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.

Canada

A Selective Test Sequence Generation Method for Protocols
(with application to LOTOS)

by

Shenyu Ren

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

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

ISBN 0-315-85808-7

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGEMENT

I would like to express my foremost gratitude to my supervisor, Dr. To-yat Cheung, for his considerate guidance, constructive advice and valuable time throughout my graduate studies. His instruction for reviewing the drafts of this thesis has greatly improved its contents and presentation.

I acknowledge with gratitude the financial support provided by Natural Sciences and Engineering Research Council of Canada, Telecommunications Research Institute of Ontario and Canadian Institute of Telecommunications Research.

I would like to thank the Protocol Research Group for providing an excellent research environment during the entire period of my study. In particular, I owe many thanks to Xing Huang for his kindness and assistance. We have had many inspiring and fruitful discussions. His useful comments on the early versions of part of the thesis were very much appreciated. I would also like to thank Xingming Ye, Yucheng Ye, Guoqiang Wang and Youwen Wu for their initiative ideas and helpful comments.

My parents and brother deserve special thanks. Their love, understanding and support have given me the courage to attain my education at this level.

ABSTRACT

This thesis proposes a new approach for generating selective test sequences from a LOTOS specification. First, based on the functional characteristics and semantic properties of the LOTOS operations, a criterion called *operational coverage* is developed, whereby, each of the LOTOS operations is assigned a mandatory set of sequences involving the actions in its operands. In general, these action sequences are not executable. A method called SELECTEST is then proposed for generating executable test sequences which satisfy the coverage criterion. SELECTEST first determines the cover for the specified operation from which a set of guides are obtained, each denoted as a possibly nonexecutable rooted path lying in the hierarchy of the LOTOS specification and passing through the operator. The guides are used to greatly speed up the search for the executable test sequences over the Petri-net representation transformed from the LOTOS specification. The coverage and method have also been generalized to the case of covering a set of LOTOS operations. The application to the Alternating Bit Protocol is included to illustrate the approach.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION AND FUNDAMENTALS

1.1 CONFORMANCE TESTING FOR DISTRIBUTED SYSTEMS

With the wide-spread acceptance of ISO's (The International Organization for Standardization) OSI Reference Model and its standardized protocols, the ISO protocols are presently being implemented by many communications software manufacturers. In order to achieve global interworking among heterogeneous systems, protocol conformance testing becomes an essential quality control process.

Conformance testing is an assessment process for determining the extent to which an implementation of a standard conforms to its specification. The assessment is carried out in a manner consistent with the implementor's declaration of the implemented capabilities stated in the Protocol Implementation Conformance Statement (PICS). The standard conformance test methodology includes the processes of abstract test suite specification, test realization, and conformance assessment. Their detailed description can be found in [RAY87, ISO9646].

Conformance testing involves applying test suites. Test suites are used to describe control and observation of the implementation under test and assign verdicts to test outcomes. A test suite comprises a number of test cases and is associated with a unique standardized test purpose. The test cases may be grouped into test groups which in turn may be grouped into larger test groups. The test cases may also be decomposed into test steps and the test steps into test events (or test sequences). Test suite generation methods, especially those based on formal specifications, have become an area of active research and development.

Formal specification techniques render the specifications of OSI standards unambiguous, precise and complete, and provide a well-defined basis for verification and conformance testing.

LOTOS (the Language fOr Temporal Ordering Specifications) is a standardized formal technique for specifying distributed systems [ISO8807]. A few methods for generating test cases from LOTOS specifications have been proposed in the literature. A review on some of these

methods is presented in Chapter 2. Most of these methods can generate all test cases and have no criteria for test selection. Since exhaustive testing is impossible, generating selective test cases from specifications is more important. Though some of those methods can generate test sequences covering subtours or cycles involving specified input or output access points, such selection criteria may not reflect the nature of LOTOS specifications. In this thesis, we focus on the problem of selective test sequence generation for LOTOS specifications. First, an operational coverage for LOTOS specifications based on the semantics of the LOTOS operations is proposed. A method called SELECTEST for selectively generating executable test sequences is then developed based on the coverage criterion and a Petri-net representation of LOTOS.

The rest of the chapter includes three more sections. LOTOS and Petri-nets are briefly introduced in Section 1.2. Motivation and contributions for this thesis are presented in Section 1.3. An outline of the thesis is given in Section 1.4.

1.2 BRIEF INTRODUCTION TO LOTOS AND PETRI-NETS

The specification of a communications service or protocol must be consistent, complete, unambiguous, precise and concise. Specification in a natural language may be easier to understand, but is often lengthy and ambiguous. Also, it cannot be used as a basis for checking the completeness and correctness of a protocol design. Hence, formal and semi-formal specification techniques have been developed. Subsections 1.2.1 and 1.2.2 briefly describe two such techniques which are related to the work in this thesis, namely, LOTOS and Petri-nets.

1.2.1 Basic LOTOS and its Graphical Representation UO-GLOTOS

LOTOS is one of the two formal description techniques (FDTs) developed by ISO for the formal specification of open distributed systems and, in particular, for network protocols.

Full LOTOS consists of two parts: the *control part* which provides the facilities for describing the external (observable) and internal (unobservable) behaviours of a system and the *data part*

which deals with the description of data types. Basic LOTOS, the control part of full LOTOS, is considered in this thesis. For the rest of the thesis, LOTOS means basic LOTOS.

LOTOS has two representations, textual [BOL87] and graphical [ISO89, CHE89]. Graphical LOTOS enhances the clarity and readability of textual LOTOS. The following brief introduction of textual LOTOS is based on the description of T. Bolognesi, et al. [BOL87] and the graphical LOTOS is based on Cheung's UO-GLOTOS [CHE89].

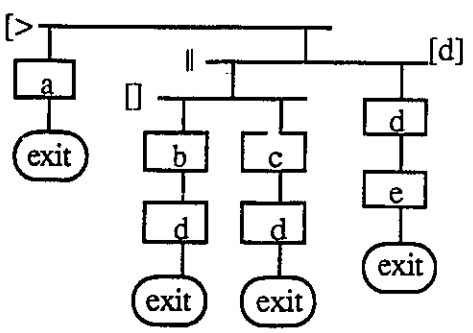
In a LOTOS specification, a distributed system is described via a hierarchy of process definitions. The essential components of a process definition are its *behaviour expressions*, which describe the possible *actions* of the system and are built by applying LOTOS operators on other behaviour expressions. A complete list of the LOTOS constructs is given in Figure 1.1 (Symbols 'B', 'B1' and 'B2' stand for behaviour expressions). For the details of the syntax and semantics of these constructs, we refer the reader to [ISO8807].

UO-GLOTOS is an integrated software environment for LOTOS research [CHE89,90,91a]. It is based on graphical-LOTOS and Petri-nets. It includes many tools and a model under which a LOTOS expression is represented as a hierarchical structure, where the root represents its 'outermost' operation and the subhierarchies represent its operands. Such a hierarchy reflects the main syntactic characteristics of textual LOTOS. The constructs of UO-GLOTOS can also be found in Figure 1.1.

The following example is a LOTOS specification in both textual and UO-GLOTOS representation.

Textual representation: S = (a;exit) [> ((b;d;exit [] c;d;exit) |[d]| (d;e;exit))

UO-GLOTOS representation:



The following notation is used in UO-GLOTOS (Figure 1.1):

B — a meta-symbol for the graphical representation of the behaviour expression B

< r > — Cross-reference number which are used as cross-reference among processes.

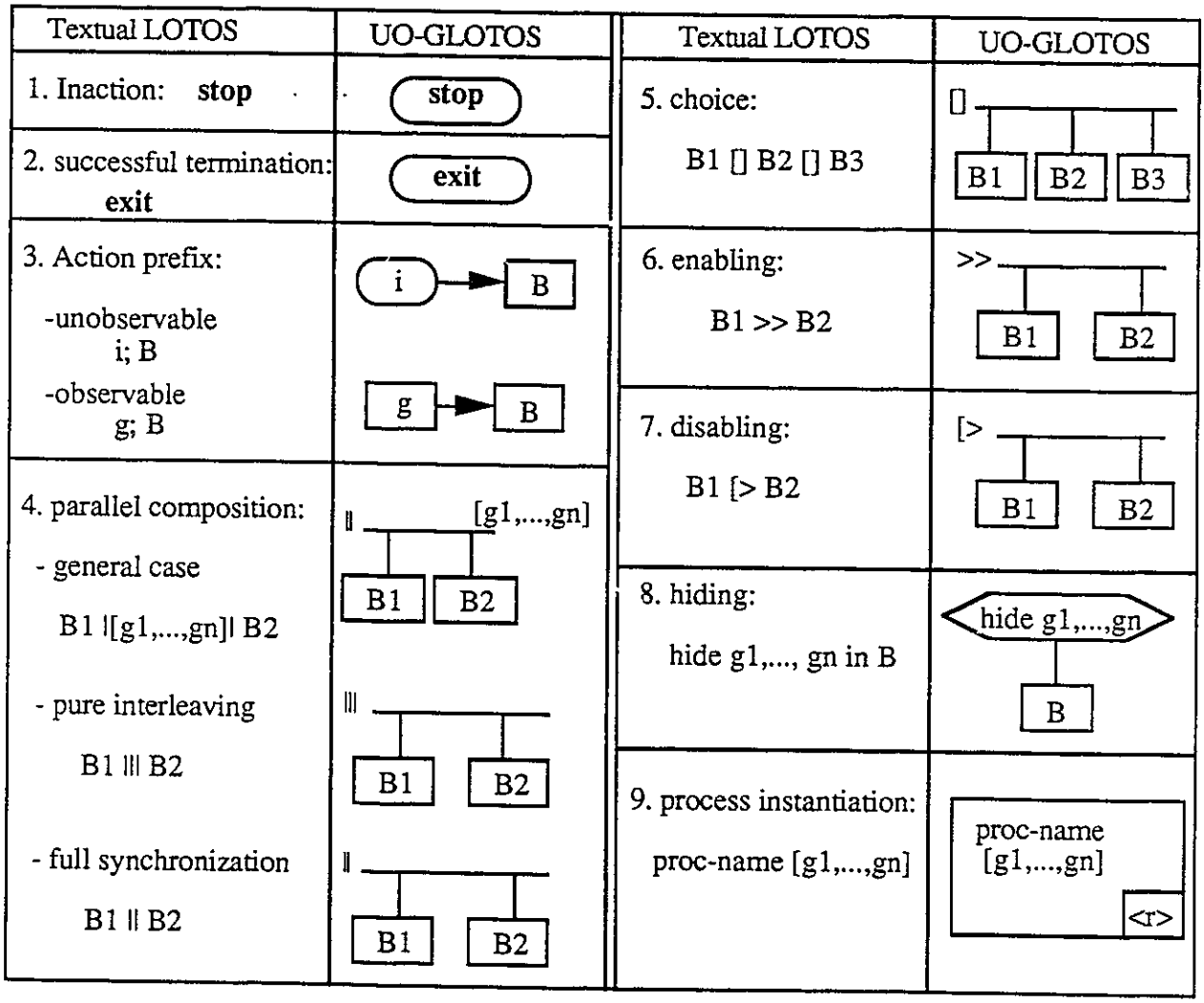


Figure 1.1. Constructs of the textual LOTOS and UO-GLOTOS

1.2.2 Petri-nets

The computational part of the method for selective test sequence generation proposed in this thesis is based on a Petri-net representation of LOTOS. A Petri-net [PET81] is a semi-formal

technique for specifying distributed systems. It is especially powerful for representing concurrent behaviours. Its definition is given below.

Definition 1.1. *(Petri-net)*

A Petri-net is a four-tuple $C = <P, T, I, O>$, where

P: A finite set of places;

T: A finite set of transitions, where $P \cap T = \emptyset$;

I: T -> P is the input function, a mapping from transitions to sets of places and;

O: T -> P is the output function, a mapping from transitions to sets of places.

A Petri-net graph consists of two types of nodes: a circle node representing a place, and a bar node representing a transition. Directed arcs may connect nodes of different types, but not nodes of the same type. For a transition t, an arc from a place to t is called an *input arc*, and an arc from t to a place is called an *output arc*. A place connected to t by an input arc is called an *input place* of t, and a place connected by an output arc is called an *output place* of t.

Usually, places and transitions play different roles. A transition represents an action, whereas its input and output places represent the preconditions of firability and postconditions after firing of the action, respectively. Fulfillment of a precondition specified by a place is indicated by the existence of a token in that place. In order to fire a transition, every input place of the transition must contain at least one token. After the transition has been fired, one token is removed from each of its input places and one token is added to each of its output places. A *marking* (Definition 1.2) is the representation of a distribution of tokens in the places, and the marking of a Petri-net at an initial moment is called the *initial marking*. Firing a transition in a marking μ results in a new marking μ', we say the μ' is *immediately reachable* from μ. If μ' is immediately reachable from μ, and μ'' is immediately reachable from μ', μ'' is *reachable* from μ.

Definition 1.2. *(The marking of a Petri-net)*

A *marking* μ of a Petri-net $C = <P, T, I, O>$ is an n-vector denoting a distribution of tokens in the places. In notation, $\mu = (M(p1), M(p2), ..., M(pn))$, where $M(pi)$ is the number of tokens residing in place pi, $i = 1, 2, ..., n$.

In the literature, the general reachability problem in the analysis of a Petri-net design has been studied. The problem can be stated as follows: Given the markings μ and μ', is μ' reachable from μ ? In this thesis, our method for generating selected test sequences over a Petri-net requires the solution for a special case of the general reachability problem. The details are given in Chapter 4.

Several means of transforming a LOTOS specification to a Petri-net representation are reviewed in Chapter 2.

1.3 MOTIVATION AND CONTRIBUTIONS OF THIS THESIS

In exhaustive testing, the number of test cases may be extremely big or even infinite. Since execution of all the test cases usually is experimentally infeasible, selecting a proper subset of the test cases for execution is more effective and economical.

Test selection should not be done arbitrarily. A strategy that satisfies certain conditions and requirements has to be worked out first, so that the selected test cases are useful for the conformance testing, in the sense that there is a big chance of detecting the types of errors as desired.

Selection may be done in the following two approaches:

(1) Selection is made from an exhaustive set of test cases already generated by some means and based on the test purposes or external knowledge, such as the user's experience and heuristics. These procedures cannot be reflected in a formal or semi-formal specification and are therefore not easy for automation. The method proposed by Brinksma, et al. [BRI91] belongs to this approach.

(2) Selection is made during the process of test case generation under the guidance of a criterion. This avoids the overproduction of test cases. Included in this approach are the technique based on state-identification (Subsection 2.3.1) and the all-DU-path and OI-path methods (Subsection 2.3.2) [SID89, URA87,91].

As far as we know, all methods having appeared in the literature for selectively generating test cases are based on the specification models of finite state machines or their extensions (such as ESTELLE). In this thesis, we propose a selective test sequence generation method based on LOTOS specifications. No methods for a similar purpose have been reported in the literature.

This thesis has two main contributions:

(1) An operational coverage for LOTOS specifications is proposed as a criterion for test sequence selection.

Based on the functional characteristics and semantic properties of the LOTOS operations, a criterion called operational coverage is developed. A cover for a single LOTOS operation is defined over the hierarchy of the LOTOS expression associated with it. It consists of a mandatory set of (possibly nonexecutable) action sequences characterized by the semantics of the operation and depending on the expressions contained in its operands.

The coverage has also been generalized to the case of multiple LOTOS operations. It includes the following special types of coverage: all-parallel, all-action-prefix, all-disable, all-enable and all-choice.

(2) A method called SELECTEST is proposed for selectively generating executable test sequences from LOTOS specifications.

Based on the coverage criterion mentioned above, SELECTEST first derives a set of "guides", each denoted as a (possibly nonexecutable) rooted path (i.e., an action sequence) in the LOTOS hierarchy and passing through the operator under consideration. It then transforms the specification into a Petri-net and applies the guides to search for some appropriate executable test sequences over the Petri-net.

In Petri-net, the problem of finding an executable test sequence in which a given nonexecutable action sequence is embedded becomes to find an executable transition sequence in which a nonexecutable transition sequence is embedded. It is a special case of the general reachability problem in Petri-net analysis. To solve this problem, we proposed a method by using the nonexecutable transition sequence as a guide to simplify the derivation of an executable one.

Our method has been applied to generate test sequences covering five operations of the Alternating Bit Protocol (ABP).

1.4 OUTLINE OF THE THESIS

The rest of the thesis is organized as follows: Chapter 2 contains surveys on three issues: Petri-net representations of LOTOS, methods for protocol selective testing, and test suite generation from LOTOS specifications. These provide the background for our operational coverage and new method for selective test sequence generation. It includes the rules of transforming a LOTOS specification to a Petri-net. Chapter 3 presents one of the main contributions of this thesis: the definitions of operational coverage for single and multiple LOTOS operations. The latter includes special types of coverage, such as all-parallel, all-action-prefix and all-choice, etc. Chapter 4 presents a method for generating selected executable test sequences, the other main contribution and core part of this thesis. The method first determines the cover for the specified operation. From the cover, a set of guides used to speed up the search for the executable test sequences over a Petri-net representation of the LOTOS specification are obtained. Lastly, in Chapter 5, the details of applying the method to the Alternating Bit Protocol are given. Chapter 6 includes some concluding remarks and possible future work of our research.

Chapter 2

REVIEW ON PETRI-NET REPRESENTATIONS OF LOTOS, METHODS FOR PROTOCOL SELECTIVE TESTING, AND TEST SUITE GENERATION FROM LOTOS

2.1 INTRODUCTION

In this thesis, a criterion called the operational coverage of LOTOS specifications is proposed for test sequence selection. Based on this criterion, a method called SELECTEST for selective test sequence generation from LOTOS specifications is also proposed. As background knowledge for these proposals, existing Petri-net representations of LOTOS, methods of protocol selective testing and test suite generation from LOTOS specifications are reviewed in this chapter. First, we give a brief overview of them. Details are presented in Sections 2.2, 2.3 and 2.4.

In the literature, there exist at least four Petri-net representations of LOTOS. The model proposed by Cheung, et al. [CHE91b] describes the control flow of LOTOS. It is a variation of a model for describing both data flow and control flow [CHE88]. Galileo net [MAR89] and the model used in [GAR90] are similar to Cheung's model, except that some of the constructs, such as DISABLE, etc. are represented in a different way. The former has been used for generating test sequences [CHE91a] and the latter for compiling and verifying LOTOS specifications [GAR90]. The Petri-net model developed by Barbeau, et. al. [BAR90] is used mainly for analyzing LOTOS specifications. As pointed out in [BAR90], there is some theoretical limitation in using Petri-nets to represent some forms of recursive processes in a LOTOS specification. In practice, however, this does not seem to create a big obstacle. Though any Petri-net representation of LOTOS can be used in SELECTEST, for illustration purpose, our examples are based on Cheung's representation [CHE91b].

Since exhaustive testing is practically impossible, research on software testing in the last decade has been focused on developing strategies for selecting some of the representative test cases. In conventional software testing, one class of strategies is called *structural testing* which is

based on the control or data flow of programs. The well-known selection criteria based on control flow are all-statement, all-branch and all-path coverage [MIL81], which execute every statement, branch or path in a program flow chart at least once, respectively. The methods based on data flow diagram, aiming at covering all DU-paths [RAP85] and OI-paths [URA88], have also been proposed for selecting test sequences. In a program, a DU-path associates a point where a variable is defined with a point where its value is used. The all-DU-path strategy covers all such associations at least once. The OI-path strategy analyzes the effects of program inputs on program outputs and requires the examination of the critical association between every input variable and the output variables that are affected by this input variable. Another class of selective testing methods, under the generic name '*functional testing*' [HOW86], is based on criteria which cover a specific set of 'functional' ingredients, such as operations, strategic control sections, etc., of the implementation under test (IUT).

Some of the selective testing strategies mentioned above have been used and adopted for protocol conformance testing which aims at demonstrating the conformance of the IUT with its specification [SID89, URA87, URA91, SAR87]. A review on these methods will be presented in Section 2.3.

LOTOS is a formal description technique for specifying distributed systems [ISO8807]. We know of six test case generation methods for a LOTOS specification: The first method [WU89] generates test sequences based on specifications transformable into external behaviour expressions (EBE). This approach ignores internal actions and thus partially the indeterministic nature of distributed systems. Though being mentioned as applicable to LOTOS, the method for transforming LOTOS specifications into EBE does not seem to have been reported in the literature. The second method [TRI89] is based on a transformation of LOTOS to a labelled transition system called CHART. CHART is then transformed into a data flow diagram, based on which test cases are then derived by finding subtours using well-known graph traversal algorithms. However, the generated test sequences are not always executable. The third method [GUE89] simply makes use of a LOTOS interpreter to generate execution paths of a LOTOS specification. Test sequences are

then selected from these paths. The fourth one, called the CO-OP method [WEZ90], constructs canonical testers which, if run currently with the IUT without resulting in an unexpected deadlock, will imply that the IUT conforms with the specification. The fifth method [CHE90] is based on a graphical LOTOS tool called UO-GLOTOS. The EXECUTOR, one of the subsystems of this tool, provides four modes of operations by which a user can create desirable test sequences in different modes. The sixth method [CHE91] is based on a transformation of a LOTOS specification to a Petri-net. It includes a probabilistic model for conformance testing, a metric for measuring the degree of indeterminism of test sequences and an algorithm for generating test sequences and their degrees of indeterminism. Among the six methods mentioned above, the first two can generate test sequences covering subtours or cycles involving specified input or output access points. Such selection criteria are widely used in software testing, but do not necessarily reflect the semantics of LOTOS. Though providing a certain degree of operational assistance, the selection processes in the third and fifth methods [GUE89,CHE90] are mainly a human 'walkthrough' techniques. The sixth method can generate all test sequences up to a specified length and is exhaustive in nature.

The rest of the chapter is organized as follows: In Section 2.2, Cheung and Zhu's Petri-net representation of a LOTOS specification [CHE91b] is described. In Section 2.3, some of the selective testing methods for protocol conformance testing [SID89, URA87, URA91, SAR87] are reviewed. In Section 2.4, the methods mentioned above for test suite generation from LOTOS specifications [WEZ89, WU89, TRI91, CHE91, GUE89, CHE90] are presented.

2.2 PETRI-NET REPRESENTATION OF LOTOS BY CHEUNG AND ZHU

In this thesis, Cheung and Zhu's Petri-net representation for LOTOS (PN net) is used as the computational basis for generating selected test sequences. Though any Petri-net representation of LOTOS discussed in Section 2.1 can be used for our method SELECTEST, for illustration purpose, the PN net is used because of its hierarchical and modular structures.

The PN net is an extension of the Petri-net with labels representing the LOTOS operators, external actions, internal action 'i', inaction 'stop' and special action 'exit'. In its representation, each LOTOS process is represented by a Petri-net and the LOTOS operations on the processes are represented by adding transitions to connect the Petri-nets of the various processes. Different operations will use different ways of connection.

Definition 2.1. *(PN net)*

A PN net is an 8-tuple S = <P, T, F, L, l, pi, px, ps>, where

P: a finite set of places;

T: a finite set of transitions;

F: a set of arcs, i.e., F = (P X T) ∪ (T X P);

L: a set of labels;

l: a labelling function, l: T -> L;

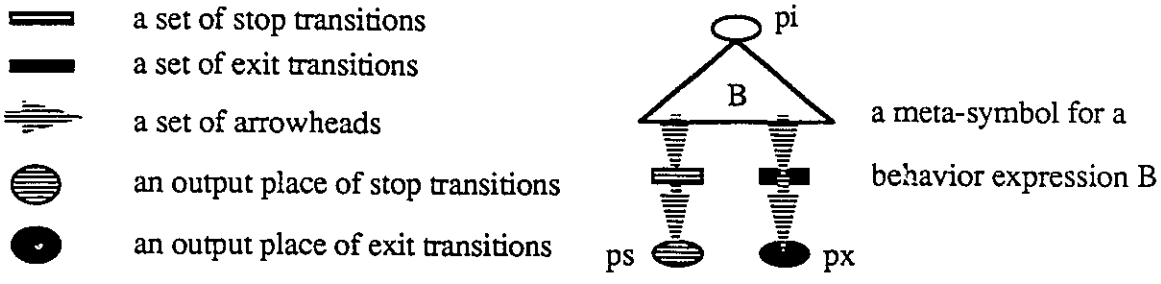
pi: the initial place;

px: the exit place;

ps: the stop place.

The transformation of a LOTOS specification to its PN net is carried out according to a set of rules for representing each of the LOTOS constructs as a Petri-net. These rules are shown in Figure 2.1 and formally described in Appendix A.

Notation 2.1. (The following notation is used in Figure 2.1)



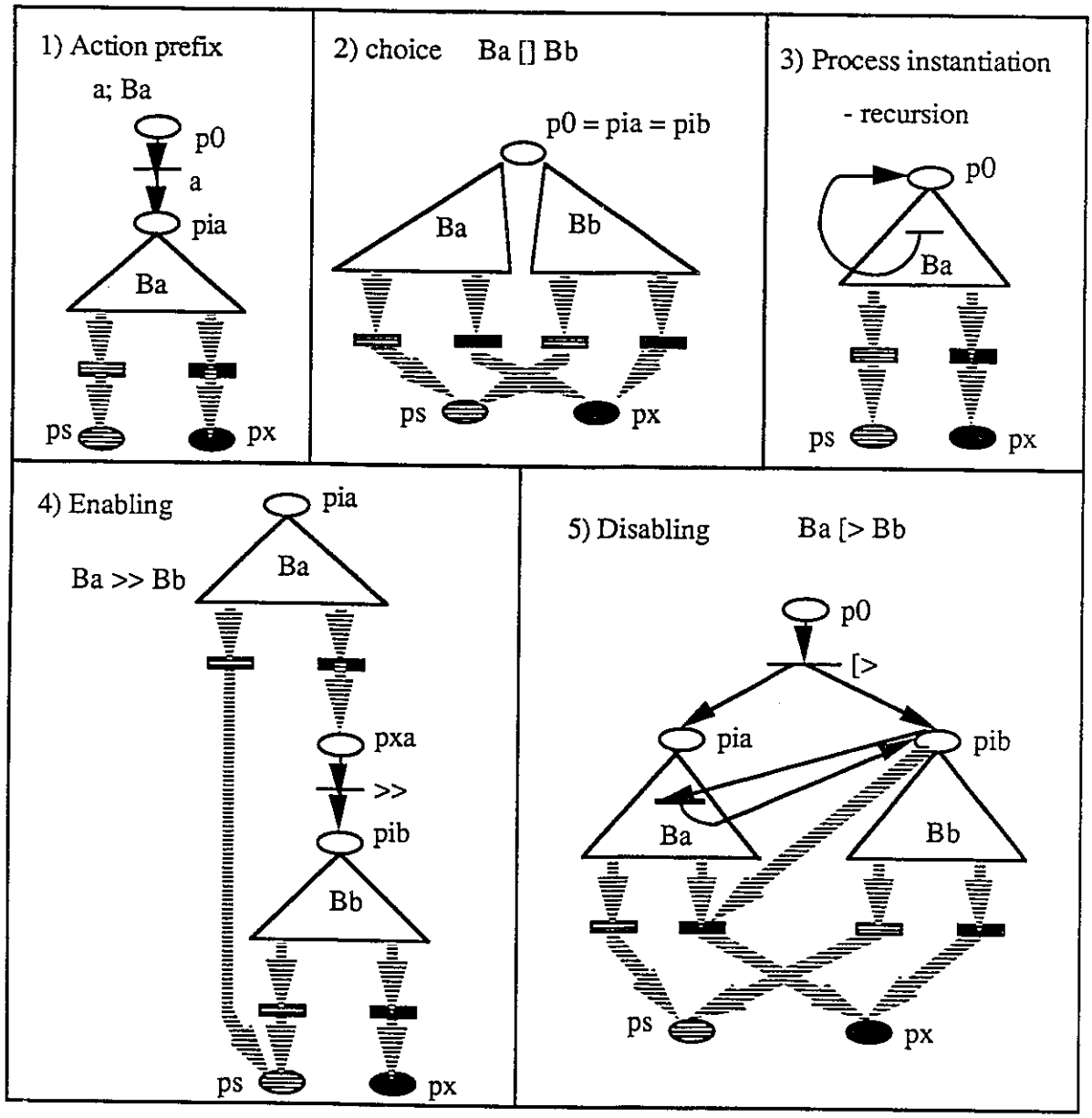


Figure 2.1. Rules for transforming LOTOS constructs to Petri-nets [CHE91b].

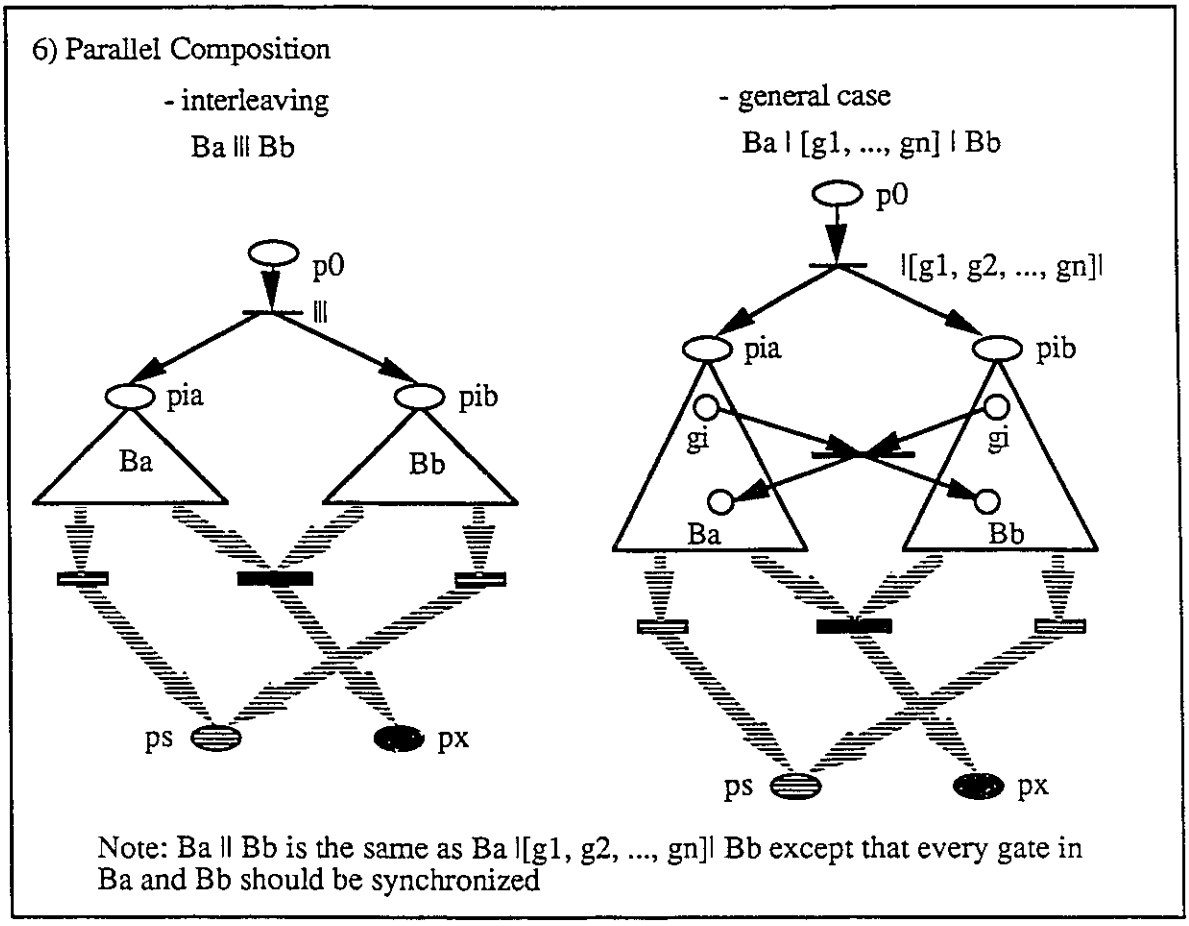


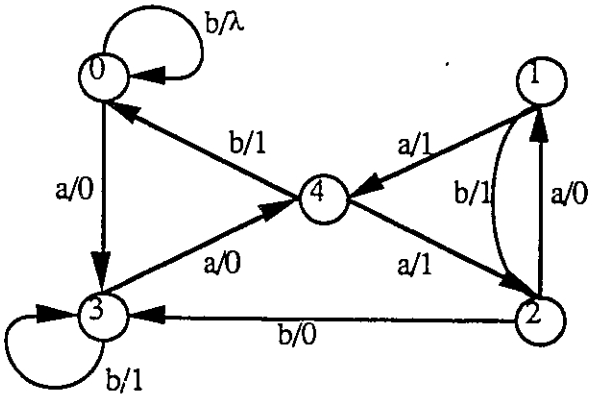
Figure 2.1. Rules for transforming LOTOS constructs to Petri-nets (cont.) [CHE91b].

2.3 SELECTIVE TESTING METHODS FOR PROTOCOLS

In this section, four well-known approaches for protocol selective testing are reviewed. The first one combines the all-branch coverage strategy in software testing and the state-identification techniques in fault-tolerance computing. It is based on a minimal and deterministic FSM [SID89, AHO88]. The second and third approaches are based on the All-DU-path and OI-path criteria. They have been applied to ESTELLE [URA87, URA91], a formal specification language defined by ISO. In the fourth one, a functional testing technique is applied to ESTELLE specifications by Sarikaya and Bochmann [SAR87].

2.3.1 The Approach Based on All-branch Testing and State-identification

From a minimal, deterministic FSM specification, this approach is intended to generate a sequence of inputs (and possibly outputs), which is capable of verifying all specified branches (i.e., transitions in FSM) when applied to the IUT. Testing each of the branches (specified as a transition t from state s to state s') proceeds as follows: a preamble first brings the IUT from the initial state to s. Then, transition t is executed at s, bringing the IUT to a state, say q. The branch is considered as having been correctly implemented if q can be identified as the expected state s'. State identification is done by applying on q a distinguishing sequence [GON70], or a unique input/output sequence (UIO sequence, Figure 2.2) [SAB88, AHO88], or a set of characterizing sequences [CHO78]. q is considered as the same as s' if and only if the response to those sequences is the one expected. In order to test all the states and branches of the IUT, one may follow a Chinese postman tour or a rural postman tour [AHO88], which traverse all the branches of the specification and tests each of the states along the tour by the above state-identification technique.



(a) A transition diagram for machine M

state	UIO
0	b/λ
1	a/1 a/1
2	b/0
3	b/1 b/0
4	a/1 a/0

(b) UIO sequences for M

Figure 2.2 An example of UIO sequences for finite state machine M [SID89].

The assumption of a deterministic FSM overly simplifies the real nature of many distributed systems. Some of the sequences (distinguish sequence etc.) used for state identification are rarely available, especially for large systems. These affect the application of this approach.

2.3.2 The Methods of Applying All-DU-path and OI-path Criteria to ESTELLE Specifications

Ural and Yang presented methods of applying all-DU-path and OI-path criteria to protocol conformance testing [URA87, URA91]. The methods are based on the normal form specification of ESTELLE and consist of the following steps:

a) The normal-form specification is transformed into a digraph which represents both its control and data flow.
b) Definitions and uses of all context variables are identified.
c) (for OI-paths only) Input and output interaction parameters employed in the specification are identified.
d) Based on the above information, associations between definitions and usages of each variable or associations between each output and those inputs that influence the output are established.
e) Test sequences are selected to cover all definition-and-use associations or all output-and-input associations so as to satisfy all-DU-path or OI-path criterion, respectively.

The set of test sequences selected in these ways determines whether an IUT establishes the desired associations among values of the input and output parameters, and context variables. The methods complement control-flow-based test sequence selection strategies.

2.3.3 The Functional Testing Method of Sarikaya, Bochmann and Cerny

Sarikaya, Bochmann and Cerny [SAR87] proposed a functional testing method for protocol conformance testing. In this method, control and data flow graphs are first obtained from the normal form of a ESTELLE specification. In the control flow graph, subtours which are sequences starting and ending at the initial state of the graph can be easily obtained. They represent distinct control phases such as connection establishment, data transfer etc. The data flow graph has four types of nodes: I-nodes represent input primitive parameters, D-nodes represent context variables and constants, O-nodes represent output primitive parameters, and F-nodes represent data operations (functions). Properly decomposing and merging the data flow graphs will result in

several functional blocks. The test sequences are then selected on the basis of the subtours of the control graph and parameter variations of the input primitives of each data flow functional block.

In this method, functional program testing techniques are applied to generate test sequences. It can detect the specific functional errors of the implementation. However, it requires considerable human efforts to identify the functions and their relationships in case of non-trivial applications.

2.4 METHODS FOR TEST SUITE GENERATION FROM LOTOS SPECIFICATIONS

In Section 2.3, several methods employing selective testing strategies for conformance testing have been described. These methods are all based on finite state machines or their extensions (e.g., ESTELLE). As far as we know, no work has been done on selective test sequence generation directly from LOTOS specifications. In this section, some of the methods for generating test suites from a LOTOS specification are briefly reviewed. However, all of them, in principle at least, can generate test cases exhaustively.

2.4.1 The CO-OP Method

The CO-OP method [WEZ89] is designed for deriving a canonical tester from a labelled transition system representing a LOTOS specification. The canonical tester is first proposed by Brinksma [BRI88]. Its basic idea is that an IUT will conform with its specification if there is no unexpected deadlock when the tester concurrently runs with the IUT. The CO-OP method first derives COmpulsory(B) and OPtions(B) of a labelled transition system B. Test cases are then constructed from them (See the definitions in the following paragraph). By emphasizing on the internal actions, the method brings out the intrinsic indeterministic nature of black-box testing. It is shown that compositional derivation of a tester of LOTOS, i.e., construction of a tester T(B1*B2) from attributes of the testers T(B1) and T(B2), is possible (' * ' represents any operator of LOTOS).

Notation 2.2

$\underset{a \in V}{[]}\ a$ = x1 [] x2 [] ... [] xn, where V = {x1, x2, ..., xn}.

orth(M) The set of all sets which are formed by choosing exactly one member from each element of M, where M = {m1, m2, ..., mn} and mi is a set, i = 1, 2, ..., n.

Definition 2.2. *(Compulsory(B))*

Let B be a labelled transition system.

Compulsory(B) = { comp $\supset$ L(B) | $\exists$ B'. B = ε => i -> and comp = {x $\in$ L(B) | B' - x -> }}.

Compulsory(B) contains a set comp with the external events that can be performed from stable state B'. A *stable state* accepts only external actions and cannot do an internal action. In each test case at least one element of each set-element of Compulsory(B) must be used.

Definition 2.3. *(Options(B))*

Let B be a labelled transition system.

Options(B) = { x $\in$ L(B) | $\exists$ B'. B = ε => B' - i -> and B' - x -> }.

Options(B) contains all external events that can be performed from unstable state B' without making any internal transitions before doing so. An *unstable state* is a state that can do an internal event. Elements of Options(B) may be used in test cases.

Definition 2.4. *(Basic test cases for B)*

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

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

T3 = i; stop

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

where V $\in$ orth(Compulsory(B)) and option $\in$ Options(B). The canonical tester for the entire specification is the test suite consisting of all the basic test cases joined by the choice operators.

The CO-OP method or canonical tester explores all and only traces in the specification. It is not intended for practical use because the data aspect of the LOTOS specifications is not considered in the derivation of the tester.

2.4.2 The Method of Wu and Chanson

The method proposed by Wu and Chanson [WU89] is based on External Behaviour Expressions (EBE) of protocols which are supposed to be obtainable from formal protocol specifications in either ESTELLE or LOTOS (as well as from documentation in English). In EBE, the data flow and control flow portions are represented by external observable Input/Output sequences and their logic (function and predicate) relations, respectively. The method first identifies all interaction paths and their I/O subpaths obtained from the EBE. An *interaction path IP* is the externally observable track on which a sequence of interactions between the protocol and its external environment occurs, starting from the initial external state S_0 of the protocol and ending in the same state. An *I/O subpath* is the externally observable track e_1, ... e_k in an IP, associated with a specific test purpose and satisfying certain conditions (e.g., e_1 is an input primitive with its parameters and e_k is an output primitive with its parameters). Test cases can be obtained by grouping these I/O-subpaths according to the test objectives.

The method ignores the internal structure of a specification and therefore ignores the indeterministic nature of a distributed system. Selective testing is possible by selecting some of the I/O subpaths associated with specific functions.

2.4.3 The Method of Tripathy and Sarikaya

Tripathy and Sarikaya presented a method for generating test cases from a LOTOS specification [TRI91]. In this method, the LOTOS specification is simplified first so that a full synchronize composition is changed to a general composition and a sequential composition to a general composition etc. Then, the specification is mapped on to an extended labelled transition system called CHART developed by [MIL84]. From CHART, control and data flow graphs can

be obtained. They reflect the changes of states and how an input primitive parameter determines the values of context variables as well as the values of output primitive parameters, respectively. Test cases are then derived from the control flow graph according to the definition of test cases in CO-OP method [WEZ89]. By partitioning the data flow graph into smaller blocks and then combining them into larger blocks which represent various protocol functions, data flow functions are obtained, too.

In the method, full LOTOS specifications are considered. The exhaustive test case generated from the control graph turn out to be the same as those obtained by the CO-OP method [WEZ89]. Based on the data flow functions, a selection of specific functions is possible.

2.4.4 The Method of Cheung, Wu and Ye

Cheung, et al. presented a method which focuses on generating and executing test cases for indeterministic distributed systems [CHE91a]. In the method, an extended Petri-net model and a metric for measuring the degrees of indeterminism of test sequences are first developed. And then a fairness model of conformance testing and an iterative algorithm called IPNTEST for generating test cases and their degrees of indeterminism are presented. The metric is used to estimate the number of times that a test sequence should be tried in order to make a fair verdict. The kth iteration of IPNTEST generates a group of test sequences of length k by reachability analysis, all having the same preamble. The number of iteration and hence lengths of the test sequences can be controlled. Each group of test sequences provides a fairly uniform coverage for the system state reachable by that preamble. The test groups, as a whole, cover all bounded firable paths of the system. IPNTEST has been implemented as a fully automated test case generator.

The algorithm can generate all test sequences by deriving all the markings of a Petri-net. Comparing with the other methods discussed in this section, it is simple and efficient, especially, for the indeterministic systems. The method is mainly for exhaustive testing though the algorithm may be modified for selective testing.

2.4.5 The Method of Gueraichi and Logrippo

Gueraichi and Logrippo presented a semi-formal method [GUE89] for deriving test cases from a basic LOTOS specification. It has been applied to the protocol LAP-B. In this method, an execution tree is first automatically generated from the specification by using a LOTOS interpreter [GUI88]. The interpreter performs a transformation of the specification by replacing all operators such as 'parallel composition' and 'disable' etc. with 'choice' only. The derived tree includes all possible execution sequences of the entity specified. Test sequences are then selected manually by using the technique discussed in Subsection 2.3.1 on the state diagram transformed from the execution tree.

The selection process in this method is done mainly by human 'walkthrough'. No systematic guidelines for this selection have been reported.

2.4.6 The Method of Cheung and Ye

The method proposed by Cheung and Ye [CHE90] is based on the UO-GLOTOS tool developed by University of Ottawa. The EXECUTOR, one of the subsystems, can be used as a semi-automatic visual tool for dynamically tracing the logical flow of the design and for generating or executing test sequences for the specified system. To help in using the tool, the LOTOS operators, executable actions, actions waiting for or already executed in rendezvous, the current trace and history of execution are all clearly revealed in a graphical representation. Based on these information, the user can dynamically select paths for execution with much less analysis of the system in advance.

The method can generates all executable paths of a LOTOS specification including data. The test cases can be selected from these paths in different ways. The selection process of this method is also done by human 'walkthrough'. No selection criteria have been proposed.

Chapter 3

OPERATIONAL COVERAGE FOR LOTOS SPECIFICATIONS

3.1 INTRODUCTION

This chapter contains the first main contribution of this thesis, namely, an operational coverage for a LOTOS specification. A coverage can be defined for individual operations or for a set of operations. As far as we know, this problem has never been considered in the literature.

In order to determine the criteria for selective testing on LOTOS specifications, we have to first consider what are the important functions or characteristics of LOTOS. Both syntactically and semantically, the operations (i.e., operators and their operands) play an important role in determining the functional and structural relationships among the components of a LOTOS specification. They describe and determine both the control flow and data flow of the specified system. Their proper functioning are crucial to the overall correctness of an IUT. Therefore, for the purpose of selective test sequence generation, the LOTOS operations become a natural target for proposing criteria.

The rest of the chapter is organized as follows: Section 3.2 defines an operational coverage for the individual LOTOS operations. Section 3.3 extends the definition to a set of LOTOS operations.

3.2 OPERATIONAL COVERAGE FOR SINGLE LOTOS OPERATIONS

A selection criterion should be intuitively simple for understanding what it covers. Many well-known criteria, such as the all-DU-paths strategies, have such a property. In the case of LOTOS, we find it conceptually clearer to define coverage in terms of the LOTOS hierarchy of operations. An operation consists of either a single primitive element (i.e., internal action, external action, exit or stop) or an operator associated with a set of operands (which are operations themselves). A cover is a set of paths (sequences of actions) starting at the root of the hierarchy representing the operation. In general, the sequences are not executable because of the possible

existence of rendezvous. Chapter 4 will present a method for generating executable test sequences based on these sequences.

Notation 3.1. In the following description, δ, θ, a, b, ... z represent single actions and α, β, υ, ϖ, σ represent sequences of actions.

S	The LOTOS specification.
$A_O(B)$	The set of observable actions in behavior expression B, abbreviated as A_O if B = S.
A_S	The set of special action and inaction of S, including exit (successful termination) and stop (unsuccessful termination), represented by the symbols δ and θ respectively.
$\alpha\cdot\beta$ or $\alpha\beta$	Concatenation of sequence β behind sequence α.
$\Pi_{i=1,n}\alpha_i$	$\alpha_1\alpha_2 \ldots \alpha_n$.
front(β)	An arbitrary initial proper subsequence of β, i.e., $\exists$ a non-empty sequence σ such that β = front(β)$\cdot\sigma$.
$\alpha - \delta$	Elimination of the last element δ from sequence α.
$\alpha \mid\mid\mid \beta$	An arbitrary interleaving of sequences α and β.
C(B, O)	The cover for a set of operators O with respect to behaviour expression B. O may be omitted when it is irrelevant or clear from the context. In particular, if O consists of a single operator op, it can be written as C(B, op).
B, B1, B2	Behaviour expression within S, where B = B1 op B2.
υ, υ'	Arbitrary sequence of C(B1).
ϖ, ϖ'	Arbitrary sequence of C(B2).

The coverage of a LOTOS operation is a set of test sequences, which reflect the semantics of the operator and depend on the expressions contained in its operands. The sequences are chosen in such a way that there is a balance between the efficiency in generating them (i.e., keeping their number small) and the sufficiency in representing the major characteristics of the operation. For example, in order to cover the major interruption capabilities of the disable operation B1 [> B2, at least three test sequences must be selected: one involving actions executed totally within B1,

another involving actions executed totally within B2, which prevent the execution of B1 from the very beginning, still another involving some actions executed within B1, and (because of interruption) some subsequent actions executed within B2.

our definition of operational coverage is divided into two cases: one (Definition 3.1) for LOTOS expressions without involving any recursions and the other (Definition 3.2) for expressions involving some recursions.

Definition 3.1. *(Operational coverage for single LOTOS operations w.r.t. an expression which does not include recursions).* That is, in the following definitions, B1 ≠ B and B2 ≠B.

A. *Coverage for the primitive elements* (exit, stop, internal action i, and $g \in A_O$):

$$C(B) = \{ \alpha \}, \text{ where } \alpha = \begin{cases} \delta & \text{if } B = \text{exit} \\ \theta & \text{if } B = \text{stop} \\ \phi & \text{if } B = i \\ g & \text{if } B = g \end{cases}$$

Explanation: δ is the symbol used to represent 'exit' in [ISO8807], and θ is our symbol for representing 'stop'. The exact meanings of these symbols depend on their usage.

B. *Coverage for the action-prefix operation:*

For B = a op B1, where $a \in A_O \cup \{ i \}$ and op = ';', we define:

$$C(B, op) = \begin{cases} \{ a \cdot \upsilon \} & \text{if } a \in A_O \\ \{ \upsilon \} & \text{if } a = i \end{cases}$$

Example 3.1

Suppose B = a ; B1, where B1 = i ; exit.

Then, $C(B1) = \{ \delta \}$ and $C(B, ;) = \{ a\delta \}$.

C. *Coverage for the choice operation:*

For B = B1 op B2, where op = '[]', we define:

$C(B, op) = \{ \upsilon, \varpi \}$.

Example 3.2

Suppose B = B1 [] B2, where B1 = a ; exit [] b ; exit, B2 = c ; exit [] d ; exit, C(B1, []) = { aδ, bδ } and C(B2, []) = { cδ, dδ }.

Then, C(B, []) = { aδ, cδ } or { aδ, dδ } or { bδ, cδ } or { bδ, dδ }.

D. *Coverage for the enable operation:*

For B = B1 op B2, where op = '>>', we define:

$C(B, op) = \{\alpha, \beta\}$, where $\alpha = \begin{cases} \upsilon & \text{if } \theta \in \upsilon \\ \phi & \text{otherwise} \end{cases}$ and $\beta = \begin{cases} (\upsilon - \delta)\cdot\varpi & \text{if } \delta \in \upsilon \\ \phi & \text{otherwise} \end{cases}$

Explanation: A sequence υ in C(B1) may end with a 'stop' or 'exit ' (symbolized as θ or δ, respectively). In the former case, the sequence α is executed entirely within B1 and B2 is never triggered. In the latter case, sequence β extends from B1 into B2 is adopted.

Example 3.3

Suppose B = B1 >> B2, where B1 = a ; b ; stop [] b ; c ; exit, B2 = d ; exit., C(B1, ;) = { abθ, bcδ }and C(B2, []) = { dδ }.

Then, C(B, >>) = { abθ, bcdδ }.

E. *Coverage for the disable operation:*

For B = B1 op B2, where op = '[>', we define:

$C(B, op) = \{ \upsilon, \varpi, front(\upsilon')\cdot\varpi' \}$.

Explanation: At least three sequences should be included: (1) υ is executed in B1 until its end and B2 is never initiated. (2) ϖ is started in B2 and B1 is never initiated. (3) The front part of υ' is executed within B1, and then, B1 is interrupted by the start of ϖ' in B2.

Example 3.4

Suppose B = B1 [> B2, where B1 = a ; b ; stop [] b ; c ; exit, B2 = d ; exit, C(B1, []) = { abθ, bcδ } and C(B2, ;) = { dδ }. Assume υ = abθ, υ' = bcδ and $\varpi = \varpi'$ = dδ.

Then, C(B, [>) = { abθ, dδ, bdδ } or { abθ, dδ, bcdδ }.

F. *Coverage for the parallel operations:*

For B = B1 op B2, where op = '|[G]|' and G = $\{g_1, g_2, ..., g_{m-1}\}$ is a list of synchronization gates, we define:

$$C(B, op) = \begin{cases} \{\Pi_{i=1...m} (\upsilon_i \,|||\, \varpi_i)\cdot g_i\} & \text{if } \exists\ \upsilon = \Pi_{i=1...m}\ \upsilon_i \cdot g_i \\ & \quad \varpi = \Pi_{i=1...m}\ \varpi_i \cdot g_i \\ & \quad \text{where } g_m = \delta \\ \phi & \text{otherwise} \end{cases}$$

Note: If G is empty, B1 |[G]| B2 is the same as B1 ||| B2. If $G = A_o(B1) \cup A_o(B2)$, B1 |[G]| B2 is the same as B1 || B2.

Example 3.5

Suppose B = B1 |[b]| B2, where B1 = a ; b ; c ; d ; stop [] c ; b ; exit, B2 = a ; f ; b ; g ; h ; exit, C(B1, []) = { abcdθ, cbδ } and C(B2) = { afbghδ }. Let υ = cbδ, ϖ = afbghδ.

Then, C(B, |[b]|) = { acfbghδ } or { cafbghδ } or {afcbghδ}.

Example 3.6

Suppose B = B1 || B2, where B1 = a ; b ; exit [] a ; c ; exit, B2 = a ; b ; exit [] a ; c ; exit, C(B1, []) = { abδ, acδ } and C(B2, []) = { abδ, acδ }.

Then, C(B, ||) = { abδ } or { acδ }.

Coverage for recursions has always been a difficult issue in software testing. Some finite models have been used to approximately represent systems involving recursions. One approach is to limit only one iterative execution of a recursive expression. We adopt this limitation in the following definition.

Definition 3.2. (*Coverage for an operation w.r.t. an expression involving a recursive expression B*)

For B = B1 op B, where B1 is non-recursive, we let B* = 'B1 op exit' and define C(B, op) as C(B1 op B*, op), i.e., C(B, op) = C(B1 op B*, op).

Explanation: (1) By replacing the first reoccurrence of B with B*, we have limited the execution of the recursive expression B to only once. (2) For the case where B = B op B1, the definition is similar.

Example 3.7

Suppose B = B1 >> B, where B1 = a ; exit [] b ; exit. Let B* = B1 >> exit = (a ; exit [] b ; exit) >> exit and redefine B = (a ; exit [] b ; exit) >> B*.

Then, C(B, >>) = {aaδ} or {abδ} or {baδ} or {bbδ}.

3.3 OPERATIONAL COVERAGE FOR A SET OF LOTOS OPERATIONS

In this section, the operational coverage for single operations is extended to cover a set of operations. Naturally, it is defined as the union of the covers of the individual operations. But, it may have redundant sequences. In order to eliminate such redundancy, we define the following *Reduction* operation in terms of paths of the LOTOS hierarchy.

Definition 3.3. (*Reduction*)

Let Q = {Q1, Q2, ... Qn}, where Qi is a set of paths in the LOTOS hierarchy, i = 1, 2, ..., n. RED[Q] is an operation on Q such that, for any $q \in Q_j$, q will be eliminated from Qj if $\exists\, p \in Q_k$, where $k \neq j$ such that q is a proper subpath of p.

(Note that the subpaths of a path within the same set should not be eliminated because each cover Qi is supposed to include a fixed number of test sequences for checking the characteristics of a specified operation. Eliminating such subpaths will eliminate some of these test sequences. Example 3.8 shows such a case.)

Example 3.8

Suppose B = B1 op B2, where B1 = a ; B, B2 = b ; exit and op = [].

Let B* = a ; exit [] B2 and redefine B = a ; B* op B2 = (a ; (a ; exit [] b ; exit)) op (b ; exit).

Coverage for the specified operation (B, op) is: C(B) = {aaδ, bδ} or {abδ, bδ}.

(Note: According to the coverage definition for choice operator, two test sequences are needed to form a cover, therefore bδ cannot be eliminated from the cover {abδ, bδ})

Definition 3.4. *(Operational coverage for a set of LOTOS operations)*

Suppose expression B is composed of the set of operations (B1, op1), ..., (Bn, opn). Let O = {op1, ..., opn}. Then, C(B, O) = RED[C(B1, op1), ..., C(Bn, opn)].

Example 3.9

Suppose B = B2 = a ; B1 op2 c ; exit, where op2 = [], B1 = b ; c ; exit op1 d ; e ; exit and op1 = [].

Coverage for the set of specified operations (B1, op1) and (B2, op2) are:

C(B1, op1) = { bcδ, deδ } and C(B2, op2) = { abcδ, cδ } or { adeδ, cδ }, respectively.

Then, C(B, O) = { abcδ, deδ, cδ } or {bcδ, adeδ, cδ}, where O is { op1, op2 }.

Similar to the all-DU-path, all-branch, all-OI-path criteria in conventional selective software testing, each of the following special cases covers all occurrences of a specified type of LOTOS operators in the specification S. Since the operations play an important role in determining the functional and structural relationships among the components of a LOTOS specification, They will serve special purposes in testing. For example, the *all-parallel* coverage include sequences extending over several processes of the system which are running concurrently.

A. *All-parallel* coverage, where O contains all the parallel operators in S.

B. *All-action-prefix* coverage, where O contains all the action prefix operators in S.

C. *All-enable* coverage, where O contains all the enable operators in S.

D. *All-disable* coverage, where O contains all the disable operators in S.

E. *All-choice* coverage, where O contains all the choice operators in S.

Chapter 4

A METHOD FOR GENERATING SELECTIVE EXECUTABLE TEST SEQUENCES

4.1 INTRODUCTION

This chapter contains the second main contribution of this thesis, called method SELECTEST, for selectively generating executable test sequences from a LOTOS specification. The method are described formally in Section 4.2, using an example as illustration. Detailed description of the concepts and algorithms involved in the method is given in Section 4.3.

In this method, selection of test sequences is based on the operational coverage for a LOTOS specification developed in Chapter 3 and the process of generating executable test sequences is carried out on its Petri-net representation. Each executable test sequence is a sequence in which its possibly non-executable action sequence is embedded. In order to generate the executable test sequences over a Petri-net, one has to solve a special case of the general reachability problem, i.e., finding an executable sequence of transitions in which a non-executable transition sequence is embedded. One approach is to derive the entire marking graph and then find a path leading from the initial state to the specified state. But, the state explosion problem makes this approach inefficient and infeasible in practice. To overcome this difficulty, SELECTEST first determines the cover for the specified operation. Based on the cover, a set of guides are obtained, each denoted as a possibly nonexecutable rooted path in the hierarchy of the LOTOS specification and passing through the operator. It then transforms the LOTOS specification into a Petri-net and applies the guides to search for executable test sequences over the Petri-net.

4.2 FORMAL DESCRIPTION OF METHOD SELECTEST

In this section, we present a method called SELECTEST for generating executable test sequences which cover a specified LOTOS operation. Since it is difficult to derive executable test sequences directly from a LOTOS specification S, our method is divided into two phases: In the

first phase, based entirely on the LOTOS hierarchy, a set of possibly non-executable test sequences, called *guides* or *local test sequences*, is obtained to cover the specified operation. In the second phase, these guides are used to derive executable test sequences based on a Petri-net representation of S. This method is summarized in Figure 4.1 followed by some explanation. A formal description is then given.

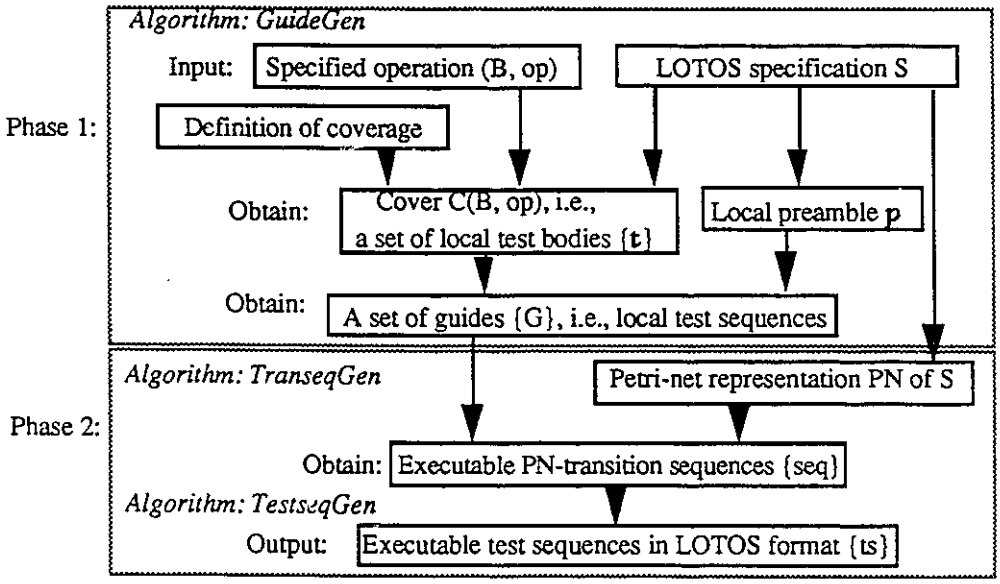


Figure 4.1. The selective test sequence generation method SELECTEST.

Explanation of method SELECTEST.

Phase 1. Generate the cover and a set of guides:

In the fault-tolerant state-identification approach to testing a system specified in a finite state machine (FSM), a test sequence is composed of two segments: a preamble followed by a test body. First, the preamble leads the IUT from the initial state to the specified state over the FSM. Then, the test body (e.g., distinguishing sequence, UIO sequence, etc.) is used to 'identify' that state. A similar but generalized idea is adopted in this phase of our method.

This phase is based on the given LOTOS hierarchy S. For clarity sake, our description is by means of UO-GLOTOS, though not necessarily so. First, a preamble leads the IUT from the

'outermost' operator to the designated operator over the LOTOS hierarchy S. Then, a set of test bodies is used to 'cover' the operator. Each test body starts from this operator and ends at one of the terminal nodes of S (Figure 4.2). The preamble, concatenated with the test bodies, form a set of *local* test sequences. Because of the existence of rendezvous in LOTOS specifications, they may not be executable and used as 'guides' in Phase 2.

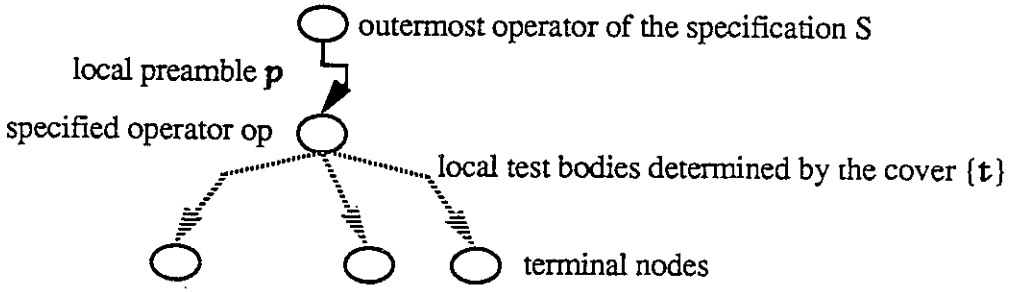


Figure 4.2. Generating guides (i.e., local test sequences) from a UO-GLOTOS specification.

Phase 2. Generate executable test sequences from the guides:

In the Petri-net representation PN of S, each guide is a possibly non-executable sequence of transitions G = {t0, t1, t2, ..., tn}, where t0 is the initial transition of PN. The problem becomes finding an executable sequence of transitions seq = {t0, t1',..., t1, t2',..., t2,..., tn} in which G is embedded. In general, whether u_{i+1} (the marking after firing t_{i+1}) can be reachable from u_i (the marking after firing t_i) is not known. In this special case, u_{i+1} can be reachable from u_i, because G is corresponding to a LOTOS action sequence which is derived based on the semantics of each LOTOS operation. This action sequence can form an executable test sequence. Therefore, in order to find an executable transition sequence seq from G, we only need to find some intermediate markings between every pair of u_i and u_{i+1}. In our method, instead of creating the whole marking graph from the initial marking u0, a partial marking graph is obtained by back up searching from u_{i+1}. In the most of cases, it can speedy the search for an executable transition sequence.

Since LOTOS is a process algebra, all the method for test derivation from LOTOS specifications are based on certain state transition models, in which control and data flow of the

LOTOS specifications can be clearly represented. Because Petri-nets are powerful in representing concurrent operations of distributed systems, the computation part of our method is based on Petri-net representations of LOTOS specifications. The main algorithm of this method is described in the following pseudo-code.

Parameters and functions used in Algorithm TranseqGen:

seq	Initially empty, used to store the executed transitions consecutively.
G	Used as a stack. Initially containing the given guide. Subsequently used to store those transitions whose executability status is not known at the moment.
POP(G)	An operation which retrieves and deletes the top element of stack G.
pre(t)	The set of pre-places of transition t in a Petri-net. t is executable only if every p in pre(t) has a token.
pre(p)	The set of pre-transitions of place p in a Petri-net.
node(t)	Those places of pre(t) which have no tokens, i.e., those p in pre(t) none of whose pre-transitions have been executed.
fire(t)	A sequence of transitions which must be executable in order for t to be executable but whose executability status is not known at this moment. If fire(t) is empty, t is currently executable and is put into seq immediately. Otherwise, fire(t) and t are stacked on top of G to be considered in the later cycles.

Algorithm TranseqGen (for generating an executable transition sequence from a guide):

```
Process: seq := ϕ.
         While G ≠ ϕ do
            begin
               t := POP(G);
               If t ∉ seq then
                  begin
```

node(t) := { p | p ∈ pre(t), pre(p) ∩ seq = ϕ };

fire(t) := Π { t | t ∈ pre(p), p ∈ node(t) } - G;

If fire(t) ≠ ϕ then G := fire(t)·t·G else seq := seq·t

end /* endif */

end. /* endwhile */

The above process proceeds as follows (Figure 4.3): Initially, the stack G = {t0, t1, ... tn} contains the given guide. In each cycle of the while loop, one transition t is popped up from G and checked for its executability, by finding whether there are some transitions which should be executed before t. If there are no such transitions or they are already in seq, t is executable currently and queued to seq immediately. Otherwise, those transitions which should be executed before t must be sought and stacked on top of G. In the first cycle, t0 is retrieved and checked for executability. In order to execute t0, p0 must contain a token. Thus, node(t0) contains p0. Since p0 has no pretransition, fire(t0) is empty and t0 is executable and put into seq. In general, suppose the sequence seq = {t0, t1, ..., t_{k-1}} has been executed and tk is retrieved from G. To fire tk, place pk must have tokens. Therefore, node(tk) contains pk and fire(tk) contains the pretransition tk' of pk. Since the executability status of tk' is still unknown, tk' and tk are stacked on top of G for future consideration.

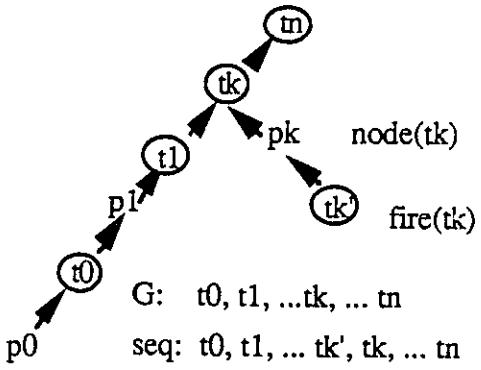


Figure 4.3. Generating an executable transition sequence from a guide on a Petri-net.

A formal description of the algorithm is given below.

METHOD SELECTEST (for executable selective test sequence generation)

INPUT : A LOTOS specification S; a specified operation (B, op).

OUTPUT : A set of executable test sequences covering (B, op).

PROCEDURE :

Phase 1. Generate the cover C(B, op) and a set of guides from S:

/* Based on the given LOTOS hierarchy S, this phase is achieved by applying Algorithm GuideGen whose pseudo-code is given in Subsection 4.3.1 */

1.1) Based on the definition of operational coverage, generate the cover C(B, op) for the specified operation (B, op) in the form of a set of local test bodies $\{\mathbf{t}\}$.

1.2) Generate the local preamble $\mathbf{p}$ for op.

1.3) For each local test body $\mathbf{t} \in$ C(B, op), form a local test sequence (i.e., a guide G) by concatenating op and $\mathbf{t}$ to the local preamble $\mathbf{p}$. That is, set $G = \mathbf{p} \cdot op \cdot \mathbf{t}$, Each guide is a rooted path in the LOTOS hierarchy (Figure 4.2). (In order to facilitate the transformation of a LOTOS-path to a Petri-net-path, nodes representing the relevant LOTOS operators will be inserted into the preamble and the test body.)

Phase 2. Generate executable test sequences based on the guides :

/* Based on a Petri-net representation PN of the given LOTOS specification S, this phase transforms the possibly non-executable guides into executable ones by inserting appropriate actions. It is achieved by applying Algorithms TranseqGen and TestseqGen whose pseudo-codes are given in Subsection 4.3.2 */

2.1) Transform S to a Petri-net representation PN and every guide obtained in Phase 1 from its UO-GLOTOS node-representation to a PN representation. (Note that, in case of the choice operator, there is no corresponding transition in PN for UO-GLOTOS node and it is mapped to a place. This is also applicable for the nodes representing recursions.)

2.2) For each guide G, generate an executable transition sequence seq based on its PN representation.

2.3) Transform these executable PN-transition sequences to test sequences in terms of LOTOS actions.

Example 9

Consider the specification S = (a ; B) |[b,d,e]| (b ; c ; d ; exit [ ] e ; exit), where B = b ; d ; exit op e ; exit and op = '[ ]'. The UO-GLOTOS and Petri-net representations of S are shown in Figure 4.4 and Figure 4.5, respectively. We want to generate test sequences covering the operation (B, op) (op is node n3 in Figure 4.4). The phases and results of applying SELECTEST are shown below:

Phase 1. Obtain the cover and guides from the UO-GLOTOS representation (Figure 4.4) :

The cover C(B, op) is {bdδ, eδ}, providing two local test bodies: {n4,n5,n6}, {n7,n8}.

The local preamble is {n2}.

The guide obtained by concatenating {n4,n5,n6} to {n2} is {n1,n2,n3,n4,n5,n6}, which is non-executable, because, in order to proceed to the rendezvous at d, it requires the execution of c (node11) which is not included in the guide.

The guide obtained by concatenating {n7,n8} to {n2} is {n1,n2,n3,n7,n8}, which is executable.

Phase 2. Obtain executable test sequences based on the guides from the Petri-net representation PN (Figure 4.5) :

The guides in PN format are {t1,t2,t3,t4,t6} and {t1,t2,t7,t8}.

The executable transition sequences are {t1,t2,t3,t5,t4,t6} and {t1,t2,t7,t8}.

(Figure 4.6 shows the detail of how Phase 2.2 is applied on the guide {t1,t2,t3,t4,t6} to generate the executable transition sequence {t1,t2,t3,t5,t4,t6}.)

The executable test sequences in LOTOS format are {a,b,c,d,exit} and {a,e,exit}.

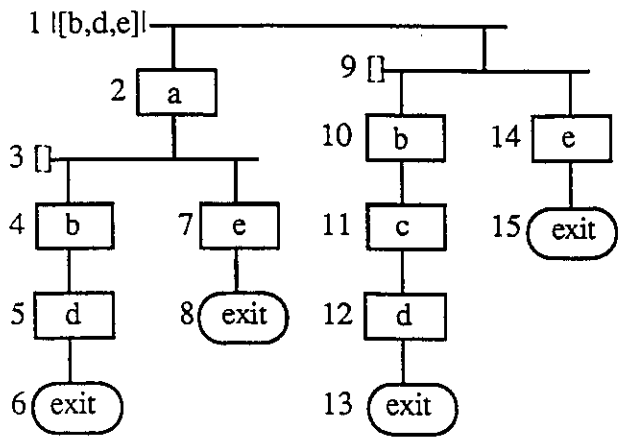


Figure 4.4. UO-GLOTOS representation of S = (a ; B) |[b,d,e]| (b ; c ; d ; exit [] e ; exit, where B = b ; d ; exit [ ] e ; exit.

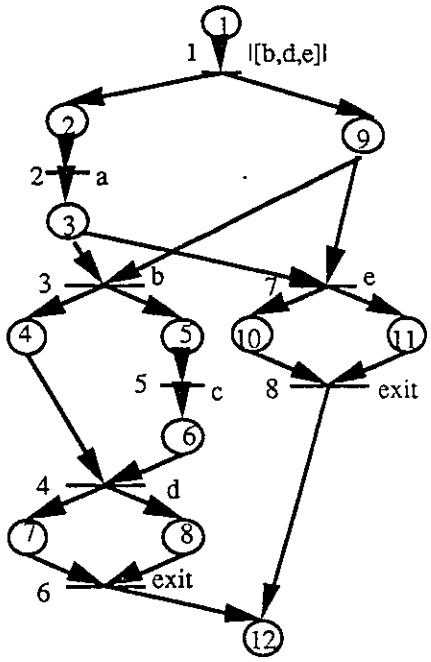


Figure 4.5. Petri-net representation of the LOTOS specification of Figure 4.4.

Cycle	t = POP(G)	node(t)	fire(t)	Guide G	Transition sequence seq
0				t1, t2, t3, t4, t6	
1	t1			t2, t3, t4, t6	t1
2	t2			t3, t4, t6	t1, t2
3	t3			t4, t6	t1, t2, t3
4	t4	p6	t5	t5, t4, t6	t1, t2, t3
5	t5			t4, t6	t1, t2, t3, t5
6	t4			t6	t1, t2, t3, t5, t4
7	t6				t1, t2, t3, t5, t4, t6

Figure 4.6. Illustration of Phase 2.2 of SELECTEST for guide G = {t1,t2,t3,t4,t6}.

4.3 DETAILED DESCRIPTION OF THE ALGORITHMS USED IN METHOD SELECTEST

In this section, the algorithms GuideGen, TranseqGen and TestseqGen involved in the method SELECTEST are described in details. Notation used in the algorithms is defined below:

Notation 4.1

$L = L_e \cup \{i\} \cup L_s$	i is the internal action; L_e is the set of labels for external actions including 'exit' and 'stop'; L_s is the set of special labels for LOTOS operators.
label(x)	Labelling function: T --> L, if x is in the transition set T of PN N --> L, if x is in the node set N of UO-GLOTOS.

The rest of this section is organized as follows: Subsection 4.3.1 gives the detailed description of Algorithm GuideGen. Subsection 4.3.2 presents Algorithms TranseqGen and TestseqGen. Subsection 4.3.3 proposes an extension of the algorithms to cover a set of LOTOS operations (Section 3.3).

4.3.1 Detailed Description of Algorithm GuideGen

Algorithm GuideGen realizes Step 1 of the method SELECTEST. It generates a set of guides based on the definition of the operational coverage. It calls two procedures Search-up and Search-down for deriving the local preambles and local test bodies, respectively.

Algorithm: GuideGen

/**

Generating a set of guides from a UO-GLOTOS specification. A preamble and a set of test bodies are obtained by executing Procedure Search-up and Procedure Search-down, respectively. Then, a set of guides are formed by concatenating each of the test bodies to the preamble.

**/

INPUT: A UO-GLOTOS specification, a specified operator represented by node n_{op}.

OUTPUT: guide(1), guide(2), guide(3).

METHOD:

begin

Step 0: /* Initialization */

preamble := body(1) := body(2) := body(3) := body' := ϕ.

Step 1: /* Generating a local preamble */

Search-up(preamble, n_{op}).

Step 2: /* Generating a set of local test bodies: body(i), where i = 1,2,3 */

begin

Case 1: /* label(n_{op}) = '>>' */

Search-down(body(1), n_{op});

Case 2: /* label(n_{op}) $\in$ (|||, |[G]|, ||, ;) */

Search-down(body(1), left-child(n_{op}));

Search-down(body', right-child(n_{op}));

/* Checking if both the test bodies have the same synchronization gates. If not, an error occurs */

if $\exists$ g1', g2' ... gn' $\in$ body(1), g1", g2" ... gn" $\in$ body'

where g1'=g1", g2' = g2"... gn' = gn" $\in$ G

then body(1) := body(1) ||| body'

else print 'error';

Case 3: /* label(n_{op}) = ' [] ' */

Search-down(body(1), left-child(n_{op}));

Search-down(body(2), right-child(n_{op}));

Case 4: /* label(n_{op}) = ' [> ' */

Search-down(body(1), left-child(n_{op}));

Search-down(body(2), right-child(n_{op}));

/* Obtaining a test body having interrupted property */

Search-down(body(3), n_{op});

body(3) := body(3) - n_{op};

Case 5: /* label(n_{op}) $\notin$ $L_e \cup L_s \cup \{ i \}$. That is the case of recursion. */

rec := rec $\cup$ {n_{op}};

Transform the recursive expression B = B1 op B to B = B1 op B',
where B' = B1 op exit. /* Similar for B = B op B1 */

end. /* endcase */

Step 3: /* Deriving a set of guides by concatenating each of the local test bodies to the local preamble */

for each body(i), guide(i) := preamble·body(i) , where i = 1, 2, 3.

end. /* GuideGen */

Procedure Search-up(preamble, node);

```
/*************************************************************************
Generate a preamble by searching from the specified operator up to the outermost operator in
the LOTOS hierarchy.
*************************************************************************/
 begin
   If node ≠ ϕ then
     begin
       If  label(node) = '>>'
            then  begin
                    If right-child(node) ∈ preamble
                      then begin
                             preamble' := ϕ;
                             Search-down(preamble', left-child(node));
                             preamble := preamble'·node·preamble
                           end
            else   preamble := node·preamble;
       Search-up(preamble, parent(node))
     end /* endif */
```

```
end. /* Search-up */
```

Procedure Search-down(body, node);

```
/*************************************************************************
```

Generate a test body by searching from the specified operator down to several terminal nodes based on the selection criterion for different operations. The recursive expressions are recorded because they need some special processing in Algorithm TranseqGen.

```
*************************************************************************/
begin
If node = ϕ
  then  begin
          If label(parent(node)) ≠ 'exit'                    /* Processing recursion */
            then  rec := rec ∪ {parent(node)}
        end  /* endif */
  else  begin
          case 1:     /* label(node) = ' >> ' */
                      Search-down(body, left-child(node));
                      body := body·node;
                      Search-down(body, right-child(node));
          case 2:     /* label(node) ∈ ( |||, |[G]|, || ) */
                      body' := body" := ϕ;
                      Search-down(body', left-child(node));
                      Search-down(body", right-child(node));
                      /* Checking if both the test bodies have the same synchronization
                         gates. If not, an error occurs */
                      if   ∃ g1', g2' ... gn' ∈ body', g1", g2" ... gn" ∈ body"
                           where g1'=g1", g2' = g2"... gn' = gn" ∈ G
```

```
                then   body := body·node·(body' ||| body")
                else   print 'error';
        case 3:  /* label(node) = ' [] ' */
                body := body·node;
                /* child(node) means right or left child */
                Search-down(body, child(node));
        case 4:  /* label(node) = ' [> ' */
                /* tag represent the number of actions occurred before interruption and
                num(body) means the number of actions in the body */
                take any tag from { 0, 1, 2, ... };
                If tag ≠ 0
                  then  begin
                          Search-down(body, left-child(node));
                          body = front(body), where num(body) = tag;
                          Search-down(body, right-child(node))
                        end /* endthen */
                  else Search-down(body, child(node));
          otherwise:  body := body·node;  Search-down(body, child(node))
      end /* endif */
end. /* Search-down */
```

4.3.2 Detailed Description of Algorithms TranseqGen and TestseqGen

Algorithms TranseqGen and TestseqGen realize Step 2 of the method SELECTEST.

Algorithm: TranseqGen(guide,seq)

```
/*************************************************************************
```

Generating an executable Petri-net-based transition sequence from a guide. Every transition in

guide is checked for its executability in Step2. If it is executable, it will be put in 'seq'. Otherwise, it will be put back to guide G with those transitions which should be executed before it and has not be executed yet (not in seq). In the case of recursion, those transitions may be already in 'seq' and needed to be repeated in 'seq' for the second execution of a recursive expression. In Step 2, special processing for recursion is provided.

**********/

INPUT: The LOTOS specification S and a guide G.

OUTPUT: An executable transition sequence seq.

METHOD:

```
begin
  Step 1.  Transform S to PN and the guides from node-representation in S to transition-
           representation in PN.
  Step 2.  /* Generating an executable transition sequence for each guide G */
           seq := ϕ;                          /* Initially, no transition has been executed */
           While G ≠ ϕ and t ∉ rec do
            begin
              t := POP(G);                    /* Take out one transition t from guide G */
              If (t ∉ seq) then
                  begin                       /* Check the executability of transition t */
                    node(t) := { p | p ∈ pre(t), pre(p) ∩ seq = ϕ };
                    fire(t) := Π { t | t is an arbitrary transition in pre(p), p ∈ node(t) } - G;
                    If fire(t) ≠ ϕ  then  G := fire(t)·t·G          /* t is executable */
                     else  seq := seq·t ;                           /* t is not executable */
                  end   /* endif, endcase 1*/
            end;
           If t ∈ rec then                    /* This is for processing recursion */
            begin
```

```
                seq' := ϕ;
                while G ≠ ϕ do
                  begin
                    t := POP(G);
                    If (t ∉ seq') then
                     begin
                      node(t) := { p | p ∈ pre(t), pre(p) ∩ seq' = ϕ };
                      fire(t) := Π { t | t is an arbitrary transition in pre(p), p ∈ node(t) } - G;
                      If (fire(t) ≠ ϕ)
                        then G := fire(t)·t·G
                        else begin  seq := seq·t;  seq' := seq'·t end  /*endif, endcase 2*/
                     end
                  end  /* endif */
              end  /* endwhile */
end. /* TranseqGen */
```

Algorithm: TestseqGen

```
/*************************************************************************
```

Generating a test sequence from an executable transition sequence by replacing transitions in PN labelling as external actions in LOTOS.

```
*************************************************************************/
```

```
INPUT:    seq     /* Output of Algorithm TranseqGen */
OUTPUT:   a test sequence ts.
METHOD:
          begin
            ts := ϕ;
            while seq ≠ ϕ do
```

```
        begin
            t := POP(seq);
            If label(t )∈ L_e and (seq = ϕ or label(t) ≠ 'exit') then ts := ts·label(t)
        end /* endwhile */
    end. /* TestseqGen */
```

4.3.3 Extension of the Algorithms

The algorithms described in Subsections 4.3.1, 4.3.2 and 4.3.3 can be applied to generate test sequences covering multiple LOTOS operations. The procedure is described below:

For a set of operators op1, op2, ..., opn involved in expression B,

(a). Apply Step 0 to Step 2 of Algorithm GuideGen to each of the operators op1, op2, ... opn.

(b). Execute the following pseudo code:

Compare the preambles of opi and opj or the test bodies of opi and opj, $i \neq j = 1, 2, ..., n$.

case 1: /* all-parallel coverage */

If preamble of opi = front(preamble of opj) then the preamble and test bodies of opj are eliminated.

case 2: /* all-enable or all-action-prefix coverage */

If preamble of opi = front(preamble of opj) then the preamble and test bodies of opi are eliminated.

case 3: /* otherwise (including all-choice and all-disable coverage) */

If body(m) of opi is a subsequence of body(n) of opj then body(m) of opi is eliminated, where m, n = 1,2,3;

(c). Apply Step 3 of Algorithm GuideGen to each of the operators op1, op2, ..., opn to obtain a set of guides.

(d). Apply Algorithm TranseqGen to each of the guides so as to obtain a set of executable transition sequences.

(e). Apply Algorithm TestseqGen to each of the executable transition sequences so as to obtain a set of test sequences.

Chapter 5

APPLICATION TO THE ALTERNATING BIT PROTOCOL

In this chapter, we apply the method SELECTEST described in Chapter 4 to the Alternating Bit Protocol (ABP). The textual LOTOS specification of ABP, listed in Appendix B, includes all the basic LOTOS operators except the parallel operator " || ". Selective test sequences covering the following six operations (specified as operators if apply) are derived:

(1) |[tm1,tm2,tm3]| (node 4 of Figure 5.1),

(2) ||| (node 26 of Figure 5.1),

(3) [] (node 28 of Figure 5.1),

(4) [> (node 18 of Figure 5.1),

(5) >> (node 39 of Figure 5.1), and

(6) recursive process instantiation (nodes 30 and 32 of Figure 5.1).

Note that, since we do not consider data, we have replaced the three constants 'timestart', 'reset' and 'expired' of gate tm in the textual ABP with the three gates tm1, tm2 and tm3, respectively.

For clarity, our notation is based on the UO-GLOTOS specification of ABP (Figure 5.1). The Petri-net representation of ABP (Figure 5.2) is transformed from its UO-GLOTOS specification of Figure 5.1. In order to testing the sixth operation, the specification of the Medium (from sender to receiver) of ABP is changed (Figure 5.3), based on our coverage definition for operations involving recursions (Definition 3.2 in Chapter 3). This change is made to limit the times of the iterative execution of a recursive expression to only once. It is widely used in testing infinitive processes. Figure 5.4 shows the correspondence between the nodes of the UO-GLOTOS specification (Figure 5.1) and the transitions of the Petri-net (Figure 5.2). Figures 5.5, 5.6 and 5.13 show some of the computed data and the generated test sequences.

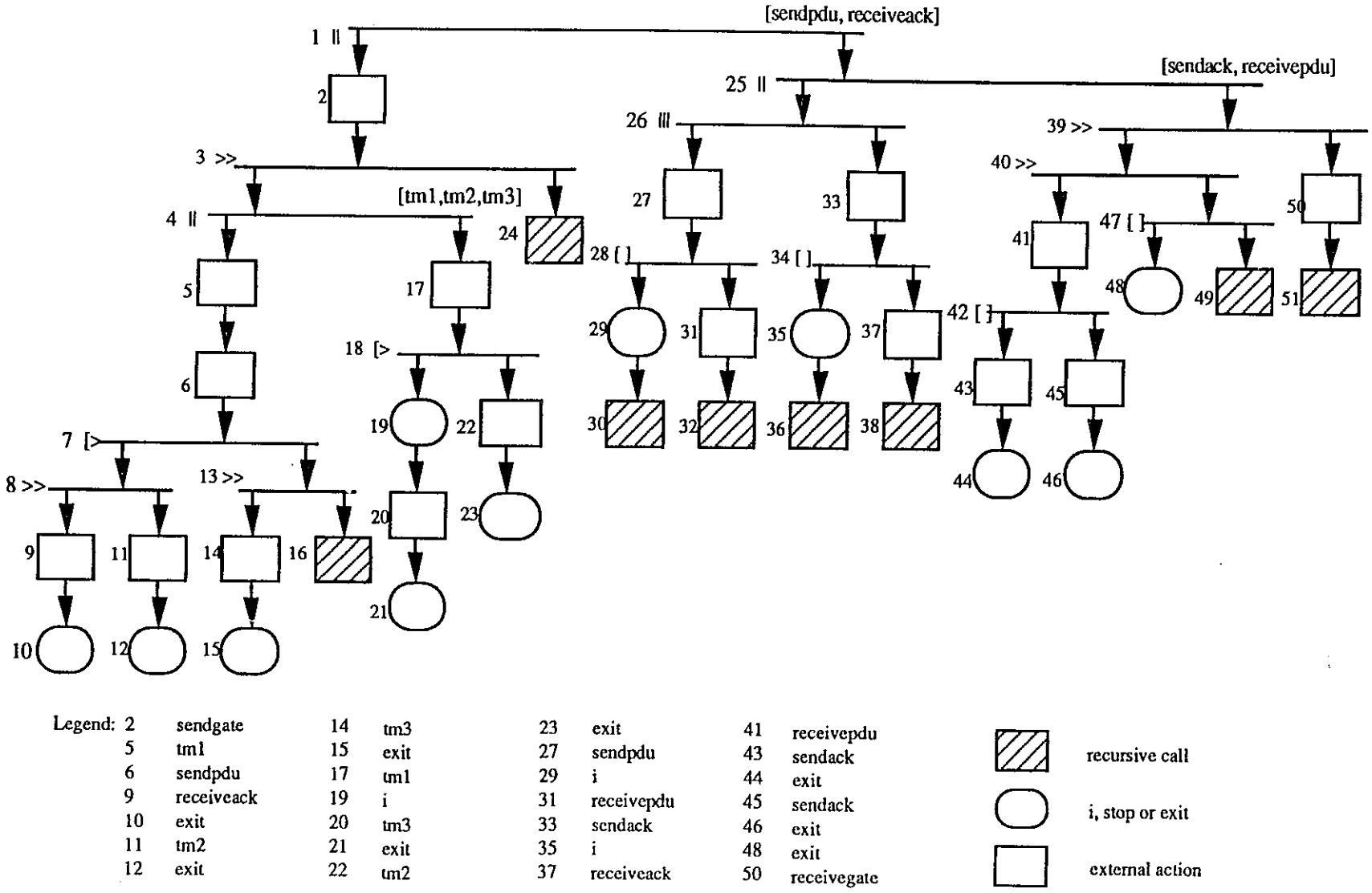
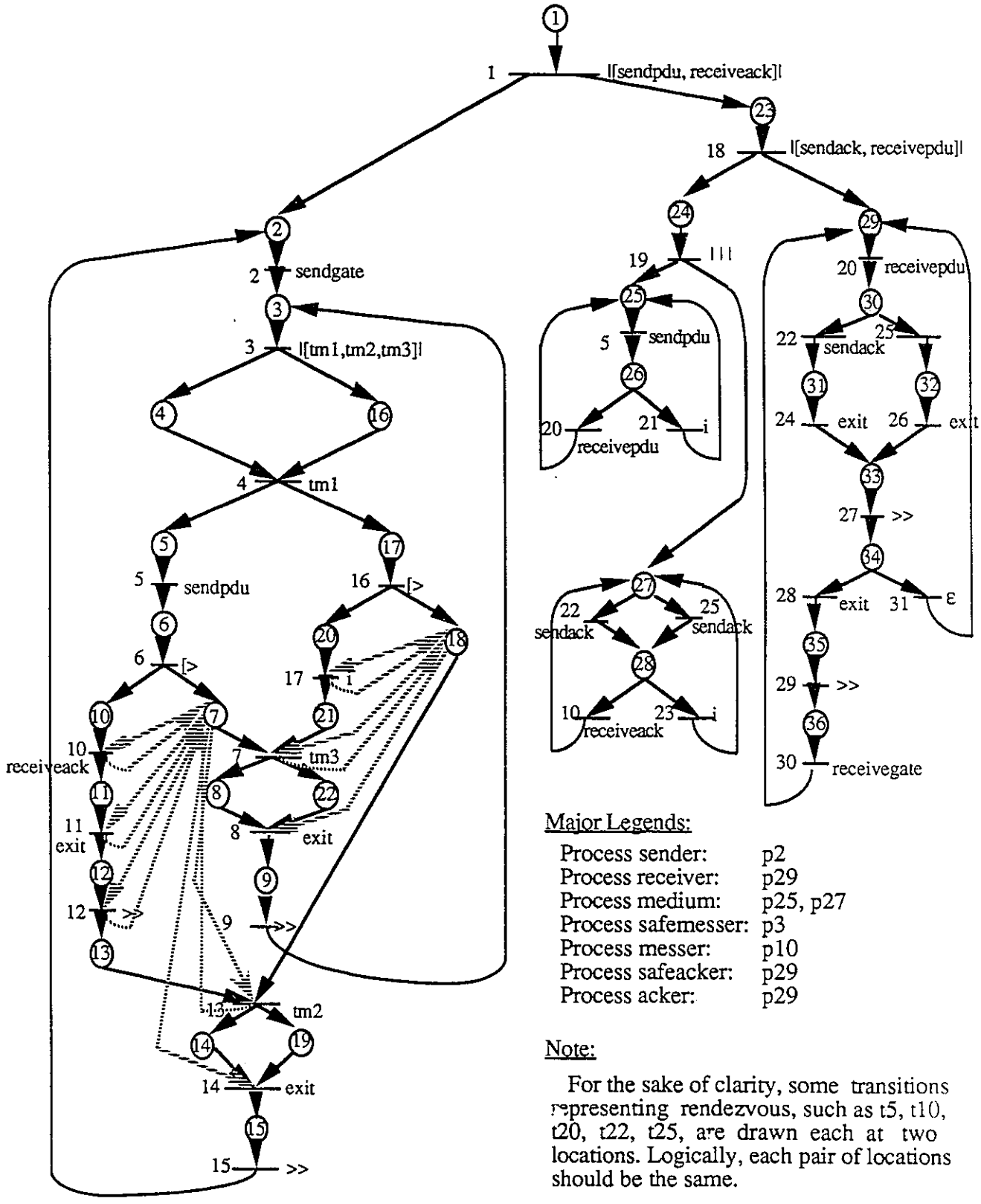


Figure 5.1. The UO-GLOTOS representation of the Alternating Bit Protocol.



Major Legends:

Process sender:	p2
Process receiver:	p29
Process medium:	p25, p27
Process safemesser:	p3
Process messer:	p10
Process safeacker:	p29
Process acker:	p29

Note:

For the sake of clarity, some transitions representing rendezvous, such as t5, t10, t20, t22, t25, are drawn each at two locations. Logically, each pair of locations should be the same.

Figure 5.2. The Petri-net representation of the Alternating Bit Protocol.

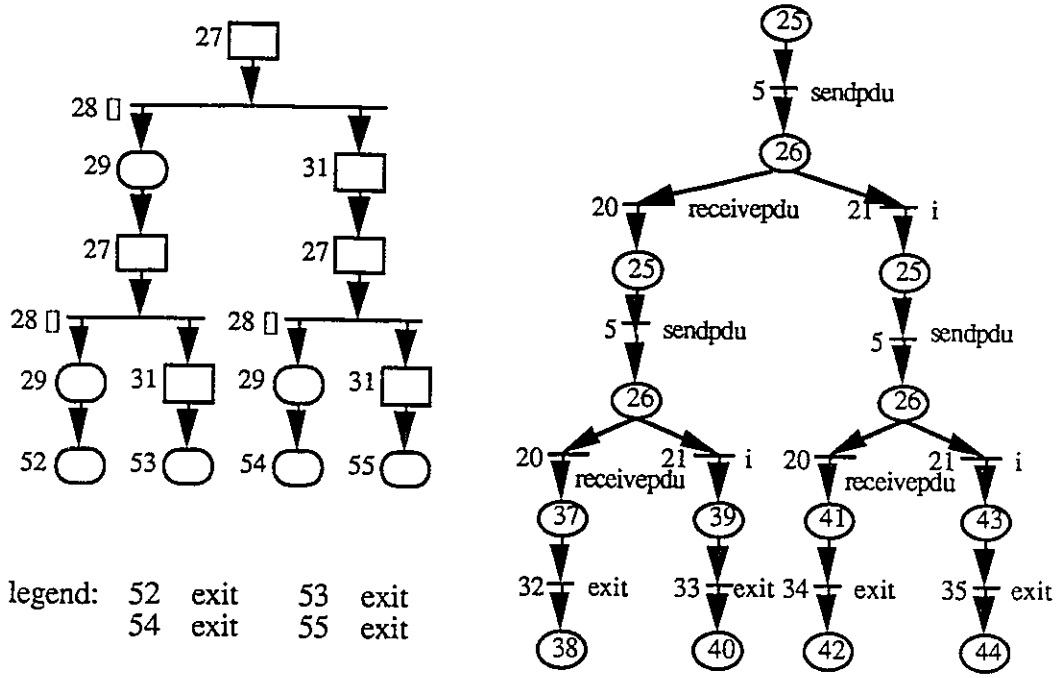


Figure 5.3 The UO-GLOTOS and Petri-net representations of the Medium (from sender to receiver) of ABP.

node	transition or place	node	transition or place	node	transition or place	node	transition or place	node	transition or place
1	1	12	14	23	14	34	place 28	45	25
2	2	13	9	24	place 2	35	23	46	26
3	15	14	7	25	18	36	place 27	47	place 34
4	3	15	8	26	19	37	10	48	28
5	4	16	place 3	27	5	38	place 27	49	place 29
6	5	17	4	28	place 26	39	29	50	30
7	6	18	16	29	21	40	27	51	place 29
8	12	19	17	30	place 25	41	20	52	35
9	10	20	7	31	20	42	place 30	53	34
10	11	21	8	32	place 25	43	22	54	33
11	13	22	13	33	22 (25)	44	24	55	32

Figure 5.4. Correspondence between node of UO-GLOTOS (Figure 5.1) and transitions (places) of PN (Figure 5.2).

Figure 5.5 shows the guides generated by Algorithm GuideGen. The inputs to this algorithm are the LOTOS specification of ABP and six designated operations (specified as node in UO-GLOTOS): (1) node 4, (2) node 26, (3) node 28, (4) node 18, (5) node 39 and (6) node 30,32. The output is a set of guides for the six operations of ABP shown below as nodes in UO-GLOTOS (Figure 5.1).

Results of applying Algorithm GuideGen			
Operator op		Cover C(E, op)	A set of guides in node format
1	|[tm1,tm2,tm3]| node n4	{tm1,sendpdu,receiveack,tm2,exit} ||| {tm1,tm2,exit}	{n1,n2,n4,n5,n6,n7,n9,n10,n8,n11,n12,n17,n18, n22,n23}
2	||| node n26	{sendpdu,receivepdu} ||| {sendack}	{n1,n25,n26,n27,n28,n31,n33,n34,n35}
3	[] node n28	ϕ {receivepdu}	{n1,n25,n26,n27,n28,n29} {n1,n25,n26,n27,n28,n31}
4	[> node n18	{tm3,exit} {tm2,exit} {(i),tm2,exit}	{n1,n2,n4,n17,n18,n19,n20,n21} {n1,n2,n4,n17,n18,n22,n23} {n1,n2,n4,n17,n18,n19,n22,n23}
5	>> node n39	{receivepdu,sendack, receivegate}	{n1,n25,n41,n42,n43,n44,n40,n47,n48,n39,n50}
6	recursive process node n30, 32	{sendpdu, receivepdu, sendpdu} {sendpdu,sendpdu}	{n1,n25,n26,n27,n28,n31,n27,n28,n29,n54} {n1,n25,n26,n27,n28,n29,n27,n28,n29,n52}

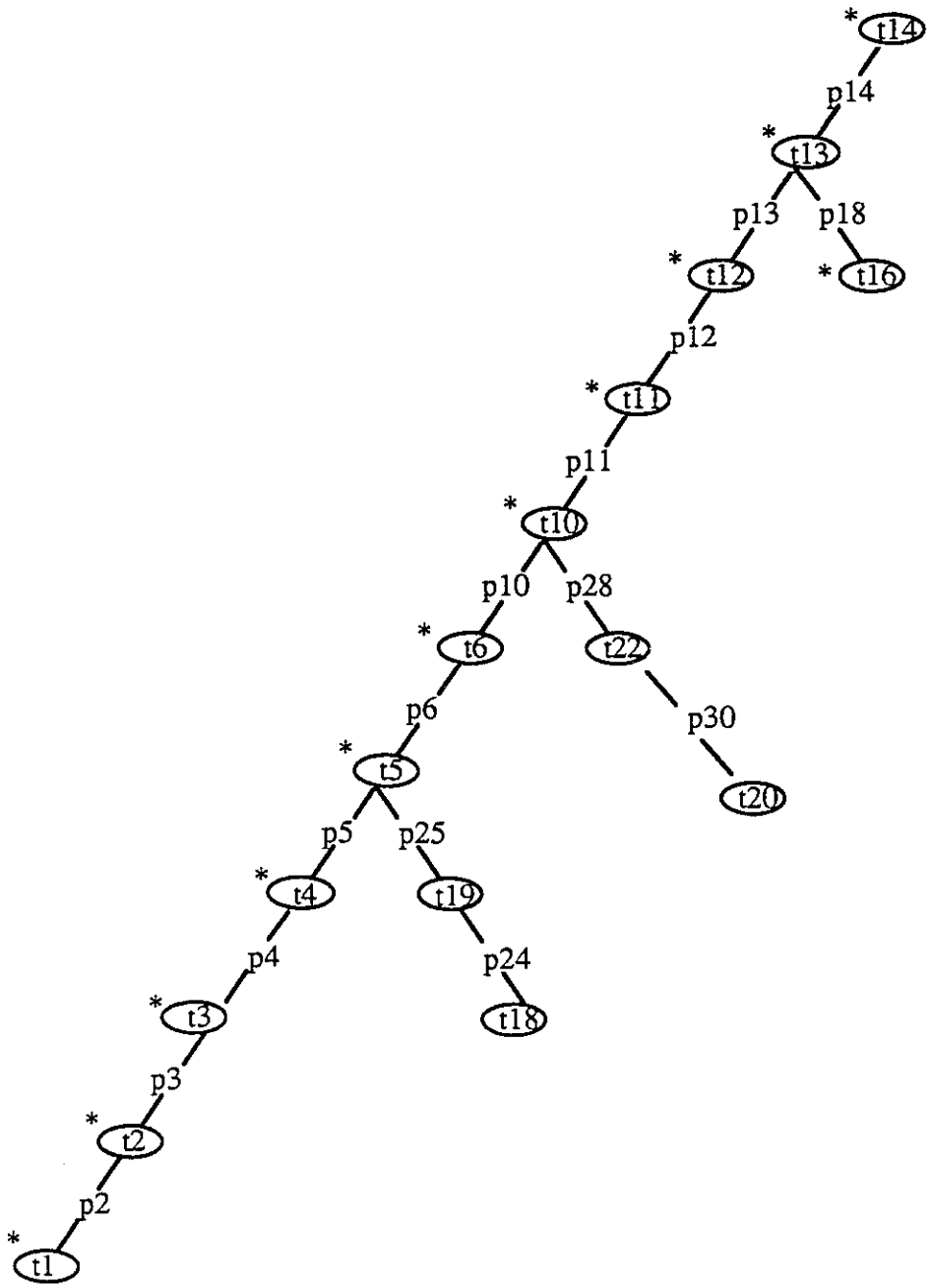
Figure 5.5. A set of guides generated by GuideGen for six operations of ABP.

Figure 5.6 shows the results of applying Algorithm TranseqGen. The input to this algorithm is the set of guides in Column 3 of Figure 5.5 after being transformed to the transition format of the Petri-net (Column 2 of Figure 5.6). The output is a set of executable transition sequences (Column 3 of Figure 5.6).

Results of applying Algorithm TranseqGen			
Operator op		A set of guides in transition format	Executable transition sequences
1	![tm1,tm2,tm3]! transition t3	{1,2,3,4,5,6,10,11,12,13,14, 4,16,13,14}	{1,2,3,4,18,19,5,6,20,22,10,11,12,16,13,14}
2	||| transition t19	{1,18,19,5,20,25,23}	{1,18,19,2,3,4,5,20,25,23}
3	[] place 26	{1,18,19,5,21} {1,18,19,5,20}	{1,18,19,2,3,4,5,21} {1,18,19,2,3,4,5,20}
4	[> transition t16	{1,2,3,4,16,17,7,8} {1,2,3,4,16,13,14} {1,2,3,4,16,17,13,14}	{1,2,3,4,16,17,18,19,5,6,7,8} {1,2,3,4,16,18,19,5,6,20,22,10,11,12,13,14} {1,2,3,4,16,17,18,19,5,6,20,22,10,11,12,13,14}
5	>> transition t29	{1,18,20,22,24,27,28,29,30}	{1,18,19,2,3,4,5,20,22,24,27,28,29,30}
6	recursive process place 25	{1,18,19,5,20,5,21,33} {1,18,19,5,21,5,21,35}	{1,18,19,2,3,4,5,20,6,16,17,7,8,9,3,4,5,21,33} {1,18,19,2,3,4,5,21,6,16,17,7,8,9,3,4,5,21,35}

Figure 5.6. Executable transition sequences generated by TranseqGen for six operations of ABP.

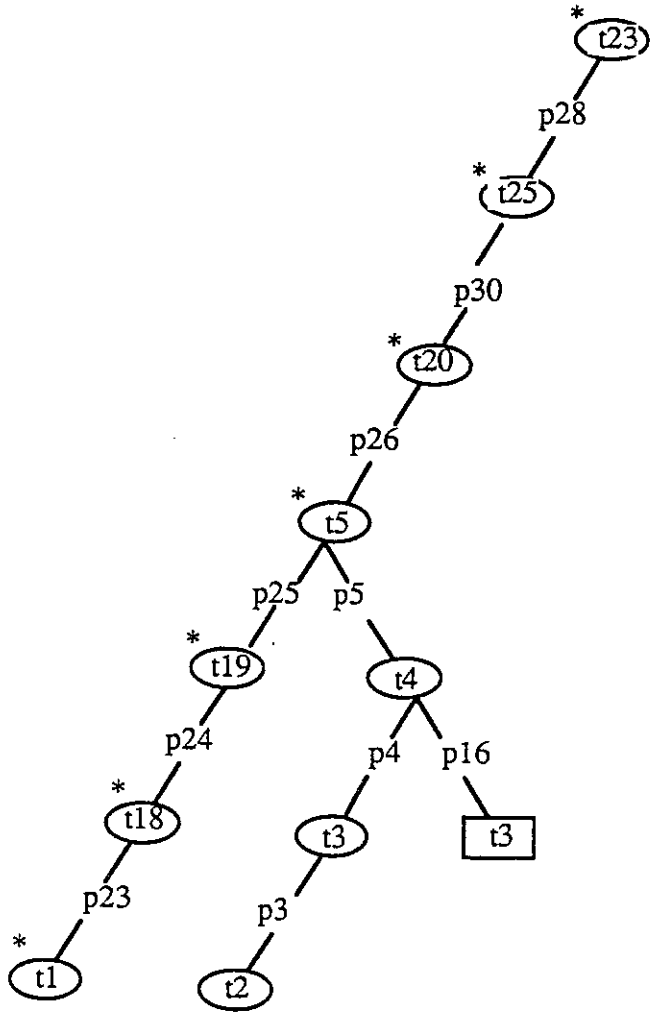
Figures 5.7 to 5.12 show the details of applying Algorithm TranseqGen (Step 2) for each of the six specified operations. In these figures, the set of transitions T in ovals with '*' are in the guide obtained from GuideGen. For every transition t in T, examine its executability and find a set of transitions T' (if there is any) which must be fired before executing t. Each of the transitions in T' is represented by a oval without '*' and attached under t in the figures. Repeat the above procedure for every transition in T' till no further such transition is found. After examining all of the transitions in T, an executable transition sequence is formed in a left-branch first order. The transitions in a square are already included in an executable transition sequence and are therefore ignored. For example, in Figure 5.7, t5 is examined. It can not be fired until t4 and t19 have been executed. Since t4 has already been in the guide -- a nonexecutable sequence, only t19 is attached under t5. Then, t19 is examined and t18 is found to be executed before t19 and be attached under it.



Guide G: t1 t2 t3 t4 t5 t6 t10 t11 t12 t13 t14 t4 t16 t13 t14

Executable transition sequence: t1 t2 t3 t4 t18 t19 t5 t6 t20 t22 t10 t11 t12 t16 t13 t14

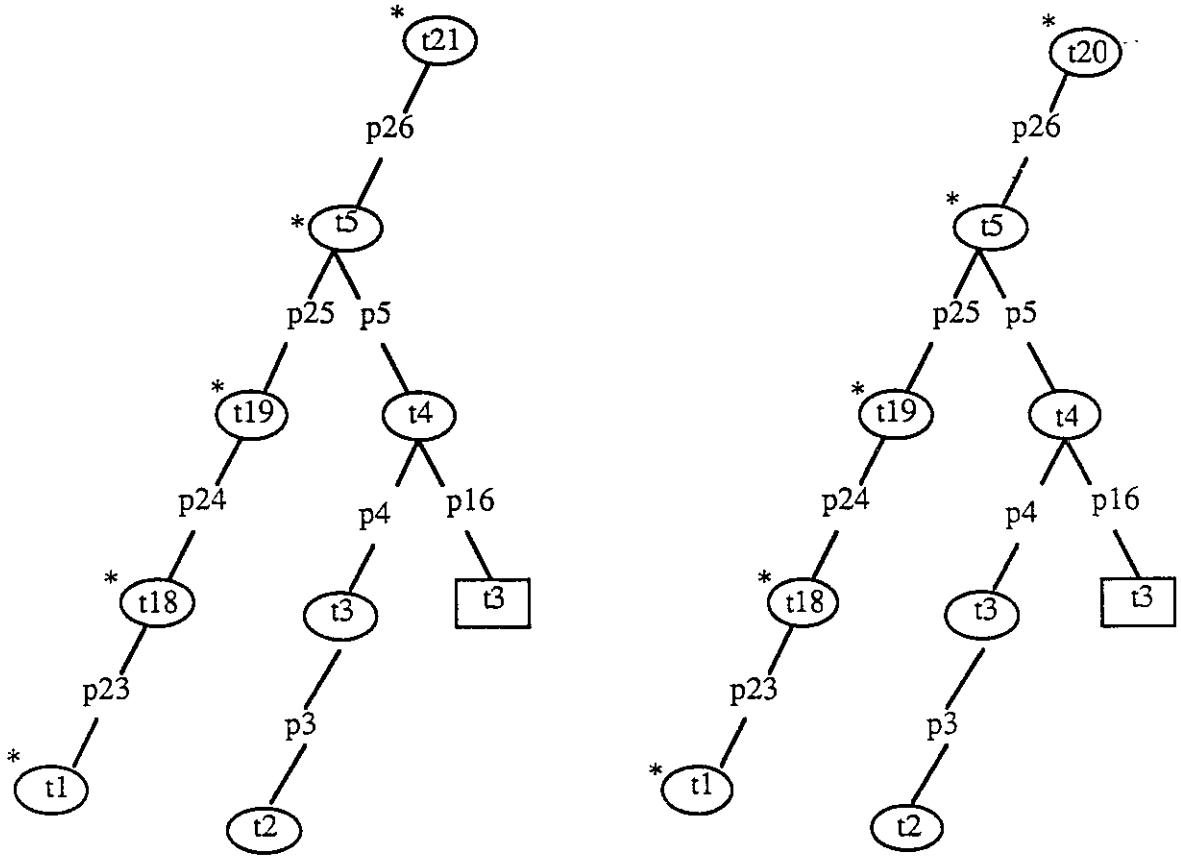
Figure 5.7. Detail of applying TranseqGen for the cover C(B, ![tm1,tm2,tm3]!).



Guide G: t1 t18 t19 t5 t20 t25 t23

Executable transition sequence: t1 t18 t19 t2 t3 t4 t5 t20 t25 t23

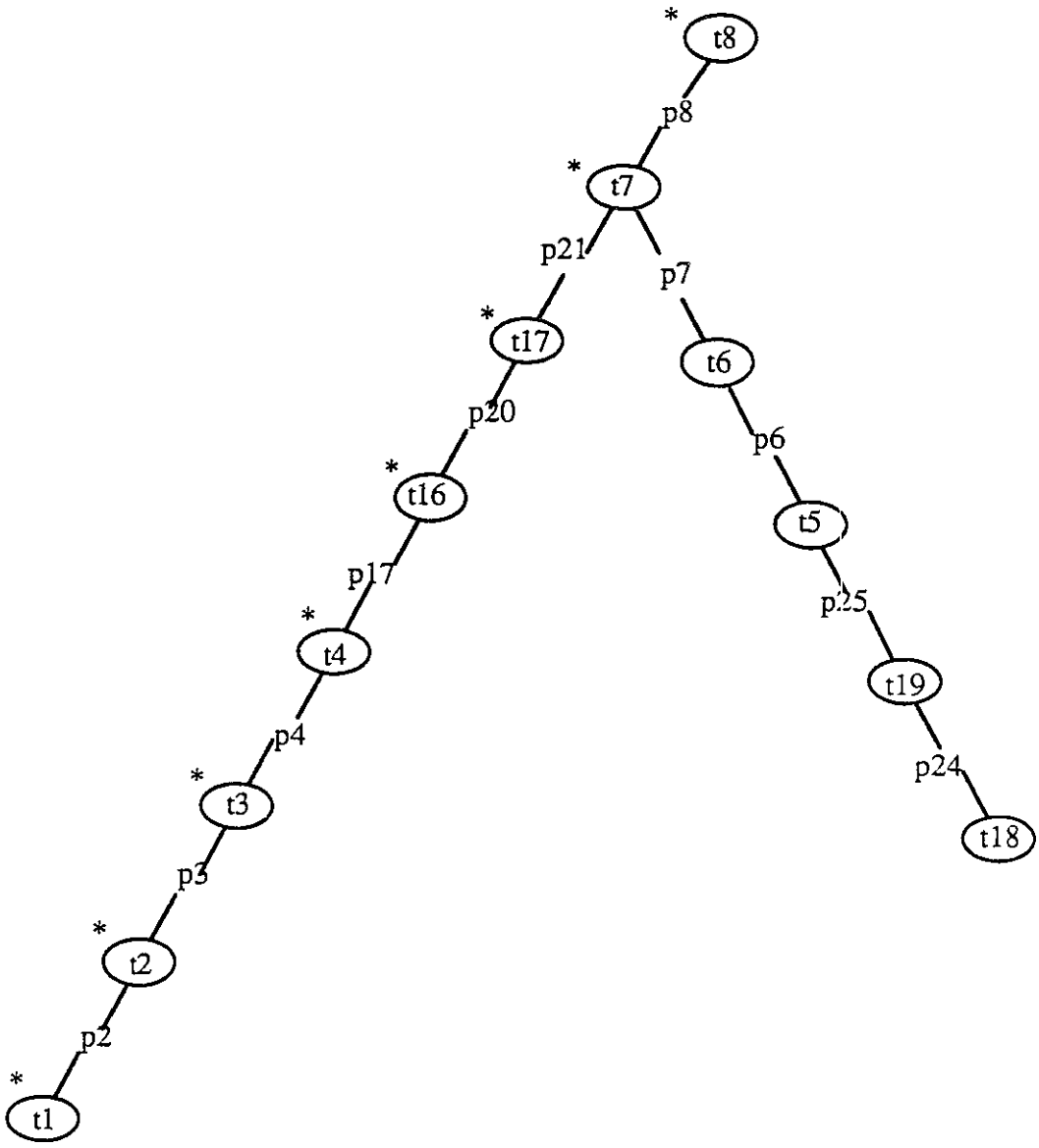
Figure 5.8. Detail of applying TranseqGen for the cover C(B, III).



Guides {G}: t1 t18 t19 t5 t21, t1 t18 t19 t5 t20

Executable transition sequences: t1 t18 t19 t2 t3 t4 t5 t21, t1 t18 t19 t2 t3 t4 t5 t20

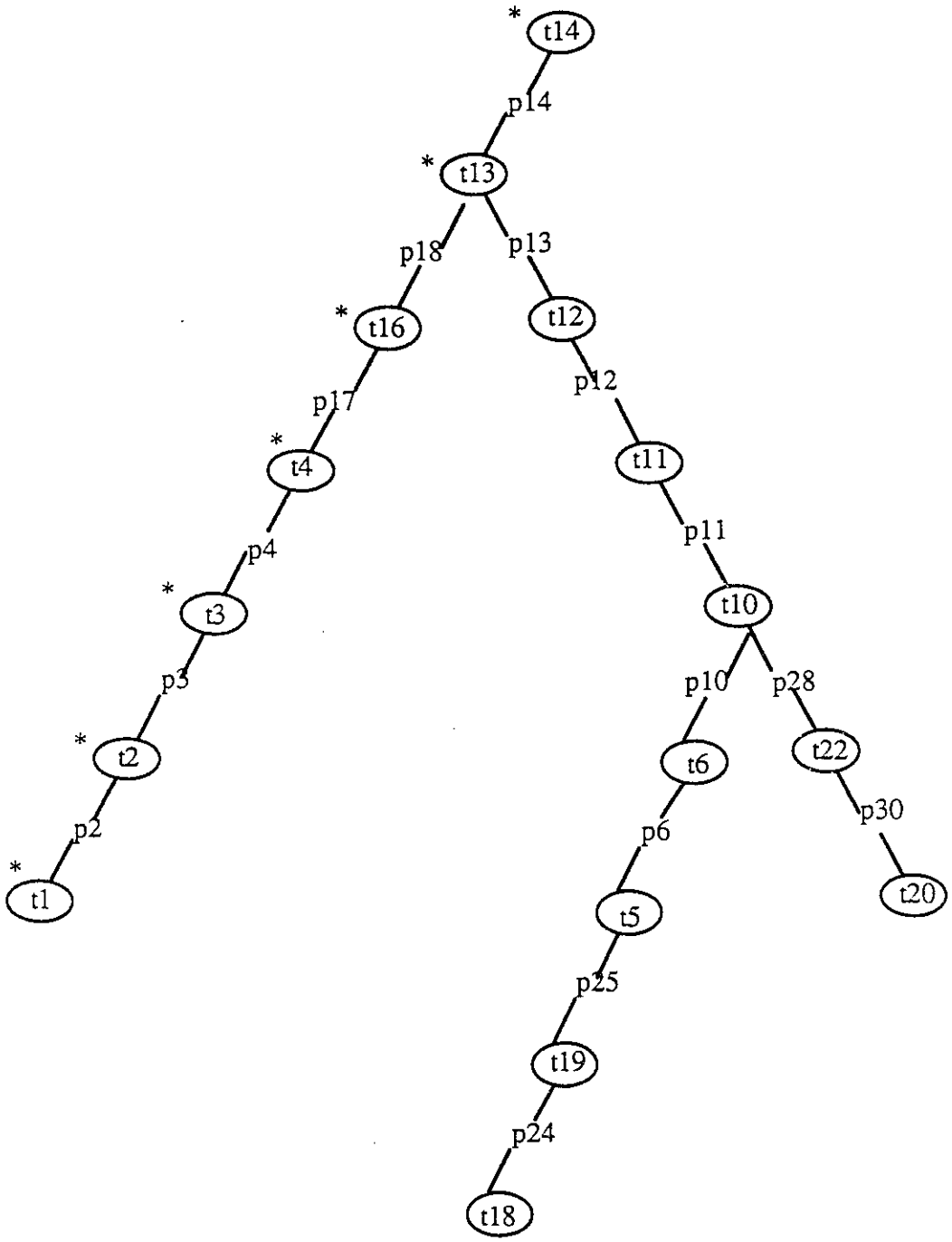
Figure 5.9. Detail of applying TranseqGen for the cover C(B, []).



Guide G: t1 t2 t3 t4 t16 t17 t7 t8

Executable transition sequence: t1 t2 t3 t4 t16 t17 t18 t19 t5 t6 t7 t8

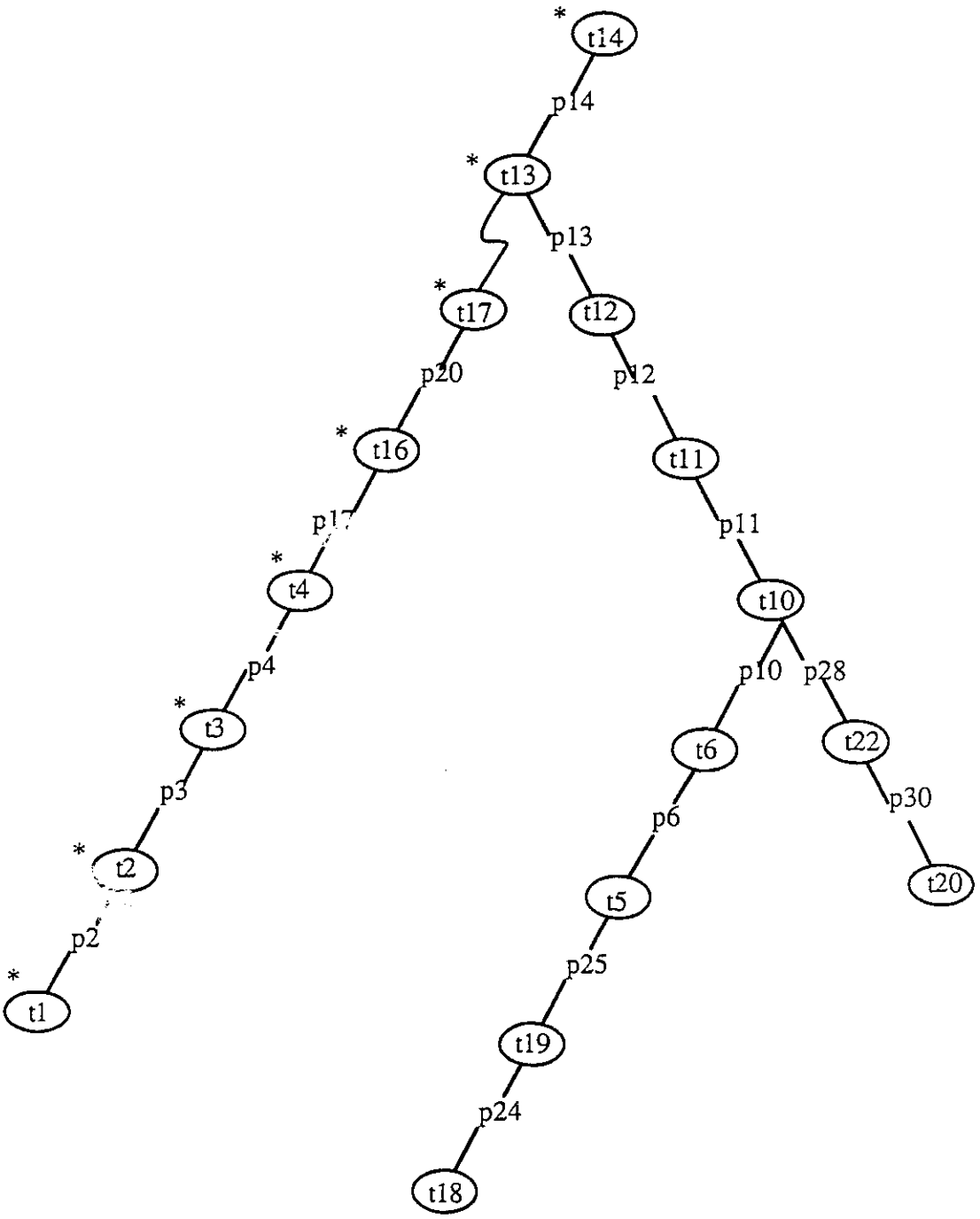
Figure 5.10.a. Detail of applying TranseqGen for the cover C(B, [>).



Guide G: t1 t2 t3 t4 t16 t13 t14

Executable transition sequence: t1 t2 t3 t4 t16 t18 t19 t5 t6 t20 t22 t10 t11 t12 t13 t14

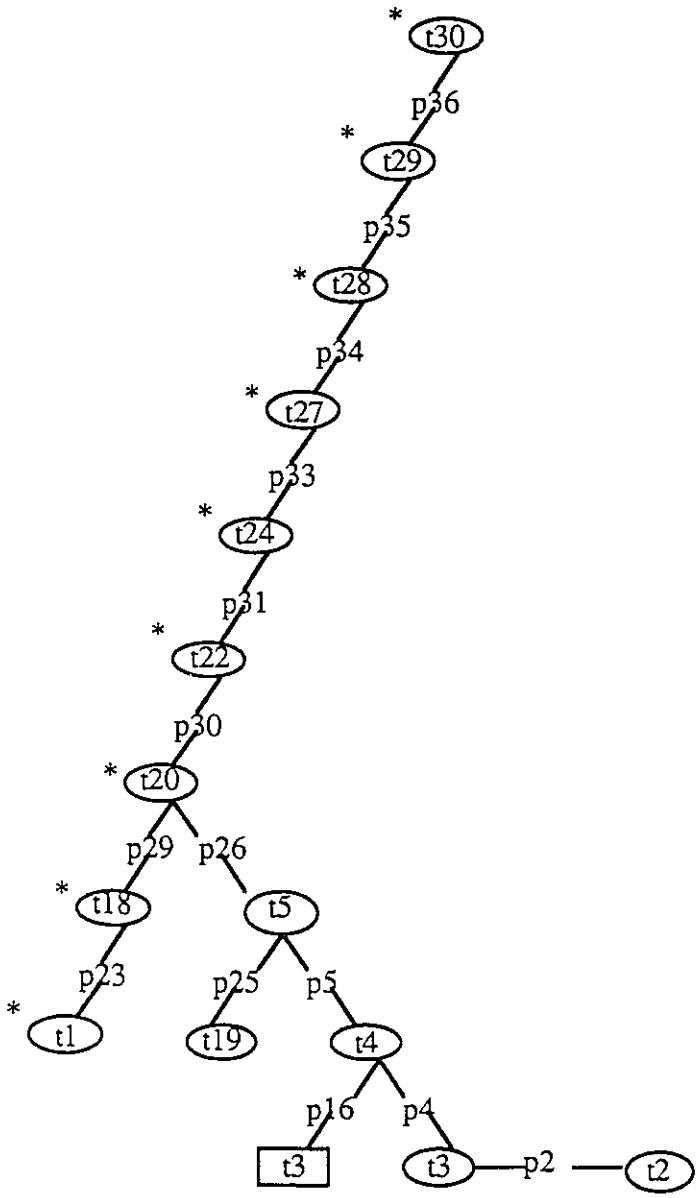
Figure 5.10.b. Detail of applying TranseqGen for the cover C(B, [>).



Guide G: t1 t2 t3 t4 t16 t17 t13 t14

Executable transition sequence: t1 t2 t3 t4 t16 t17 t18 t19 t5 t6 t20 t22 t10 t11 t12 t13 t14

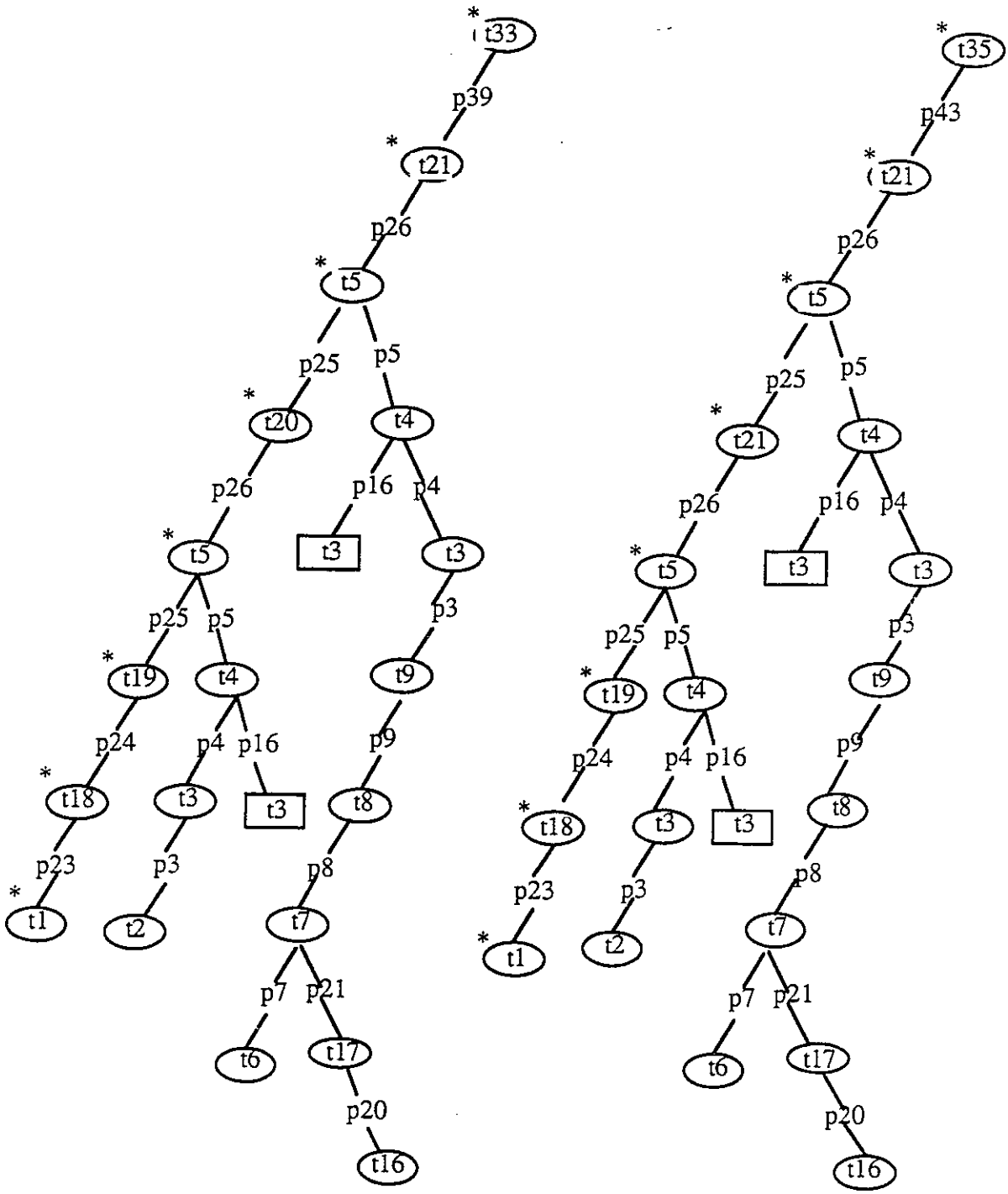
Figure 5.10.c. Detail of applying TranseqGen for the cover C(B, [>).



Guide G: t1 t18 t20 t22 t24 t27 t28 t29 t30

Executable transition sequence: t1 t18 t19 t2 t3 t4 t5 t20 t22 t24 t27 t28 t29 t30

Figure 5.11. Detail of applying TranseqGen for the cover C(B, >>).



Guides {G}: t1 t18 t19 t5 t20 t5 t21 t33, t1 t18 t19 t5 t21 t5 t21 t35

Executable transition sequences: t1 t18 t19 t2 t3 t4 t5 t20 t6 t16 t17 t7 t8 t9 t3 t4 t5 t21 t33,
t1 t18 t19 t2 t3 t4 t5 t21 t6 t16 t17 t7 t8 t9 t3 t4 t5 t21 t35

Figure 5.12. Detail of applying TranseqGen for the recursive process instantiation operation.

Figure 5.13 shows the set of executable test sequences obtained by applying Algorithm TestseqGen. The input to the algorithm is the set of transition sequences in Column 3 of Figure 5.6, and the output is a set of executable test sequences (Column 2 of Figure 5.13).

Results of applying Algorithm TestseqGen		
Operator op		Executable test sequences
1	|[tm1,tm2,tm3]|	{sendgate,tm1,sendpdu,receivepdu,sendack,receiveack,tm2,exit}
2	|||	{sendgate,tm1,sendpdu,receivepdu,sendack}
3	[ ]	{sendgate,tm1,sendpdu} {sendgate,tm1,sendpdu,receivepdu}
4	[>	{sendgate,tm1,sendpdu,tm3,exit} {sendgate,tm1,sendpdu,receivepdu,sendack,receiveack,tm2,exit} {sendgate,tm1,(i),sendpdu,receivepdu,sendack,receiveack,tm2,exit}
5	>>	{sendgate,tm1,sendpdu,receivepdu,sendack,receivegate}
6	recursive process	{sendgate,tm1,sendpdu,receivepdu,tm3,tm1,sendpdu,exit} {sendgate,tm1,sendpdu,tm3,tm1,sendpdu,exit}

Figure 5.13. Executable test sequences generated by TestseqGen covering six operations of ABP.

Chapter 6

CONCLUSION AND FUTURE RESEARCH

In this thesis, we report our research on an automated method for selective test sequence generation from LOTOS specifications. It consists of the proposal of an operational coverage criterion for single and multiple LOTOS operations and the development of a method called SELECTEST for selectively generating executable test sequences based on the criterion.

All the methods reported in the literature for test case generation from LOTOS specifications are exhaustive, in the sense they can generate all the test cases without allowing any systematic and automated means for selection. Though some of these methods are claimed to be selective, their selection processes are either carried out by human 'walkthrough' or based on some state transition models. In our method, the selection is carried out based on LOTOS specifications during the process of generation. Based on the operational coverage criterion, SELECTEST generates the guides and then the test sequences for the specified operations. These test sequences are able to detect implementational errors of functional and structural nature, as the operations of a LOTOS specification play a major role in these aspects.

It should be pointed out that there exists an unsolved issue in using Petri-nets to represent some forms of LOTOS recursive process instantiations. The first form is the recursive process instantiation occurring in the left sub-expression B1 of B1 >> B2 or B1 [> B2. For example, B = (a; B [] b; exit) >> c; d; exit. The second one is the recursive definition of a process occurring in sub-expression B1 or B2 of B1 |[G]| B2. In this form, the LOTOS expression may not be represented by a Petri-net if the number of synchronization gates in B1 and B2 is not the same. For instance, B = (a; b; a; B) |[a]| (a; a; c; a; exit). This problem exists not only in Cheung's representation, but also in all other Petri-net representations, such as Galileo net and the one used in [GAR90]. In fact, the problem of representing recursion exists not only in Petri-nets, but also in other finite models. In this thesis, however, general representation of LOTOS by Petri-nets is not our concern. In our method for selective testing, our criterion allows only one iteration of a

recursive process and thus we transform an infinite system into a finite one. Since the exhaustive testing is impossible, especially for an infinite system, this kind of restrictions for processing recursions is quite common in practice.

Future research includes the following works:

At present, SELECTEST is applied to basic LOTOS (i.e. the control part of LOTOS) only. Including data in the selection process is one area of further study. The data aspect of LOTOS specifications plays an important role in execution. For instance, predicates and parametric values may determine whether an action can be executed or not. Eventually, involvement of data in test sequence generation is necessary in practice. Another selective testing method based on DU-paths of the data part of a LOTOS specification is being investigated under the direction of Dr. Cheung. We may combine that method with ours for selective test sequence generation.

In addition, the operational coverage criterion proposed in this thesis may be further refined. For example, in our definition, only one local test sequence is chosen for the interleaving and general-parallel operations, regardless of what order of actions and how many actions are involved in these operations. We may define a criterion which selects several local test sequences in different orders of actions. But, then there may not exist feasible test sequences fulfilling the criterion, because the parallel operations at an upper level may affect the order of actions at lower levels during execution. Therefore, as in all other static-information-based test sequence selection methods, the feasibility of test sequences has to be enforced during actual testing.

If the test sequences derived according to one criterion are a subset of those derived according to another criterion, we say the latter criterion is stronger than the former one. Currently, the strength of the criterion proposed in this thesis has not been compared with others, because there exist no other selection criteria for LOTOS specifications. In the future, when other criteria for selective test sequence generation from LOTOS are available, we can compare the strength of the different criteria.

We have dealt only with the problem of generating selected test sequences from LOTOS specifications. There are other difficult related issues about testing LOTOS implementations. One

of them is the non-deterministic nature of distributed systems specified in LOTOS. Because of nondeterminism, a test suite applied to the IUT may result in a "non-conclusive" verdict. That is, the IUT may respond differently when the same test sequence is applied under the same conditions but at different times. Cheung, et al. propose an approach for generating test sequences including non-deterministic property and a fairness model for conformance testing. Since it is out of the scope of this thesis, we will not discuss it further. Interested readers are referred to [CHE91].

APPENDIX

A. RULES FOR TRANSFORMING LOTOS CONSTRUCTS TO PETRI-NETS

This appendix contains the formal definition of the rules of transforming LOTOS specifications to Cheung's Petri-net representations. A pictorial description of these rules is given in Chapter 2. The following notation will be used in the transformation rules.

Notation.

Tx(S)	= {t | t ∈ T(S), l(t) = 'exit'}, i.e., a finite set of 'exit' transitions.
Ts(S)	= {t | t ∈ T(S), l(t) = 'stop'}, i.e., a finite set of 'stop' transitions.
Txs(S)	= Tx(S) ∪ Ts(S).
G	= {g1, g2,..., gn}, i.e., a finite set of gates in LOTOS.
Tg(G, S)	= {t | t ∈ T(S), l(t) ∈ G}, i.e., a finite set of transitions labelled as gi, i = 1, 2, ..., n.
Fx(S)	= {(p, t) (t, q) | (p, t), (t, q) ∈ F(S), t ∈ Tx(S)}, i.e., a finite set of arcs from and to 'exit' transitions.
Fs(S)	= {(p, t) (t, q) | (p, t), (t, q) ∈ F(S), t ∈ Ts(S)}, i.e., a finite set of arcs from and to 'stop' transitions.
Fxs(S)	= Fx(S) ∪ Fs(S).
Fg(G, S)	= {(p, t) (t, q) | (p, t), (t, q) ∈ F(S), t ∈ Tg(G, S)}, i.e., a finite set of arcs to and from transitions labelled as gi, i = 1, 2, ..., n.
tij = ti x tj	Merging transitions ti and tj to the transition tij.

The Transformation Rules:

LOTOS construct	Petri-net representation (PN net) of LOTOS constructs
a; Ba	p0 -> t0; t0 -> pia; l(t0) = a.

Ba [] Bb	p0 = pia =pib; pxa ≡ pxb; psa ≡ psb; P(Ba[]Bb) = P(Ba) ∪ (P(Bb) - {pxb, psb} - pib}; T(Ba[]Bb) = T(Ba) ∪ T(Bb); F(Ba[]Bb) = F(Ba) ∪ F(Bb); l(Ba[]Bb) = l(Ba) ∪ l(Bb); pi = p0; px = pxa = pxb; ps = psa = psb.
Ba >> Bb	pxa -> ε; ε -> pib; psa ≡ psb; P(Ba >> Bb) = (P(Ba) - {psa}) ∪ P(Bb); T(Ba >> Bb) = T(Ba) ∪ T(Bb) ∪ {ε}; F(Ba >> Bb) = F(Ba) ∪ F(Bb) ∪ {(pxa, ε), (ε, pib)}; l(Ba >> Bb) = l(Ba) ∪ l(Bb) ∪ {(ε, '>>'}; pi = pia; px = pxb; ps = psa = psb.
Ba [> Bb	p0 -> ε; ε -> pia; ε -> pib; psa ≡ psb; pxa ≡ pxb; P(Ba [> Bb) = P(Ba) ∪ (P(Bb) - {pxb, psb}) ∪ {p0}; T(Ba [> Bb) = T(Ba) ∪ T(Bb) ∪ {ε}; F(Ba [> Bb) = F(Ba) ∪ F(Bb) ∪ F([>), where F([>) = {(pib, t), (t, pib) | (t ∈ T(Ba) - Txs (Ba)) ∧ (t ≠ ε)} ∪ {(pib, t) | t ∈ Txs (Ba)} ∪ {(p0, ε), (ε, pia), (ε, pib)}; l(Ba [> Bb) = l(Ba) ∪ l(Bb) ∪ {(ε, ' [>')}; pi = p0; px = pxa = pxb; ps = psa = psb.
Ba ||| Bb	p0 -> ε; ε -> pia; ε -> pib; psa ≡ psb; pxa ≡ pxb; P(Ba ||| Bb) = (P(Ba) - {pxa}) ∪ (P(Bb) - {pxb, psb}) ∪ {p0, p1}; T(Ba ||| Bb) = (T(Ba) - Tx(Ba)) ∪ (T(Bb) - Tx(Bb)) ∪ T(|||) ∪ {ε}, where T(|||) = {tij |tij = ti x tj, where ti ∈ Tx(Ba), tj ∈ Tx(Bb)};

	F(Ba ||| Bb) = (F(Ba) - Fx(Ba)) ∪ (F(Bb) - Fx(Bb)) ∪ F(|||), where F(|||) = {(p, tij) | tij ∈ T(|||) ∧ ((p,ti) ∈ Fx(Ba) ∨ (p,tj) ∈ Fx(Bb))} ∪ {(tij, p1) | tij ∈ T(|||)} ∪ {(p0, ε), (ε, pia), (ε, pib)};
	l(Ba ||| Bb) = (l(Ba) ∪ l(Bb) - {(t, 'exit') | t ∈ Tx(Ba) ∪ Tx(Bb)}) ∪ {(ε, '|||')} ∪ {(tij,'exit') | tij ∈ T(|||)};
	pi = p0; px = pxa = pxb; ps = psa = psb.
Ba |[G]| Bb	p0 -> ε; ε -> pia; ε -> pib; psa ≡ psb; pxa ≡ pxb;
G = [g1,...,gn]	P(Ba |[G]| Bb) = P(Ba ||| Bb);
	T(Ba |[G]| Bb) = T(Ba ||| Bb) - Tg(G, Ba) - Tg(G, Bb)) ∪ T(|[G]|), where T(|[G]|) = {tij | tij = ti X tj, where ti ∈ Tg(Ba) and tj ∈ Tg(Bb)};
	F(Ba |[G]| Bb) = (F(Ba ||| Bb) - Fg(G, Ba) - Fg(G, Bb)) ∪ F(|[G]|), where F(|[G]|) = {p,tij) | tij ∈ T(|[G]| ∧ ((p, ti) ∈ Fg(G, Ba) ∨ (p,tj) ∈ Fg(G, Bb))} ∪ {(tij,p) | tij ∈ T(|[G]|) ∧ ((ti, p) ∈ Fg(G, Ba) ∨ (tj,p) ∈ Fg(G, Bb))};
	l(Ba |[G]| Bb) = (l(Ba ||| Bb) - {(t, l(t)) | t ∈ Tg(G, Ba) ∪ Tg(G,Bb)}) ∪ {(tij, l(ti)) | tij ∈ T(|[G]|)} ∪ {(ε, '|[]|')};
	pi = p0; px = pxa = pxb; ps = psa =psb.
process instantiation	A process instantiation P[G] is formed by a process identifier P with an associated gate list G = g1, g2,..., gn. Let G' = g1', g2',..., gn'. If G' is not equal to G, P[G'] describes a different process from P[G]. The corresponding PN net of P[G'] is the same as PN net of P[G] with a new labelling function defined below: $l(t) = \begin{cases} (t, gi') & t \in T(P) \wedge l(t) \neq gi \text{ where } i = 1...n \\ (t, gi) & \text{otherwise} \end{cases}$

B. THE LOTOS SPECIFICATION OF THE ALTERNATING BIT PROTOCOL

This appendix contains the textual LOTOS specification of ABP extracted from [BER88]. For easy comparison, place number pxx or transition number tyy corresponding to those in Figure 5.2 have been added in the rightmost column.

```
                                                                    Place or transition No.
specification  alternate [sendgate,receivegate] : noexit                          p1
behaviour
 hide  sendpdu,receiveack,receivepdu,sendack   in
  (      sender[sendgate,sendpdu,receiveack] (0 of Bit)                         p2
    |[sendpdu,receiveack]|                                                      t1
       (  medium[sendpdu,receivepdu]                                            p25
        |||                                                                     t19
          medium[sendack,receiveack]                                            p27
       )
    |[sendack,receivepdu]|                                                      t18
       receiver[receivegate,receivepdu,sendack] (0 of Bit)  )                   p29
where
process    medium[inporte,out] : noexit :=                    p25     p27
 inporte  ? msg : message;                                    t5      t22
 (  (  i; medium[inporte,out] )                               t21,p25 t23,p27
       [ ]                                                    p26     p28
       out ! msg;  medium[inporte,out]  )                     t20,p25 t10,p27
endproc
process    sender[sendgate,sendpdu,receiveack] (seqsender : Bit) : noexit :=  p2
 sendgate ? datmsg : datamsg;                                                 t2
 safemesser[sendpdu,receiveack] (datmsg,seqsender)                            p3
 >>                                                                           t15
```

```
sender[sendgate,sendpdu,receiveack] (Succ(seqsender))                    p2
where
process  safemesser[sendpdu,receiveack] (datmsg : datamsg,               p3
   seqsender : Bit) : exit :=
   hide tm in
   (  (   tm ! timestart;                                                t4
          sendpdu ! makepdu(datmsg,seqsender);                           t5
          (   (  messer[receiveack]                                      p10
                 >>                                                      t12
                 tm ! reset ; exit )                                     t13,t14
             [>                                                          t6
                 (tm  !  expired ; exit )                                t7,t8
                 >>                                                      t9
                 safemesser[sendpdu,receiveack](datmsg,seqsender) ) ) )  p3
    |[tm]|                                                               t3
      timer[tm]  )                                                       p16
   where
      process  messer[receiveack] : exit :=                              p10
         receiveack ? ack : message ; exit                               t10,t11
      endproc
      process    timer[tm] : exit (Bool) :=                              p16
         tm  !  timestart ;                                             t4
         ( i ; tm !  expired ;  exit(false)                              t17,t7,t8
           [>                                                            t16
           ( tm  !  reset ;  exit(true) ))                               t13,t14
      endproc
endproc
```

```
endproc
process receiver[receivegate,receivepdu,sendack] (seqreceiver : Bit :        p29
 noexit :=
 safeacker[receivepdu,sendack] (seqreceiver)                                 p29
 >>                                                                          t29
 accept datmsg : datamsg in
    (  receivegate ! datmsg ;                                                t30
       receiver[receivegate,receivepdu,sendack] (Succ(seqreceiver))  )       p29
 where
 process safeacker[receivepdu,sendack] (seqreceiver : Bit) :                 p29
    exit(datamsg) :=
    acker[receivepdu,sendack] (seqreceiver)                                  p29
  >>                                                                         t27
    accept pdu : message, ok : Bool in
       (  [ok]     ->  exit(datamessage(pdu))                                t28
        [ ]                                                                  p34
          [not(ok)] ->  safeacker[receivepdu,sendack] (seqreceiver) )        p29
    where
    process acker[receivepdu,sendack] (seqreceiver : Bit) :                  p29
       exit(message,Bool) :=
       receivepdu ? pdu : message ;                                          t20
       (  [is-equal (sequence(pdu), seqreceiver)] ->
             sendack ! makeack(seqreceiver) ; exit(pdu,true)                 t22,t24
        [ ]                                                                  p30
          [is-not-equal (sequence(pdu),seqreceiver)] ->
             sendack ! makeack(Succ(seqreceiver)) ; exit(pdu,false)  )       t25,t26
    endproc
```

```
    endproc
  endproc
endspec
```

REFERENCES

[AHO88] A. V. Aho, A.T. Dahbura, D. Lee, M. U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours", Protocol Specification, Testing and Verification VIII, Edited by S. Aggarwal and K. Sabanani, (1988), pp.75-86.

[BAR90] M. Barbeau and G.V. Bochmann, "Deriving analysable Petri-nets from LOTOS specifications", Tech. Report, Universite de Montreal, (1990).

[BER88] P. Berlinguette and D. Gueraichi, "The alternating bit protocol: textual and graphical representations", Tech. Report TR-88-25, Dept. of Computer Science, Univ. of Ottawa. (Nov. 1988)

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

[BRI88] E. Brinksma, "A theory for the derivation of tests", Protocol Specification, Testing and Verification VIII, Edited by S. Aggarwal and K. Sabanani, (1988), pp. 63-73.

[BRI91] E. Brinksma, J. Tretmans and L. Verhaard, " A framework for test selection", Proc. 11th International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification, Stockholm, (June 1991), pp. 216-231.

[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 Technique for Distributed Systems and Communication Protocol, Vancouver, (1989), pp. 33-48.

[CHE90] T. Y. Cheung and Y. C. Ye, "An executor for graphical LOTOS", Proc. 3rd International Conference on Formal Description Technique for Distributed Systems and Communication Protocols, Madrid, Spain, (1990), pp.667-670.

[CHE91a] T. Y. Cheung, Y. Wu and X. Ye, " Generating test sequences and their degrees of indeterminism for protocols", Proc. 11th IFIP International Symposium on Protocol Specification, Testing and Verification, (1991), pp. 278-293.

[CHE91b] T. Y. Cheung, X. Zhu, Private communication.

[CHO78] T. Chow, " Testing software design modeled by finite-state machines", IEEE Trans. Software Engineering, Vol. SE-4, (1978), pp. 178-187.

[GAR90] H. Garavel, J. Sifakis, "Compilation and verification of LOTOS specifications", Protocol Specification, Testing and Verification X, Edited by L. Logrippo, R. L. Probert and H. Ural, (1990), pp. 379-394.

[GON70] G. Gonenc, " A method for the design of fault detection experiments", IEEE Trans. Computers., Vol C-19, (1970), pp. 551-558.

[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. Sabanani, (1988), pp.339-410.

[HOW86] W. E. Howden, "A functional approach to program testing and analysis", IEEE Trans. Software Engineering, SE-12(10), (1986), pp.997-1005.

[ISO8807] ISO8807, "Information processing systems - open systems interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behavior".

[ISO89] ISO/IEN JTC1/SC21 N3253, "G-LOTOS: a Graphical Syntax for LOTOS", 1989.

[ISO9646] ISO9646, "Information processing systems, open systems interconnection, OSI conformance testing methodology and framework", (1989).

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

[MIL81] E. Miller and W. E. Howden, "Software testing and validation techniques, IEEE tutorial, second edition", IEEE Computer Society Press, Los Alamitos, CA, (1981).

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

[NAI81] S. Naito and M. Tsunoyama, " Fault detection for sequence machines by transition tours", Proc. IEEE Fault Tolerant Computing Conference, (1981).

[PET81] J. Peterson, "Petri-net theory and the modeling of systems".

[RAP85] S. Papps and E. Weyuker, "Selecting software test data using data flow information", IEEE Trans. Software Engineering, SE-11(4), (1985), pp. 367-375.

[RAY87] D. Rayner, "OSI conformance testing", Computer Networks and ISDN Systems, Vol. 14, (1987), pp. 79-98.

[SAB88] K. Sabnani and A. Dahbura, " A protocol test generation procedure", Computer Networks & ISDN Systems, Vol.15, (1988), pp.285-297.

[SAR87] B. Sarikaya, G. Bochmann and E. Cerny, "A test design methodology for protocol testing", IEEE Trans. Software Engineering, Vol. SE-13, NO. 5, (1987), pp. 518-531.

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

[TRI91] P. Tripathy and B. Sarikaya, "Test generation from LOTOS specifications", IEEE Trans. on Computers, Vol. 40, No. 4, (1991), pp. 543-552.

[URA87] H. Ural, "A test derivation method for protocol conformance testing", Protocol Specification, Testing and Verification VII, edited by H. Rudin and C. H. West, (1987), pp. 347-357.

[URA88] H. Ural and B. Yang, "A structural test selection criterion", Information Processing Letters, Vol.28, (1988), pp. 157-163.

[URA91] H. Ural and B. Yang, "A test sequence selection method for protocol testing", IEEE Trans. on Communications, Vol. 39, No. 4, (1991), pp. 514-523.

[WEZ89] C. D. Wezeman, "The CO-OP method for compositional derivation of conformance testers", Protocol Specification, Testing and Verification IX, Edited by E. Brinksma, G. Scolco and C. A. Vissers, (1989), pp.145-158.

[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).