National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform 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.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme 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.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

Test-Sequence Generation for Distributed Systems Specified in Petri-nets
(with application to LOTOS)

by

Youwen Wu

A M.Sc. Thesis

submitted to the School of Graduate Studies and Research
in partial fulfilment of the requirements for the
Master of Computer Science Degree*

University of Ottawa
Ottawa, Ontario
Canada

September 1990

* The Master of Computer Science Program is a joint program with Carleton University, administrated by the Ottawa-Carleton Institute for Computer Science

© Youwen Wu, Ottawa, Canada, 1991

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-67999-9

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGEMENT

I am very grateful to my supervisor, Dr. To-yat Cheung, for his guidance and advice throughout my graduate studies.

I acknowledge with gratitude the financial support provided by the Telecommunications Research Institute of Ontario.

I would like to thank the Protocol Research Group for providing an excellent research environment for the entire period of my study leading to this thesis. In particular, discussions with members of the Group, especially Xinming Ye and Guoqiang Wang, have turned out to be extremely helpful.

ABSTRACT

This thesis proposes a new approach for the generation of test-sequences for indeterministic distributed systems. The system is specified in a Petri-net extended with internal and special labels so as to include the representation of indeterministic and unusual behaviours. Based on this extended model, a metric called 'degree of indeterminism' is developed for estimating the number of times a test sequence should be executed before a verdict can be made. A fairness model of conformance testing is then presented on the basis of this metric. Also, an iterative algorithm called IPNTEST is developed for generating test sequences and their degrees of indeterminism. The kth iteration of IPNTEST generates groups of test sequences which all have the same length k. Test sequences belonging to a group have the same preamble, while different groups may have different preambles. Together, the test groups cover all the bounded feasible paths of the system under test. IPNTEST has been applied to generate test sequences for LOTOS specifications. Based on IPNTEST, an automatic test case generator called LOTOS-TCG has been implemented in the Sun Workstation.

TABLE OF CONTENTS

ACKNOWLEDGEMENT i

ABSTRACT ii

TABLE OF CONTENTS iii

LIST OF FIGURES v

LIST OF TABLES vi

CHAPTER 1 INTRODUCTION AND FUNDAMENTALS 1

1.1 Preliminaries of formal description techniques for protocol specification 1

1.1.1 Basic LOTOS 2

1.1.2 Labelled transition systems 3

1.1.3 Labelled Petri-Nets 5

1.2 Motivations and contributions of the thesis 7

CHAPTER 2 REVIEW ON PETRI-NET REPRESENTATIONS AND TEST SUITE GENERATION FOR LOTOS SPECIFICATIONS 12

2.1 Introduction 12

2.2 Petri-net representations of LOTOS 14

2.2.1 Galileo nets 14

2.2.2 Petri-net representation of LOTOS by Cheung and Zhu 20

2.3 Test suite generation methods for LOTOS specifications 21

2.3.1 The theory of Canonical testers 21

2.3.2 The CO-OP method 23

2.3.4 Tripathy and Sarikaya's method 26

2.3.3 Guerauchi and Logrippo's method 26

CHAPTER 3 GENERATING TEST SEQUENCES AND THEIR DEGREES OF INDETERMINISM FROM A PETRI-NET-BASED SPECIFICATION......... 28
3.1 Introduction.. 28
3.2 The model of indeterministic Petri-nets.. 28
3.3 Generation of test sequences and their degrees of indeterminism (Algorithm IPNTEST).. 35

CHAPTER 4 APPLICATION OF ALGORITHM IPNTEST TO LOTOS (LOTOS-TCG - A TEST CASE GENERATOR FOR LOTOS).................... 42
4.1 Introduction.. 42
4.2 Functions of test case generator LOTOS-TCG... 42
4.3 Subprogram - TRANSFORMATION.. 44
4.3.1 Data structures used in subprogram TRANSFORMATION............................ 45
4.3.2 Algorithm used in subprogram TRANSFORMATION................................. 48
4.4 Subprogram - GENERATION... 51
4.4.1 Data structures used in subprogram GENERATION.................................... 52
4.4.2 Functions of subprogram GENERATION.. 53

CHAPTER 5 AN EXAMPLE (THE CLASS 0 TRANSPORT PROTOCOL).................... 55
5.1 Exemplifying the data structures and algorithm IPNTEST.................................. 55
5.2 The output of LOTOS-TCG... 61
5.3 Appendix: the LOTOS specification Handler.. 67

CHAPTER 6 SUMMARY AND FUTURE WORKS.. 69

REFERENCES... 72

LIST OF FIGURES

Figure 3.1 Different types of markings of a marking graph.. 30
Figure 3.2 An indeterministic Petri-net..34
Figure 3.3 Marking tree of the indeterministic Petri-net of Figure 3.2........................... 35
Figure 4.1 A LOTOS-based graphical environment for distributed computing research at University of Ottawa.. 43
Figure 4.2 Functional diagram of the test case generator LOTOS-TCG.......................... 44
Figure 4.3 Functional diagram of the subprogram TRANSFORMATION.........................45
Figure 5.1 Tree of internal node table for the connection Phase of the Class 0 Transport Protocol...57
Figure 5.2 Binary tree of the Connection Phase of the Class 0 Transport Protocol...............58
Figure 5.3 Indeterministic Petri-net representation of LOTOS specification for the Connection Phase of the Class 0 Transport Protocol...58
Figure 5.4 Marking subtrees of the indeterministic Petri-net of Figure 5.3........................60
Figure 5.5 Indeterministic Petri-net representation of LOTOS specification for the Class 0 Transport Protocol...66

LIST OF TABLES

Table 1.1 Syntax of basic LOTOS 3

Table 1.2 Notations for labelled transition systems........ 5

Table 2.1 Galileo net representation for LOTOS constructs........ 18

Table 2.2 Extensions in Cheung's Pet t representation of LOTOS construcs........ 20

Table 4.1 The eight fields of a row of the internal node table........ 47

Table 4.2 The five fields of a node of the binary tree........ 47

Table 4.3 Twelve fields of an element of gnet........ 48

Table 4.4 Data structures used in an iteration........ 52

Table 4.5 Structure of a node of a marking_subtree........ 53

Table 5.1 Matrix representation of the Connection Phase of the Class 0 Transport Protocol 59

Table 5.2 Markings under consideration........ 60

Table 5.3 Results of applying IPNTEST to the Connection Phase of the Class 0 Transport Protocol........ 61

Chapter 1
INTRODUCTION AND FUNDAMENTALS

In recent years, communications protocols have become more and more complex because of the growing demands for more services to be provided by the networks. In response to this trend, the International Standardization Organization (ISO) has proposed a number of formal description techniques (FDTs) designed for the specification of the behaviours of communicating entities. The main objective of FDTs is to allow the production of unambiguous specifications and to provide a well-defined basis for the validation of the design and for the testing of their conformance with the specifications.

Most of the existing results of protocol testing methods are based on FDTs for describing deterministic systems. However, because of the different levels of details in a description, a specification at a certain level may exhibit indeterministic behaviours of the system. This implies that if a test sequence generated from a specification with indeterminism is applied to the IUT under the same condition but at different times, the IUT may end at different states or respond differently. Therefore, a test sequence may have a 'no-conclusion' verdict after testing. In this thesis, a metric for measuring the degree of indeterminism of test sequences and a new algorithm (and its implementation) for generating test sequences and their degrees of indeterminism are proposed for attacking these problems. The algorithm has been applied to the Language fOr Temporal Ordering Specifications (LOTOS).

In this chapter, the terminology and preliminaries for our methods are given. The motivation and outline of the thesis are also described.

1.1 PRELIMINARIES OF FORMAL DESCRIPTION TECHNIQUES FOR PROTOCOL SPECIFICATION

In the past, most specification methods are based on finite state machines [HOP79] and Petri-nets [PET81]. Recently, an algebraic language LOTOS has been standardized as a language for

specifying distributed systems under the Open Systems Interconnection architecture (OSI). In this thesis, our major research results are applied to LOTOS through transforming a LOTOS specification to a labelled Petri-net, a special case of labelled transition systems [HOP79]. To provide a background for such developments, the preliminaries of basic LOTOS, labelled transition systems and labelled Petri-nets are described in this section.

1.1.1 Basic LOTOS

LOTOS (Language fOr Temporal Ordering Specification) was developed by ISO/TC97/SC21/WG16-1/FDT/Subgroup C mostly in the period 1981-1986 for describing distributed systems. It is now an ISO standard [ISO8807].

A full LOTOS specification has essentially two parts: the control part which consists of processes and the data part which consists of data structures [BOL87]. The control part has a foundation based on a modification of Milner's Calculus of Communicating Systems (CCS) [MIL80] and Hoare's Communicating Sequential Processes (CSP) [HOA85]. The processes describe the control flow of the system by means of interactions. The data structures are based on the abstract data types ACT ONE [EHR85]. Since the data part of the full LOTOS specifications is not the concern of this thesis, it will not be discussed any further. Interested readers are referred to [BOL87].

In this thesis, we consider only basic LOTOS, a subset of the full LOTOS constructs, which can be used to specify the control flow part only. In basic LOTOS, processes interact with one another by pure synchronizations, i.e., interprocess interactions without value passing. It includes all the LOTOS process operations except those which must include data such as 'par', 'let', etc. The syntax of basic LOTOS is summarized in Table 1.1. The details of the syntax and semantics of full LOTOS are given in [BOL87].

Notations used in Table 1.1:

B, B_1, B_2... represent processes. a, a_1, a_2 ... represent external actions.

g, g_1, g_2 ... represent gates.

name	behaviour expression	
inaction	stop	
termination	exit	
action-prefix	a; B or i; B	(a is an external action,
		i is an internal action)
choice	B1 [] B2	
parallel-composition	B1 | [a1, ... , an] | B2	(general format)
	B1 ||| B2	(pure interleaving)
	B1 || B2	(full synchronization)
enabling	B1 >> B2	
disabling	B1 [> B2	
hiding	B\ [g1, ... , gn]	

Table 1.1. Syntax of basic LOTOS

LOTOS specifications can be converted to labelled transition systems or Petri-nets [GAR90, MAR89, WEZ89]. These two representations are described below. Some transformation techniques from LOTOS specification to Perti-nets are described in Chapter 2.

1.1.2 Labelled Transition Systems

In this section, the concept of labelled transition systems is described. They can be considered as extensions of finite state machines [HOP79].

Definition (*labelled transition system*)

A *labelled transition system* TS is a 4-tuple $< S, L, T, s_0 >$, where

S is a non-empty set of states;

L is a set of labels representing actions;

T: a transition in S x L x S; and

$s_0 \in$ S is the initial state of TS.

Some existing theory and methods for conformance testing based on labelled transition systems are given in [BRI86, BRI88, WEZ89]. The notations for labelled transition systems given in Table 1.2 are used in Chapter 2 for describing the theory and method of canonical testers. In particular, three types of actions are mentioned: Internal actions are unobservable and usually depend on the implementation of the specification. External actions are observable. They usually represent the interactions among processes. Special actions play a special role in a specification. For example, when a LOTOS specification is transformed to a labelled transition system, 'exit' (successful termination) can be represented by a special action δ [BRI88, WEZ89].

Notation	Meaning
B, P, Q, R...	states in S
$\mu_1, \mu_2, \ldots, \mu_n$	individual actions in L
i	internal action in L
a, b, c, ...	external actions in L
δ	a special action, it reaches a termination state
L_c	$L_c = L - \{i\}$, called the set of actions
ε	empty sequence
$\sigma, \alpha, \beta, \gamma, \ldots$	finite sequence from L_c, called a external path
L_c^*	$L_c^* = \{ \varepsilon, \sigma, \alpha, \beta, \gamma, \ldots\}$
$P - \mu -> Q$.	$(P, \mu, Q) \in T$
$P - \mu_1, \ldots, \mu_n -> Q$	$\exists P_j (1 \leq j \leq n-1)$, s.t. $P - \mu_1 -> P_1 - \mu_2 -> \ldots\ P_{n-1} - \mu_n -> Q$

$P - \mu_1, ..., \mu_n ->$	$\exists Q$, s.t. $P - \mu_1, \mu_2, ..., \mu_n -> Q$
$P -/ \mu_1, ..., \mu_n ->$	not $P - \mu_1, \mu_2, ..., \mu_n ->$
$P = \varepsilon => Q$	$\exists n\ (n \geq 1)$, s.t. $P - i^n -> Q$
$P = a => Q$	$\exists P_1, P_2$, s.t. $P = \varepsilon => P_1 - a -> P_2 = \varepsilon => Q$
$P = a_1, ..., a_n => Q$	$\exists P_j (1 \leq j \leq n-1)$, s.t. $P = a_1 => P_1 = a_2 => ...\ P_{n-1} = a_n => Q$
$P = \sigma =>$	$\exists Q$, s.t. $P = \sigma => Q$
$P \neq \sigma =>$	not $P = \sigma => Q$
Tr(P)	$\{\sigma \mid \sigma \in L_c^*$ and $P = \sigma =>\}$
out(P)	$\{a \mid a \in L_c$ and $P = a =>\}$
$P = a_1, ..., a_n -> Q$	$\exists P_1$, s.t. $P = a_1, a_2, ..., a_{n-1} => P_1 - a_n -> Q$

Table 1.2. Notations for labelled transition systems

In Table 1.2, two types of notations have been used between actions: -> and =>. The difference between the two notations is that the notation => omits all internal actions, the notation -> does not omit any action (nether internal nor external actions). For example, P => Q may represent P -> i -> Q. Tr(P) is the set of external action sequences. out(P) is the set of external actions which can be reached through internal actions (≥ 0) by state P.

1.1.3 Labelled Petri-Nets

A labelled Petri-net is a formal method for expressing the control flow of a system. It is particularly powerful for those systems which exhibit concurrent behaviours [PET81]. The new algorithm proposed in this thesis for generating test sequences is based on labelled Petri-nets.

Definition (*labelled Petri-net*)

A *labelled Petri-net* PN is a 6-tuple $< P, T, F, M_0, L, h >$, where

P is a nonempty set of places;

T is a set of transitions, s.t. $P \cap T = \phi$;

F is a set of arcs, $F \subseteq (P \times T) \cup (T \times P)$;

M_0 is the initial marking;

L is a set of labels representing actions; and

h is a mapping: T -> L.

For $x \in P \cup T$, $.x = \{y \mid (y, x) \in F\}$ is called the preset of x, and $x. = \{y \mid (x, y) \in F\}$ is called the postset of x.

A labelled Petri-net consists of two types of nodes called places and transitions, usually denoted as circles and bars, respectively. Normally, places and transitions play different roles. A transition represents an action with the places in its preset and postset representing the preconditions and postconditions of the action, respectively. The fulfillment of a condition specified by a place is indicated by the existence of a token in that place. A state of the system is expressed in terms of a vector M called marking, whose p^{th} component M(p) denotes the number of tokens in place p. The state space of a labelled Petri-net is the set of all markings. The initial marking denotes the initial system state. The firing rules of Petri-nets are given in the following definition.

Definition (*Firing Rules*)

A transition t is said to be firable at a marking M, if for every $p \in .t$: $M(p) \geq 1$.

The firing of a transition makes the net evolve from one state to another. If the transition t is fired at a marking M, a new marking M′ is reached, where:

$$M'(p) = \begin{cases} M(p) - 1 \text{ if } p \in .t - t.; \\ M(p) + 1 \text{ if } p \in t. - .t; \\ M(p) \quad \text{otherwise} \end{cases}$$

Mathematically, a labelled Petri-net can be represented by two matrices D^- and D^+. Each matrix has *m* rows (one for each place) and *n* columns (one for each transition). $D^-[i, j] = 1$ means

that place i belongs to the preset of transition j. $D^-[i, j] = 0$ means that place i does not belong to the preset of transition j. $D^+[i, j] = 1$ means that place i belongs to the postset of transition j. $D^+[i, j] = 0$ means that place i does not belong to the postset of transition j. The matrix representations of labelled Petri-nets are mainly applied in the implementation of algorithm IPNTEST of this thesis.

In the matrix representation, suppose that a vector X of firing transitions lead a Petri-net from marking M to M'. The new marking M' is given by the following matrix equation:

$$M' = M + X \cdot D \qquad \text{where } D = D^+ - D^- .$$

The marking M' is said to be reachable from the marking M if M' is obtained by firing a sequence of transitions starting from M. The reachability tree (or marking graph, marking tree) consists of those states reachable from the initial state. It represents the behaviours of the specified system.

There exist several means of transforming a LOTOS specification to a labelled Petri-net. The relevant details are given in Chapter 2. For the rest of the thesis, 'Petri-nets' means 'labelled Petri-nets'.

1.2 MOTIVATIONS AND CONTRIBUTIONS OF THE THESIS

The purpose of conformance testing is to determine whether or not an IUT (implementation under test) behaves according to the specification. A lot of research efforts on conformance testing is devoted to the derivation of test suites for such a purpose.

In the literature, a test suite has appeared in at least two formats: a canonical tester and a set of test sequences.

A canonical tester [BRI88] is the specification of another system derived from the specification of the SUT (system under test). It is intended to be executed concurrently with the IUT. For example, in LOTOS, the canonical tester and IUT are combined and executed as a single

system under the full synchronization operation (||). Each concurrent execution tests one of the external paths of the specification of the SUT. A conformance test requires at least as many executions as the number of external paths of the specification. Also, since there is no way to ensure that the same external path will not be repeatedly executed in different executions, it may require a tremendous number of the executions in order to exhaust all the external paths of the specification. Hence, it may be concluded that this approach is not practical for conformance testing. It can be used a a theoretical basis for other developments.

Except for the canonical tester mentioned above, all test suites reported in the literature appear in the format of a set or sets of test sequences. The methods for the generation and verdict-imposition of test sequences are greatly influenced by two important characteristics of distributed systems, namely, concurrency and indeterminism.

Concurrency of actions means that more than one of them can occur simultaneously. In most of the traditional methods, concurrent actions are treated as a set of all possible interleavings of sequential actions. For example, the simultaneous occurrence of actions a and b is expressed either as 'a occurs before b' or as 'b occurs before a'. A different approach is considered in [POM85] for handling concurrent actions represented by a Petri-net. In that approach, the concurrent actions (firable transition) at a marking are represented by a single set without distinguishing their order of executions. In this thesis, the traditional method is adopted for handling concurrency.

Indeterminism is the phenomenon in which the next state or action of the system is unpredictable. This means that, if a sequence of external actions is applied to an IUT under the same circumstance but at different times, the IUT may react differently, reaching different states or responding with different external actions. In a specification, indeterminism is expressed in terms of either internal actions or two or more identical external actions initiated at the same state under the same input.

In the literature, two general approaches have been adopted for conformance testing. In both approaches, the indeterminism property is not taken into consideration in the process of test sequence generation. They differ in the way the system is specified:

(1) In the first approach, test sequences are generated from a specification without indeterminism. When such a test sequence is applied to the IUT, two kinds of verdicts [ISO9646] can be made: *pass* and *fail*. *Pass* means that the result of the test is as expected. *Fail* means that the test result is not (but should be) as expected. The methods proposed by Sidhu, Ural, et al. [SID89, URA87a, URA87b, URA88] belong to this approach.

(2) In the second approach, the original specification includes indeterminism but, for the purpose of test sequence generation, is first transformed into one without it. Test sequences are then generated from the derived specification. When such a test sequence is applied to the IUT, three kinds of verdicts [ISO9646] can be made: *pass*, *fail* and *no-conclusion*. *Pass* and *fail* have the same meanings as above. *No-conclusion* means that although an unexpected result (e.g., an unexpected deadlock) occurs, a *fail* conclusion cannot be made yet because when the same test sequence is tried again at the same state but at another time, the result may be as expected. Interpreted from another point of view, such a phenomenon is due to the multiplicity of the expected results (i.e., indeterminism) of a single test sequence. The problem is that it is not known how many times a test sequence should be re-applied to the IUT in order to reach a definite correct verdict. The method of Wu and Chanson [WU89] belongs to this approach.

In summary, current research in test sequence generation has two major unsolved problems (among others): (1) The indeterminism property is not taken into consideration in the process of test sequence generation. (2) Very little has been done on the 'no-conclusion issue' of verdict-imposition on test sequences [ISO9646]. While these problems are not obstacles for sequential program testing, they are important issues for indeterministic distributed systems, such as network protocols.

In this thesis, in order to solve these two problems, we propose a new theory and a new algorithm for test sequence generation. They take indeterminism into consideration and attack the *no-conclusion* issue. Following is a list of contributions of this thesis:

a) An indeterministic Petri-net model is presented by classifying the labels of the Petri-net into three different kinds: external labels, representing explicitly defined actions; internal labels, representing those unspecified actions dependent on the implementation; and special labels, representing unusual actions. Special actions play special roles, such as termination of a process, etc. In LOTOS, for example, stop (inaction) and exit (successful termination) can be represented by such labels.

b) Based on the indeterministic Petri-net model and a uniform probability distribution for resolving indeterminism, a metric called 'degree of indeterminism' is developed for estimating the number of times a test sequence should be executed in order that a verdict can be made.

c) Based on the degree of indeterminism of test sequences, a fairness model of conformance testing is proposed. In this model, an IUT is said to be conforming with the specification if every test sequence passes at least once after trying a designated 'fair' number of times. As far as we know, it is the first time that the 'fairness' concept is used in conformance testing. Though not the main objective of this thesis, this concept initiates an approach for the study of performance analysis of conformance testing for distributed systems with indeterministic behaviours.

d) The main contribution is an iterative algorithm called IPNTEST for generating test sequences and their degrees of indeterminism from the Petri-net representation of the system. We believe that this is the first algorithm for generating test sequences from specifications including the indeterminism property. The algorithm is iterative. The kth iteration generates a set of test cases. Each test case consists of test sequences which all have the same length k and the same preamble. The number of iterations and hence lengths of the test sequences can be controlled by the user.

e) This thesis includes also a substantial amount of experimental work. IPNTEST is applied to LOTOS and implemented in a test case generator LOTOS-TCG. LOTOS-TCG is a subsystem of UO-GLOTOS, a research environment for graphical LOTOS research

development at University of Ottawa. It accepts a specification in textual or graphical LOTOS and transforms it into a Galileo net [MAR89]. Test sequences and their degrees of indeterminism are then generated by a strategy similar to reachability analysis. The length of the test sequences can be controlled by the user. LOTOS-TCG has been applied to the Class 0 Transport Protocol (details see Chapter 5). The application and implementation of IPNTEST serve the following purposes: 1) To show the feasibility of our theory and Algorithm IPNTEST. 2) To provide a prototype for commercial applications. 3) To provide a tool for further research.

The rest of the thesis is organized as follows. Chapter 2 includes a survey on the Petri-net representations of LOTOS and the methods for generating test cases from LOTOS specifications. This provides the background based on which our new theory and algorithm are developed. The new algorithm IPNTEST is described in Chapter 3. In this central part of the thesis, at first, an indeterministic Petri-net model is defined; and, on the basis of the marking graph of the Petri-net, a metric called 'degree of indeterminism' is proposed. Then, the algorithm IPNTEST is described in detail. The algorithm generates test sequences together with their degrees of indeterminism. Lastly, a fairness model is presented for attacking the no-conclusion issue in conformance testing. Chapter 4 describes the application of IPNTEST to LOTOS. A system called LOTOS-TCG for generating test sequences from LOTOS specifications is implemented. LOTOS-TCG transforms a LOTOS specification to a Petri-net and generates test sequences from it by means of IPNTEST. The implementation of LOTOS-TCG is given in detail. Chapter 5 contains a complete example of the Class 0 Transport Protocol. It is used for exemplifying our Algorithm IPNTEST in Chapter 3, the data structures described in Chapter 4, and the application of LOTOS-TCG. In Chapter 6, besides summarizing and evaluating the work of our study, the foreseeable future work is outlined.

Chapter 2

REVIEW ON PETRI-NET REPRESENTATIONS AND TEST SUITE GENERATION FOR LOTOS SPECIFICATIONS

2.1 INTRODUCTION

Part of the contribution of this thesis is a new algorithm (IPNTEST) for generating test sequences for distributed systems. To apply Algorithm IPNTEST, the system has to be specified as an indeterministic Petri-net. Also, IPNTEST has been applied to LOTOS by transforming the latter to such a Petri-net. In this chapter, as background information, existing Petri-net representations and test suite generation methods for LOTOS are reviewed.

In the literature, there exist at least three Petri-net representations of LOTOS [CHE88, GAR90, MAR89]. The two Petri-net representations proposed by Cheung, et al. and Garavel, et al. [CHE88, GAR90] preserve the modular design of LOTOS, i.e., the structures and relationships among the LOTOS processes remain as part of the characteristics of their Petri-net representations. In order to do so, Cheung and Garavel represent each process by a Petri-net and use additional transitions for representing the LOTOS operators. The third representation, called Galileo Petri-net, is proposed by Marchena and Leon [MAR89]. It does not preserve the modular design of LOTOS. In this representation, all the LOTOS processes and operations are absorbed into a single Petri-net.

A lot of results have been obtained as a consequence of the tremendous efforts made in the research of test sequence generation. However, for distributed systems, most of the methods are applied to their control flow only. They do not seriously consider the indeterminism issue and are based on extended finite state machines, labelled transition systems [BRI86, BRI88, CHO78, KOH78, SID89], and the normal form of ESTELLE [FAV86, URA88]. Recently, methods taking data into consideration and making use of data flow analysis techniques begin to appear. For example, in the methods proposed by Ural [URA87a, URA87b], the generated test sequences cover all definition-and-usage pairs of a data flow diagram.

LOTOS is an algebraic FDT different from the above ones for specifying indeterministic distributed systems. As for generating test suites for LOTOS specifications, we know of only the three methods as summarized below.

1) The CO-OP method [WEZ89] is used to construct canonical testers applied to a labelled transition system. A canonical tester [BRI88] is a distributed process to be run concurrently with the IUT. Theoretically, if no unexpected deadlock occurs when a canonical tester runs concurrently with an IUT, it means that the IUT conforms with its specification. By emphasizing on the internal actions, the CO-OP method brings out the intrinsic indeterminism nature of black-box testing. However, the method is not practical, partly because it cannot easily distinguish the individuality of the test sequences.

2) The method proposed by Tripathy and Sarikaya [TRI89] is applied to a model called CHART, a labelled transition system extended with some facilities for handling data flow. However, no details are given as how to handle the data.

3) The method proposed by Gueraichi and Logrippo [GUE89] makes use of a LOTOS interpreter for generating executable paths of a LOTOS specification. Test sequences are then selected from these paths. However, the selection process is done mainly by human 'walkthrough'. No systematic or complete guidelines for this process have been reported. Naturally, no implementation of this selection step has been done.

Another test sequence generation method based on external behaviour expressions (EBE) is proposed by Wu and Chanson [WU89]. EBE ignores indeterminism in the specification. For specifications involving indeterminism, a preliminary process is required for transforming them into an EBE. Thus, basically, the EBE model generation process does not allow indeterminism in a specification. Though it has been claimed that the method can be applied to LOTOS, the rules for transforming LOTOS specifications to EBE have not been reported in the literature.

In Section 2.2, two Petri-net representations of LOTOS specifications [CHE88, MAR89] are described. In Section 2.3, the test suite generation methods [GUE89, TRI89, WEZ89] for LOTOS specifications are reviewed.

2.2 PETRI-NET REPRESENTATIONS OF LOTOS

Many Petri-net-based techniques have been used in protocol specification, verification and testing [BER89, GAR90, SAN86a, SAN86b]. In order to apply the well-known techniques of Petri-nets to LOTOS, some methods have been proposed to transform LOTOS specifications to Petri-nets. There exist at least three such methods in the literature: 1) Galileo nets [MAR89]; 2) the Petri-net representation proposed by Garaval and Sifakis [GAR90]; and 3) the Petri-net representation proposed by Cheung and Zhu [CHE88]. The first two methods are used as tools for verification purposes, i.e., Galileo Petri Net Analizer [SAN86a, SAN86b] for the traditional verification purpose and CAESAR [GAR90] mainly for verifying the synchronization gates. The third one can be used for studying the complexity and test coverage problems of LOTOS specifications. These representations, except Garaval's whose transformation rules are not formally and completely described in [GAR90], are briefly reviewed in the following subsections.

2.2.1. Galileo Nets

In this thesis, Galileo net is used as the FDT for generating test sequences since it has been more fully developed than the other methods. A Galileo net is a formal specification method for describing concurrent systems. It consists of two parts: a control part based on Petri-nets and a functionality part based on PASCAL data types. In this section, only a review on the control part is made because no details of the functionality part have been reported [MAR89, PAV89]. Hereafter, 'Galileo net' means the control part of a Galileo net only.

Basically, a Galileo net is an extension of a labelled Petri-net with the inclusion of three special labels "i", "exit" and "stop".

<u>Definition</u> (*Galileo net*)

A *Galileo net* is a 10-tuple $N = \langle P, P_i, P_o, T, F, M_0, L, f, C_i, C_o \rangle$, where

P is a nonempty set of places;

P_i is a subset of P called input places;

P_o is a subset of P called output places;

T is a set of transitions, $T \cap P = \phi$;

F is a set of arcs;

M_0 is the initial marking;

L is the labelling function;

f is the functionality (if $Po \neq \phi$, then f = "exit", else f = "noexit");

C_i is the input connectivity (C_i = Cardinal (P_i), i.e., the numbers of input places); and

C_o is the output connectivity (C_o = Cardinal (P_o), i.e. the numbers of output places).

The transformation from a LOTOS specification to a Galileo net is based on a set of rules for representing each of the LOTOS constructs as a Petri-net. These rules are shown in Table 2.1. In this table, symbol X_a with subscript a (resp., b) indicates an entity related to the LOTOS process B_a (resp., B_b), which is represented by the Galileo net N_a (resp., N_b).

LOTOS construct	Petri-net representation
stop	$P = \{p_0, p_1\}$, $P_i = \{p_0\}$, $P_o = \phi$, $T = \{t_0\}$, $F = \{(p_0, t_0), (t_0, p_1)\}$
	$M_0 = \{(p_0,1)\}$, $L = \{(t_0, stop)\}$, f = "noexit", $C_i = 1$, $C_o = 0$
exit	$P = \{p_0, p_1\}$, $P_i = \{p_0\}$, $P_o = \{p_1\}$, $T = \{t_0\}$, $F = \{(p_0, t_0), (t_0, p_1)\}$
	$M_0 = \{(p_0,1)\}$, $L = \{(t_0, exit)\}$, f = "exit", $C_i = 1$, $C_o = 1$
g; B_b	"a" or "i" is represented by N_a.
or	$P_a = \{p_0, p_1\}$, $P_{ia} = \{p_0\}$, $P_{oa} = \{p_1\}$, $T_a = \{t_0\}$, $F_a = \{(p_0, t_0), (t_0, p_1)\}$
i; B_b	$M_{0a} = \{(p_0,1)\}$, $L_a = \{(t_0, \text{"a"})\}$, where "a" is "g" or "i",
	f_a = "exit", $C_{ia} = 1$, $C_{oa} = 1$

$P = (P_a - P_{oa}) \cup (P_b - P_{ib}) \cup (P_{oa} \times P_{ib}), P_i = P_{ia}, P_o = P_{ob}, T = T_a \cup T_b,$

$F = (F_a - \{(t_a, p_a) \mid p_a \in P_{oa})\}) \cup (F_b - \{(p_b, t_a) \mid p_b \in P_{ib})\}) \cup$

$\{(t_a, (p_a, p_b)) \mid (t_a, p_a) \in F_a, (p_a, p_b) \in P\} \cup$

$\{((p_a, p_b), t_a) \mid (p_b, t_b) \in F_b, (p_a, p_b) \in P\}$

$M_0 = M_{0a}, L = (L_a \cup L_b), f = f_b, C_i = C_{ia}, C_o = C_{ob}$

B_a [] B_b

$P = (P_a - P_{ia} - P_{oa}) \cup (P_b - P_{ib} - P_{ob}) \cup (P_{ia} \times P_{ib}) \cup (P_{oa} \times P_{ob})$

$P_i = (P_{ia} \times P_{ib}), P_o = (P_{oa} \times P_{ob}), T = (T_a \cup T_b),$

$F = (F_a - \{(p_a, t_a) \mid p_a \in P_{ia})\} - \{(t_a, p_a) \mid p_a \in P_{oa})\}) \cup$

$(F_b - \{(p_b, t_b) \mid p_b \in P_{ib})\} - \{(t_b, p_b) \mid p_b \in P_{ob})\}) \cup$

$\{((p_a, p_b), t_a) \mid (p_a, t_a) \in F_a, (p_a, p_b) \in P\} \cup$

$\{((p_a, p_b), t_b) \mid (p_b, t_b) \in F_b, (p_a, p_b) \in P\} \cup$

$\{(t_a, (p_a, p_b)) \mid (t_a, p_a) \in F_a, (p_a, p_b) \in P\} \cup$

$\{(t_b, (p_a, p_b)) \mid (t_b, p_b) \in F_b, (p_a, p_b) \in P\}$

$M_0 = \{((p_a, p_b), 1) \mid (p_a \in P_{ia}), (p_b \in P_{ib})\}, L = (L_a \cup L_b)$

$f =$ if $(f_a = f_b)$ then f_a else "exit", $C_i = (C_{ia} \times C_{ib}), C_o = (C_{oa} \times C_{ob}),$

B_a |[E]| B_b

$P = (P_a \cup P_b), P_i = (P_{ia} \cup P_{ib}), P_o = (P_{oa} \cup P_{ob})$

$T = (T_a - T_{xa}) \cup (T_b - T_{xb}) \cup$

$\{(t_a, t_b) \mid (t_a \in T_{xa}), (t_b \in T_{xb}), (L(t_a) = L(t_b)), \text{possible } (t_a, t_b)\}$

where $T_{xa} = \{t_a \mid (t_a \in T_a), ((L(t_a) \in E \cup \{exit\})\}$

$T_{xb} = \{t_b \mid (t_b \in T_b), ((L(t_b) \in E \cup \{exit\})\}$

$F = (F_a - \{(p_a, t_a), (t_a, p_a) \mid (p_a \in P_a), (t_a \in T_{xa})\}) \cup$

$(F_b - \{(p_b, t_b), (t_b, p_b) \mid (p_b \in P_b), (t_b \in T_{xb})\}) \cup$

$\{(p_a, (t_a, t_b)) \mid (p_a, t_a) \in F_a, (t_a, t_b) \in T\} \cup$

$\{((t_a, t_b), p_a) \mid (t_a, p_a) \in F_a, (t_a, t_b) \in T\} \cup$

$\{(p_b, (t_a, t_b)) \mid (p_b, t_b) \in F_b, (t_a, t_b) \in T\} \cup$

$\{((t_a, t_b), p_b) \mid (t_b, p_b) \in F_b, (t_a, t_b) \in T\}$

$M_0 = M_{0a} \cup M_{0b}$

$L = (L_a - (t_a, L(t_a)) \mid (t_a \in T_{xa})) \cup (L_b - (t_b, L(t_b)) \mid (t_b \in T_{xb})) \cup$

$\{((t_a, t_b), L(t_a)) \mid (t_a, t_b) \in T\}$

$f =$ if $(f_a = f_b)$ then f_a else "noexit"

$C_i = (C_{ia} + C_{ib}), C_o = (C_{oa} + C_{ob})$

$B_a >> B_b$ $\quad P = (P_a - P_{oa}) \cup (P_b - P_{ib}) \cup (P_{oa} \times P_{ib}),\ P_i = P_{ia}, P_o = P_{ob}, T = (T_a \cup T_b)$

$F = (F_a - \{(t_a, p_a) \mid (p_a \in P_{oa})\}) \cup (F_a - \{(p_b, t_b) \mid (p_b \in P_{ib})\}) \cup$

$\{(t_a, (p_a, p_b)) \mid (t_a, p_a) \in F_a, (p_a, p_b) \in P\} \cup$

$\{((p_a, p_b), t_b) \mid (p_b, t_b) \in F_b, (p_a, p_b) \in P\}$

$M_0 = M_{0a}, L = (L_a \cup L_b), f = f_b, C_i = C_{ia}, C_o = C_{ob}$

$B_a\ [> B_b$ $\quad P = (P_a - P_{oa}) \cup (P_b - P_{ib} - P_{ob}) \cup (P_{oa} \times P_{ob}), P_i = P_{ia}, P_o = (P_{ob} \times P_{ob})$

$T = T_a \cup (T_b - T_{ib}) \cup (PT_a \times T_{ib})$ where

$T_{ib} = \{t_b \mid (t_b \in T_b), (.t_b \in P_{ib})\}$

$PT_a = \{X \mid (X \subset P(P_a)), \text{reachable } (N, X)\}$

$P(P_a)$ is the power-set of P_a.

$F = (F_a - \{(t_a, p_a) \mid (t_a \in T_a), (p_a \in P_{oa})\}) \cup$

$(F_b - \{(p_b, t_b) \mid (p_b \in P_{ib})\} - \{(t_b, p_b) \mid (t_b \in T_{ib})\} -$

$\{(t_b, p_b) \mid p_b \in P_{ob})\}) \cup \{(t_a, (p_a, p_b)) \mid (t_a, p_a) \in F_a, (p_a, p_b) \in P\} \cup$

$\{(t_b, (p_a, p_b)) \mid (t_b, p_b) \in F_b, (p_a, p_b) \in P\} \cup$

$\{(p_a, (X, t_b)) \mid (X \in PT_a), (p_a \in X), (X, t_b) \in T\} \cup$

$\{((X, t_b), p_a) \mid (X \in PT_a), (t_b \in T_{ib}), (t_b, p_b) \in F_b\}$

$M_0 = M_{0a}, L = (L_a \cup L_b), f =$ if $(f_a = f_b)$ then f_a else "exit"

$C_i = C_{ia}, C_o = (C_{oa} \times C_{ob})$

Hide E in A: Hiding makes the observable actions in the set E which appear in the behaviour expression A become unobservable. In particular, they become unavailable for synchronization with other processes. If N_a is the Galileo net of A, the resulting net N = (hide E in N_a) is the same as N_a with a new labelling function L, which is defined as:

$L = (t_a, i)$ where $(t_a \in T_a)$ and $(t_a \in E)$

$L = (t_a, L(N_a))$ otherwise

Table 2.1 Galileo net representation for LOTOS constructs

There are two kinds of recursions in LOTOS: simple recursion, where a process invokes itself; and composed recursion, where a set of processes invoke one another. In a Galileo net, recursions must be well-formed. In a well-formed recursion, all processes should be preceded by the prefix operator ";" . Infinite behaviours arising from recursions are denoted in a Galileo net by cycles. Following are the transformation rules for these two kinds of recursions.

(a) Simple Recursion: Process A := ..., A endproc (* A *)

Let N_a the equivalent net of the process A without recursion, and $P_x = \{p_a | (p_a \in P_{fa}), (t_a \in .p_a), (L(t_a) \neq \text{"exit"}), (L(t_a) \neq \text{"stop"})\}$ is the output recursive place set. The recursive net is defined as:

$F = (P_a - P_x), P_i = P_{ia}, P_o = (P_{oa} - P_x)$

$T = T_a$

$F = (F_a - \{(t_a, p_a) \mid (t_a, p_a) \in F_a, (p_a \in P_x)\}) \cup \{(t_a, p_a) \mid (t_a \in .P_x), (p_a \in P_{ia})\}$

$M_0 = M_{0a}, L = L_a$

$f = f_a, C_i = C_{ia}, C_o = \text{Cardinal}(P_o)$

(b) Composed Recursion: Process A_0 := ..., where

process A_1 endproc (* A_1 *)

...

process A_n endproc (* A_n *)

endproc (* A_0 *)

Suppose the recursive process A_0 be composed of a set of recursive processes A_1, ..., A_n. Let N_{a0}, ..., N_{an} be the Galileo net representations for A_0, ..., A_n, respectively. The sets P_{xi} and R_{xi} are defined for each process A_i in the following ways:

$P_{xi} = \{p_a \mid (p_a \in P_{oai}), (t_a \in .p_a), (L(t_a) \neq \text{"exit"}), (L(t_a) \neq \text{"stop"})\}$

$R_{xi} = \{(p_a, a_j) \mid p_a \in P_{xi}), (a_j \in Id)\}$

where $Id = \{A_0, ..., A_n\}$ is the process set of identifiers.

The composition net N for processes A_0, ..., A_n is defined as:

$P = (P_{a0} - P_{x0}) \cup (P_{a1} - P_{x1}) \cup ... \cup (P_{an} - P_{xn})$

$P_i = P_{ia0}, P_o = (P_{oa0} - P_{x0}) \cup (P_{oa1} - P_{x1}) \cup ... \cup (P_{oan} - P_{xn})$

$T = T_{a0} \cup T_{a1} \cup ... \cup T_{an}$

$F = (F_{a0} - \{(t_a, p_a) \mid (t_a, p_a) \in F_{a0}, (p_a \in P_{x0})\}) \cup$

$(F_{a1} - \{(t_a, p_a) \mid (t_a, p_a) \in F_{a1}, (p_a \in P_{x1})\}) \cup$

$... \cup$

$(F_{an} - \{(t_a, p_a) \mid (t_a, p_a) \in F_{an}, (p_a \in P_{xn})\}) \cup$

$\{(t_a, p_a) \mid (t_a \in .P_{x0}), (p_a, a_j) \in R_{x0}, (p_a \in P_{iaj})\} \cup$

$\{(t_a, s_a) \mid (t_a \in .P_{x1}), (p_a, a_j) \in R_{x1}, (p_a \in P_{iaj})\} \cup$

$... \cup$

$\{(t_a, p_a) \mid (t_a \in .P_{x1}), (p_a, a_j) \in R_{x1}, (p_a \in P_{iaj})\} \cup$

$M_0 = M_{0a0}, L = (L_{a0} \cup L_{a1} ... L_{an})$

if (f_{a0} = "exit" or f_{a1} = "exit" or ... or f_{a0} = "exit")

then f = "exit" else "noexit"

$C_i = C_{ia0}, C_o = \text{Cardinal}(P_o)$

2.2.2 Petri-net Representation of LOTOS by Cheung and Zhu

The Petri net representation of LOTOS specifications proposed by Cheung and Zhu [CHE88] preserves the modular design in LOTOS. In the model, each LOTOS process is represented by a Petri-net and the LOTOS operations on the processes are represented by additional transitions for connecting these Petri-nets for processes. Different operations will use different ways of connection. The model also takes care of the data part of LOTOS by extending the Petri-net with special kinds of tokens, transitions, places and firing rules. A summary of these extensions is given in Table 2.2. The details of the transformation rules are given in [CHE88].

Petri-net feature	addition to an ordinary Petri-net	Purpose
token	control token:	for control flow
	data token:	for data flow
transition	data-transitions:	for describing special LOTOS operators: a-transition: for representing value-assignment statements g-transition: for representing guarded-choice i-transition: for representing interprocess communication m-transition: for matched synchronization
	operator transitions:	for representing operators
place	initial places:	for representing a process
firing rule	data conditions	in order for a transition to be firable, both control and data conditions have to be satisfied

Table 2.2 Extensions in Cheung's Petri-net representation of LOTOS constructs

Some improvements of this model are needed for it to become useful. For example, representation of the parallel operator has to be simplified, representations for recursions have to be added, etc. When these improvements are completed, Algorithm IPNTEST can also be applied on the basis of this model.

2.3 TEST SUITE GENERATION METHODS FOR LOTOS SPECIFICATIONS

The indeterminism property of a LOTOS specification makes endeavors for generating test suites from LOTOS more difficult than from other FDTs, such as ESTELLE and SDL. One of the problems is how verdicts should be defined and made for test sequences generated from specifications with indeterminism. Not much work has been done in this area of research. In this section, a theory and three methods concerning the generation of test suites from a LOTOS specification are reviewed. Among them, the canonical tester theory and the CO-OP method, though not practical in application, provide some new insights into the techniques of handling indeterminism in LOTOS specifications. The method presented by Tripathy and Sarikaya generates the same test cases as the CO-OP method, but is based on a different model called CHART. The method proposed by Guerauchi and Logrippo is a semi-automatic method. First, an execution tree is automatically produced for a LOTOS specification by using a LOTOS interpreter [GUI88]. Then a finite state machine is generated by studying the execution tree. The indeterminism property of LOTOS specifications is not considered in the finite state machine. Test sequences are generated from the finite state machine.

2.3.1 The Theory of Canonical Testers

Brinksma [BRI86, BRI88] presents a theory for the derivation of canonical testers from a specification. This theory is applicable to a system specified as a labelled transition system with internal actions. The translation of LOTOS constructs into labelled transition sytems is described in [BRI86, BRI88]. Therefore, the canonical theory is also applicable to LOTOS.

In the following definitions, the notations described in Table 1.2 of Section 1.1.2 are

adopted. First, the conformance relation between two processes is defined. Then, the terms 'test suite' and 'canonical tester' are defined. These concepts are the basis of the CO-OP method described in Section 2.3.2.

Definition (*conformance*)

Let B1 and B2 be two processes.

B1 *conf* B2 iff, $\forall \sigma \in Tr(B2)$ and $\forall A \subset L_c$, the following is true:

If $\exists B1', \forall a \in A, B1 = \sigma \Rightarrow B1' \neq a \Rightarrow$

then $\exists B2', \forall a \in A, B2 = \sigma \Rightarrow B2' \neq a \Rightarrow$

This means that B1 conforms to B2 if B1 contains no unexpected deadlocks with respect to the external path of B2. However, it allows B1 to possess external path not specified in B2.

As defined below, the term 'test suite' used in the canonical tester theory has a meaning slightly different from how it is commonly used.

Definition (*test suite*)

(a) A *test suite* TS is a set of processes. The elements of a test suite are termed test cases.

(b) Let $\sigma \in L_c^*$, T be a test case and B be a behaviour, then a derivation $T \| B = \sigma \Rightarrow T' \| B'$ is a test run of T and B; a test run is completed if $T' \| B' \Rightarrow$ stop occurs.

(c) A completed test run $T \| B = \sigma \Rightarrow T' \| B'$ is successful if $\sigma = \sigma'\delta$ for some $\sigma' \in L_c^*$; a completed test run fails if it is not successful.

(d) For T a test case Succ(T, B) $=_{df}$ all completed test runs of T and B are successful; for TS a test suite Succ(TS, B) $=_{df} \forall T \in TS$ Succ(T, B).

By the above definition, a test case is a process that acts as the environment for the IUT. A test run is a particular derivation of the parallel composition of the behaviour of a IUT and a test case.

Definition (*Canonical tester*)

Let S be a process specification. A *canonical tester* for S is a process T(S) such that:

(i) Tr(T(S)) = Tr(S); and (ii) for every process B, for every $\sigma \in L_c^*$. B conf S

iff B || T(S) = σ => stop implies T(S) = σ => stop

There are two basic ideas of a canonical tester T(S) for a system S: (1) It is capable of exploring those and only those external paths of S. (2) B conforms to S iff every deadlock between B and the tester T(S) can be explained as the tester having reached a terminal state. The main disadvantage is that running a canonical tester T(S) concurrently with an IUT once will test only one external path in S. For conformance testing, the tester has to be rerun until it has tested all external paths of S.

2.3.2 The CO-OP Method

Based on the theory reviewed in Section 2.3.1, Wezeman [WEZ89] proposed the CO-OP method for producing the canonical tester from a LOTOS specification based on basic LOTOS.

For the CO-OP method, indeterminism of a specification should all be due to internal actions. Any indeterminism due to multiple external actions should first be converted to indeterminism due to internal actions.

The following definitions are used for describing the CO-OP method.

Definition (*stable, unstable state*)

At a *stable state*, the system will accept only external actions and cannot perform an internal actions. At an *unstable state*, the system can perform an internal action.

Definition (*Compulsory* (B), *Options* (B))

Let B be a labelled transition system.

Compulsory (B) = $\{L_e \supset$ comp | $\exists$ B′, B = ε => B′ -/ i -> and

$$comp = \{x \in L_c \mid B' \text{-} x \text{->} \}\}$$

Options(B) = $\{ x \in L_c \mid \exists B', B = \varepsilon \Rightarrow B' \text{ - i ->} \text{ and } B' \text{ -x ->}\}$

Compulsory (B) is a set of sets of events. For each stable state B′ for which $B = \varepsilon \Rightarrow B'$ holds, Compulsory (B) contains a set comp with the external events that can be performed from B′. In each test case at least one element of each-element of Compulsory (B) must be used.

Options(B) is a set of events. For each unstable state B′ for which $B = \varepsilon \Rightarrow B'$ holds, this set contains all external events that can be performed from B′ without making any internal transitions before doing so. Elements of Option(B) may be used in test cases.

Definition (*orth* (M))

For a given set M, $M = \{m_1, m_2, ..., m_n\}$, where each m_i is a set, *orth* (M) denotes the set of all sets which are formed by choosing exactly one member from each element of M.

The following notation is used for defining test basic cases.

Notation:

$$\underset{a \in V}{[]} \; a = x_1 \; [] \; x_2 \; [] \; ... \; [] \; x_n \qquad \text{where } V = \{x_1, x_2, ... \, x_n\}$$

Test cases can be classified as basic or composite. A test case is said to be basic if its behaviour is defined by one of the following expressions.

Definition (*basic test cases*)

$T1 = \underset{a \in V}{[]} \; a; ...$

$T2 = (\underset{a \in V}{[]} \; a; ... \qquad) \; [] \; option; ...$

$T3 = i; stop$

$T4 = i; stop\ [\,]\ a; \ldots$ for some $a \in out(B)$

where $V \in orth(Compulsory(B))$ and $option \in Options(B)$. The test cases T3 and T4 are used for the special situation: where processes $\exists B'$, $B = \varepsilon \Rightarrow B'$, B' is a stop. Here B can move internally to a state where it will not accept any $a \in out(B)$.

Composite test cases are constructed by combining several basic test cases with the choice operator. That is, if t1, t2, ..., tm are basic test cases, then tn = t1 [] t2 [] ... [] tm is a composite test case. The canonical tester for the entire specification is the test suite consisting of all the basic test cases joined by the choice operators.

Definition (B *after* a)

Let B be a labelled transition system. Let $a \in out(B)$. The behaviour of B after interaction in a is specified by:

$$B \textit{ after } a = \left(\underset{B_i \in AS_a(B)}{[\,]} i; B_i\right) \text{ where } AS_a(B) = \{ B' \mid B = a \rightarrow B' \}$$

The main result of the CO-OP method can be summarized in the following algorithm.

ALGORITHM (CO-OP method)

Given a labelled transition system B, its canonical tester T(B) can be constructed (recursively) by the following steps:

(1) Construct the sets Compulsory(B) and Options(B).

(2) For each $a \in out(B)$ construct (B after a).

(3) If $\phi \notin Compulsory(B)$

$$\text{then } T(B) = \underset{V \in orth(Compulsory(B))}{[\,]} i; \left(\underset{a \in V}{[\,]} a; T(B \text{ after } a)\right)$$

$$[\,]\ option; T(B \text{ after } option)$$

$$\text{option} \in \text{Option}(B)$$

$$\text{else } T(B) = \quad i; \text{stop}$$

$$[]\ ([]_{a \in out(B)}\ a; T(B \text{ after } a))$$

The CO-OP method can be applied to specifications written in labelled transition systems. In order to apply the CO-OP method to LOTOS, a set of rules for transforming each basic LOTOS expression to a labelled transition system is given in [WEZ89]. It has been implemented by Alderden [ALD89] using the Cornell Synthesizer Generator [REP89]. In this implementation, any sub-specification can be chosen for deriving a canonical tester.

2.3.3 Tripathy and Sarikaya's Method

Tripathy and Sarikaya [TRI89] present a method for generating test cases from a LOTOS specification. In this method they apply the set of rules proposed by Karjoth [KAR88] to transform a LOTOS specification to an extended labelled transition system CHART. It is developed by [MIL84]. It is modelled by a control graph and a data flow graph. Test cases and their data flow functions are then derived from these graphs. The derived test cases turn out to be the same as those obtained by the CO-OP method [WEZ89]. In fact, this method is simply a combination of the three methods: the CHART developed by Milner, the transformation rules proposed by Karjoth, and the test cases defined by CO-OP method.

2.3.4 Guerauchi and Logrippo's Method

Gueraichi and Logrippo [GUE89] present a semi-formal method for deriving test cases for the protocol LAPB specified in LOTOS. In this method, an execution tree is first automatically generated from the specification by using a LOTOS interpreter [GUI88]. Test cases are then obtained by manually inspecting the derived tree. The interpreter performs a transformation of the specification by replacing the operators such as 'parallel composition', and 'disable' etc. to a tree

consisting of only 'choices'. This tree represents all possible (up to a specified length) execution sequences of the entity specified. As an example, the method is applied to the protocol LAPB. Based on the knowledge of the protocol, the specification is divided into three phases: 1) Connection Phase; 2) Data transfer phase; 3) Termination phase. The tree is represented with a state diagram. The diagram is a finite state machine. Test cases are generated from the finite state machine by using the UIO (Unique Input-Output) [SID89] method. In general, given a current state, the transition tree for this state is obtained. Next, test cases for this state are generated. The preambles of the new states are obtained by concatenating the preamble of the current state with those edges which lead to these new states.

Chapter 3

GENERATING TEST SEQUENCES AND THEIR DEGREES OF INDETERMINISM FROM A PERTI-NET-BASED SPECIFICATION

(ALGORITHM IPNTEST)

3.1 INTRODUCTION

This chapter presents a new approach for generating test sequences for distributed systems which include both concurrency and indeterminism. It includes four parts: (1) An extended Petri-net model - indeterministic Petri-nets (IPN) - is developed. (2) Based on the IPN, a metric called 'degree of indeterminism' is developed for estimating the number of times a test sequence should be executed before a verdict can be made. (3) An algorithm called IPNTEST is proposed for generating test sequences and their degree of indeterminism. (4) A fairness model of conformance testing is suggested. Implementation and application of IPNTEST are reported in Chapters 4 and 5.

The rest of the chapter is organized as follows: Section 3.2 defines an indeterministic Petri-net - IPN. A metric called 'degree of indeterminism' is developed. Section 3.3 describes the algorithm IPNTEST. A suggestion on how to use of this metric in conformance testing is also made.

3.2. THE MODEL OF INDETERMINISTIC PETRI-NETS

Concurrency and indeterminism are two common characteristics of many distributed systems. A Petri-net is well known for its capability in handling concurrency. In order to specify indeterminism, the basic labelled Petri-net has to be extended. This is done by classifying its labels and actions into 'external', 'internal' and 'special'. Here, we do not intend to give general formal definitions for these terms because they usually depend on the individual applications. In practice, such as in LOTOS, external actions are those observable interactions between processes and the environment, internal actions are those operations whose behaviours cannot be observed by other

processes or the environment, and special actions are those which play a special role in the specification, such as 'exit' (successful termination) and 'stop' (inaction).

Definition (IPN, *indeterministic labelled Petri-net)*.

An *indeterministic labelled Petri-net* IPN is a 6-tuple $< P, T, F, M_0, L, h >$, where

P is a non-empty set of places;

T is a set of transitions, where $P \cap T = \phi$;

F is a set of arcs, i.e., $F \subseteq (P \times T) \cup (T \times P)$;

M_0 is the initial marking;

$L = L_c \cup \{i\} \cup L_s$ is a set of labels, where L_c is the set of external labels, i is the internal label and L_s is the set of special labels (such as terminations); and

h: is a mapping T-> L.

Note: The concepts of token, marking and firing are the same as for PN.

Remember that a marking is a vector the value of whose j^{th} element denotes the number of tokens existing in the jth place of IPN. In the following development, we shall encounter the three types of markings defined below.

Definition (*firable*(M), *observable, internal-absent and internal-present markings*).

Let M be a marking of IPN and *firable*(M) be the multi-set $\{t \mid t \in L, t \text{ is firable at } M\}$.

(Note: firable(M) may have duplicate elements.)

– M is said to be *observable* iff $\text{firable}(M) \cap (L_c \cup L_s) \neq \phi$.

– M is said to be *internal-absent* iff $i \notin \text{firable}(M)$.

– M is said to be *internal-present* iff $i \in \text{firable}(M)$.

Figure 3.1 shows a marking tree with different types of markings. Terminal markings will be defined later. Obviously, some markings may belong to two different types. Each arc of the tree represents a firable label.

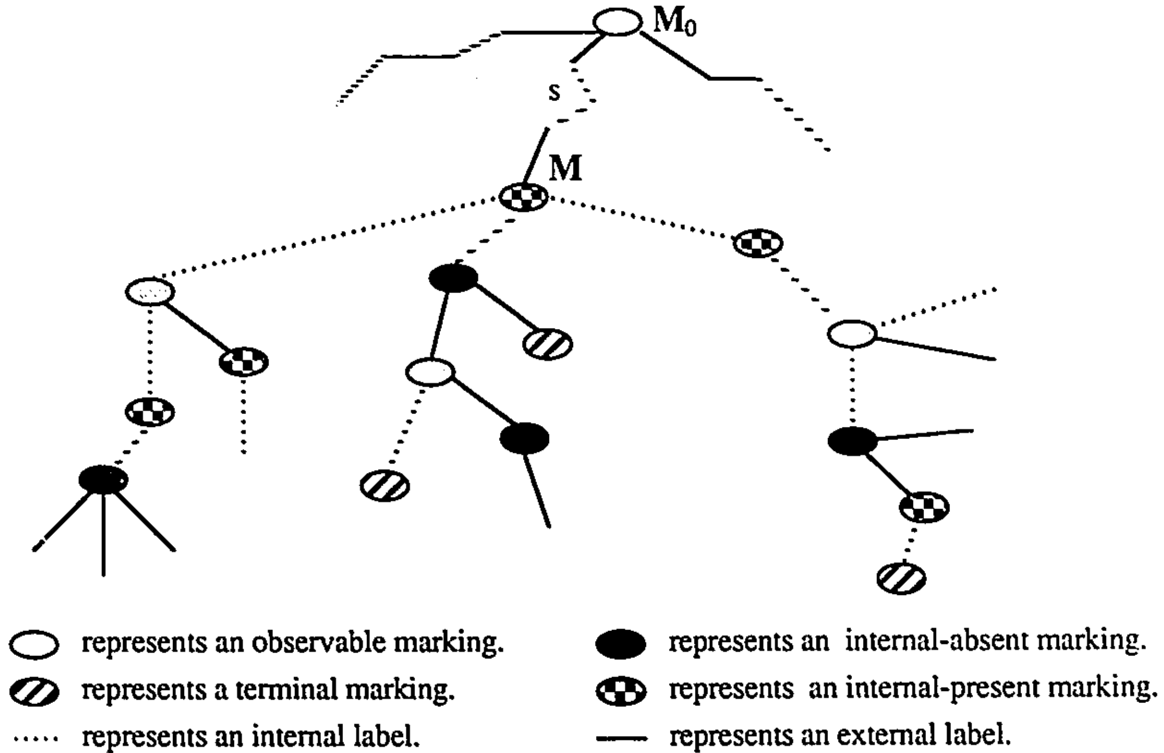


Figure 3.1. Different types of markings of a marking graph.

In the following notations, a, b, c, ... represent elements of L_c, i represents the internal label, x, x_j represent elements of L_s, and t, t_j represent elements of L.

Notation	Meaning
S*	The set of finite sequences of elements of S.
ε, t^0	ε denotes the empty sequence. For any $t \in L$, $t^0 = \varepsilon$.
M $[t_1,t_2, ..., t_n>$ M′	Firing the label sequence $t_1,t_2, ..., t_n$ changes marking M to marking M′.

$M\ [i^n>\ M'$	Firing the internal label i n times changes marking M to marking M′, i.e., $M\ [i, i, ..., i>\ M'$.
$M\ [a>>\ M'$ $a \in L_e$	There exist markings M_1 and M_2 such that $M\ [i^n>\ M_1\ [a>\ M_2\ [i^m>\ M'$.
$M\ [a_1a_2, ..., a_n>>\ M'$ $a_j \in L_e, j = 1, 2, ..., n$	There exist markings M_j, $j = 1, 2, ..., n-1$, such that $M\ [a_1>>\ M_1\ [a_2>> ... M_j\ [a_{j+1}>> ...\ M_{n-1}\ [a_n>>\ M_n = M'$.
$M\ [a_1a_2...a_n>>$ $a_j \in L_e, j = 1, 2, ... n$	There exists a marking M′ such that $M\ [a_1a_2...a_n>>\ M'$.
$\lvert t_1t_2 ... t_n \rvert$	External length (i.e., number of external labels) of the sequence $t_1t_2 ... t_n$.
#(t, M)	The number of times the label t and the internal label i appear in firable(M). In particular, if t = i, #(t, M) is just the number of i's appearing in firable(M).

When testing a deterministic process, executing the same sequence of actions under the same circumstances but at different times should always lead to the same state and have the same expected response (if any). For an indeterministic process, however, executing the same sequence of actions (at the same state) at different times may not always lead to the same state or get the same response. In the following, we propose a metric for measuring the probability that a specified marking can be reached by a specified sequence of actions. We assume that a uniform probability distribution is used for resolving indeterminism.

Consider the single transition: $M_1\ [t>\ M$ which is defined in the specification of a system. When executing the action t at marking M_1, an internal decision of the system may lead to $\#(t, M_1)$ possible outcomes and reaching M is only one of them. Hence, assuming a uniform probability distribution for these outcomes, $1/\#(t, M_1)$ will be the probability that M will be reached by

executing t. Hence, $\#(t, M_1)$ can be used as a measure for the degree of indeterminism of t from M_1 to M. In the following, we generalize this idea to a sequence of actions.

<u>Definition</u> (*degree of indeterminism, reachable, deterministically reachable, internally reachable, externally-last reachable*).

For a sequence $s = t_1 t_2 \dots t_n \in L^*$ and markings $M_1, M_2, M_3, \dots, M_n, M$, suppose that

$$M_1\ [t_1\!>\ M_2\ [t_2\!>\ M_3 \dots M_n\ [t_n\!>\ M \qquad (1)$$

is a feasible path (a sequence of firable actions) obtained from the IPN specification of a system.

– The *degree of indeterminism* of s between M_1 and M is defined to be

$$ind(M_1; s, M) = \prod_{j=1,\dots,n} \#(t_j, M_j)$$

In particular, when M_1 is the initial marking, the notation $ind(M_1; s, M)$ can be abbreviated as $ind(s, M)$.

– M is said to be *reachable* from M_1 by s with a degree of indeterminism $ind(M_1; s, M)$.

– M is said to be *deterministically reachable* by s from M_1 iff $ind(M_1; s, M) = 1$.

– M is said to be *internally reachable* from M_1 iff $t_j = i$ or ε, $j = 1, 2, \dots, n$. In particular, a marking is internally reachable from itself.

– M is said to be *externally-last reachable* from M_1 iff $t_n \in L_e$.

<u>PROPOSITION</u> (metric for measuring the probability of reachability).

For the sequence s specified as in (1), $1/ind(M_1; s, M)$ is used as the metric for measuring the probability that M is reachable from M_1 by s.

In the following, for a given marking M, we classify those observable markings internally reachable from M into two groups.

Definition (*deterministic set* of marking M, *indeterministic set* of marking M).

Let M be a marking of an IPN.

– $D(M) = \{D_1(M), D_2(M), \ldots, D_n(M)\}$ is called the *deterministic set* of M, where $D_j(M) = \text{firable}(M_j)$, M_j is an observable and internal-absent marking deterministically and internally reachable from M, and n is the number of such markings.

– $I(M) = \{I_1(M), I_2(M), \ldots, I_m(M)\}$ is called the *indeterministic set* of M, where $I_j(M) = \text{firable}(M_j) \cap (L_c \cup L_s)$, for some marking M_j which satisfies one of the following two conditions: (i) M_j is an observable and internal-present marking internally reachable from M. (ii) M_j is an observable and internal-absent marking internally-but-not-deterministically reachable from M; and m is the number of such markings.

Definition (*terminal marking*).

A marking M is said to be *terminal* iff both D(M) and I(M) are empty.

Note: M is terminal if no observable markings can be reached by it.

EXAMPLE 3.1

For the IPN shown in Figure 3.2 and its marking tree shown in Figure 3.3, we have:

$L_e = \{a, b, c, e, f, h\}$ is the set of external labels.

$L_s = \{x\}$ is the set of special labels.

$L = L_e \cup L_s \cup \{i\} = \{a, b, c, e, f, h, i, x\}$ is the set of labels.

For the following markings:

$M_0 = (1\,0\,0\,0\,0\,0\,0\,0\,0\,0)$, $M_1 = (0\,1\,0\,0\,0\,0\,0\,0\,0\,0)$, $M_2 = (0\,0\,1\,0\,0\,0\,0\,0\,0\,0)$

$M_4 = (0\,0\,0\,0\,1\,0\,0\,0\,0\,0)$, $M_6 = (0\,0\,0\,0\,0\,0\,1\,0\,0\,0)$, $M_7 = (0\,0\,0\,0\,0\,0\,0\,1\,0\,0)$ and

$M_8 = (0\,0\,0\,0\,0\,0\,0\,0\,1\,0)$,

we have:

M_2 is internal-absent because firable(M_2) = {b, c} contains only external labels b and c.

M_4 is observable and internal-present because firable(M_4) = {h, h, i, i} contains two external labels h and two internal labels i.

M_7 is internally reachable from M_4, i.e., M_4 [i > M_6 [i> M_7.

M_7 is not deterministically reachable from M_0, because ind(eii, M_7) = #(e, M_0) ·#(i, M_4) ·#(i, M_6) = 2 ·2 ·1 = 4. This means that there is only a probability of 1/4 that M_7 can be reached by firing the sequence eii from M_0.

M_8 is terminal because D(M_8) = I(M_8) = ϕ.

M_8 is deterministically reachable from M_0 by the sequence iac, because ind(iac, M_8) = 1.

D(M_0) = {{a}} and I(M_0) = {{e}}.

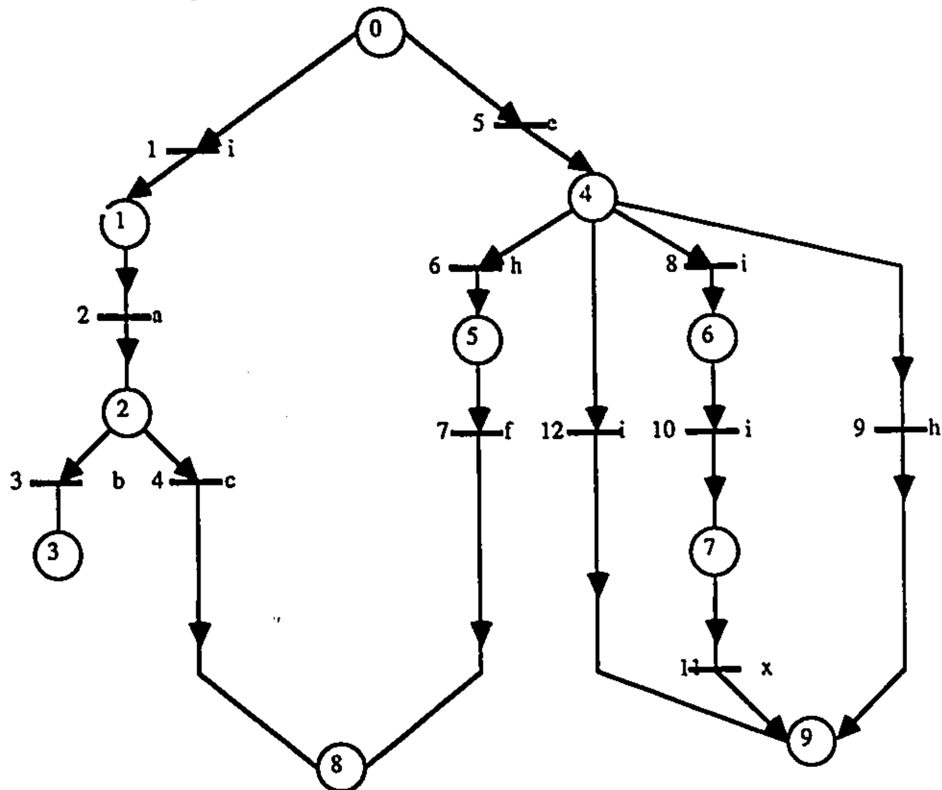


Figure 3.2. An indeterministic Petri-net
(digits represent transition numbers; letters represent their labels).

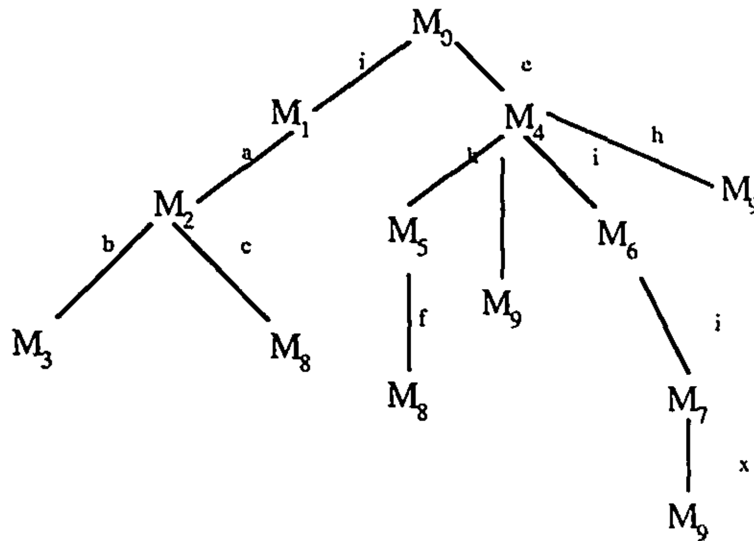


Figure 3.3. Marking tree of the indeterministic Petri-net of Figure 3.2.

3.3. GENERATION OF TEST SEQUENCES AND THEIR DEGREES OF INDETERMINISM (ALGORITHM IPNTEST)

In this section, we present an algorithm for generating test cases, their associated markings and their degrees of indeterminism for an IPN. Let us first give our definitions of test sequences.

Notation (Tr(IPN), *external path*, Tr(k, IPN)).

– Tr(IPN) = $\{s \mid s \in (L_e \cup L_s)^*$ and M_0 [s>> }. Each s is called an *external path* of IPN.

– Tr(k, IPN) = $\{s \mid s \in$ Tr(IPN) and $|s| = k\}$.

Note: If the referenced Petri-net IPN is obvious from the context, the symbol IPN may be omitted from the notation.

Definition (*test sequence* for IPN).

$s = a_1 a_2 \ldots a_k$ is said to be a *test sequence* for IPN iff the following two conditions are satisfied: (1) Each a_j is an external action or special action. (2) There exists an external path s′ =

$a_1'a_2' \dots a_k'$ of IPN with the property that, for every j, if a_j' is an output operation (resp., input operation), a_j is an input operation (resp., output operation) of the same message; or, if a_j' is a special action, $a_j' = a_j$.

Note that the terms 'external path' and 'test sequence' used in our context are the same as 'feasible path' and 'feasible test sequence' as used in the literature. Since it is trivial to convert a path to a test sequence, we shall hereafter use the two terms 'external path' and 'test sequence' interchangeably, provided that there is no confusion in the context.

Definition (*preamble and test sequence for a marking*).

- s is said to be a *preamble* of marking M iff $s \in Tr(IPN)$ and M_0 [s>> M.
- sa is said to be *a test sequence for a marking* M iff s is a preamble of M and $a \in L_c \cup L_s$.

  Note: By this definition, only non-terminal markings may have test sequences.

Notation (to be used for describing mapping g and Algorithm IPNTEST).

- For a collection of sets $A_1, A_2, \dots, A_n$, $(b_1, b_2, \dots, b_n) \in (A_1, A_2, \dots, A_n)$ iff $b_i \in A_i$, $i = 1, 2, \dots n$.
- $\mathcal{R}(k)$ is a set of pairs of the form (s, M) generated in the kth iteration, where M is a marking reachable by the preamble s and k is the external length of s.
  - If the pair (s, M) is known, ' s/ ' may be used as an abbreviation for ' s/ind(s,M) '.
- $\mathcal{P}(k)$ is the set of all test sequences generated in the kth iteration. It is also the set of preambles of the markings for the next iteration.
- G(M, k) denotes a collection of test cases for M, all whose test sequences are of length k.

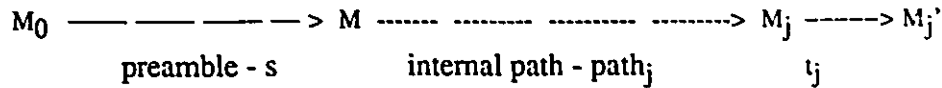
The degrees of indeterminism of the generated test sequences can also be calculated as follows:

Calculating the degrees of indeterminism:

For each $t_j = a_j$ (resp., b_j) and each $M_j \in D_j(M)$ (resp., $I_j(M)$) in Formula (2), where M_j is reachable from M by $path_j$ (a sequence of internal labels) and $M_j\ [t_j>\ M_j'$, the degree of indeterminism of the generated test sequence st_j can be calculated by the following formula:

$$st_j/ \overset{def}{=} st_j/ind(st_j, M_j'),$$

where $ind(st_j, M_j') = ind(s, M) \cdot ind(M, path_j, M_j) \cdot ind(M, t_j, M_j')$



PROPOSITION (*test-case-generating mapping g*).

For a given marking M with preamble s where $|s| = k$, the mapping g described below expands s into a set of test cases for M by appending the elements of the deterministic set D(M) and indeterministic set I(M) to s.

mapping g: $\mathcal{R}(k) \rightarrow G(M, k+1)$

$(s, M) \rightarrow g(s, M) = \{[sa_1/, sa_2/, ..., sa_n/, sb_1/, sb_2/, ..., sb_m/] - [s/] \mid$ for (2)

some $(a_1, a_2, ..., a_n) \in (D_1(M), D_2(M), ..., D_n(M))$ and

some $(b_1, b_2, ..., b_m) \in (I_1(M) \cup \{\varepsilon\}, I_2(M) \cup \{\varepsilon\}, ..., I_m(M) \cup \{\varepsilon\})\}$.

Notes:

(a) The test coverage aspect of mapping g will be discussed after the description of Algorithm IPNTEST.

(b) Since D(M) may be empty (then $a_1 = a_2 \ldots a_n = \varepsilon$) or some b_j may be ε, some of generated test sequences may be the same as the preamble s itself. The mapping g omits such test sequences from the test cases. Therefore, g(s, M) may be empty.

(c) Since $path_j$ may be non-unique, the degree of indeterminism of a test sequence may also be non-unique. In practice, for example, one may let $path_j$ be one of the shortest internal paths or let ind($path_j$, M_j) be the average of the degrees of indeterminism over all such internal paths.

ALGORITHM IPNTEST

INPUT

An indeterministic Petri-net (IPN) and an integer $m \geq 1$.

OUTPUT

1) All the markings externally-last-reachable by a path of external length (i.e., number of its external labels) k, where $k < m$.

2) Preambles of these markings.

3) Test cases for the initial marking and the generated markings, if they are not terminal.

4) Degrees of indeterminism of the generated test sequences.

METHOD

Starting from the initial marking M_0, the observable markings and their preambles are generated interactively as follows: In the kth iteration, for each externally-last-reachable marking M, where M_0 [s>> M and $|s| = k-1$, generate: (a) A group of test cases g(s, M) of the form [sa_1, sa_2, ..., sa_n, sb_1, sb_2, ..., sb_m] as described in Formula (2). (b) All the non-terminal markings M externally-last-reachable by a path of external length k. Details are given below.

Initialization Set $\mathcal{R}(1) := \{(\varepsilon, M_0)\}$ and $\mathcal{P}(0) := \phi$. Start the kth iteration with $k = 1$.

k^{th} iteration Suppose $\mathcal{R}(k)$ and $\mathcal{P}(k-1)$ have been obtained during the $(k-1)^{th}$ iteration. Obtain the set of preambles $\mathcal{P}(k)$, the set of test cases g(s, M) and $\mathcal{R}(k+1)$ by the following steps:

Step 1. Set $\mathcal{P}(k) := \phi$. For every pair (s, M) in $\mathcal{R}(k)$, do the following:

a) Generate g(s, M), the set of test cases and their degrees of indeterminism, by using the mapping g.

b) Set $\mathcal{P}(k) := \mathcal{P}(k) \cup (\bigcup_{c \in g(s,M)} c)$

Step 2. If $k = m$ or $\mathcal{P}(k) = \phi$, stop; otherwise, go to Step 3.

Step 3. Set $\mathcal{R}(k+1) := \{(st, M') \mid$ for every $st \in \mathcal{P}(k)$ and M' such that $M\ [i^n\!> M''\ [t\!> M'\}$.

Set $k := k+1$ and go to the next iteration.

Discussion on Algorithm IPNTEST

(a) The generated test cases can be grouped into two types: Those including both deterministic test sequences, (i.e., sequences where degree of indeterminism is equal to 1) and indeterministic test sequences (i.e., sequences with degree of indeterminism greater than 1); and those including only deterministic test sequences.

(b) All generated test sequences are firable (with respect to control flow only), i.e., each action is firable at the marking reached from the initial marking by its preamble.

(c) All firable test sequences of length at most m of IPN have been generated. Hence, the test cases cover all transitions firable at those places of IPN which are reachable by a path of external length less than m. The test cases also cover all the neighboring places of these transitions.

(d) In general, the degree of indeterminism of a given test sequence may not be unique because of the multiple choice of internal or mixed paths. Also in our development, uniform probability distribution has been adopted for resolving indeterminism. In practice, depending on the application, specific rules for solving these problems may be adopted.

(e) Since IPNTEST generates test sequences 'level by level' and marking by marking, it can be easily modified for selective test sequence generation, e.g., by avoid to generate those unneeded testing sequences for a particular test purpose.

Application of Algorithm IPNTEST to conformance testing:

As explained at the beginning of this chapter, the kind of verdicts made on a test sequence depends on whether the IUT (implementation under test) is deterministic or indeterministic. If the IUT is for a deterministic program, one can conclude that the IUT has an error if a correctly derived test sequence fails after a single trial (e.g., ending at an unexpected deadlock). When testing an IUT which exhibits indeterminism, however, a test sequence may fail in one trial but pass in another. For such IUTs, a different criterion should be adopted for making verdicts on the test results.

Definition (*pass definitely*).

A test sequence *passes definitely* iff no unexpected deadlock occurs during the application of the test sequence.

PROPOSITION (*conformance testing*).

An implementation under test *conforms to a specification with respect to a test suite* if every test sequence of the suite passes definitely at least once after trying a 'fair' number of times.

PROPOSITION.

In conformance testing (according to the above definition), when executing a test sequence generated by IPNTEST, its degree of indeterminism can be used as the fair number of trials.

The 'fairness' concept is first time introduced into conformance testing as far as we know. A starting point is set for the study of performance analysis of conformance testing for distributed systems with indeterminism.

It should be noted that it is not the objective of this thesis to develop an advanced, complete probabilistic theory or performance analysis model for solving the 'fairness' issue. It serves just as a starting point towards such a target.

Chapter 4

APPLICATION OF ALGORITHM IPNTEST TO LOTOS

(LOTOS-TCG _ A TEST CASE GENERATOR FOR LOTOS)

4.1 INTRODUCTION

In this chapter, we present the second major contribution of this thesis: Application of Algorithm IPNTEST to the generation of test sequences from a LOTOS specification and the implementation of IPNTEST as a test case generator LOTOS-TCG.

LOTOS-TCG is a subsystem of UO-GLOTOS, an interactive menu-driven graphical system developed under the direction of Cheung [CHE89] for research in LOTOS. UO-GLOTOS is based on a limited set of simple graphical patterns for representing the LOTOS constructs and a hierarchical structure for representing a LOTOS specification. Besides LOTOS-TCG, four other subsystems, including a graphical editor, two translators between textual and graphical LOTOS and an interpreter, have been implemented. Other subsystems such as syntax-verifier, display of Petri-net, are also being developed. An organizational diagram of the entire system UO-GLOTOS is shown in Figure 4.1.

4.2 FUNCTIONS OF TEST CASE GENERATOR LOTOS-TCG

LOTOS-TCG accepts a graphical or textual LOTOS specification as input and it outputs test sequences and their degrees of indeterminism by applying the Galileo net transformation described in Chapter 2 and the algorithm described in Chapter 3. The LOTOS specification is produced by either the editor or the translators of UO-GLOTOS. The internal representation of a LOTOS specification in UO-LOTOS is an internal node table.

Functionally, LOTOS-TCG is divided into two subprograms (Figure 4.2):

(1) TRANSFORMATION, which transforms a LOTOS specification produced by the editor or the translators of UO-GLOTOS into an indeterministic Petri-net (Galileo net);

(2) GENERATION, which generates test cases by applying Algorithm IPNTEST on the

indeterministic Petri-net.

Starting with the internal node table, TRANSFORMATION produces the matrix representation of an indeterministic Petri-net. GENERATION then generates the test sequences and their degrees of indeterminism and groups them into test cases through executing the indeterministic Petri-net by the algorithm IPNTEST described in Chapter 3. In the following subsections, some details of the implementation of these two programs are given. LOTOS-TCG is implemented with the C language on a SUN Workstation under UNIX system.

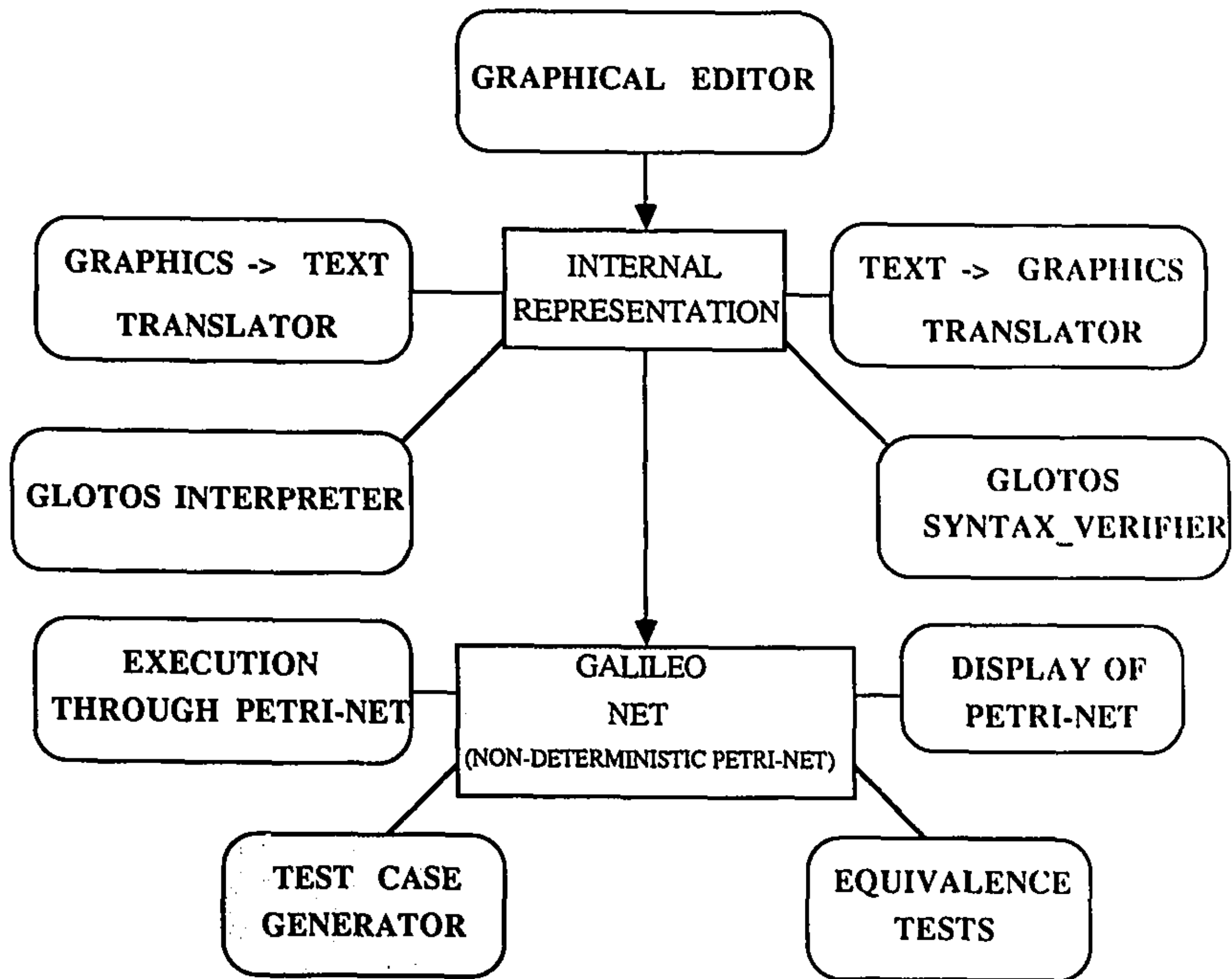


Figure 4.1. A LOTOS-based graphical environment for distributed computing research at University of Ottawa (The shaded components form LOTOS-TCG)

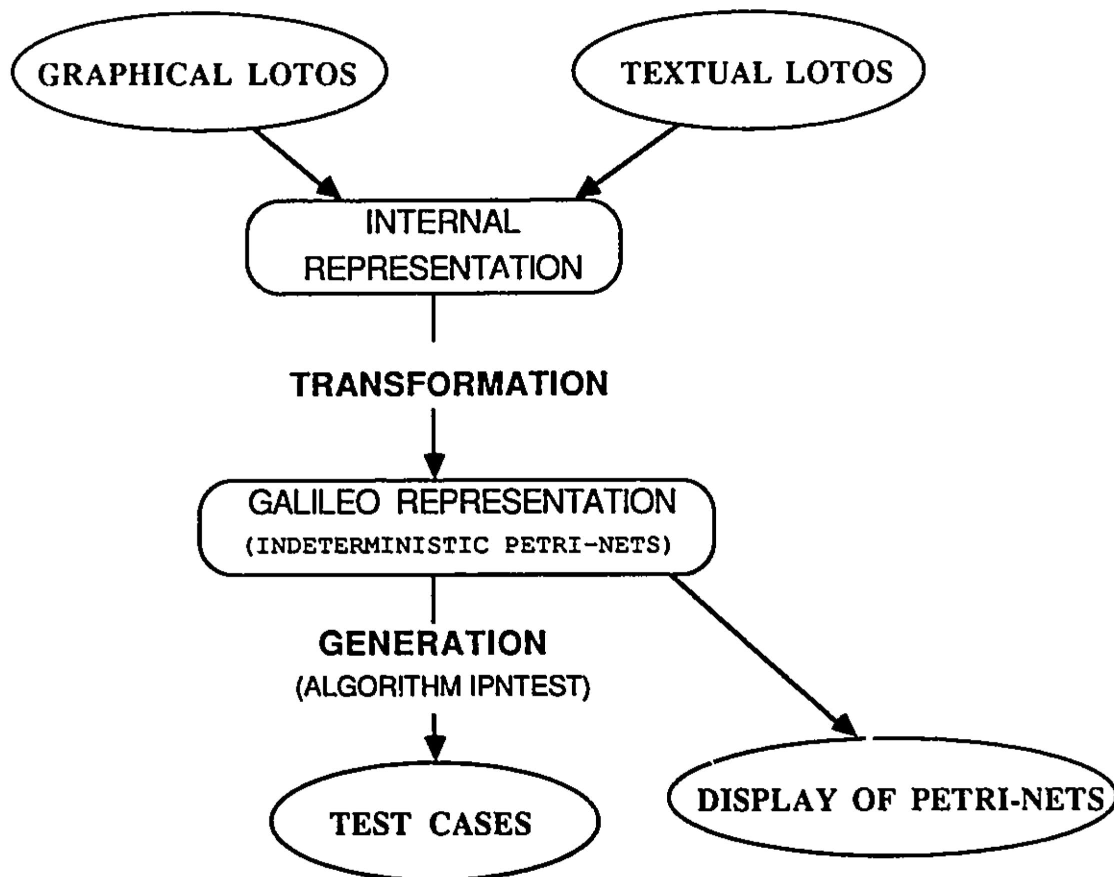


Figure 4.2. Functional diagram of the test case generator LOTOS-TCG

4.3 SUBPROGRAM - TRANSFORMATION

The internal representation of the LOTOS specification, internal node table, is transformed into an indeterministic Petri-net in the subprogram TRANSFORMATION. There are two steps: 1) The internal node table is converted into a binary tree by using the algorithm INT-BT. 2) The binary tree is transformed into a Petri-net by using the algorithm PT with application of the Galileo transformation rules. The algorithms, data structures and functions of TRANSFORMATION are shown in Figure 4.3.

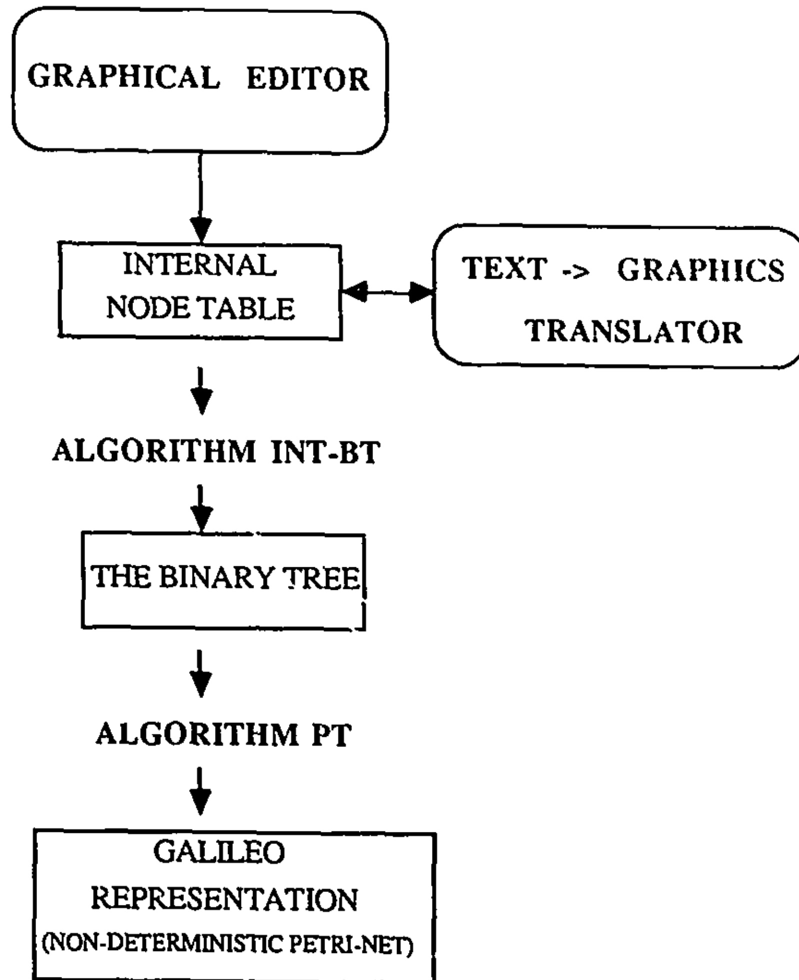


Figure 4.3. Functional diagram of the subprogram TRANFORMATION

Among the various Petri-net models, Galileo net is chosen as the internal representation of LOTOS because it is applicable to basic LOTOS and has been more fully developed than the other methods reported in the literature. The Galileo nets (extension of Petri-nets) and the transformation rules were proposed by Marchena and Leon [MAR89] and are reviewed in Chapter 2. The data structures and algorithms used in TRANSFORMATION are described in section 4.3.1 and 4.3.2.

4.3.1 Data Structures Used in Subprogram TRANSFORMATION

The subprogram TRANSFORMATION uses three data structures: the internal node table, the binary tree and a stack called gnet. Both the internal node table and the binary tree are used for

representing a LOTOS specification. The purpose of converting the internal node table to the binary tree is to make the creation of the Petri-net representation efficient and easy. The internal node table is the input data structure of TRANSFORMATION. It represents a LOTOS specification. The gnet is the output data structure of TRANSFORMATION. It represents an indeterministic Petri-net.

1) The internal node table

The internal node table is an internal representation of a LOTOS specification. It is produced by the editor or the translators of UO-GLOTOS and used by the other subsystems. In UO-GLOTOS, each node on the screen is a symbol representing a process, or an operator, or an action of the specification. Each of these nodes is represented in the table as a row with eight different fields (Table 4.1). The category field of the node indicates what a node represents for. For example, the value 5 of the category field means that the node is an external action. As indicated by the last three fields of a node, the set of nodes in the table form a tree, i.e., a hierarchical structure representing the LOTOS specification. In UO-GLOTOS, an operator has a variable number of operands. In the hierarchy represented by the tree, the first operand represents its son and the other operands as the brothers of its son. Also, in the internal node table, the action preceding the action-prefix operator is treated as the father of the action-prefix. In this case, action-prefix has only one operand. This makes operator action-prefix different from other operators. This is not efficient in the transformation of a LOTOS specification to an Petri-net. Because of the above reasons, a new internal representation, a binary tree, is proposed and given as follows.

field	data type	usage
category	integer	what the node represents for
row	integer	the position of the node in the screen
column 1	integer	the position of the node in the screen
column 2	integer	the position of the node in the screen
point of father	integer	the node number of its father

point of son	integer	the node number of its son
point of brother	integer array	the node number of its brother(s)
rec	integer	the recursion point

Table 4.1. The eight fields of a row of the internal node table

2) The binary tree

The binary tree is another internal representation of the LOTOS specification. It is transformed from the internal node table. It is different from the internal node table in one aspect: the internal node table can represent an operator which has a variable number of operands, but the operations in the binary tree are always binary operations. Every operator has two operands. In the binary tree representation, an operator has two children: the left child represents one of operands and the right child represents the other. A node represents an operation or an action. All leaf-nodes are actions and all non-leaf-nodes are operators. The fields of each node in the binary tree is shown in Table 4.2.

field	type	usage
category	integer	what the node represents for
leftchild	integer	the node number of its left child
rightchild	integer	the node number of its right child
father	integer	the node number of its father
recson	integer	the recursion point

Table 4.2. The five fields of a node of the binary tree

3) The indeterministic Petri-net representation

A data structure called gnet is used to record the indeterministic Petri-net. It is used as a stack in the transformation from LOTOS to indeterministic Petri-nets. Each element of gnet represents a Petri-net. The structure of its elements is shown in Table 4.3.

field	data type	usage
s	integer array	places
si	integer array	input places
so	integer array	output places
t	integer array	transitions
fst	integer array	arcs from places to transitions
fts	integer array	arcs from transitions to places
mi	integer array	the initial marking
l	character array	labels
l_type	integer array	label types
f	integer	'exit' or 'noexit'
ci	integer	numbers of input places
co	integer	number of output places

Table 4.3. Twelve fields of an element of gnet

4.3.2 Algorithms Used in Subprogram TRANSFORMATION

Two algorithms are used in TRANSFORMATION: INT-BT and PT. Algorithm INT-BT converts the internal node table into a binary tree. Algorithm PT traverses the binary tree and applies a set of transformation rules to create the indeterministic Petri-net.

1) Algorithm INT-BT

Algorithm INT-BT is used to transform the internal node table into a binary tree. It transforms all LOTOS operators into binary operators. This includes two things: 1) The operators which have more than two operands are changed to binary operators. For example, an operator choice has three operands: a, b, and c. In the internal node table, if the choice operator [] is represented by a node, a is its son, b is a's brother and c is b's brother. In the algorithm INT-BT, this is changed to a [] b [] c. In the binary tree, if the choice between a and b is a node, a is its left child, the choice between b and c is its right child, b is the left child of the choice between b and c,

c is the right child of the choice between b and c. 2) The operator action-prefix in UO-GLOTOS has only one operand. For example, for a LOTOS expression a; exit, if action-prefix is a node, a is its father and exit is its son. In the algorithm INT-BT, the operator action-prefix will have two operands. For the same expression a; exit, if action-prefix is a node, a is its left child and exit is its right child. A pseudo-code description of the algorithm is given below.

<u>ALGORITHM INT-BT</u>

<u>INPUT</u>

The internal node table INT representation of a LOTOS specification.

<u>OUTPUT</u>

The binary tree BT representation of a LOTOS specification.

<u>INITIALIZATION</u>

BT is an empty binary tree

<u>METHOD</u>

```
For each node ∈ INT,
  If the operator OP which has more than two operands in INT
    add OP to any two operands,
    expansion BT:
         for each operator OP:
                represent the first operand of OP as its left child and the other
                operand as its right child;
         endfor
  if the operator action-prefix
    for the operator action-prefix,
    expansion BT: change its father in INT to its left child in BT, its son in INT to
                   its right child.
  otherwise
```

copy the node in INT to BT

endfor

2) Algorithm PT

Algorithm PT is used to create an indeterministic Petri-net by traversing the binary tree in post-order. The algorithm is similar to the well-known technique of transforming an arithmetic expression into a backus normal form. A stack gnet is used for traversal. The structure of its elements is given in Table 4.3. During traversal, there are two possibilities: (1) If an action node is met, an indeterministic Petri-net will be formed according to the transformation rules and be pushed into the stack. (2) If an operator node is met, two Petri-nets representing its operands will be popped up from the stack to form a new Petri-net according to the transformation rules. The new Petri-net is pushed back into the stack. An indeterministic Petri-net is created when traversal of the binary tree is completed.

A pseudo-code description of the algorithm is given below.

ALGORITHM PT

INPUT

The binary tree representation of a LOTOS specification.

OUTPUT

The indeterministic Petri-net transformed from the input LOTOS specification.

METHOD

Post-traverse the binary tree, apply the transformation rules in the traversing.

Initialization: tp = 1 is the pointer of the node number in the binary tree

procedure:

```
traverse (tp)
integer tp;
{
```

```
 if (the node of the_binary_tree is not empty)
 then
       {
         traverse ( the_binary_tree[tp]. leftchild);
         traverse ( the_binary_tree[tp]. rightchild);
         process(tp);
        }
}end traverse

process (tp)
integer tp;
{
 If (the node is an action)
 then
       {
         form an indeterministic Petri-net;
         push it into the stack;
        }
 else
       {
         two indeterministic Petri-nets pop out of the stack;
         applying the transformation rules to the operator;
        }
 endif
} end process
```

4.4 SUBPROGRAM - GENERATION

The subprogram GENERATION is mainly an implementation of Algorithm IPNTEST. The input to GENERATION is an indeterministic Petri-net obtained in TRANSFORMATION. The outputs of GENERATION are produced by Algorithm IPNTEST.

4.4.1 Data Structures Used in Subprogram GENERATION

IPNTEST is an iterative algorithm. Two kinds of data structures are used in GENERATION: 1) data structures used in every iteration; 2) data structures used for a particular marking.

1) Data structures used in every iteration

The data structure begin_marking is used to record the markings obtained for one iteration from the last iteration, i.e., $\mathcal{R}(k)$ in Algorithm IPNTEST. The structure of begin_marking is shown in Table 4.4.

The data structure end_marking is used to collect the observable markings reached by the markings stored in begin-marking. The purpose of the collection is to produce the markings in begin_marking of next iteration, i.e., $\mathcal{R}(k + 1)$ in Algorithm IPNTEST. Both of end_marking and begin_marking have the same structure as shown in Table 4.4.

field	data type	usage
preamble	integer array	the preamble of the marking
degree	integer	the 'degree of indeterminism' of the preamble
marking	integer array	the marking

Table 4.4. Data structures used in an iteration.

2) Data structures used for a particular marking

The data structure marking_subtree is used to record a subtree of the marking tree of an indeterministic Petri-net. It consists of some of the reachable markings from a particular marking. The root of the subtree is a marking in begin_marking. The leaves of the subtree are all the observable markings internally reachable from the root marking. It is used for a particular marking. It will be erased after the deterministic set and the indeterministic set of the marking are decided. Each node in the marking_subtree records the feature of a marking. The structure of a node of the marking_subtree is shown in Table 4.5.

field	data type	usage

marking	integer array	the marking
father	integer	the node number of its father
from_transition	integer	the last fired transition to reach the marking
out_external	integer array	all firable external transitions of the marking
out_internal	integer array	all firable internal transitions of the marking
in_pre_marking	integer	the flag of internal-present-marking
in_ab_marking	integer	the flag of internal-absent-marking
ob_marking	integer	the flag of observable marking
degree	integer	the degree of indeterminism of an action
sons	integer array	the son markings of the marking

Table 4.5. Structure of a node of the marking_subree.

The data structure de_inde_set is used to record both the deterministic set and the indeterministic set of a marking. The structure of de_inde_set is an integer array.

4.4.2 Functions of Subprogram GENERATION

A pseudo-code description of the implementation of IPNTEST is given below:

```
Initialization:   Send the initial marking to begin_marking; k = 1;
Iteration:        For iteration k
                  {
                     end_marking = ϕ;
                     For all markings ∈ begin_marking(k)
                        {
                         generating its marking subtree, i.e., getting its external markings;
                         generating its deterministic set and indeterministic set;
                         generating its test sequences, degree of indeterminism and group
                         them into test cases;
                         end_marking = end_marking ∪ all its external markings
                        }
                     begin_marking = ϕ;
                     Getting begin_marking for next iteration from end_marking:
```

```
        if (a marking ∈ end-marking) and ( not a terminal marking)
              begin_marking =  begin_marking ∪ the marking
}
```

Chapter 5

AN EXAMPLE (THE CLASS 0 TRANSPORT PROTOCOL)

In this chapter, we apply the test case generator LOTOS-TCG to the Class 0 Transport Protocol for exemplifying our test sequence generation method.

Our test sequence generation method, Algorithm IPNTEST, is described in Chapter 3. LOTOS-TCG is described In Chapter 4.

The LOTOS specification *Handler* of the Class 0 Transport Protocol is described in the [BOL87]. There are five processes in the specification. In Section 5.1, we use one of the processes as an example to explain in detail the data structures used in LOTOS-TCG and the main steps of Algorithm IPNTEST. In Section 5.2, an example of the output of LOTOS-TCG is given. The entire LOTOS specification *Handler* is given in Section 5.3.

5.1 EXEMPLIFYING THE DATA STRUCTURES AND ALGORITHM IPNTEST

In this section, the LOTOS process *Connection-phase* for the Connection Phase of the Class 0 Transport Protocol is used as an example to explain Algorithm IPNTEST and the usage of the internal data structures.

First, the LOTOS specification of *Connection-phase* is input to the text-to-graphics translator of UO-GLOTOS. Then, test sequences and test cases are generated by LOTOS-TCG. In the following, the inputs and outputs of the UO-GLOTOS translator, the subprogram TRANSFORMATION and the subprogram GENERATION are described.

<u>UO-GLOTOS (the translator):</u>

The input to the translator is the LOTOS process *Connection-phase* shown in Section 5.3. The output from the translator is an internal node table whose tree representation is given in Figure 5.1.

TRANSFORMATION:

The input to the subprogram TRANSFORMATION is the internal node table which is the output of UO-GLOTOS. The binary tree, as the internal representation of the LOTOS specification, is produced by TRANSFORMATION (Figure 5.2). The output of TRANSFORMATION is the indeterministic Petri-net shown in Figure 5.3 and its matrix representation is given in Table 5.1.

GENERATION:

The input to the subprogram GENERATION is the matrix representation of the indeterministic Petri-net shown in Table 5.1. Table 5.2 includes all the markings under consideration. The marking subtrees of the Petri-net are shown in Figure 5.4. The results of GENERATION, i.e., applying algorithm IPNTEST, is shown in Table 5.3.

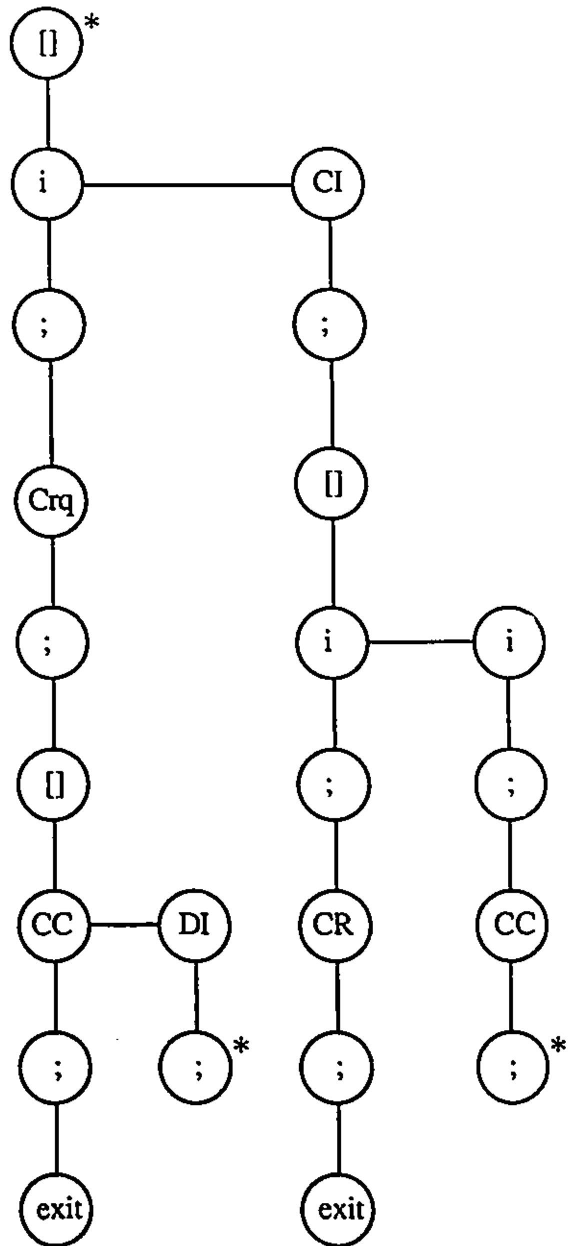


Figure 5.1. Tree of internal nodes for the Connection Phase of the Class 0 Transport Protocol
(* indicates a recursion)

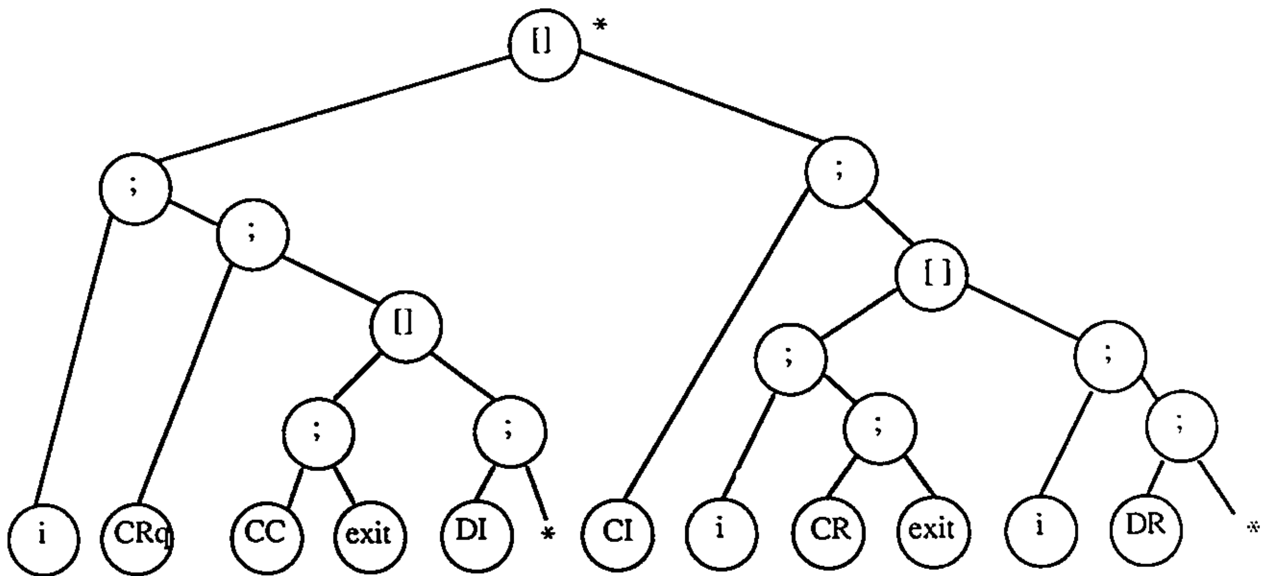


Figure 5.2. Binary tree of Connection Phase of the Class 0 Transport Protocol

(* indicates a recursion)

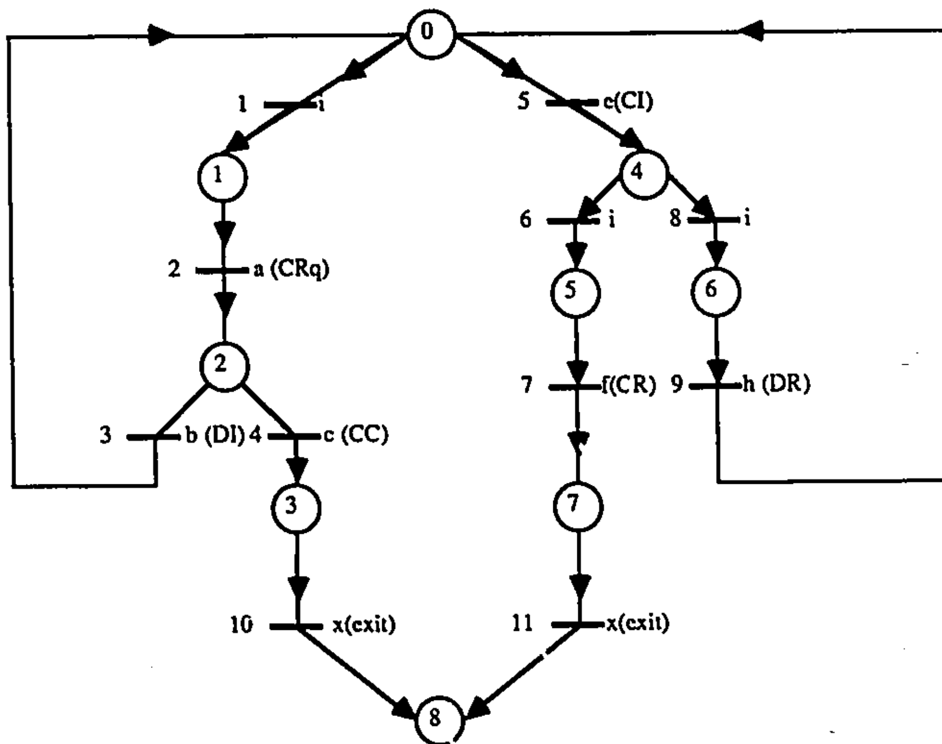


Figure 5.3. Indeterministic Petri net representation of the LOTOS specification of the Connection Phase of the Class 0 Transport Protocol

D^- =

	0	1	2	3	4	5	6	7	8
1(i)	1								
2(CRq)		1							
3(DI)			1						
4(CC)			1						
5(CI)	1								
6(i)					1				
7(CR)						1			
8(i)					1				
9(DR)							1		
10(exit)				1					
11(exit)								1	

D^+ =

	0	1	2	3	4	5	6	7	8
1(i)		1							
2(CRq)			1						
3(DI)	1								
4(CC)				1					
5(CI)					1				
6(i)						1			
7(CR)								1	
8(i)							1		
9(DR)	1								
10(exit)									1
11(exit)									1

Table 5.1. Matrix representation of the Connection Phase of the Class 0 Transport Protocol

name	marking	name	marking
M_0	(1 0 0 0 0 0 0 0 0)	M_1	(0 1 0 0 0 0 0 0 0)
M_2	(0 0 1 0 0 0 0 0 0)	M_3	(0 0 0 1 0 0 0 0 0)
M_4	(0 0 0 0 1 0 0 0 0)	M_5	(0 0 0 0 0 1 0 0 0)
M_6	(0 0 0 0 0 1 0 0 0)	M_7	(0 0 0 0 0 0 0 1 0)
M_8	(0 0 0 0 0 0 0 0 1)		

Table 5.2. Markings under consideration

iter.	(preamble, marking)	det. set	indet. set	set of test cases	test sequences
k	$\mathcal{R}$(k): (s, M)	D	I	g(s, M)	$\mathcal{P}$(k)
1	(ε, M0)	{a}	{e}	[a/1], [a/1, e/2]	{a, e}
2	(a, M2)	{b, c}		[ab/1], [ac/1]	
	(e, M4)		{f},{h}	[ef/4], [eh/4], [ef/4, eh/4]	{ab, ac, ef, eh}
3	(ab, M0)	{a}	{e}	[aba/1], [aba/1, abe/2]	
	(ac, M3)	{x}		[acx/1]	
	(eh, M0)	{a}	{e}	[eha/4], [eha/4, ehe/8]	
	(ef, M7)	{x}		[efx/4]	{aba, abe, acx, efx, eha, ehe}

Table 5.3. Results of applying IPNTEST to the Connection Phase of the Class 0 Transport Protocol (In the symbols s/ind, ind is 'the degree of indeterminism of sequence s'.)

5.2 THE OUTPUT OF LOTOS-TCG

In this section, the LOTOS-TCG is applied to the LOTOS specification *Handler*. The generated indeterministic Petri-net representation of *Handler* is shown in Figure 5.5. On entering the input *Handler* to LOTOS-TCG, the user is asked to input the length of test sequences (the first statement). In this example, we enter the number 4. Then LOTOS-TCG outputs the preambles, markings, test sequences with their degrees of indeterminism, and test cases for four iterations. After browsing the results, the user will be asked whether to continue or not (the last sentence), the user can input y(es) or n(o). In this example, we enter n to end the execution of LOTOS-TCG. If y is entered the system will go back to the beginning (the first sentance). The example of applying LOTOS-TCG is as follows:

```
Please input the length of test sequences: 4
Iteration 1:

    preamble:
    marking: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

        Test sequences:
                CI /2.
                CRq /1.
        Test cases:
                CRq /1. ]
                [CI /2. CRq /1. ]

Iteration 2:

    preamble: CI
    marking: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

        Test sequences:
                CI CR /4.
                CI DR /4.
        Test cases:
                [CI DR /4. ]
                [CI CR /4. ]
                [CI CR /4. CI DR /4. ]

    preamble: CRq
    marking: 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

        Test sequences:
                CRq DI /1.
                CRq CC /1.
```

Test cases:
[CRq DI /1.]
[CRq CC /1.]

Iteration 3:

preamble: CI CR
marking: 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CI CR exit /4.
Test cases:
[CI CR exit /4.]

preamble: CI DR
marking: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Test sequences:
CI DR CI /8.
CI DR CRq /4.
Test cases:
[CI DR CRq /4.]
[CI DR CI /8. CI DR CRq /4.]

preamble: CRq DI
marking: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CRq DI CI /2.
CRq DI CRq /1.
Test cases:
[CRq DI CRq /1.]
[CRq DI CI /2. CRq DI CRq /1.]

preamble: CRq CC
marking: 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CRq CC exit /1.
Test cases:
CRq CC exit /1.]

Iteration 4:

preamble: CI CR exit
marking: 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CI CR exit DI /12.
CI CR exit DtI /12.
CI CR exit DtR /16.
CI CR exit DI /16.
CI CR exit DR /8.
CI CR exit DR /8.

Test cases:
[CI CR exit DR /8.]
[CI CR exit DR /8.]
[CI CR exit DR /8. CI CR exit DR /8.]
[CI CR exit DtR /16 .]
[CI CR exit DtR /16. CI CR exit DR /8.]
[CI CR exit DtR /16 . CI CR exit DR / 8 .]
[CI CR exit DtR /16. CI CR exit DR /8. CI CR exit DR /8.]
[CI CR exit DI /16.]
[CI CR exit DI /16. CI CR exit DR /8.]
[CI CR exit DI /16. CI CR exit DR /8.]
[CI CR exit DI /16. CI CR exit DR /8. CI CR exit DR /8.]
[CI CR exit DI /12.]
[CI CR exit DI /12. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DR /8. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DtR /16.]
[CI CR exit DI /12. CI CR exit DtR /16. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DtR /16. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DtR /16. CI CR exit DR /8. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DI /16.]
[CI CR exit DI /12. CI CR exit DI /16. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DI /16. CI CR exit DR /8.]
[CI CR exit DI /12. CI CR exit DI /16. CI CR exit DR /8. CI CR exitDR /8.]
[CI CR exit DtI /12.]
[CI CR exit DtI /12. CI CR exit DR /8.]
[CI CR exit DtI /12. CI CR exit DR /8.]
[CI CR exit DtI /12. CI CR exit DR /8. CI CR exit DR /8.]
[CI CR exit DtI /12. CI CR exit DtR /16.]
[CI CR exit DtI /12. CI CR exit DtR /16. CI CR exit DR /8.]
[CI CR exit DtI /12. CI CR exit DtR /16. CI CR exit DR /8.]
[CI CR exit DtI /12. CI CR exit DtR /16. CI CR exit DR /8. CI CR exit DR /2.]
[CI CR exit DtI /12. CI CR exit DI /16.]
[CI CR exit DtI /12. CI CR exit DI /16. CI CR exit DR /8.]
[CI CR exit DtI /12. CI CR exit DI /16. CI CR exit DR /8.]
[CI CR exit DtI /12. CI CR exit DI /16. CI CR exit DR /8. CI CR exit DR /2.]

preamble: CI DR CI
marking: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CI DR CI CR /16.
CI DR CI DR /16.

Test cases:
[CI DR CI DR /16.]
[CI DR CI CR /16.]
[CI DR CI CR /16. CI DR CI DR /16.]

preamble: CI DR CRq
marking: 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CI DR CRq DI /4.
CI DR CRq CC /4.
Test cases:
[CI DR CRq DI /4.]
[CI DR CRq CC /4.]

preamble: CRq DI CI
marking: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CRq DI CI CR /4.
CRq DI CI DR /4.
Test cases:
[CRq DI CI DR /4.]
[CRq DI CI CR /4.]
[CRq DI CI CR /4. CRq DI CI DR /4.]

preamble: CRq DI CRq
marking: 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CRq DI CRq DI /1.
CRq DI CRq CC /1.
Test cases:
[CRq DI CRq DI /1.]
[CRq DI CRq CC /1.]

preamble: CRq CC exit
marking: 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Test sequences:
CRq CC exit DI /3.
CRq CC exit DtI /3.
CRq CC exit DtR /4.
CRq CC exit DI /4.
CRq CC exit DR /2.
CRq CC exit DR /2.
Test cases:
[CRq CC exit DR /2.]
[CRq CC exit DR /2.]
[CRq CC exit DR /2. CRq CC exit DR /2.]
[CRq CC exit DtR /4.]
[CRq CC exit DtR /4. CRq CC exit DR /2.]
[CRq CC exit DtR /4. CRq CC exit DR /2.]
[CRq CC exit DtR /4. CRq CC exit DR /2. CRq CC exit DR /2.]
[CRq CC exit DI /4.] [CRq CC exit DI /4. CRq CC exit DR /2.]
[CRq CC exit DI /4. CRq CC exit DR /2.]
[CRq CC exit DI /4. CRq CC exit DR /2. CRq CC exit DR /2.]
[CRq CC exit DI /3.] [CRq CC exit DI /3. CRq CC exit DR /2.]
[CRq CC exit DI /3. CRq CC exit DR /2.]
[CRq CC exit DI /3. CRq CC exit DR /2. CRq CC exit DR /2.]
[CRq CC exit DI /3. CRq CC exit DtR /4.]

[CRq CC exit DI /3. CRq CC exit DtR /4. CRq CC exit DR /2.]
[CRq CC exit DI /3. CRq CC exit DtR /4. CRq CC exit DR /2.]
[CRq CC exit DI /3. CRq CC exit DtR /4. CRq CC exit DR /2. CRq CC exit /2.]
CRq CC exit DI /3. CRq CC exit DI /4.]
[CRq CC exit DI /3. CRq CC exit DI /4. CRq CC exit DR /2.]
[CRq CC exit DI /3. CRq CC exit DI /4. CRq CC exit DR /2.]
[CRq CC exit DI /3. CRq CC exit DI /4. CRq CC exit DR /2. CRq CC exit /2.]
[CRq CC exit DtI /3.]
[CRq CC exit DtI /3. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DR /2. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DtR /4.]
[CRq CC exit DtI /3. CRq CC exit DtR /4. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DtR /4. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DtR /4. CRq CC exit DR /2. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DI /4.]
[CRq CC exit DtI /3. CRq CC exit DI /4. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DI /4. CRq CC exit DR /2.]
[CRq CC exit DtI /3. CRq CC exit DI /4. CRq CC exit DR /2. CRq CC exit DR /2]

Do you like to continue? (y/n): n

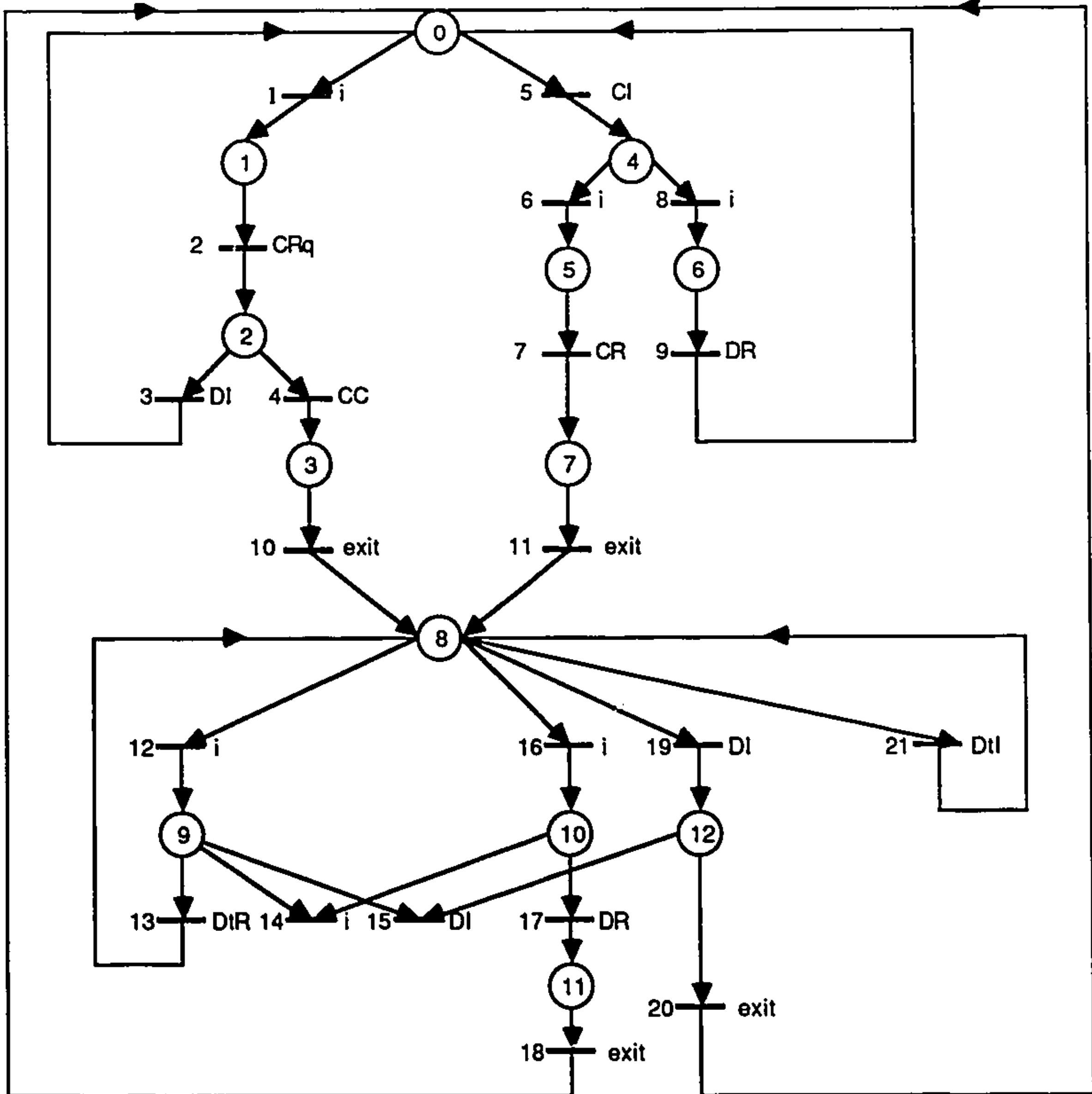


Figure 5.5. Indeterministic Petri-net representation of LOTOS specification for the Class 0 Transport Protocol

5.3 APPENDIX: THE LOTOS SPECIFICATION HANDLER [BOL87]

The specification *Handler* of the Class 0 Transport Protocol has five processes: *Connection-phase, Calling, Called, Data-phase and Termination-phase.*

```
specification Handler [ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd]: =
noexit
behaviour
Handler [ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd]
where
Process Handler [ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd] : =
(Connection-phase [ConReq, ConInd, ConRes, ConCnf, DisReq, DisInd]
>>
( ( Data-phase [DatReq, DatInd] )
[>
(Termination-phase [DisReq, DisInd])
)
>> Handler [ConReq, ConInd, ConRes, ConCnf, DatReq, DatInd, DisReq, DisInd])
where
process Connection-phase [CRq, CI, CR, CC, DR, DI]: =
(i; Calling [CRq, CI, CR, CC, DR, DI])
[]
(Called [CRq, CI, CR, CC, DR, DI])
where
      process Calling [CRq, CI, CR, CC, DR, DI] : =
      CRq; ((CC; exit)
      []
      (DI; Connection-phase [CRq, CI, CR, CC, DR, DI]))
      endproc
      process Called [CRq, CI, CR, CC, DR, DI] : =
      CI; ((i; CR; exit)
      []
      (i; DR; Connection-phase [CRq, CI, CR, CC, DR, DI]))
      endproc
endproc
process Data-phase [DtR, DtI] : =
( i; DtR; Data-phase [DtR, DtI] )
[]
(DtI; Data-phase [DtR, DtI])
endproc
process Termination-phase [DR, DI] :=
```

```
( i; DR; exit)
[]
(DI, exit)
endproc
endproc
endspec
```

Chapter 6
SUMMARY AND FUTURE WORKS

In this thesis, we have reported our research works for resolving the problems of automatic generation of test sequences for indeterministic systems based on a formal description technique. Our results include the proposal of an indeterministic Petri-net model for describing distributed systems which exhibit indeterminism, development of a metric for measuring the degree of indeterminism of a test sequence, proposal of a fairness model for conformance testing, design of an algorithm for generating test sequences of controlled lengths and for calculating the degrees of indeterminism of the test sequences, and application of the algorithm to LOTOS. Besides these theoretical developments, a substantial amount of work is involved in the implementation of a graphical system for realizing these results in the Sun Workstation.

In the literature, transformation of LOTOS specifications to Petri-nets is mainly for the purpose of execution [GAR90] and verification [MAR89, GAR90] of the specification. Our application of indeterministic Petri-nets to LOTOS is for protocol testing and seems to be the first attempt of this kind.

In protocol testing, the indeterminism issue is seldom taken into consideration. In this thesis, we propose a few new methods for solving this problem. Wezeman discussed indeterminism in the CO-OP method [WEZ89]. However, this method is for producing a Canonical Tester. It is not practical and is quite different from the traditional approach of generating test sequences. In contrast, we include the treatment of indeterminism into the test sequence generation process. A metric - degree of indeterminism - has been developed for measuring indeterminism of a test sequence. With the metric, a degree of indeterminism is produced in the generation of a test sequence for a specification with indeterminism. It is used to estimate the number of times needed in applying the test sequence to the IUT. Our algorithm - IPNTEST - is iterative. It takes indeterminism into consideration when generating test sequences. It generates test sequences with

length controlled by the user and also generates the degrees of indeterminism for the test sequences.

Algorithm IPNTEST has two limitations: 1) IPNTEST generates all possible test sequences. It does not do test sequence selection. 2) It is only applied to basic LOTOS. That means that only control flow is considered in the generation of test sequences.

Future works for this research include: 1) modification of Algorithm IPNTEST to provide the option for selective test sequence generation, 2) involvement of data in test sequence generation, and 3) generation of concurrent test sequences. More elaboration of these points follows.

Since the stepwise test sequence generation process of Algorithm IPNTEST is iterative and extends over the Petri-net in a breadth-first-search manner, modification for selective testing sequence generation will be quite straightforward. Selective testing can be done in two ways: a) Generating test sequences for a sub-specification designated by a user. The graphical system UO-GLOTOS will offer such a flexibility to a user, i.e. a user can choose any part of a specification which is displayed on the screen. b) Generating test sequences for certain specified purposes. However, the challenge is in the development of a theory and the corresponding selection criteria to back up the selective testing purpose. A Petri-net based model for test coverage is presently being devoloped under Cheung's guidance.

Considerably more work will have to be done in order to include data flow in test-sequence generation. Since the Galileo approach of nesting all LOTOS operators into a single net is probably too complicated for handling some of the operators in LOTOS, such as disable, etc., a more desirable approach may be using Cheung's modular Petri-net representation of LOTOS [CHE89], i.e., extension of Petri-nets to include features for handling data flow.

In our test sequence generation method, the traditional method in which concurrent actions are treated as a set of sequential actions is adopted for handling concurrency property of a specification. As mentioned in Chapter 1, a different approach for handling concurrency is proposed by Pomello [POM85] for specifications described in Petri-nets. In this method, the transitions firable at a marking are represented by a single set without distinguishing the order of

executions. The order of the executions may be decided in the testing. Accordingly, the concurrency property is preserved in the generated test sequences (we may called them "concurrent test sequences"). Since our test sequence generation method is based on Petri-nets, the modification for generating concurrent test sequences would not be difficult. The important thing is to find the advantage for using the concurrent test sequences.

Another result of this thesis which needs further research is the concept of fairness and the concept of degree of indeterminism. The 'fairness' concept has been in use in several areas of protocol research, such as synthesis [GOU84]. In this thesis, fairness is used as an approach for making verdicts on test sequences. It is based on a metric called 'degree of indeterminism' for measuring indeterminism. The metric is in turn based on a uniform probability distribution for resolving indeterminism. Though not fully developed (for not being the main objective of the thesis), the two concepts as used here allow us to present a practical and efficient approach for solving problems in test sequence generation for indeterministic systems. Our approach may be generalized to dealing with other probabilistic distributions and will be useful for developing a theory for test performance analysis or a probabilistic model for conformance testing.

Our research serves just as a starting point in the pursuits for dealing with indeterminism in conformance testing. Hopefully more fruitful results will follow.

REFERENCES

[ALD89] R. Alderden, "COOPER the compositional construction of a canonical tester", Proc. 2nd International Conference on Formal Description Techniques for Distributed systems and Communications Protocols, Vancouver, Canada, (1989), pp. 13-18.

[BER89] B. Baumgarten, A. Giessler and R. Platten, "Test derivation from net models", Proc. 2nd International Workshop on Protocol Test Systems, Belin, Germary, (1989), pp 91-109.

[BOL87] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems 14 (1987) pp. 25-59.

[BRI86] E. Brinksma, G. Scollo and C. Steenbergen, "LOTOS specification, their implementations and their tests", Proc. Sixth IFIP International Symposium on Protocol Specification, Testing, and Verification, Montreal, Canada, (1986), pp. 10.1-10.38.

[BRI88] E. Brinksma, "A theory for the derivation of tests", Proc. Eighth IFIP International Symposium on Protocol Specification, Testing, and Verification, Atlantic City, USA, (1988).

[CHE88] T.Y. Cheung and Y. Zhu, "A Petri-net-based method for specifying distributed systems and deriving executable graphical LOTOS and ESTELLE", Tech. Report TR-88-04, Dept. of Computer Science, Univ. of Ottawa. (Feb. 1988).

[CHE89] T.Y. Cheung, Y.C. Ye, X. Ye and G.Q. Wang, "UO-GLOTOS: a syntax/system for representing, editing and translating graphical LOTOS", Proc. 2nd International Conference on Formal Description Techniques for Distributed systems and Communications Protocols, Vancouver, Canada, (1989), pp. 33-48.

[CHO78] T.S. CHOW, "Testing software designs modeled by Finite-State machine", IEEE Trans. on Software Engineering, Vol.SE-4, No.3, (1978), pp.178-187.

[EHR85] H. Ehrig and B. Mahr, Fundamentals of Algebraic Specification 1, Springer-Verlag, Berlin, 1985.

[FAV86] J.P. Favreau and R.J. Linn, "Automatic generation of test scenarios from protocol specifications written in Estelle", Proc. Sixth IFIP International Symposium on Protocol Specification, Testing, and Verification, Montreal, Canada, (1986).

[GAR90] Garavel, H. and Sifakis, J., " Compilation and verification of LOTOS specification", Proc. Eighth IFIP International Symposium on Protocol Specification, Testing, and Verification, Atlantic City, (1988), pp.359-376.

[GOU84] Gouda, M.G. and Yu, Y.T., "Synthesis of communicating finite state machines with guaranteed progress", IEEE Trans. on Communications, Vol. COM-32, No. 7 (1984), pp. 779-788.

[GUE89] D. Gueraichi and L. Logrippo, "Derivation of test cases for LAPB from a LOTOS specification", Proc. 2nd International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, Vancouver, Canada, (1989), pp. 489-508.

[GUI88] R. Guillemont, M. Haj-hussein, L. Logrippo. "Executing large LOTOS specifications", Protocol Specification, Testing, and Verification VIII, edited by S. Aggarwal and K.Sabnani, (1988), pp 399-410.

[HOA85] C.A.R. Hoare, Communicating Sequential Processes, Prentice-hall Intl., 1985.

[HOP79] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Language, and Computation, Addison-Wesley (1979).

[ISO8807] "Information processing system - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour", ISO publication (1988).

[ISO9646] ISO/TC97/SC21, OSI Conformance Testing Methodology and Framework - Parts 1-5, ISO 2nd DP9646-1 revised test, edited by D. Rayner, Vancouver, (1987).

[KAR88] G. Karjoth, "Implementing Process Algebra specifications by state machines", Proc. Eighth IFIP International Symposium on Protocol Specification, Testing, and Verification, Atlantic City, (1988).

[KOH78] Z. Kohavi, Switching and Finite Automata Theory, New York: Mcgraw-Hill, (1978).

[MAR89] S. Marchena and G. Leon, "Transformation from LOTOS to Galileo nets", Formal Description Techniques, Elsevier Science Publishers, (1989), pp. 217-230.

[MIL80] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[MIL84] R. Milner, "A complete inference system for a class of regular behaviors", Journal of Computer and System Sciences 28, 1984, pp. 439-466.

[PAV89] J. Pavon and S. Marchena, "Analysis of LOTOS specification with Petri Nets".

[PET81] J.L. Peterson, Petri Net Theory and the Modelling of Systems, Prentice-Hall, 1981.

[POM85] L. Pomello, "Some equivalence notations for concurrent systems. An overview", in: Advances in Petri Nets 1985, edited by G. Rozenberg, LNCS, Vol. 222, (1985), pp. 381-400.

[REP89] T. Resp and T. Teitelbaum, "The synthesizer Generator", Springer Verlag, 1989.

[SAN86a] C. Sanchez, "Galileo, Model Language, and Tools", Electrical Communications Vol. 60, no. 3-4, (1986), pp. 216-224.

[SAN86b] C. Sanchez, "Galileo: Model Definition", Alcatel SESA internal report, 86-TR-8403, (1986).

[SID89] D. Sidhu and T-K Leung, "Formal methods for protocol testing: a detailed study", IEEE Trans. on Software Engineering, Vol.15, No.4, (1989), pp.413-426.

[TRI89] P. Tripathy and B. Sarikaya, "Test generation from protocol specification", Proc. 2nd International Conference on Formal Description Techniques for Distributed systems and Communications Protocols, Vancouver, Canada, (1989), pp. 455-468.

[URA87a] H. Ural, "A test deriva.'on method for protocol conformance testing", Proc. Seventh IFIP International Symposium on Protocol Specification, Testing, and Verification, Switzerland, (1987).

[URA87b] H. Ural, "Test sequence selection based on static data flow analysis", Computer Communication, Vol.10, No.5, (1987), pp.234-242.

[URA88] H. Ural, B. Yang and R.L. Probert, "A test selection method for protocol specified in Estelle", Technical Report TR-88-18, Dept. of Computer Science, University of Ottawa, (1988).

[WEZ89] C.D. Wezeman, "The CO-OP method for compositional derivation of conformance testers", Proc. Ninth IFIP International Symposium on Protocol Specification, Testing, and Verification, Armsterdam, the Netherlands, (1989).

[WU89] J.P. Wu and S.T. Chanson, "Test sequence derivation based on external behaviour expression", Proc. 2nd International Workshop on Protocol Test Systems, Berlin (West), Germany, (1989).