection_map [i, destination] := false;
( => connection_map [m, n] = false )
process : call_process [m]
    ASSERT VC

```

2.2.2.3.1.2

```

path condition : trunk [m] ? onhook
( caller terminates the call )
process : terminal_process [m]
    trunk [m] ? onhook;
    terminal_process [n] ! sv_onhook;
process : terminal_process [n]
    terminal_process [m] ? sv_onhook;
    trunk [n] ! onhook;
process : terminal_process [m]
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process [m] ? call_terminated;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [n] := idle;
    network_process ! remove_connection (n);
process : network_process
    call_process [m] ? remove_connection (destination);
( => destination = n )
    connection_map [i, destination] := false;
( => connection_map [m, n] = false )
process : call_process [m]
    ASSERT VC

```


2.2.2.3.1.3

```

path condition : trunk [n] ? onhook
  ( called party terminates the call )
process : terminal_process [n]
  trunk [n] ? onhook;
  terminal_process [m] ! sv_onhook;
process : terminal_process [m]
  terminal_process [n] ? sv_onhook;
  trunk [m] ! onhook;
process : terminal_process [n]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [n] ? call_terminated;
  stop_billing (m);
  trunk_state [m] := idle;
  trunk_state [n] := idle;
  network_process ! remove_connection (n);
process : network_process
  call_process [m] ? remove_connection (destination);
{
  => destination = n
  connection_map [i, destination] := false;
{
  => connection_map [m, n] = false
}
process : call_process [m]
  ASSERT VC

```

2.2.2.3.2

```

path condition : trunk [m] ? onhook
  ( caller terminates the call )
process : terminal_process [m]
  trunk [m] ? onhook;
  terminal_process [n] ! sv_onhook;
process : terminal_process [n]
  terminal_process [m] ? sv_onhook;
  trunk [n] ! onhook;
process : terminal_process [m]
  call_process [m] ! call_terminated;
process : call_process [m]
  terminal_process [m] ? call_terminated;
  stop_billing (m);
  trunk_state [m] := idle;
  trunk_state [n] := idle;
  network_process ! remove_connection (n);
process : network_process
  call_process [m] ? remove_connection (destination);
{
  => destination = n
  connection_map [i, destination] := false;
{
  => connection_map [m, n] = false
}
process : call_process [m]
  ASSERT VC

```

2.2.2.3.3

```

path condition : trunk [n] ? onhook
  ( called party terminates the call )
process : terminal_process [n]

```

```

    trunk [h] ? onhook;
    terminal_process [m] ! sv_onhook;
process : terminal_process [m]
    terminal_process [n] ? sv_onhook;
    trunk [m] ! onhook;
process : terminal_process [n]
    call_process [m] ! call_terminated;
process : call_process [m]
    terminal_process [n] ? call_terminated;
    stop_billing (m);
    trunk_state [m] := idle;
    trunk_state [n] := idle;
    network_process ! remove_connection (n);
process : network_process
    call_process [m] ? remove_connection (destination);
{   => destination = n }
    connection_map [i, destination] := false;
{   => connection_map [m, n] = false   }
process : call_process [m]
    ASSERT VC

```

REFERENCES

- [APT80] Apt, K.R., Frances, N. and de Roever, W.P., "A Proof System for Communicating Sequential Processes", ACM TOPLAS, Vol.2, No.3, Jul. 1980, pp.359-385.
- [ASHC75] Ashcroft, E.A., "Proving Assertions about Parallel Programs", Journal of Computer & System Sciences, Vol.10, 1975, pp.110-135.
- [BART69] Bartlett, K.A., "A Note on Reliable Full-Duplex Transmission Over Half-Duplex Links", Comm. ACM, Vol.12, No.5, May 1969, pp.260-261.
- [BETH82] Bethelot, G. and Terrat, R., "Petri Nets Theory for the Correctness of Protocols", IEEE Trans. Comm., Vol.COM-30, No.12, Dec. 1982, pp.2497-2505.
- [BIRK72] Birke, D.M., "State Transition Programming Techniques and Their Use in Producing Teleprocessing Device-Control Programs", IEEE Trans. Comm., Vol.COM-20, No.3, Jun. 1972, pp.569-575.
- [BIRM76] Birman, A. and Joyner, W.H., "A Problem Reduction Approach to Proving Simulation between Programs", IEEE Trans. Soft. Eng., Vol.SE-2, No.2, Jun. 1976, pp.87-96.
- [BIRM74] Birman, A., "On Proving Correctness of Microprograms", IBM J. Res. Develop., May 1974, pp.254-260.
- [BOCH80] Bochmann, G.V. and Sunshine, C., "Formal Methods in Communication Protocol Design", IEEE Trans. Comm. Vol. COM-28, No.4, Apr. 1980, pp.643-650.

- [BOCH78] Bochmann, G.V., "Finite State Description of Communication Protocols", Computer Networks, Vol.2, Oct. 1978, pp.361-372.
- [BOCH77] Bochmann, G.V. and Gecsei, J., "A Unified Method for the Specification and Verification of Protocols", Proc. of IFIP Congress, Toronto, Canada, Aug. 1977, pp.229-234.
- [BOUR80] Bourne, J., Perry, J. and Workman, R., "The DMS-100 Distributed Control System : Fault Tolerance Digital Style", Telesis, Vol.7, No.4, 1980, pp.6-12.
- [BRAN78] Brand, D. and Joyner, W.H., "Verification of Protocols Using Symbolic Execution", Computer Network Protocols, 1978.
- [CARL80] Carlson, D.E., "Bit-Oriented Data Link Control Procedures", IEEE Trans. Comm., Vol.COM-28, No.4, Apr. 1980, pp.455-467.
- [CASH80] Cashin, P.M., "Inter-Process Communications", Study Report 8005014, Bell-Northern Research, Jun. 1980.
- [CCIT76] CCITT, "Functional Specification and Descriptive Language (SDL)", Orange Book, Vol.VI.4-Rec. Z.101, 1976.
- [CHAN79] Chandy, K.M. and Misra J., "Deadlock Absence Proofs for Networks of Communicating Processes", Info. Proc. Lett., Vol.9, No.4, Nov. 1979, pp.185-189.
- [CHEN83] Chen, B.S. and Yeh, R.T., "Formal Specification and Verification of Distributed Systems", IEEE Trans. Soft. Eng., Vol.SE-9, No.6, Nov. 1983, pp.710-721.

- [CLIN73] Clint, M., "Program Proving : Coroutines", Acta Informatica, Vol.2, 1973, pp.50-63.
- [DANN82] Dannenberg, R.B. and Ernst, G.W., "Formal Program Verification Using Symbolic Execution", IEEE Trans. Soft. Eng., Vol.SE-8, No.1, Jan. 1982, pp.43-52.
- [DANT78] Danthine, A. and Bremer, J., "Modeling and Verification of End-to-End Transport Protocols", Computer Networks, Vol.2, Oct. 1978, pp.381-395.
- [DAY79] Day, J.D., "A Bibliography on the Formal Specification and Verification of Computer Network Protocols", Comput. Comm. Rev., Vol.9, Oct. 1979.
- [DIJK76] Dijkstra, E.W., A Discipline of Programming, New Jersey: Prentice Hall, 1976.
- [DIJK75] Dijkstra, E.W., "Guarded Commands, Non-Determinacy and Formal Derivation of Programs", Comm. ACM, Vol.18, No.8, Aug. 1975, pp.453-457.
- [DIJK71] Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes", Acta Informatica, Vol.1, 1971, pp.115-138.
- [FLON81] Flon, L. and Suzuki, N., "The Total Correctness of Parallel Programs", SIAM J. Comput., Vol.10, No.2, May 1981, pp.227-246.
- [FLOY67] Floyd, R.W., "Assigning Meaning to Programs", Proc. of Symposium Applied Math., Vol.COM-28, No.4, Apr. 1980, pp.455-467.
- [GRIE76] Gries, D., "An Illustration of Current Ideas on the Derivation of Correct Proofs and Correct Programs", IEEE Trans. Soft. Eng., Vol.SE-2, No.4, Dec. 1976,

pp.238-244.

- [HANT76] Hantler, S.L. and King, J.C., "An Introduction to Proving the Correctness of Programs", ACM Comput. Surveys, Vol.8, No.3, Sep. 1976, pp.331-353.
- [HARA78] Harangozo, J., "Protocol Definition with Formal Grammars", Computer Network Protocol Symposium, Liege, Belgium, 1978.
- [HOAR78] Hoare, C.A.R., "Communicating Sequential Processes", Comm. ACM, Vol.21, No.8, Aug. 1978, pp.666-677.
- [HOAR71] Hoare, C.A.R., "Proof of a Program : FIND", Comm. ACM, Vol.14, No.1, Jan. 1971, pp.39-45.
- [HOWA76] Howard, J.H., "Proving Monitors", Comm. ACM, Vol.19, No.5, May 1976, pp.273-279.
- [KELL76] Keller, R.M., "Formal Verification of Parallel Programs", Comm. ACM, Vol.19, No.7, Jul. 1976, pp.371-384.
- [KING76] King, J.C., "Symbolic Execution and Program Testing", Comm. ACM, Vol.19, No.7, Jul. 1976, pp.385-394.
- [KOHL81] Kohler, W.H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", ACM Computing Surveys, Vol.13, No.2, Jun. 1981, pp.149-183.
- [LAMP80] Lamport, L., "The 'Hoare Logic' of Concurrent Programs", Acta Informatica, Vol.14, No.21, 1980, pp.21-37.
- [MERL76] Merlin, P.M. and Farber, D.J., "Recoverability of

Communication Protocols - Implications of a Theoretical Study", IEEE Trans. Comm., Vol.COM-24, No.9, Sep. 1976, pp.1036-1043.

[MISR81] Misra, J., Chandy, K.M., "Proofs of Networks of Processes", IEEE Trans. Soft. Eng., Vol.SE-7, No.4, Jul. 1981, pp.417-426.

[NELS83] Nelson, R.A., Haibat, L.M. and Sheridan, P.R., "Casting Petri Nets into Programs", IEEE Trans. Soft. Eng., Vol. SE-9, No.5, Sep. 1983, pp.590-602.

[OWIC82] Owicki, S. and Lamport, L., "Proving Liveness Properties of Concurrent Programs", ACM TOPLAS, Vol.4, No.3, Jul. 1982, pp.455-495.

[OWIC76a] Owicki, S. and Gries, D., "An Axiomatic Proof Technique for Parallel Programs I", Acta Informatica, Vol.6, 1976, pp.319-340.

[OWIC76b] Owicki, S. and Gries, D., "Verifying Properties of Parallel Programs : An Axiomatic Approach", Comm. ACM, Vol.19, No.5, pp.279-285.

[PENN82] Penney, B.K. and Williams, J.W., "The Software Architecture for a Large Telephone Switch", IEEE Trans. Comm., Vol.COM-30, No.6, Jun. 1982, pp.1369-1379.

[PENN80] Penney, B., "The DMS-100 : a Switch that Takes Care of Itself", Telesis, Vol.7, No.4, 1980, pp.41-43.

[PETE77] Peterson, J.L., "Petri Nets", ACM Comput. Surveys, Vol.9, No.3, Sep. 1977, pp.223-252.

[POST76] Postel, J.B. and Farber, D., "Graph Modelling of Computer Communications Protocols", Proc. 5th Texas

Conf. Comput. Syst., Austin, TX, Oct. 1976.

- [POUZ78] Pouzin, L. and Zimmermann, H., "A Tutorial on Protocols", Proc. of the IEEE, Vol.66, No.11, Nov. 1978, pp.1346-1370.
- [RICA81] Ricart, G and Agrawala, A.K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks", Comm. ACM, Vol.24, No.1, Jan. 1981, pp.9-17.
- [SCHU80] Schultz, G.D. et. al., "Executable Description and Validation of SNA", IEEE Trans. Comm., Vol.COM-28, No.4, Apr. 1980, pp.661-677.
- [SCHW82] Schwartz, R.L. and Melliar-Smith, P.M., "From State Machines to Temporal Logic : Specification Methods for Protocol Standards", IEEE Trans. Comm., Vol.COM-30, No.12, Dec. 1982, pp.2486-2496.
- [SEVI80] Sevia, R., "Control of Peripheral Modules", Telesis, Vol.7, No.4, 1980, pp.24-26.
- [SILB83] Silberschatz, A., "Extending CSP to Allow Dynamic Resource Management", IEEE Trans. Soft. Eng., Vol.SE-9, No.4, Jul. 1983, pp.527-531.
- [STEN76] Stenning, N.V., "A Data Transfer Protocol", Computer Networks, Vol. 1, Sep. 1976, pp.99-110.
- [SUNS79] Sunshine, C., "Formal Techniques for Protocol Specification and Verification", Computer, Sep. 1979, pp.20-29.
- [SUNS78] Sunshine, C., "Survey of Protocol Definition and Verification Techniques", Computer Networks, Vol.2, Oct. 1978, pp.346-350.

- [TALL79] Talley, D., "Basic Telephone Switching Systems", Hayden Book Company Inc., New Jersey, 2nd ed., 1979.
- [WEGN83] Wegner, P. and Smolka, S.A., "Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives", IEEE Trans. Soft. Eng., Vol.SE-9, No.4, Jan. 1983, pp.446-462.
- [WIRT71] Wirth, N., "Program Development by Stepwise Refinement", Comm. ACM, Vol.14, No.4, Apr. 1971, pp.221-227.
- [YAU83] Yau, S.S. and Caglayan, M.U., "Distributed Software System Design Representation Using Modified Petri Nets", IEEE Trans. Soft. Eng., Vol.SE-9, No.6, Nov. 1983, pp.733-745.
- [ZAFI80] Zafiropulo, P. et. al., "Towards Analysing and Synthesising Protocols", IEEE Trans. Comm., Vol. COM-28, No. 4, Apr. 1980, pp.651-660.

END

12-06-186

FIN