National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

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.

Canadä

Towards the Validation of Distributed Systems Based on Data Flow Analysis

(with application to LOTOS)

by

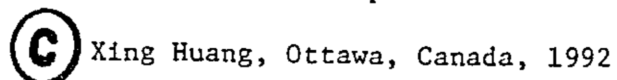
Xing Huang

A M.Sc. Thesis

submitted to the School of Graduate Studies and Research
in partial fulfillment 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


© Xing Huang, Ottawa, Canada, 1992

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

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

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

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

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

ISBN 0-315-85786-2

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGEMENT

I am very grateful to my supervisor, Dr. To-yat Cheung, for his considerate guidance, constructive advice and patience throughout my graduate studies. His instruction for revising 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 for the entire period of my study. In particular, I owe many thanks to Shenyu Ren for her kindness and help. The many discussions we have had have been of great influence and assistance. I would also like to thank Yucheng Ye and Guoqiang Wang, for their helpful comments.

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

ABSTRACT

This thesis proposes a new approach for the detection of data flow anomalies and generation of selective test sequences for distributed systems specified in LOTOS. It includes a *Data Petri-Net* (DPN) model for system specification. The model is an ordinary Petri-net extended with the capabilities of handling abstract data types, unusual actions and data synchronization. Based on the DPN, parameter (variable/constant) occurrences are classified as definition, undefinition and use. A method called DETANOM is then developed for detecting data flow anomalies, such as *undef-use*, *def-def* and *def-undef* anomalies. To facilitate the selection of test sequences, three families of data-flow-oriented coverage criteria are proposed, which include *all-defs covers*, *all-uses covers* and *all-du-paths covers*. Test sequences selected according to these criteria aim at checking whether an implementation under test possesses the desired associations among the values of the input and output parameters. A method called GENTEST is also developed for generating selective test sequences according to these criteria. Finally, these DPN-based methods are applied to the validation of LOTOS specifications. Details of an application to the Alternating Bit Protocol are included.

TABLE OF CONTENTS

ACKNOWLEDGEMENT i

ABSTRACT ii

TABLE OF CONTENTS iii

LIST OF FIGURES vi

LIST OF TABLES vii

Chapter 1 INTRODUCTION AND FUNDAMENTALS.......... 1

1.1 VALIDATION OF DISTRIBUTED SYSTEMS 1

1.2 TWO SPECIFICATION TECHNIQUES 2

1.2.1 LOTOS 2

1.2.2 Predicate/Action-Net 6

1.3 MOTIVATIONS AND CONTRIBUTIONS OF THE THESIS 7

1.4 OUTLINE OF THE THESIS 8

Chapter 2 REVIEW ON DATA-FLOW-BASED METHODS FOR ANOMALY DETECTION AND TEST SEQUENCE SELECTION 10

2.1 INTRODUCTION 10

2.2 DETECTING DATA FLOW ANOMALIES 11

2.3 COVERAGE CRITERIA FOR THE SELECTION OF TEST SEQUENCES ...12

2.4 COVERAGE CRITERIA OF RAPPS AND WEYUKER 13

2.5 A COMPARISON OF THE COVERAGE CRITERIA 15

Chapter 3 REVIEW ON PETRI-NET-BASED REPRESENTATIONS AND TEST SEQUENCE GENERATION METHODS FOR LOTOS 17

3.1 INTRODUCTION 17

3.2 PETRI-NET-BASED REPRESENTATIONS FOR LOTOS 18

3.2.1 Communicating Systems of Petri-Nets of Cheung and Zhu 18

3.2.2 Galileo Nets of Marchena and Leon 19

3.2.3 Place/Transition-Nets of Barbeau and Bochmann 19

3.2.4 Networks of Garavel and Sifakis 20

3.3 METHODS FOR GENERATING TEST SEQUENCES FOR LOTOS 21

3.3.1 The Method of Gueraichi and Logrippo 21

3.3.2 The Methods of Wezeman and Pitt 21

3.3.3 The Method of Cheung and Ye 22

3.3.4 The Method of Tripathy and Sarikaya 22

3.3.5 The Method of Cheung, Wu and Ye 22

3.3.6 The Method of Cheung and Ren 23

Chapter 4 A DATA PETRI-NET MODEL FOR THE VALIDATION OF DISTRIBUTED SYSTEMS 25

4.1 INTRODUCTION 25

4.2 THE DATA PETRI-NET MODEL 25

4.3 VALIDATION METHODS FOR DATA PETRI-NETS BASED ON DATA FLOW ANALYSIS 30

4.3.1 Preliminaries of Data Flow Analysis 30

4.3.2 A Method for Detecting Data Flow Anomalies 32
4.3.3 Criteria for Selecting Test Sequences 37
4.3.4 A Method for Generating Selective Test Sequences 40

Chapter 5 VALIDATION OF LOTOS SPECIFICATIONS VIA DATA PETRI-NETS........ 44
5.1 A VALIDATION METHOD BASED ON DATA PETRI-NETS 44
5.2 TRANSFORMATION OF LOTOS SPECIFICATIONS TO DATA PETRI-NETS 47
5.2.1 Informal Description of the Transformation Process 47
5.2.2 Formal Description of the Transformation Rules for Behaviour Expressions 54

Chapter 6 APPLICATION TO THE ALTERNATING BIT PROTOCOL........ 63
6.1 INTRODUCTION 63
6.2 REPRESENTATION OF SENDER IN DATA PETRI-NETS 63
6.3 DETECTION OF DATA FLOW ANOMALIES FOR SENDER 65
6.4 GENERATION OF TEST SEQUENCES FOR SENDER 68

Chapter 7 SUMMARY AND FUTURE RESEARCH........ 71

REFERENCES 74

APPENDIX: LOTOS SPECIFICATION OF SENDER 78

LIST OF FIGURES

Figure 1.1	LOTOS specification of the extended natural numbers	3
Figure 1.2	LOTOS specification of an external action	4
Figure 2.1	A comparison of the coverage criteria	16
Figure 4.1	(Data) synchronization in a (Data) Petri-Net	27
Figure 4.2	The Data Petri-Nets N4.1 for Example 4.1	29
Figure 4.3	Data flow anomalies in a Data Petri-Net	36
Figure 4.4	A comparison of the coverage criteria for Data Petri-Nets	39
Figure 4.5	An example on the comparison of the coverage criteria	40
Figure 4.6	The marking graph G4.4 used for test sequence generation	42
Figure 5.1	Transformation of the "atomic" expressions	49
Figure 5.2	Transformation of the "unary" operations	49
Figure 5.3	Transformation of the action-prefix and enabling operations	50
Figure 5.4	Transformation of the choice and disabling operations	51
Figure 5.5	Transformation of the parallel operations and process instantiations	52
Figure 5.6	Simplification of the transformation	54
Figure 6.1	The Data Petri-Net N.Sender for the Sender of the Alternating Bit Protocol	64
Figure 6.2	The data types, data sorts, variables, constants and functions for N.Sender	64
Figure 6.3	The marking graph G.Sender for the Data Petri-Net N.Sender	66

LIST OF TABLES

Table 1.1 The syntax of LOTOS operations 4

Table 1.2 Three types of interaction in LOTOS 5

Table 4.1 Three types of interaction in a Data Petri-Net 27

Table 4.2 The status of transitions, places, variables and constants after each iteration 36

Table 4.3 Identification of the complete paths covering all definitions of sort Bool 43

Table 4.4 Generation of the test sequences covering all definitions of sort Bool 43

Table 5.1 Transformation of LOTOS specifications to Data Petri-Nets 47

Table 5.2 Notation used in the transformation of behaviour expressions 54

Table 6.1 Labels for the transitions of the Data Petri-Net N.Sender 65

Table 6.2 The label function for the marking graph G.Sender 67

Table 6.3 The markings for the marking graph G.Sender 68

Table 6.4 The du-paths covering all definitions in G.Sender 69

Table 6.5 The complete paths and test sequences covering all definitions in G.Sender 69

Chapter 1
INTRODUCTION AND FUNDAMENTALS

1.1 VALIDATION OF DISTRIBUTED SYSTEMS

Validation of a distributed system includes two areas: verification for showing that its design is "error-free" or satisfies some predefined properties (e.g., liveness and safety), and testing for showing that its implementation conforms to the specification. Many specification models have been proposed for such purposes. Among them, FSMs (Finite State Machines), Petri-nets, LOTOS [BOL89, ISO88], ESTELLE [BUD87, ISO87] and SDL [CCI87a, 87b] have the most widespread applications.

Several well-known techniques have been developed for verification. For example, reachability analysis is used for checking the absence of state deadlocks, unspecified receptions and non-executable transitions. Invariance analysis is used for finding invariant properties. Equivalence relationships are used for proving the similarity or dissimilarity of two specifications (design). However, these approaches do not fulfill another objective of verification, namely, detecting data flow anomalies in a specification. To do this requires data flow analysis techniques.

Test sequence generation is an important part of a testing process. However, because of the complexity of distributed systems, exhaustive testing is impossible. It becomes important to be able to select the desirable test sequences. Tremendous effort has been put into the research of test sequence generation based on various specification techniques, such as FSMs [SID89], ESTELLE [URA87, 91] and LOTOS [CHE91, 92, GUE89, PIT90, TRI91, WEZ89]. However, only a few of them take both control and data flow information into consideration. This involves data flow analysis techniques.

Data flow analysis is an important approach for the validation of distributed systems. Most of the existing validation methods, especially those for LOTOS, do not consider this issue. In this thesis, we propose a Petri-net-based model called Data Petri-Net (DPN) and two data-flow-oriented methods for the validation of distributed systems.

In the following sections, we first give a brief introduction to two specification techniques, LOTOS and Predicate/Action-net. Then, we describe the motivations and outline of this thesis.

1.2 TWO SPECIFICATION TECHNIQUES

This section briefly introduces two specification techniques, namely LOTOS and Predicate/Action-net. They are closely related to our work.

1.2.1 LOTOS

LOTOS (Language fOr Temporal Ordering Specification) is one of the two formal description techniques standardized by ISO (International Organization for Standardization) [ISO88] for the formal specification of distributed systems and, in particular, for communication protocols. A LOTOS specification has two components: 1) the data part which deals with the description of data structures and value expressions; and 2) the control part which deals with the description of processes and their interactions via behaviour expressions. The data part is based on the abstract data type specification language ACT ONE [EHR85]. The control part is based on a modification of Milner's Calculus of Communicating Systems (CCS) [MIL80] and Hoare's Communicating Sequential Processes (CSP) [HOA85].

Abstract data types

An abstract data type is a collection of data domains called *sorts*, design-based abstract data items and operations on these data items in their data domains. All data items of the data sorts can be generated from the abstract data items by means of the operations. Formally, the specification of a data type includes four clauses: **type**-clause, which defines the name of the type definition; **sorts**-clause, which defines the names of the involved data sorts; **opns**-clause, which defines the syntax of all operations; and **eqns**-clause, which defines the semantics of all operations through a set of equations. An example is given below.

Example 1.1 (data types in LOTOS, Figure 1.1)

Figure 1.1 presents the data type specification of the extended natural numbers (extracted from [BOL89]), where the data type *Extended-nat-numbers* includes a sort called *nat*, a constant called *0* and two operations called *succ* and +.

```
type   Extended_nat_numbers  is
sorts  nat
opns   0       : -> nat
       succ    : nat -> nat
       _+_     : nat, nat -> nat
eqns
    forall     x, y : nat
    ofsort     nat
       x + 0          = x;
       x + succ(y)    = succ(x+y);
endtype
```

Figure 1.1 LOTOS specification of the extended natural numbers

LOTOS has the following facilities and capabilities for the manipulation of data types: 1) a library of predefined data types; 2) extensions and combinations of existing data types; 3) parameterization and actualization of parameterized data types; and 4) renaming of data types. The details are given in [BOL89, ISO88].

Behavior expressions

As a basic behavior expression, an external *action* is specified in form of "g α [Pt]", where g is a *gate* which indicates the interaction point; α is a list of zero or more *events* which can be either an output or an input event. An input event is denoted as a value declaration (! e). An output event is denoted as a variable declaration (? x : s), respectively; and Pt is an optional *predicate* which

establishes a condition on the values that can be exchanged. The internal action is denoted as a particular symbol **i**.

Example 1.2 (actions in LOTOS, Figure 1.2)

A typical specification of an external action is given in Figure 1.2, where the variable *x* gets a new value which is greater than *succ(0)* and value *succ(0)* is offered at gate *g*.

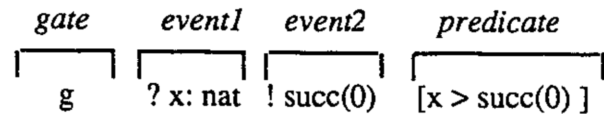


Figure 1.2 LOTOS specification of an external action

A behavior expression can be formed by applying an operator on other behavior expressions. It may also include the instantiations of processes.

For the methods we developed later, all the LOTOS operations except sum-expressions and par-expressions are included because sum-expressions and par-expressions can be represented by other expressions. The syntax of these operations is summarized in Table 1.1. The semantics of these operations is presented in Chapter 5 along with their transformation rules. A complete description of all LOTOS expressions can be found in [ISO88].

name	LOTOS operation	
inaction	**stop**	
successful-termination	**exit** $(e_1, ..., e_n)$	
guarding	$[P_t]$ -> B_b	
local-definition	**let** $x_1 = e_1, ... , x_n = e_n$ **in** B_b	
action-prefix	$g\ \alpha\ [P_t]$; B_b	($g\ \alpha\ [P_t]$ is an external action)
	i ; Bb	(**i** is an internal action)
enabling	B_a >> **accept** x_1: s_1, ..., x_n: s_n **in** B_b	

Table 1.1 The syntax of LOTOS operations (to be continued)

name	LOTOS operation	
choice	B_a [ ] B_b	
disabling	B_a [> B_b	
parallel-composition	$B_a \mid [g_1, .., g_n] \mid B_b$	(general format)
	$B_a \mid\mid\mid B_b$	(pure interleaving)
	$B_a \mid\mid B_b$	(full synchronization)
process-instantiation	$P[g1,...,gm]\ (e_1, ..., e_n)$	
hiding	**hide** $g_1, ..., g_n$ **in** B_b	

Table 1.1 The syntax of LOTOS operations (continued)

Data synchronization

Data synchronization is a function common to many distributed systems. LOTOS handles synchronization via its parallel operator ($\mid [g_1, .., g_n] \mid$). Synchronization between two processes may take place if the processes have triggered one or more events at an identical gate. By providing different events, three types of interaction can be represented in LOTOS. They are summarized in Table 1.2 (extracted from [BOL89] with modified notation), where x1 and x2 are variables; e1 and e2 are value expressions; s1 and s2 are sort names

process A	process B	syn. condition	interaction type	effect
g ! e1	g ! e2	value(e1) = value(e2)	value matching	synchronization
g ! e1	g ? x2 : s2	value(e1) $\in$ domain(s2)	value passing	after synchronization x2 = value(e2)
g ? x1 : s1	g ? x2 : s2	s1 = s2	value generation	after synchronization x1= x2 = e for e $\in$ domain(s1)

Table 1.2 Three types of interaction in LOTOS

1.2.2 Predicate/Action-Net

The Predicate/Action-net (PA-net) was first used for verifying parallel programs [KEL76] and later applied to specify and analyze communication protocols [DIA82, 87]. It has been recently adopted as part of Numerical Petri Nets [BIL88]. According to [KEL76], a PA-net is a Petri-net extended with a set of variables and labels of the form "predicate; action" attached to the transitions. Despite its various applications, a PA-net has not been formally defined in the literature. We give a formal definition below.

Definition 1.2 (*PA-net*)

A *PA-net* (Predicate/Action-net) is a 6-tuple $N = \langle P, T, F, V, L, h \rangle$, where P is a nonempty set of places; T is a set of transitions, $T \cap P = \emptyset$; F is a set of arcs, $F \subset (P \times T) \cup (T \times P)$; V is a set of variables; L is a set of labels; and h is a mapping: T -> L.

For transition t, its label h(t) includes two clauses: the **when**-clause *when(t)* specifies a predicate and consists of a Boolean expression over V; the **do**-clause *do(t)* includes a vector assignment, V := Ft(V), where Ft is a function over V.

The state of the system can be represented by means of a *marking* vector $\mu = (\mu(p_1), \mu(p_2), \ldots, \mu(p_n); \mu(x_1), \mu(x_2), \ldots, \mu(x_m))$, where $n = |P|$, $m = |V|$, $\mu(p_i)$ is the number of tokens at $p_i \in P$, $\mu(x_j)$ is the value of $x_j \in V$. The initial marking $\mu 0$ denotes the initial system state.

The execution of a system is expressed by the firing of transitions. At marking μ, a transition $t \in T$ is said to be *firable* if when(t) has a true value and $\mu(p) \geq 1$ for every $p \in$ pre(t), where pre(t) = $\{p \mid p \in P, (p, t) \in F\}$. *Firing* t makes the system evolve from the current marking μ to a new marking μ' updated by: 1) removing a token from every $p \in$ pre(t) and adding a token to every $p \in$ post(t), where post(t) = $\{p \mid p \in P, (t, p) \in F\}$; and 2) setting $\mu'(x)$:= Ft(x) for every $x \in V$.

1.3 MOTIVATIONS AND CONTRIBUTIONS OF THE THESIS

Besides control flows, a model used for the specification and validation of distributed systems is expected to be capable of handling data types, data synchronization and data flows. LOTOS provides the facilities for defining abstract data types and various kinds of interaction, including value matching, value passing and value generation [BOL89]. But, most of its existing validation methods (as reviewed in Chapter 3) do not consider data flow analysis. Data flow analysis methods have been applied to ESTELLE for selecting test sequences [URA87, 91]. But, It does not support abstract data types. By directly coupling two processes, a PA-net can handle certain kinds of data synchronization, such as value passing [DIA82]. But, PA-net does not support abstract data types either.

In this thesis, a model called Data Petri-Net (DPN) is developed for the handling of these problems. A DPN is a Petri-net extended with sets for accommodating abstract data types and labels for manipulating data synchronization. Data flow analysis is the main approach in the development of our validation methods for detecting anomalies and for generating selective test sequences.

Our methods can be applied to distributed systems specified in other specification techniques which can be transformed to DPNs. LOTOS is chosen as our application in this thesis. The application requires the transformation of LOTOS to DPNs. Although four transformation methods have been proposed in the literature, they either do not consider data or are not formally defined. (See Chapter 3 for a survey of these transformations.) So, a transformation between LOTOS and DPNs is developed in this thesis.

Following is a summary of the contributions of this thesis.

a) An extended Petri-net model called Data Petri-Net (DPN) is presented, in which a specific label is associated with every transition. The label includes a *gate-clause* for indicating the interaction point, an *event-clause* for specifying the interaction to be completed, a *do-clause* for assigning values to variables and a *when*-clause for specifying the firing condition. Such transitions can present various kinds of data synchronization such as those in LOTOS. To

accommodate data types, a DPN includes four sets called DT(Data Type), DS(Data Sort), C(Constant) and V (Variable) and a function called f (DT->DS). (Chapter 4)

b) Based on the traditional method for generating marking vectors, a method called DETANOM is developed for detecting three types of data flow anomalies: *undef-use*, *def-def* and *def-undef* anomalies, for distributed systems specified in DPNs. (Chapter 4)

c) Three families of data-flow-oriented path coverage criteria, namely *all-defs covers*, *all-uses covers* and *all-du-paths covers*, are proposed. Within each family, three criteria are suggested for covering paths related to the designated parameter, to all parameters of the designated sort and to all parameters of the designated data type. Test sequences selected according to these criteria aim at checking whether an implementation under test possesses the desired associations among the values of the input and output parameters. (Chapter 4)

d) A method called GENTEST is developed for generating test sequences according to the above criteria. (Chapter 4)

e) The above validation methods developed for DPNs are applied to distributed systems specified in a subset of LOTOS. The transformation rules from this subset of LOTOS to DPNs are also developed. (Chapters 5 and 6)

1.4 OUTLINE OF THE THESIS

The rest of this thesis is organized as follows. Chapter 2 presents a review on software validation methods based on data flow analysis. This provides the background based on which this thesis is developed. Chapter 3 includes a survey on the Petri-net-based representations of LOTOS and methods for generating test sequences from LOTOS specifications. It summarizes the relevant works about the representation and validation of LOTOS. Chapter 4 includes the main contributions of this thesis. Firstly, a model called Data Petri-Net (DPN) is proposed for the specification of distributed systems. Secondly, the concepts of the definitions, undefinitions and uses of parameter (variable/constant) occurrences are defined. They form the basis of data flow analysis. Thirdly, a method called DETANOM is developed for detecting on DPNs three kinds of

data flow anomalies called *undef-use*, *def-def* and *def-undef*. Lastly, three families of test sequence selection criteria, namely *all-defs covers*, *all-uses covers* and *all-du-paths covers*, are proposed. These criteria aim at covering paths with certain data flow properties. A method called GENTEST for generating test sequences based on these criteria is also developed. Chapter 5 shows the application of the DPN-based methods to the validation of LOTOS specifications. In this approach, a LOTOS specification is transformed to a DPN. The transformation rules are also provided. Chapter 6 contains a detailed example of the Alternating Bit Protocol. It illustrates our approach to the validation of LOTOS specifications via DPNs. Lastly, Chapter 7 summarizes our work and outlines possible future research.

Chapter 2

REVIEW ON DATA-FLOW-BASED METHODS FOR ANOMALY DETECTION AND TEST SEQUENCE SELECTION

2.1 INTRODUCTION

Part of the contributions of this thesis is the proposal of methods for validating Data Petri-Nets based on data flow analysis. Our work has been stimulated greatly by traditional software (in particular, program) validation methods. As background information, some of these methods are reviewed in this chapter.

Execution of a program normally includes the input of data, operations on them, and output of the results in an order determined by the program control and the input data. This sequence of events is viewed as *data flow*. The analysis of data flows can be applied to the validation of programs, including the detection of data flow anomalies and the selection of test sequences.

Data flow analysis is normally performed in a flow graph where a node represents a fragment of the statements of the program and an edge expresses the possible transfer of control between two nodes. For ease in discussion, we assume that every node represents a single statement. Before reviewing the methods for program validation, some basic terms are defined below.

Definition 2.1 (*terminology related to graphs*)

Consider a *directed graph* G<N, E>, where N is a set of nodes and $E \subset N \times N$ is a set of directed edges. For $n \in N$, $pre(n) = \{m \mid (m, n) \in E\}$ is called the *preset* of n, $post(n) = \{m \mid (n, m) \in E\}$ is called the *postset* of n. n is called an *entry* (resp., *exit*) node of G if its preset (resp., postset) is empty. n is called a *successor* (resp., *predecessor*) of node m if $n \in post(m)$ (resp., $n \in pre(m)$).

A *path* is a finite sequence of nodes $(n_1, n_2, ..., n_{k-1}, n_k)$, where $(n_j, n_{j+1}) \in E$, $j = 1, ..., k-1$. A path is called a *cycle* if its first node and last node are identical. A path is said to be *cycle-*

free if all its nodes are distinct. A path is said to be *complete* if its first node is an entry node of G and its last node is an exit node of G.

The *concatenation* operator, denoted as a dot (·), is defined as follows: Let p and q be two paths, p·q represents a path which begins with p and is followed by q.

Definition 2.2 (*terminology related to data flow*)

When a statement associated with the label at node n is executed, the status of a variable x may be affected in the following ways: 1) x is said to be *used* at n if the execution requires the value of x. 2) x is said to be *defined* at n if the execution results in assigning a new value to x. 3) x is said to be *undefined* at n if the execution makes it unavailable for later use.

For a node n, *def(n)*, *use(n)*, and *undef(n)* denote the sets of variables defined, used and undefined at n, respectively.

A path p is said to be *def-clear* with respect to (w.r.t.) a variable x, if $x \notin def(n)$, for every node n of p.

For two nodes m and n, def(m) is said to *reach* use(n) w.r.t. a variable x, if there exists a path (m)·p·(n) where $x \in (def(m) \cap use(n))$ and p is a def-clear path w.r.t. x.

The rest of this chapter is organized as follows. Section 2.2 defines three types of data flow anomalies. Section 2.3 reviews several classes of test-sequence-selection criteria. Section 2.4 presents formally the criteria proposed by Rapps, et al. Section 2.5 compares all of the criteria reviewed in this chapter.

2.2 DETECTING DATA FLOW ANOMALIES

The data flow of a program is expected to be consistent in two ways: 1) If a variable is used at some step, it should have been defined at an earlier step. 2) If a variable is defined at some step, it should be used at a later step. Otherwise, a data flow anomaly occurs. The following three types of anomaly are pointed out by Fosdick, et al. [FOS76].

Definition 2.3 (*data flow anomaly*)

A program is said to have an *undef-use* anomaly if there exists any variable which is used before it is defined.

A program is said to have a *def-def* anomaly if there exists any variable which is defined twice without being used between the definitions.

A program is said to have a *def-undef* anomaly if there exists any defined variable which is undefined without being used first.

Based on the above definitions, Fosdick, et al. [FOS76] design and implement a system, called DAVE, for analyzing FORTRAN programs. DAVE performs an exhaustive and depth-first search for data flow anomalies with a time complexity linearly proportional to the product of the number of edges of the flow graph and the number of program variables. It is based on a static analysis of the program, i.e., it does not have to execute the program.

2.3 COVERAGE CRITERIA FOR THE SELECTION OF TEST SEQUENCES

Data flow analysis has been widely applied in the selection of test sequences. In the literature, there exist at least four classes of selection criteria, proposed by Laski and Korel [LAS83], Ntafos [NTA84], Rapps and Weyuker [RAP85], and Ural and Yang [URA88, 91], separately. These criteria aim at covering paths capturing certain data flow associations.

Laski and Korel [LAS83] emphasize the fact that a given node may contain the uses of several variables, where each use may be reached by several definitions occurring at different nodes. The set of these definitions is called a *definition context.* They propose three criteria, namely, Reach Coverage, Context Coverage and Ordered Context Coverage, which aim at selecting paths along which the various combinations of definitions may reach the node.

Ntafos [NAT84] realizes the fact that a variable x may be defined in terms of another variable, i.e., each definition reaches its use at a node where another definition occurs. Hence, a path contains a chain of alternating definitions and uses, called *du-interactions*. A class of path

selection criteria, called Required k-Tuples, is proposed to cover every path containing i ($2 \leq i \leq k$) du-interactions. Also, these criteria require every loop to be traversed in a particular way.

The criteria proposed by Rapps and Weyuker [RAP85] account for how the different kinds of use of variables affect the control flow in a program. In these criteria, every use of a variable is further classified as either a predicate use or a computation use. Then, the criteria attempt to cover the various combinations of definitions and uses. The criteria are applied later for selecting test sequences for protocols specified in ESTELLE [URA87]. Since our works are influenced mostly by Rapps and Weyuker's criteria, these criteria are presented in more details in Section 2.4.

Ural and Yang single out one kind of definition, called *input*, and one kind of c-use, called *output* for investigation. An *input* is the definition of a variable that occurs in: 1) an input statement, or 2) the left-hand side of an assignment statement whose right-hand side contains constants only, or 3) in the output parameter of a called subprogram such that this variable is undefined before the call, but is defined by the called subprogram. An *output* is the use of a variable/constant occurring in an output statement. An input is said to *influence* an output if it can reach this output either directly or through other variables. They propose a class of criteria, called All-simple-OI-paths and All-k-iteration-OI-paths, for covering paths whose last node contains an output which is influenced by the input in its first node. Also, these criteria require every loop to be traversed in a particular way. The criteria are modified for selecting test sequences for protocols specified in ESTELLE [URA91].

A comparison of the above criteria is presented in Section 2.5.

2.4 COVERAGE CRITERIA OF RAPPS AND WEYUKER [RAP85, CLA89]

In this section, we present formally Rapps and Weyuker's path selection criteria which are based on data flow and control flow. These criteria are redefined and compared with other criteria by Clarke, et al.[CLA89]. A method for generating complete paths which satisfy the strongest criterion, namely, all-du-paths, is implemented by Bieman, et al. [BIE89].

The following description is based on [CLA89].To prevent a flow graph G<N, E> from having any defined but not used variable or having any used but undefined variable, it is assumed that G satisfies the following conditions: 1) Every definition of any variable x reaches at least one use of x. 2) Every use of any variable x is reached by at least one definition of x.

Definition 2.4 (*p-use, c-use*)

The use of a variable at a node n is called a *predicate use* (*p-use*), if n represents the predicate of a conditional branch statement; otherwise, the use is called a *computation use* (*c-use*).

Definition 2.5 (*Rapps and Weyuker's criteria*)

Let P be a set of complete paths of the flow graph G<N, E>.

* P is an *all-defs* cover of G, if, $\forall$ n $\in$ N and $\forall$ x $\in$ def(n), P includes at least one path (n)·p·(m), where p is def-clear w.r.t. x and x $\in$ use(m).
* P is an *all-p-uses* cover of G, if, $\forall$ n $\in$ N and $\forall$ x $\in$ def(n), the following condition is satisfied: For $\forall$ m $\in$ N which has a p-use of x reached by def(n), and $\forall$ successor m' of m, P includes at least one path (n)·p·(m, m'), where p is def-clear w.r.t. x.
* P is an *all-p-uses/some-c-uses* cover of G, if, $\forall$ n $\in$ N and $\forall$ x $\in$ def(n), one of the following conditions is satisfied:
  1) $\forall$ m $\in$ N which has a p-use of x reached by def(n), and $\forall$ successor m' of m, P includes at least one path (n)·p·(m, m'), where p is def-clear w.r.t. x.
  2) There is no p-use of x reached by def(n). Also, P includes at least one path (n)·p·(m), where p is def-clear w.r.t. x and m $\in$ N which has a c-use of x.
* P is an *all-c-uses/some-p-uses* cover of G, if, $\forall$ n $\in$ N and $\forall$ x $\in$ def(n), one of the following conditions is satisfied:
  1) $\forall$ m $\in$ N which has a c-use of x reached by def(n), P includes at least one path (n)·p·(m), where p is def-clear w.r.t. x.
  2) There is no c-use of x reached by def(n). Also, P includes at least one path (n)·p·(m), where p is def-clear w.r.t. x and m $\in$ N which has a p-use of x.

* P is an *all-uses* cover of G, if, $\forall$ n $\in$ N and $\forall$ x $\in$ def(n), the following conditions are satisfied:
  1) $\forall$ m $\in$ N which has a c-use of x reached by def(n), P includes at least one path (n)·p·(m), where p is def-clear w.r.t. x.
  2) $\forall$ m $\in$ N which has a p-use of x reached by def(n), and $\forall$ successor m' of m, P includes at least one path (n)·p·(m, m'), where p is def-clear w.r.t. x.
* P is an *all-du-paths* cover of G, if, $\forall$ n $\in$ N and $\forall$ x $\in$ def(n), the following conditions are satisfied: $\forall$ m $\in$ N which has a use of x reached by def(n), and $\forall$ successor m' of m, P includes every path (n)·p·(m, m'), where p is cycle-free and def-clear w.r.t. x.

Based on control flow only, Rapps and Weyuker also propose the following three criteria:

* P is an *all-nodes* cover of G, if, $\forall$ n $\in$ N, P includes at least one path where n occurs.
* P is an *all-edges* cover of G, if, $\forall$ (m, n) $\in$ E, P includes at least one path containing (m, n).
* P is an *all-paths* cover of G if P includes every path of G.

2.5 A COMPARISON OF THE COVERAGE CRITERIA [CLA89, YAN88]

By analyzing the strength of the criteria proposed separately by Rapps, et al., Ntafos, and Laski, et al., Clarke, et al. [CLA89] find that some of them fail to attain certain minimum program coverage requirements. Specifically, the Required k-tuples do not ensure that each definition in a program is used at least once, and the Reach Coverage, Context Coverage and Ordered Context Coverage all fail to ensure that all statements are covered. Then, they introduce minor changes to the original criteria and show the relationship among the modified criteria. Also, Yang [YAN88] compares their criteria with the others. Their results are summarized in Figure 2.1, where each criterion modified by Clarke, et al. is marked with a plus (+). The conditions for the comparison are defined below.

Definition 2.6 (*conditions for comparing coverage criteria*)

Let C1 and C2 be two coverage criteria.

C1 is said to be *stronger* than C2, denoted as C1 -> C2, iff, for any given flow graph, any set of complete paths that satisfies C1 also satisfies C2, but not vice versa.

C1 is said to be *equivalent* to C2, denoted as C1 <-> C2, iff, for any given flow graph, any set of complete paths that satisfies C1 also satisfies C2, and vice versa.

C1 is said to be *incomparable* with C2, iff, none of the following holds: C1 -> C2, C2 -> C1 and C1 <-> C2.

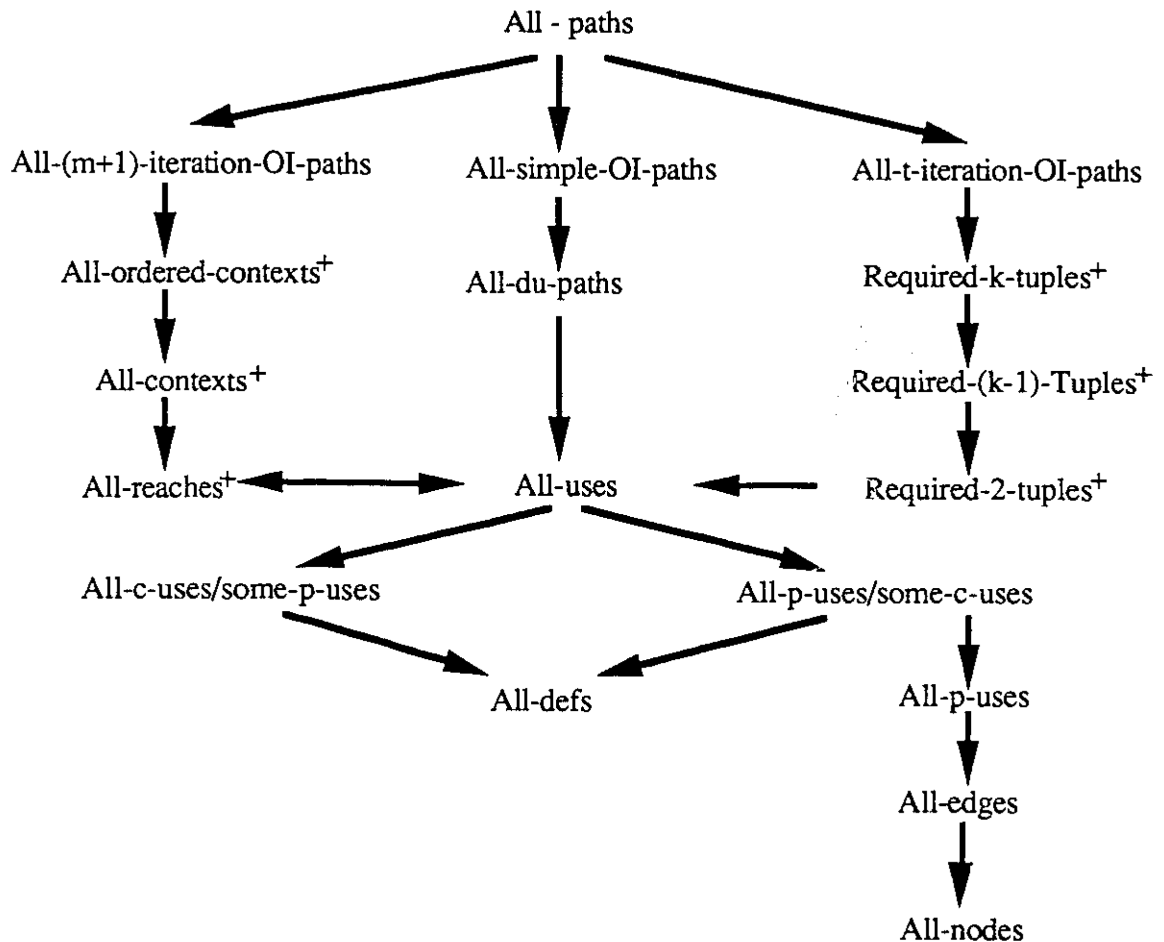


Figure 2.1 A comparison of the coverage criteria

Chapter 3

REVIEW ON PETRI-NET-BASED REPRESENTATIONS AND TEST SEQUENCE GENERATION METHODS FOR LOTOS

3.1 INTRODUCTION

In this thesis, a Petri-net-based model called Data Petri-Net (DPN) is proposed for representing and validating distributed systems based on data flow analysis. It will be applied to LOTOS. As background information, some similar models and test sequence generation methods for LOTOS are reviewed in this chapter. Most of these methods do not take data flow analysis into consideration.

A Petri-net is a well-developed model for the specification and validation of concurrent systems [DIA87]. At least four Petri-net-based models have been proposed for the representation and validation of LOTOS specifications. They are the Communicating System of Petri-Nets (CSPN) proposed by Cheung, et al. [CHE88, 92], the Galileo net proposed by Marchena, et al. [MAR89], the Place/ Transition-net proposed by Barbeau, et al. [BAR90], and the Network proposed by Garavel, et al. [GAR90]. Only CSPN and Network can represent full LOTOS. They are intended for the execution and verification of LOTOS specifications. Recently, test sequence generation methods based on the variations of these models have been reported [CHE91, 92].

A labelled transition system is another means for expressing LOTOS. In this kind of representation, the parallel operators of LOTOS, such as pure interleaving, general synchronization, etc., are replaced by choices through some expansion theorems. Hence, no real concurrence remains in the model. Such representations are applied in the study of behavioural equivalence [BOL89] and test sequence generation. The canonical tester of Brinksma [BRI89], the state diagram of Gueraichi, et al.[GUE89], and the CHART of Tripathy, et al. [TRI91] belong to this class.

Graphical LOTOS is the third kind of representation models for LOTOS. There exist at least two proposals, UO-GLOTOS proposed by Cheung, et al. [CHE89] and ISO-GLOTOS proposed

by ISO [ISO89]. In UO-GLOTOS, a LOTOS expression is represented as a hierarchical structure whose root represents its 'outermost' operation and subhierarchies represent its operands. Such a hierarchy reflects the syntactic characteristics and enhances the clarity and readability of textual LOTOS. Based on this model, a graphical-LOTOS-based software environment has been developed at the University of Ottawa for research on LOTOS. This environment has the following features: 1) representation of a specification in UO-GLOTOS; 2) transformation between textual LOTOS and UO-GLOTOS; 3) execution of a UO-GLOTOS specification; and 4) test sequence generation. Based on an executor [CHE90], a method has been proposed for generating test sequences. Recently, selective test sequences with data can also be generated in a TTCN form. ISO-GLOTOS provides similar graphical features as UO-GLOTOS. But, no significant ISO-GLOTOS-based application has been developed.

The rest of this chapter is organized as follows. Section 3.2 reviews the Petri-net-based models. Section 3.3 describes the methods for generating test sequences based on various representations.

3.2 PETRI-NET-BASED REPRESENTATIONS FOR LOTOS

In this section, we review the four Petri-net-based representations for LOTOS, i.e., the Communicating Systems of Petri-Nets (CSPN) proposed by Cheung, et al. [CHE88, 92], Galileo net proposed by Marchena, et al. [MAR89], Place/Transition-net proposed by Barbeau, et al. [BAR90], and Network proposed by Garavel, et al. [GAR90].

3.2.1 Communicating Systems of Petri-Nets of Cheung and Zhu [CHE88, 92]

A Communicating System of Petri-nets (*CSPN*) is proposed by Cheung and Zhu for the representation of the processes of a distributed system specified in full LOTOS [CHE88, 92]. Briefly, each LOTOS process is represented by a CSPN and LOTOS operations on the processes are represented by adding transitions to connect the CSPNs. As a result, the transformed CSPN preserves the modular design of the LOTOS specification.

In CSPN, there are two types of tokens: control-tokens (c-tokens) and data-tokens (d-tokens). c-tokens are used to regulate the control flow of the system. d-tokens are vehicles for transferring data values. Each d-token is associated with a data list carrying three kinds of information: value-expressions, sorts and usage status (i.e., "has been defined" or "to be defined"). In order to describe special actions, CSPN introduces four types of transitions: assignment-transitions (a-transition), guarded-choice-transitions (g-transition), interactive-transitions (i-transitions) and matched-transitions (m-transitions). a-transitions represent groups of value-assignment statements. g-transitions represent guard-choice. i-transitions are primitives for interprocess communication. m-transitions represents matched synchronizations.

3.2.2 Galileo Nets of Marchena and Leon [MAR89]

Galileo is an environment that helps software and hardware designers to model, analyze and simulate concurrent systems by means of a *Galileo net*, a Petri-net with the extension of data operations based on Pascal data types [SAN86]. Data places associated with transitions are introduced to Galileo nets for the manipulation of data operations. The firing of a transition may depend on the condition of the data contained in the associated data places and may modify these data.

For the purpose of analysis, a LOTOS specification is transformed to a Galileo net [MAR89]. In this approach, only well-formed recursions are considered, i.e., all recursions must be prefixed by the action prefix (;). Although Galileo supports data operations, it cannot handle abstract data types in a formal way. Furthermore, the transformed net absorbs all the LOTOS processes and operations, i.e., the structures and relationships among the LOTOS processes do not remain as part of the characteristics of their Galileo net representation.

3.2.3 Place/Transition-Nets of Barbeau and Bochmann [BAR90]

Barbeau and Bochmann introduce a Place/Transition-net (*P/T-net*) semantics for LOTOS [BAR90]. Basically, it is a Petri-net whose transitions may be labelled with a user-defined action, a successful termination action (∂) or an internal action (i). It can represent a subset of basic LOTOS

with the following constraints on recursions: 1) A process instantiation must be well-defined, i.e., it must be prefixed by an action prefix, or must be in the right sub-expression of an enabling operator (>>) or a disabling operator ([>). 2) The general parallel operator (| [ ] |) may not be allowed within a recursive process definition.

Barbeau and Bochmann prove that every specification based on this subset of LOTOS can be transformed to an equivalent P/T-net, and vice versa. However, P/T-net is insufficient for practical usage because it lacks of data handling facilities and no validation method is reported.

3.2.4 Networks of Garavel and Sifakis [GAR90]

The *network* proposed by Garavel and Sifakis is a basic Petri-net with the extension of data operations at the transitions and a partition of the places [GAR90]. Each transition is attached with three components: 1) a gate which can be a visible LOTOS gate, "τ" or "ε"; 2) an offer which is a list of value/variable declarations; and 3) an action which can be a null action or a composition of an assignment, a condition and/or an iteration. The partition of places is determined by a set of special components called *units*. Intuitively, each unit is a subset of places which represents a sequential behaviour. A unit can also be hierarchically refined into subunits to express that the corresponding behaviour is composed of several concurrent sub-behaviours. However, no details of units are reported.

In Garavel's network, an inaction "stop" is represented as one place. An action is represented as two places connected by a transition. A process is represented as a set of places and transitions connected according to the operators involved. Similar to other Petri-net-based models, some constraints are imposed on recursive processes. For instance, no recursion is allowed in the following components: 1) the operand of a general parallel operator, 2) the operand of operator "par", and 3) the left sub-expressions of the enable or disable operators. The model has been implemented for verifying LOTOS specifications. Their verification method is model checking based on a specific version of temporal logic.

3.3 METHODS FOR GENERATING TEST SEQUENCES FOR LOTOS

In this section, we review some methods for generating test sequences from various LOTOS-representation models, such as labelled transition systems [GUE89, WEZ89, PIT90, TRI91], UO-GLOTOS [CHE90] and Petri-nets [CHE91, 92].

3.3.1 The Method of Gueraichi and Logrippo [GUE89]

Gueraichi and Logrippo propose a method for generating test sequences from a specification based on full LOTOS [GUE89]. In this method, a LOTOS specification is first transformed by a LOTOS interpreter to a symbolic tree. Then, the tree is represented as a state diagram. Finally, unique input/output (UIO) sequences for the state diagram are derived as test sequences by applying methods similar as the ones described in [SID89]. As an example, the method is applied to the protocol LAP-B. Although data are involved in the specification, data flow analysis is not considered. Besides, the test sequences are selected manually.

3.3.2 The Methods of Wezeman and Pitt [WEZ89, PIT91]

The methods proposed independently by Wezeman [WEZ89], and Pitt and Freestone [PIT91] are based on the canonical tester theory developed by Brinksma [BRI89]. According to Brinksma's theory, a process B1 *conforms* to another process B2 if B1 does not have unexpected deadlocks w.r.t. the external paths (i.e., paths consisting of only observable actions) of B2. Thus, an implementation under test (IUT) is said to conform to a specification if no unexpected deadlock occurs when this IUT runs concurrently with the canonical tester consisting of external paths correctly derived from the specification. Based on this theory, Wezeman proposes the CO-OP method to produce the canonical tester [WEZ89], where only finite processes are considered. Pitt and Freestone also work out a method for generating test sequences for conformance testing [PIT91]. Unlike Wezeman's approach, Pitt and Freestone consider specifications including only action prefix (;) and choice ([]). But the infinite processes resulting from recursions are allowed in Pitt's approach. Generally, the canonical tester methods are considered to be impractical because they consider no data operation and are mainly for exhaustive testing.

3.3.3 The Method of Cheung and Ye [CHE90]

The method proposed by Cheung and Ye makes use of the UO-GLOTOS executer developed at the University of Ottawa [CHE90]. The method includes three phases: Firstly, all executable paths of a LOTOS specification are generated by the executor; Then, some of these paths are selected automatically for including special actions, such as "stop", "exit", parallel actions, etc. Finally, test sequences are generated for covering the selected paths. Execution and selection are facilitated by the graphical representation. The generation of test sequences in TTCN or LOTOS is under development.

3.3.4 The Method of Tripathy and Sarikaya [TRI91]

Tripathy and Sarikaya realize the deficiency in methods without data handling and present a method for test sequence generation for full LOTOS [TRI91]. In their method, the specification is first transformed to a simpler form in which every full parallel composition (full synchronization) or sequential composition (enabling) is replaced by general parallel compositions, every variable is renamed to avoid global conflicts. Secondly, the simpler specification is mapped to a CHART which is a particular kind of transition system defined by Milner [MIL84]. Thirdly, control flow and data flow graphs are generated from CHART. Finally, test sequences are generated from the control flow graphs and selected according to the protocol functions captured by the data flow graphs. The transformation of LOTOS specifications to CHARTs and the generation of control flow and data flow graphs are given in [TRI91]. However, no selection criterion is proposed.

3.3.5 The Method of Cheung, Wu and Ye [CHE91]

In conformance testing, an IUT is said to conform to a specification if no unexpected deadlock occurs when this IUT runs concurrently with a correctly generated set of test sequences. Because of the unpredictable behaviour of an indeterministic distributed system, this raises a very important question: how many times should a test sequence be run with the IUT in order to make a fair verdict ? Cheung, Wu amd Ye propose a method for attacking this problem. In the method, a

labelled Petri-net model including both external and internal actions is developed to represent an indeterministic system. A metric for measuring the degrees of indeterminism of the test sequences is provided. The metric is used to estimate the number of times that a test sequence should be tried in order to make a fair verdict. Then, a fairness model for conformance testing and an iterative algorithm called IPNTEST for generating test sequences and their degrees of indeterminism are presented. The kth iteration of IPNTEST generates a group of test sequences of length k, all having the same preamble. Each group of test sequences provides a fairly uniform coverage for the system states reachable by that preamble. The test groups, as a whole, cover all bounded firable paths of the system. The number of iterations and hence lengths of the test sequences can be controlled to prevent infinite action sequences. IPNTEST has been implemented as an integrated part of UO-GLOTOS environment. As an application, the method is applied to a LOTOS specification represented in Galileo nets. The method is simple and efficient, especially for indeterministic systems. However, it lacks data involvement and is essentially for exhaustive testing.

3.3.6 The Method of Cheung and Ren [CHE92]

Cheung and Ren propose a new approach for selectively generating executable test sequences from a LOTOS specification. First, based on the functional characteristics of the LOTOS operations, a criterion called *operational coverage* is developed, whereby, for testing purposes, each of the LOTOS operations is assigned a mandatory set called *cover* which consists of sequences of the actions involving in its operands. A method called SELECTEST is then presented for generating executable test sequences which fulfill the coverage criterion. SELECTEST first determines a set of guides, each denoted as a possibly nonexecutable rooted path containing all actions of the cover over the hierarchy of the LOTOS specification and passing through the operator (or construct in LOTOS terms). It then transforms the specification into a modified Communicating System of Petri-Nets (CSPN) [CHE88] and applies the guides to simplify the search for executable test sequences over the CSPN. The coverage and method have also been

generalized to the case of covering a set of LOTOS operations. An example on the Alternating Bit Protocol is presented. The method can generate test sequences without generating the entire marking graph. It may be quite complicated for covering multi-operators. Also, the method does not take data into consideration.

Chapter 4
A DATA PETRI-NET MODEL FOR THE VALIDATION OF DISTRIBUTED SYSTEMS

4.1 INTRODUCTION

This chapter contains the main contributions of this thesis. A model called Data Petri-Net (DPN) and a few methods are proposed for the validation of distributed systems. The model basically is a Petri-net extended with the capabilities of handling three special problems relevant to data: abstract data types, data synchronization and data flow analysis. As discussed in the previous chapters, several other models can handle these problems individually. But, DPN combines them and handles some of them in a different way.

The rest of this chapter is organized as follows. Section 4.2 introduces the Data Petri-Net model and its capabilities in handling abstract data types and data synchronization. Section 4.3 develops two data-flow-related methods for detecting data anomalies and for generating selective test sequences for Data Petri-Nets.

4.2 THE DATA PETRI-NET MODEL

A Data Petri-Net is a basic Petri-net extended with the following features for handling abstract data types and data synchronization.

Extension for handling abstract data types

The specification of a distributed system usually includes the description of various data structures and data operations. Abstract data types are developed for managing this description in a formal way. The definition of an abstract data type normally includes four clauses (as those ones in LOTOS): a **type**-clause for defining the name of the data type, a **sorts**-clause for defining the names of the involved data sorts, an **opns**-clause for defining the syntax of all operations, and an **eqns**-clause for defining the semantics of all operations. Since the syntax and semantics given in the opns-clause and eqns-clause do not affect the description of our validation methods and are

required only for implementation, they are not explicitly described in our model. To accommodate the remaining information, the DPN includes four sets DT (Data Type), DS(Data Sort), C(Constant) and V(Variable) and a function f (DT -> DS).

Extension for handling data synchronization

Data synchronization is a feature common to many distributed systems. In basic Petri-nets, as shown in Figure 4.1, synchronization between processes A and B at places p1 and p3 is represented by directly connecting these places by a transition t1. To represent various kinds of actions (e.g., input, output, exit, etc.) and process interactions, every transition in the DPN is associated with a label containing four clauses as defined below.

Definition 4.1 (*the label of a transition*)

The *label* of a transition t is a 4-tuple *label(t)* = <gate(t), event(t), do(t), when(t)>, where

gate(t) is one of the following: an external gate-name, τ (an internal gate), ∂ (an exit gate) or Ø (a null gate); [for indicating an interaction point (if any)]

event(t) is one of the following: ? x : s (an input event, same as a LOTOS variable declaration), ! e (an output event, same as a LOTOS value declaration) or Ø (a null event); [for specifying the interaction (if any) to be completed when firing t]

do(t) is either $(x_i: s_i):= e_i$ (a set of assignment statements) or Ø (a null statement); [for assigning values to variables] and

when(t) is either a Boolean expression or Ø (a null expression). [for specifying the firing condition]

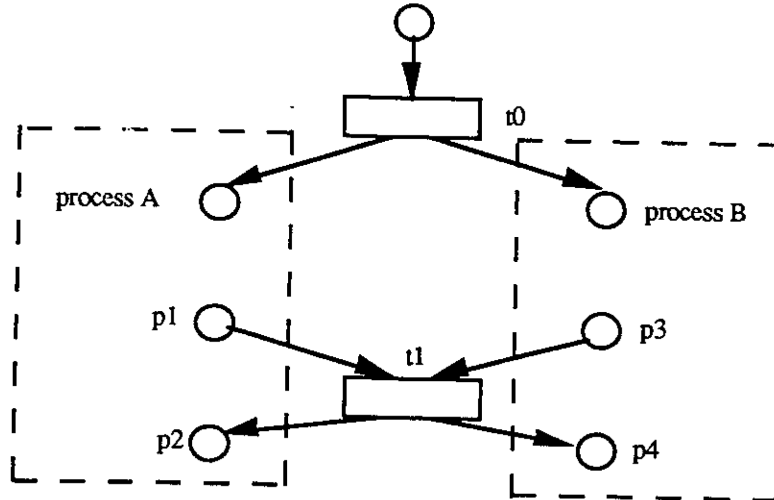


Figure 4.1 (Data) synchronization in a (Data) Petri-Net

By providing different labels, several types of interaction can be represented in a DPN. The basic Petri-net in Figure 4.1 is modified to a DPN by associating its transitions (e.g., t1) with labels. The three types of synchronization (as summarized in Table 1.2) can be represented as shown in Table 4.1, where expression e1 and variable x1 of sort s1 (resp., expression e2 and variable x2 of sort s2) are provided and used by process A (resp., B).

interaction type	label(t1)	effect
value matching	g, ! e1, Ø, value(e1) = value(e2)	synchronization
value passing	g, ! e1, (x2: s2) := e1, sort(e1) = s2	after synchronization x2 = value(e1)
value generation	g, ? x1:s1, (x2: s2) := x1, s1 = s2	after synchronization x1= x2 = e for e ∈ domain(s1)

Table 4.1 Three types of interaction in a Data Petri-Net

According to the contents of gate(t), a transition t can represent four types of actions: an *external action* if gate(t) is an external gate-name, an *internal action* (i) if gate(t) is τ, a *successful termination* (*exit*) if gate(t) is ∂ and a *null action* if gate(t) is Ø. The first three types of actions are similar those used in LOTOS. The last one is introduced for carrying relevant information (such as

predicates) and for accommodating two processes. For example, in Figure 4.1, transition t0 is used to synchronize processes A and B, and it does not involve any other action.

Formal definition of a Data Petri-Net:

Based on the above description, a Data Petri-Net can be defined as follows.

Definition 4.2 (*Data Petri-net*)

A *Data Petri-Net* (DPN) is an 11-tuple N = <P, T, F, DT, DS, V, C, L, f, label, $\mu 0$>, where

P is a set of places, $P \neq \emptyset$;

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

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

DT is a set of data types;

DS is a set of data sorts;

V is a set of pairs $\{(x : s) \mid s \in DS$ and x is a variable identifier of sort s$\}$;

C is a set of pairs $\{(c : s) \mid s \in DS$ and c is a constant identifier of sort s$\}$;

L is a set of labels;

f is a mapping: DT -> DS;

label is a mapping: T -> L; and

$\mu 0$ is the initial marking (to be defined later, possibly omitted if it is not considered.).

The behaviour part of a DPN can be graphically presented, in which an oval (resp., rectangle) is used to represent a place (resp., transition) with its identifier written inside (resp., outside). An example is given below.

Example 4.1 (Data Petri-Net, Figure 4.2)

Consider the DPN N4.1 = <P, T, F, DT, DS, V, C, L, f, label>, where P = {p1, p2}, T = {t1}, F = {(p1, t1), (t1, p2)}, DT = {Boolean, NaturalNumber}, DS = {Bool, Nat}, V = {(x : Nat)}, C = {(0 : Nat), (true : Bool), (false : Bool)}, L = {(g, ? x : Nat, $\emptyset$, $x \leq 0$)}, f(Boolean)

={Bool}, f(NaturalNumber) = {Bool, Nat}, label(t1) = <gate(t1), even(t1), do(t1), when(t1)>, where gate(t1) = "g", event(t1) = "? x : Nat", do(t1) = Ø and when(t1) ="x ≤ 0".

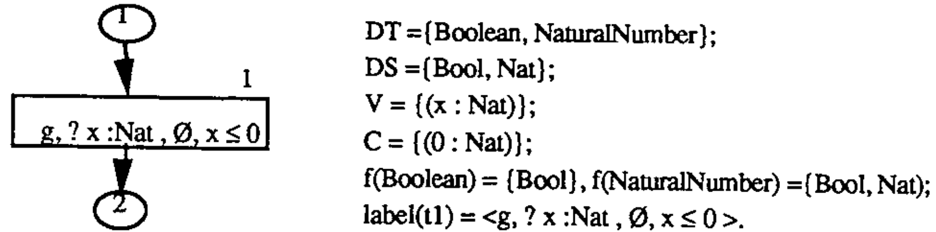


Figure 4.2 The Data Petri-Net N4.1 for Example 4.1

Execution of a Data Petri-Net

The state of the system is represented in terms of a *marking* vector, $\mu = (\mu(p_1), \mu(p_2), \ldots, \mu(p_n); \mu(x_1), \mu(x_2), \ldots, \mu(x_m))$, where $n = |P|$, $m = |V|$, $\mu(p_i)$ is the number of tokens at $p_i \in P$, $\mu(x_j)$ is the value of $x_j \in V$. The initial marking $\mu 0$ denotes the initial system state.

Firing a transition t involves the interaction specified in gate(t) and event(t), and the data operations specified in do(t). The firing rule is defined as follows:

Definition 4.3 (*firing rule*)

For a transition $t \in T$, let pre(t) = $\{p \mid p \in P, (p, t) \in F\}$and post(t) = $\{p \mid p \in P, (t, p) \in F\}$. t is said to be *firable* at marking μ if when(t) has a true value and $\mu(p) \geq 1$ for every $p \in$ pre(t). *Firing* t makes the DPN evolve from μ to a new marking μ' as defined below:

a) For every $p \in P$,

$$\mu'(p) = \begin{cases} \mu(p) - 1 & \text{if } p \in \text{pre}(t) - \text{post}(t) \\ \mu(p) + 1 & \text{if } p \in \text{post}(t) - \text{pre}(t) \\ \mu(p) & \text{otherwise.} \end{cases}$$

b) For every (x:s) $\in$ V,

$$\mu'(x) = \begin{cases} e_i & \text{if event(t) contains "? x:s" and } e_i \text{ is the received value} \\ e_j & \text{if do(t) contains "x := } e_j \text{ "} \\ \mu(x) & \text{otherwise.} \end{cases}$$

In the execution of a DPN, a variable may be undefined to prevent its further usage. A symbolic constant und is introduced for this purpose.

Definition 4.4 (und)

The symbolic constant und (*undefined*) is defined for every sort as follows: When a variable is associated with und, it looses its value.

4.3 VALIDATION METHODS FOR DATA PETRI-NETS BASED ON DATA FLOW ANALYSIS

This section includes a few methods for the validation of distributed systems via data flow analysis. It is organized as follows. Subsection 4.3.1 gives the preliminaries of data flow analysis. Subsection 4.3.2 develops the method for detecting anomalies in DPNs. Subsection 4.3.3 proposes three families of coverage criteria. Finally, Subsection 4.3.4 develops a method for generating selective test sequences based on these criteria.

4.3.1 Preliminaries of Data Flow Analysis

The marking graph of a given DPN can be used as the basis for data flow analysis. Its generation process is the same as for an ordinary Petri-net except that every edge is associated with a node called *transition node*. Hence, the marking graph of a DPN has two types of nodes: marking nodes and transition nodes, attached with reachable marking vectors and the labels of fired transitions, respectively. The marking graph inherits the data type definitions from the DPN. It also satisfies the basic properties described in Definitions 2.1-2.2 (Chapter 2).

Definition 4.5 (*marking graph*)

A *marking graph* of a DPN N<P, T, F, DT, DS, V, C, L, f, label, μ0> is a 11-tuple G = <M, T^{μ}, F^{μ}, DT, DS, V, C, L^{μ}, f, $label^{\mu}$, μ0>, where

M is the set of all reachable markings of the DPN;

T^{μ} is the set of transitions;

F^{μ} is the set of edges, i.e. $F^{\mu} \subset (M \times T^{\mu}) \cup (T^{\mu} \times M)$;

L^{μ} is the set of labels; and

$label^{\mu}$ is a mapping: T^{μ} -> L^{μ}.

Data flow analysis starts with identifying the definitions, undefinitions or uses for variables and constants. Based on a marking graph, every variable/constant can be identified in the following way.

Definition 4.6 (*define, undefine, use*)

A constant *c* with sort s is said to be *defined*, if (c : s) ∈ C. Otherwise, c is said to be *undefined*. It is assumed that every constant is defined at the initial marking.

A variable *x* with sort s is said to be *defined* at a transition *t*, if one of the following conditions occurs in the label of t: 1) event(t) contains "? x : s". 2) do(t) contains "(x : s) := e", where e ≠ und.

A variable *x* with sort s is said to be *undefined* at a transition *t*, if one of the following conditions occurs in the label of t: 1) do(t) contains "(x : s) := und". 2) gate(t) =∂. It is assumed that every variable is undefined at the initial marking.

A constant/variable x is said to be *used* at a transition *t*, if one of the following conditions occurs in the label of t: 1) event(t) contains "! e" and x appears in e. 2) do(t) contains "(y : s) := e" and x appears in e. 3) when(t) contains x.

In the following description, a *parameter* is either a constant or a variable.

4.3.2 A Method for Detecting Data Flow Anomalies

Normally, a parameter cannot be used unless it has been defined earlier. Also, a parameter should not be defined twice unless it is used in between. Otherwise, the DPN is said to have data flow anomalies. The three types of data flow anomalies, namely undef-use, def-def and def-undef presented in Definition 2.3 are adopted for describing anomalies in DPNs.

In our method, the anomaly-detection process is integrated into the process of generating a marking graph. Starting from the initial marking $\mu0$, the marking graph is expanded by firing a transition t of a given DPN. Anomalies of a parameter x are checked when firing t, provided that the status of every parameter is known.

Anomalies detected from the label of a transition

Consider a transition t with label(t) = <gate(t), event(t), do(t), when(t)>. In general, firing t includes three stages:

a) Evaluate the condition specified in when(t). Since x may be used in when(t), it may cause an undef-use anomaly.

b) Perform the interaction specified in event(t). Since x may be either used or defined in event(t), it may cause an undef-use or def-def anomaly.

c) Manipulate the data operations specified in do(t). Since x may be used, defined or undefined in do(t), it may cause an undef-use, def-def or def-undef anomaly.

Information required for anomaly detection

To detect any of these anomalies for a parameter x associated with a transition firable at marking μ, the *usage status* of x is required, i.e., whether x has been used since its last definition. This is provided by a vector *status* which is associated with μ and defined as follows.

$$\text{status}(\mu, x) = \begin{cases} \text{und} & \text{if x is undefined} \\ \text{true} & \text{if x is defined and has been used since the last definition} \\ \text{false} & \text{if x is defined but has not been used since the last definition} \end{cases}$$

In our method, a marking graph is generated by using the traditional breadth-first exploration technique. A set, denoted as *reach(k)* (for $k \geq 1$), is used to contain all markings reached from $\mu 0$ through a sequence of (k-1) transitions. This marking graph will be used for generating test sequences later.

METHOD DETANOM (generation of marking graph and detection of data flow anomaly)

INPUT: The DPN N<P, T, F, DT, DS, V, C, L, f, label, $\mu 0$>.

OUTPUT: 1) The marking graph G<M, T^{μ}, F^{μ}, DT, DS, V, C, L^{μ}, f, $label^{\mu}$, $\mu 0$>.

2) The list of information on data flow anomalies.

PROCEDURE:

Starting from the initial marking $\mu 0$, in the process for generating the marking graph, data flow anomalies are detected iteratively as follows: In the k-th iteration, if there exists a transition t, whose label(t) = <gate(t), event(t), do(t), when(t)> and each of whose preplaces has at least one token at marking $\mu \in$ reach(k), check any anomaly in t and fire t. Details are given below.

Initialization Set reach(1) := $\{\mu 0\}$.

For every $(x : s) \in V$, set status($\mu 0$, x) := und.

For every $(x : s) \in C$, set status($\mu 0$, x) := false.

Start the following iterative process with k = 1.

k-th iteration Suppose reach(k) has been obtained during the (k-1)-th iteration.

Obtain reach(k+1), expand G and detect anomalies as follows:

Step 1. Set reach(k+1) := $\emptyset$.

For each marking $\mu \in$ reach(k) and each transition $t \in \{t1 \mid \mu(p) \geq 1$ for every $p \in pre(t1)\}$, generate the next possible marking μ' as follows:

/*Whenever an undef-use anomaly is detected, the process for generating μ' is terminated*/

1.0) Initialize $\mu'(x) := \mu(x)$ for every $x \in V$ and

status(μ', y) := status(μ,y) for every $y \in (V \cup C)$.

1.1) Determine the firability of t

1.1.1) For every x *used* in when(t), detect any undef-use anomaly as follows: If status(μ', x) = und, report x as an undef-use anomaly and stop the process for generating μ'; otherwise, set status(μ', x) :=true.

1.1.2) If when(t) is evaluated to true, go to Step 1.2; otherwise, stop the process for generating μ'.

1.2) <u>Expand the marking graph G</u> /* t is firable */

1.2.1) For every x *used* in event(t), detect any undef-use anomaly as follows: If status(μ', x) = und, report x as an undef-use anomaly and stop the process for generating μ'; otherwise, set status(μ' x) :=true.

For every x *defined* in event(t), detect any def-def anomaly and modify μ'(x) as follows: If status(μ', x) = false, report x as a def-def anomaly and modify μ'(x) [firing rule]; otherwise, set status(μ', x) := false and modify μ'(x) [firing rule].

1.2.2) For every x *used* in do(t), detect any undef-use anomaly as follows: If status(μ', x) = und, report x as an undef-use anomaly and stop the process for generating μ'; otherwise, set status(μ', x) :=true.

For every x *defined* in do(t), detect any def-def anomaly and modify μ'(x) as follows: If status(μ', x) = false, report x as a def-def anomaly, modify μ'(x) [firing rule]; otherwise, set status(μ', x) := false and modify μ'(x) [firing rule].

For every x *undefined* in do(t), detect any def-undef anomaly and modify μ'(x) as follows: If status(μ', x) = false, report x as a def-undef anomaly, set status(μ', x) := μ'(x) :=und [firing rule]; otherwise, set status(μ', x) := μ'(x) := und [firing rule].

/* No undef-use anomaly is found in N. Expand G with t and μ'.*/

1.2.3) Calculate μ'(y) for every $y \in P$ [firing rule]. If $\mu' \notin M$, add μ' and t to G, and go to Step 1.3; otherwise, add t to G and stop the process for generating μ'.

1.3) <u>Set</u> reach(k+1) := reach(k+1) $\cup$ {μ'}.

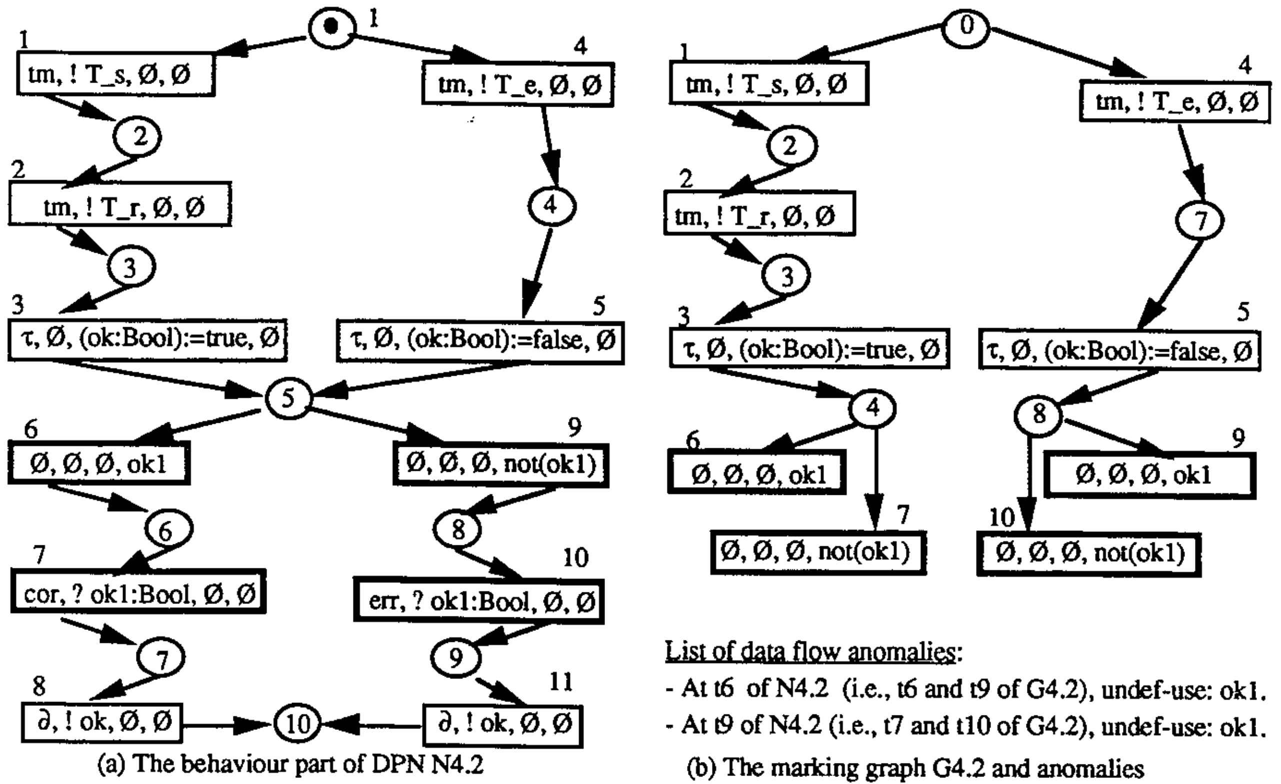
Step 2. If reach(k+1) $\neq \emptyset$, set k := k+1 and go to the next iteration; otherwise stop.

Example 4.2 (detection of data flow anomalies, Figure 4.3 and Table 4.2)

Consider the DPN N4.2 <P, T, F, DT, DS, V, C, L, f, label, μ0> shown in Figure 4.3 (a) for its behaviour part and in Figure 4.3 (c) for its data type part. There exists a deliberate error: variable *ok1* is a mistyping of variable *ok* . As a result, every path ending at t6 or t9 has an undef-use variable *ok1*.

These anomalies can be detected by applying method DETANOM on N4.2. The output includes the marking graph G4.2 and the list of data flow anomalies shown in Figure 4.3(b). Table 4.2 shows the status of the transitions, places, variables and constants after each iteration. For example, at iterations 3, *ok1* is found to be undef-use at t6 and t9 in the following paths of G4.2: μ0-t4-μ7-t5-μ8-t9 and μ0-t4-μ7-t5-μ8-t10, respectively. At iteration 4, *ok1* is also found to be undef-use at t6 and t9 in the following paths of G4.2: μ0-t1-μ2-t2-μ3-t3-μ4-t6 and μ0-t1-μ2-t2-μ3-t3-μ4-t7, respectively.

The meanings of the columns in Table 4.2 are as follows: The column "k" indicates the iteration number. The column "Trans_fired" includes those transitions which have been fired. The column "reach(k+1)" contains all markings reached from μ0 through a sequence of k transitions (listed in column "Trans_fired"). The subcolumn "places" (under column "marking μ") consists of places containing tokens. The column "status of parameter" lists the usage status for every parameter. Under this column, the symbol '*' indicates the parameter which is defined but unused when an undef-use anomaly is detected. The column "Trans_fireable" contains the transitions each of whose preplaces has at least one token.



(a) The behaviour part of DPN N4.2

(b) The marking graph G4.2 and anomalies

DT ={Boolean, time}; DS ={Bool, tim}; f (Boolean) ={Bool}; f(time) ={tim};

V = {(ok : Bool), (ok1 : Bool)}; C = { (true : Bool), (false : Bool), (T_s : tim), (T_r : tim), (T_e : tim)}.

(c) The data type sets and functions of N4.2

Figure 4.3 Data flow anomalies in a Data Petri-Net

k	Trans_ fired	reach (k+1)	marking μ places	ok	ok1	status of parameters ok	ok1	T_s	T_r	T_e	true	false	Trans_ firable
0		μ0	p1	und	und	und	und	false	false	false	false	false	t1, t4
1	t1	μ2	p2	und	und	und	und	true	false	false	false	false	t2
	t4	μ7	p4	und	und	und	und	false	false	true	false	false	t5
2	t1,t2	μ3	p3	und	und	und	und	true	true	false	false	false	t3
	t4,t5	μ8	p5	false	und	false	und	false	false	true	false	true	t6, t9
3	t1,t2,t3	μ4	p5	true	und	false	und	true	true	false	true	false	t6, t9
	t4,t5,t6					*		*	*		*		
	t4,t5,t9					*		*	*		*		
4	t1,t2,t3,t6					*				*		*	
	t1,t2,t3,t9					*				*		*	

Table 4.2 The status of transitions, places, variables and constants after each iteration

4.3.3 Criteria for Selecting Test Sequences

Theoretically, every class of criteria reviewed in Section 2.3 can be modified and applied to the selection of test sequences for Data Petri-Nets. Rapps and Weyuker's criteria are chosen as the basis of our work. As summarized in Section 2.4, Rapps' criteria fall in two categories: 1) Those based on control flows. This includes the "all-nodes", "all-edges" and "all-paths" criteria. These three criteria can be directly adopted to our marking graph. 2) Those based on data flows. This includes the "all-defs", "all-uses" and "all-du-paths" criteria. These three criteria are extended with the covers of the designated parameter (variable or constant), of all parameters of the designated data sort and of all parameters of the designated data type, as defined below.

Definition 4.7 (*three families of coverage criteria*)

Let P be a set of complete paths of the marking graph G<M, T^{μ}, F^{μ}, DT, DS, V, C, L^{μ}, $label^{\mu}$, f, $\mu 0$>, *x* be a parameter, *s* be a data sort and *t* be a data type.

(1) The family of all-defs covers:

* P is an *all-defs-of-par(x)* cover of G, if, $\forall$ n $\in$ (M $\cup$ T^{μ}) where x $\in$ def(n), P includes *at least one* path (n)·p·(m), where m $\in$ T^{μ} which has a use of x and p is def-clear w.r.t. x.
* P is an *all-defs-of-sort(s)* cover of G, if, $\forall$ (x : s) $\in$ (V $\cup$ C), P is an all-defs-of-par(x) cover of G.
* P is an *all-defs-of-type(t)* cover of G, if, $\forall$ s $\in$ f(t), P is an all-defs-of-sort(s) cover of G.
* P is an *all-defs* cover of G, if, $\forall$ t $\in$ DT, P is an all-defs-of-type(t) cover of G.

(2) The family of all-uses covers:

* P is an *all-uses-of-par(x)* cover of G, if, $\forall$ n, m $\in$ (M $\cup$ T^{μ}) where x $\in$ (def(n) $\cap$ use(m)) and def(n) can reach use(m) w.r.t. x, P includes *at lease one* path (n)·p·(m), where p is def-clear w.r.t. x.
* P is an *all-uses-of-sort(s)* cover of G, if, $\forall$ (x : s) $\in$ (V $\cup$ C), P is an all-uses-of-par(x) cover of G.

* P is an *all-uses-of-type(t)* cover of G, if, $\forall$ s $\in$ f(t), P is an all-uses-of-sort(s) cover of G.
* P is an *all-uses* cover of G, if, $\forall$ t $\in$ DT, P is an all-uses-of-type(t) cover of G.

(3) The family of all-du-paths covers:

* P is an *all-du-paths-of-par(x)* cover of G, if, $\forall$ n, m $\in$ (M $\cup$ T^{μ}) where x $\in$ (def(n) $\cap$ use(m)) and def(n) can reach use(m) w.r.t x, P includes *every path* (n)·p·(m), where p is cycle-free and def-clear w.r.t. x.
* P is an *all-du-paths-of-sort(s)* cover of G, if, $\forall$ (x : s) $\in$ (V $\cup$ C), P is an all-du-paths-of-par(x) cover of G.
* P is an *all-du-paths-of-type(t)* cover of G, if, $\forall$ s $\in$ f(t), P is an all-du-paths-of-sort(s) cover of G.
* P is an *all-du-paths* cover of G, if, $\forall$ t $\in$ DT, P is an all-du-paths-of-type(t) cover of G.

The first criterion in each family, namely the *criterion on par(x)*, aims at covering paths related to the designated parameter x. The second one, namely the *criterion on sort(s)*, aims at covering paths related to all parameters of the designated sort s. The third one, namely the *criterion on type(t)*, aims at covering paths related to all parameters of the designated type t.

The completeness of a path needs more discussion. Consider the definition given in Definition 2.1: "A path is said to be *complete* if its first node is an entry node of G and its last node is an *exit* node of G". In a marking graph G, the initial marking can be used as an entry node. However, if the DPN includes a recursive process, its marking graph may not have any *exit* node, i.e., no path has an exit node. To handle this situation, in practice, a specific marking, namely *terminal marking*, can be chosen by the system designer so that those paths ending at this marking are considered as complete. An example is shown in Figure 4.5, where the initial marking $\mu 0$ is chosen as the terminal marking.

Discussion (comparison of the coverage criteria, Figures 4.4 and 4.5)

A comparison of the proposed criteria is given in Figure 4.4, where the three criteria based on control flows, namely *all-nodes*, *all-edges* and *all-paths*, are the same as the ones proposed by Rapps, et al. (as reviewed in Section 2.4). In this figure, a symbol "->" attached with a condition means that, when this condition is satisfied, the comparison result (i.e., "stronger than") is correct.

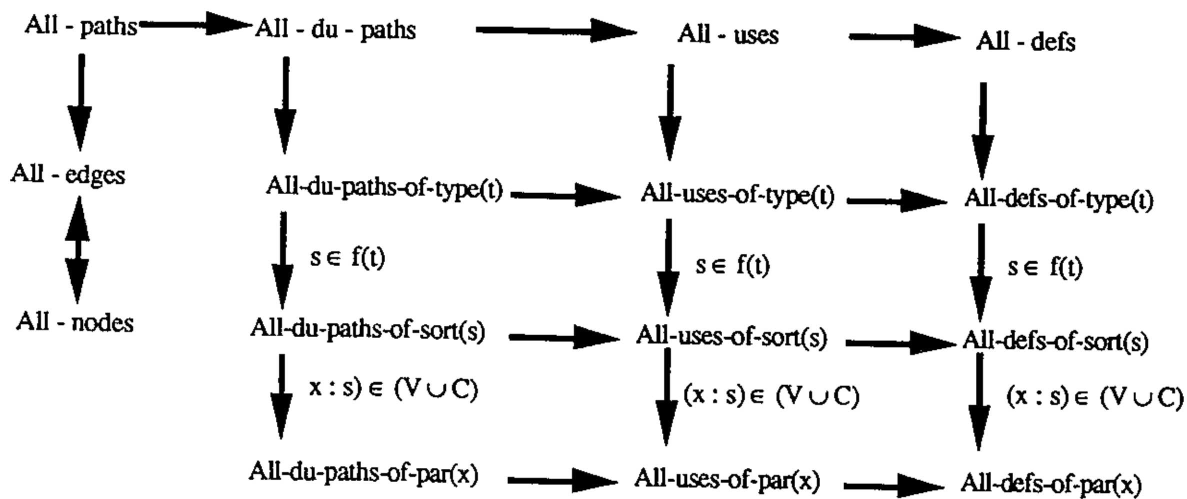


Figure 4.4 A comparison of the coverage criteria for Data Petri-Nets

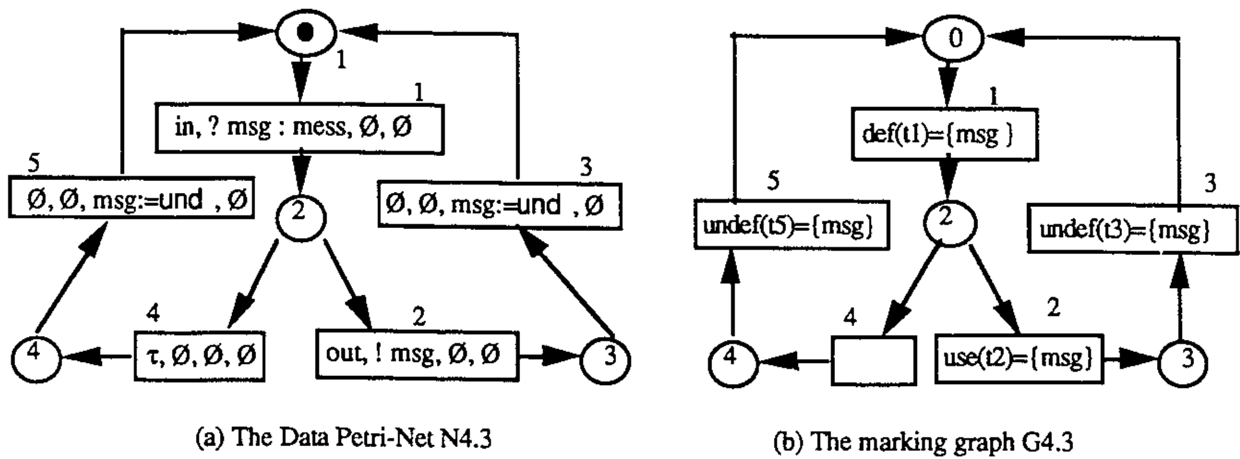
All cases in Figure 4.4 can be deduced immediately from the definitions of the criteria, except the following two ones:

a) An "all-edges" criterion is equivalent to an "all-nodes" criterion. In our marking graph, every transition has one in-coming and one out-going edge, and every edge is connected to one and only one transition node. So, covering all nodes (including all transitions) implies covering all edges. On the other hand, all nodes (transitions and markings) are connected by edges. So, covering all edges implies covering all nodes.

b) An "all-du-paths" criterion is not stronger than an "all-nodes" criterion. Consider a transition node whose label is either <τ, Ø, Ø, Ø> or <Ø, Ø, Ø, Ø>. In either case, no data is involved. Hence, an all-du-paths cover does not necessarily include this transition. An example is given below.

Example 4.3 (comparison of criteria, Figure 4.5)

Consider the DPN N4.3 as shown in Figure 4.5(a). Its marking graph G4.3 has the same structure and label function. The definition, use and undefinition sets are summarized in Figure 4.5(b). There exists only one du-path: t1-μ2-t2. Assuming that μ0 is the initial/terminal marking, the complete path set P ={μ0-t1-μ2-t2-μ3-t3-μ0} satisfies all-du-paths criterion. But P does not satisfy all-nodes criterion because nodes t4, t5 and μ4 are not contained in P.



(a) The Data Petri-Net N4.3

(b) The marking graph G4.3

Figure 4.5 An example on the comparison of the coverage criteria

4.3.4 A Method for Generating Selective Test Sequences

Based on one of the criteria proposed in Subsection 4.3.3, a set of test paths can be found from a marking graph. A set of test sequences is then generated for covering these test paths. This is achieved by the following method.

METHOD GENTEST (generation of selective test sequences)

INPUT: 1) A specified coverage criterion.

2) The marking graph G<M, T^{μ}, F^{μ}, DT, DS, V, C, L^{μ}, f, $label^{\mu}$, μ0>.

OUTPUT: A set of test sequences.

PROCEDURE:

Step 1. Identify those parameters involved in the specified coverage criterion. The result is a set par_set defined as follows:

1.1) If the criterion is on par(x), set par_set(x) := {x} and go to Step 2; otherwise, go to Step 1.2.

1.2) If the criterion is on sort(s), set par_set(s) := {x | (x : s) ∈ (V ∪ C)} and go to Step 2; otherwise go to Step 1.3.

1.3) If the criterion is on type(t), set par_set(t) := {x | s ∈ TS(t) and (x : s) ∈ (V ∪ C)} and go to Step 2; otherwise, go to Step 1.4.

1.4) Set par_set(all) := { x | s ∈ DS and (x : s) ∈ (V ∪ C)}.

Step 2. For each parameter x in par_set, identify those du-paths involved in the specified criterion as follows:

2.1) Locate the definitions and uses of x in the marking graph.

2.2) According to the criterion, find all the du-paths, on each of which x is defined at the first nodes and used at the last node.

Step 3. Generate a set of complete paths covering all the du-paths identified in Step 2.

Step 4. Convert each complete path generated in Step 3 to a test sequence as follows:

4.1) For every transition t with label(t) = <gate(t), event(t), do(t), when(t)> and gate(t) ∉ {τ, Ø}, generate a corresponding action in the test sequence.

4.2) Refine this sequence by the information carried in transitions whose gate-clauses are in set {τ, Ø}.

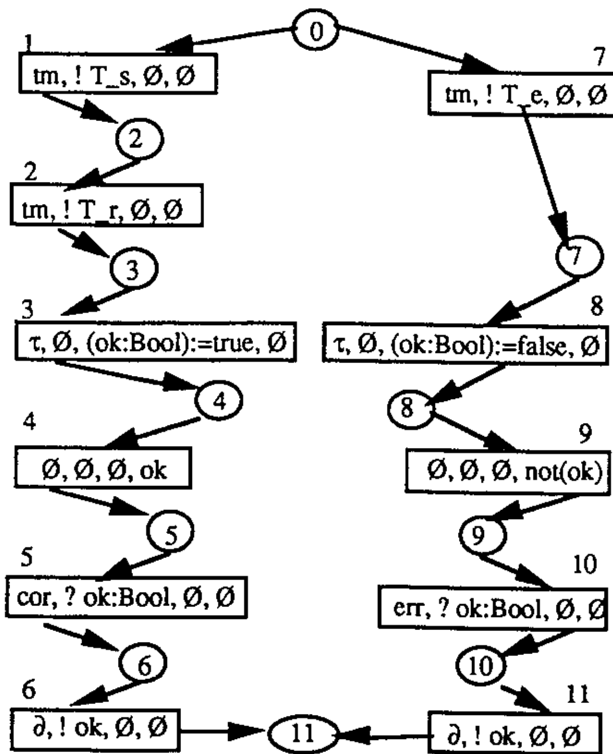
<u>Discussion</u> (definition of test sequences)

The definition of a test sequence may vary according to the different applications. For example, a test sequence in [CHE91] is defined as a sequence of reverse external actions of a test path, i.e., if there exists an *input* action in the test path, there must be in the test sequence an *output* action of the same message, and vice versa. In [FAC91], if there exists an *input/output* action in the test path, there must be in the test sequence an *output* action of the same message. It is trivial to

convert an external test-path action to a test-sequence action according to either definition. In our method, a sequence of external test-path actions is used as a pseudo-test sequence which can be easily converted to a real test sequence at the stage of implementation.

Example 4.4 (selective test sequence generation, Figure 4.6)

The errors in the DPN N4.2 used in Example 4.2 can be removed by changing every "ok1" to a "ok". The marking graph for the modified DPN is G4.4 shown in Figure 4.6, where μ0 and μ11 are the initial marking and the terminal marking, respectively. The symbol "o1" (resp., "o2") under the column "μ(ok)" is a symbolic value assigned to variable "ok" when transition t7 (resp., t9) is fired. To cover all definitions related to sort *Bool*, method GENTEST is applied to G4.4 with the criterion "all-defs-of-sort(*Bool*)" as follows.



μ	places	μ(ok)
μ0	p1	
μ2	p2	
μ3	p3	
μ4	p5	true
μ5	p6	true
μ6	p7	o1
μ7	p4	
μ8	p5	false
μ9	p8	false
μ10	p9	o2
μ11	p10	

Note: Only places having tokens are listed

The data type sets and functions of G4.4:

DT ={Boolean, time}; DS ={Bool, tim}; f (Boolean) ={Bool}; f(time) ={tim};

V = {(ok : Bool)}; C = { (true : Bool), (false : Bool), (T_s : tim), (T_r : tim), (T_e : tim)}.

Figure 4.6 The marking graph G4.4 used for test sequence generation

The criterion indicates that only parameters of sort Bool are involved. So, par_set(Bool) ={ok, true, false}. The identification of involved du-paths and complete paths is shown in Table 4.3, where the column "co-no." lists the reference numbers of the complete paths. The generation of the test sequences from the complete paths is included in Table 4.4.

par.	def. at	used at	du-paths	complete paths	co-no.
true	μ0	t3	μ0-t1-μ2-t2-μ3-t3	μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5--t5-μ6-t6-μ11	1
false	μ0	t8	μ0-t7-μ7-t8		
ok	t3, t5	t4, t6	t3-μ4-t4, t5-μ6-t6	μ0-t7-μ7-t8-μ8-t9-μ9-t10--μ10-t11-μ11	2
	t8, t10	t9, t11	t8-μ8-t9, t10-μ10-t11		

Table 4.3 Identification of the complete paths covering all definitions of sort Bool

co-no.	trans.	actions	test sequences	note on the test sequences
1	t1	tm ! T_s	tm ! T_s;	The null actions corresponding to transitions t3 and t4 are not included in the test sequence.
	t2	tm ! T_r	tm ! T_r;	
	t3	ok := true	cor ? ok : Bool;	
	t4	[ok]	exit(ok)	
	t5	cor ? ok : Bool		
	t6	exit(ok)		
2	t7	tm ! T_e	tm ! T_e;	The null actions corresponding to transitions t8 and t9 are not included in the test sequence.
	t8	ok := false	err ? ok : Bool;	
	t9	[not(ok)]	exit(ok)	
	t10	err ? ok : Bool		
	t11	exit(ok)		

Table 4.4 Generation of the test sequences covering all definitions of sort Bool

Chapter 5

VALIDATION OF LOTOS SPECIFICATIONS VIA DATA PETRI-NETS

5.1 A VALIDATION METHOD BASED ON DATA PETRI-NETS

LOTOS has the facilities for handling abstract data types and data synchronization. However, among its validation methods as reviewed in Chapter 3, only Tripathy's method [TRI91] applies data flow analysis for the selection of test sequences. But, it does not consider any specific selection criterion or data flow anomalies. In this chapter, as one of the contributions of this thesis, the DPN-based validation methods developed in Chapter 4 are applied to LOTOS specifications.

Although four transformation methods have been proposed in the literature, they either do not consider data or are not formally defined. (See Chapter 3 for a survey of these transformations.) Our contribution includes the development of a transformation from a subset of LOTOS to DPNs (Section 5.2).

In our method, a LOTOS specification S is first transformed to a DPN N. Then, N is checked for potential errors through the detection of data flow anomalies and is used to generate test sequences selectively. The method is described formally below.

METHOD VALOTOS (validation of a LOTOS specification)

INPUT: 1) A LOTOS specification S.

2) A specified coverage criterion.

OUTPUT: 1) A list of data flow anomalies.

2) A set of test sequences.

PROCEDURE:

Step 1. Transform S to a DPN N (to be described in Section 5.2).

Step 2. Detect data flow anomalies by applying method DETANOM (Subsection 4.3.2) on N.

Step 3. Identify any error by analyzing the results of Step 2. If any error is identified in N, correct S and go to Step 1; otherwise, go to Step 4.

Step 4. Generate test sequences by applying method GENTEST (Subsection 4.3.4).

An example is given below for illustrating the application of the above method to LOTOS. In fact, details of two of the steps (Step 2 and Step 4) have been given in Chapter 4 (Examples 4.2 and 4.4, resp.) for illustrating their individual methods. The example just shows an integration of these methods.

Example 5.1 (validation of LOTOS specifications)

Consider the following LOTOS specification of process S5_1. For anomaly detection (Step 2), the following error is deliberately inserted: The variable *ok1* appearing at lines 15 and 17 is a misprint of *ok* appearing at line 14. This error causes some undef-use anomalies in the DPN representation. For test sequence generation (Step 4), the criterion "all-defs-of-sort(*Bool*)" is used for covering all definitions related to the data sort *Bool*.

```
1     process    S5_1 [tm, cor, err] : exit :=
2         library
3             Boolean
4         endlib
5         type time is
6          sort  tim
7          opns
8             T_s    : -> tim
9             T_r    : -> tim
10            T_e    : -> tim
11        endtype
12        ( ( tm ! T_s; tm ! T_r; exit (true) )
```

```
13              [ ] ( tm ! T_e; exit (false) )
14            ) >> accept ok : Bool  in
15            ( [ ok1 ] -> cor ? ok1:Bool ; exit(ok)
16              [ ]
17              [ not(ok1) ] -> err ? ok1 :Bool ; exit(ok) )
18    endproc
```

Following are the steps of applying the method VALOTOS on S5_1:

Step 1 (transformation). The LOTOS specification of S5_1 is transformed to its DPN representation N4.2 as used in Example 4.2 (Figure 4.3 (a), (c)).

Step 2 (anomaly detection). S5_1 is checked for data flow anomalies by applying method DETANOM on N4.2. This process was described in Example 4.2. Two undef-use anomalies of ok1 are found at the transitions t6 and t9 of N4.2 (Figure 4.3(b)).

Step 3 (error identification). To identify the errors which cause these anomalies, we consider the status of the parameters given in Table 4.2. For instance, at iterations 3 and 4, ok is found to be defined but unused (marked with the symbol '*') while ok1 is undefined but used. This coincidence suggests that ok and ok1 may be the same variable. If really so, this error in S5_1 can be removed by changing every "ok1" to a "ok". The new DPN has the marking graph G4.4 as used in Example 4.4 (Figure 4.6).

Step 4 (selective test sequence generation). The required test sequences can be generated by applying method GENTEST on G4.4 with the criterion "all-defs-of-sort(Bool)". This process was described in Example 4.4. The following two test sequences are generated.

1) tm ! T_s; tm ! T_r; cor ? ok : Bool; exit(ok).

2) tm ! T_e; err ? ok : Bool; exit(ok).

5.2 TRANSFORMATION OF LOTOS SPECIFICATIONS TO DATA PETRI-NETS

Like other net representations reviewed in Chapter 3, our transformation is applicable only to a subset of LOTOS with the following constraints: 1) Sum-expressions and par-expressions are not considered. 2) Recursive process instantiations may not be allowed in the left sub-expression of an enabling/disabling operator or in the operands of a parallel operator.

In the rest of this section, the process for transforming LOTOS specifications to Data Petri-Nets is described first informally in Subsection 5.2.1 and then formally in Subsection 5.2.2.

5.2.1 Informal Description of the Transformation Process

This subsection includes an informal description of the transformation process of a LOTOS specification S to a Data Petri-Net N. The transformation consists of two parts: one for abstract data types and the other for behaviour expressions as summarized in Table 5.1.

LOTOS specification S	DPN N<P, T, F, DT, DS, V, C, L, f, label>
abstract data types	DT, DS, C, f
behaviour expressions	P, T, F, V, L, label

Table 5.1 Transformation of LOTOS specifications to Data Petri-Nets

Transformation of abstract data types

The transformation of abstract data types is straightforward. It just creates the sets DT, DS, C and function f as follows. For every abstract data type t defined in S, do the following:

a) Set $DT := DT \cup \{t\}$.

b) Identify all the data sorts $s_1, .., s_n$ involved in t, and set $DS := DS \cup \{s_1, .., s_n\}$, $f(t) := \{s_1, .., s_n\}$.

c) For every data sort s_i , $i = 1, ..., n$, of t, identify all the constants of s_i, say $c_{1i}, ..., c_{mi}$, and set $C := C \cup \{(c_{1i} : s_i), ..., (c_{mi} : s_i)\}$.

Example 5.2 (transformation of abstract data types)

Consider the process S5_1 in Example 5.1. It defines two data types: standard type Boolean and a user-defined type time. Based on the standard [ISO88], Boolean consists of a data sort Bool, and the sort Bool in turn has two constants: true and false. Based on the specification in S5_1, time consists of a data sort tim, and the sort tim in turn has three constants: T_s, T_r, and T_e. As a result, the DPN for S5_1 has the following sets and functions: DT = {Boolean, time}, DS = {Bool, tim}, C = {(true : Bool), (false : Bool), (T_s : tim), (T_r : tim), (T_e : tim)}, f(Boolean) = {Bool} and f(time) = {tim}.

Transformation of behaviour expressions

In general, a LOTOS expression B is either atomic or recursively defined by applying an operator on other sub-expressions. The transformation of B to a DPN representation N is accomplished by bottom-up synthesis in the following way: First, transform the sub-expressions involved to their DPNs. Then, connect these DPNs by some places and transitions according to the operator involved. During the connection of two DPNs, two kinds of places are frequently involved: an *input place* which will receive tokens from the preceding NPN; an *output place* which will pass tokens to the succeeding NPN.

Following is an informal description of the transformation of behaviour expressions:

a) B is one of the four "atomic" expressions: inaction "stop", external action "g α [Pt]", internal action "i" and successful termination "exit(e_1, ..., e_k)".

If B is an inaction, N consists of one place; otherwise, N consists of two places p0 and p1 connected by transition t0.

Example 5.3

The DPNs for the four atomic expressions are shown in Figure 5.1.

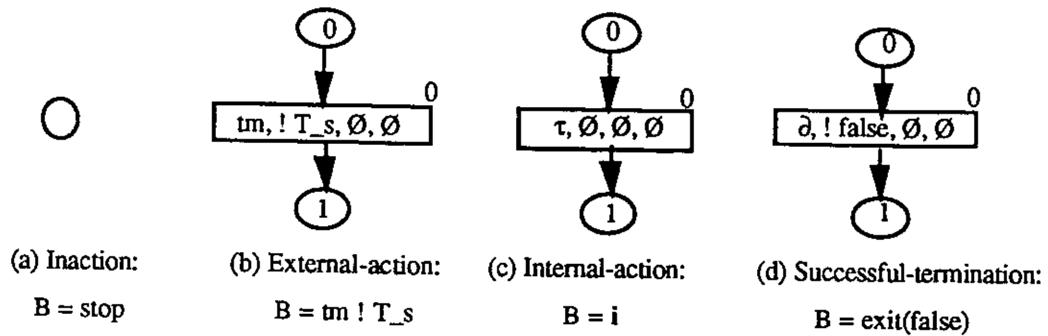


Figure 5.1 Transformation of the "atomic" expressions

b) B is a "unary" operation, i.e., a guarding operation "[Pt] -> B_b" or a local definition "let x_1: $s_1 = e_1, \ldots, x_k : s_k = e_k$ in B_b".

N is constructed in two steps: First, B_b is transformed to its DPN N_b. Then, a place p0 is created and connected to the input place of N_b by a transition t0.

Example 5.4

The DPNs for the guarding operation "B = [ok] -> B_b" and local-definition "B = let ok : Bool = true in B_b" are given in Figure 5.2 (a) and (b), respectively, where B_b = cor ! ok ; exit. [The transformation of B_b is given later in the description for action-prefix operations.]

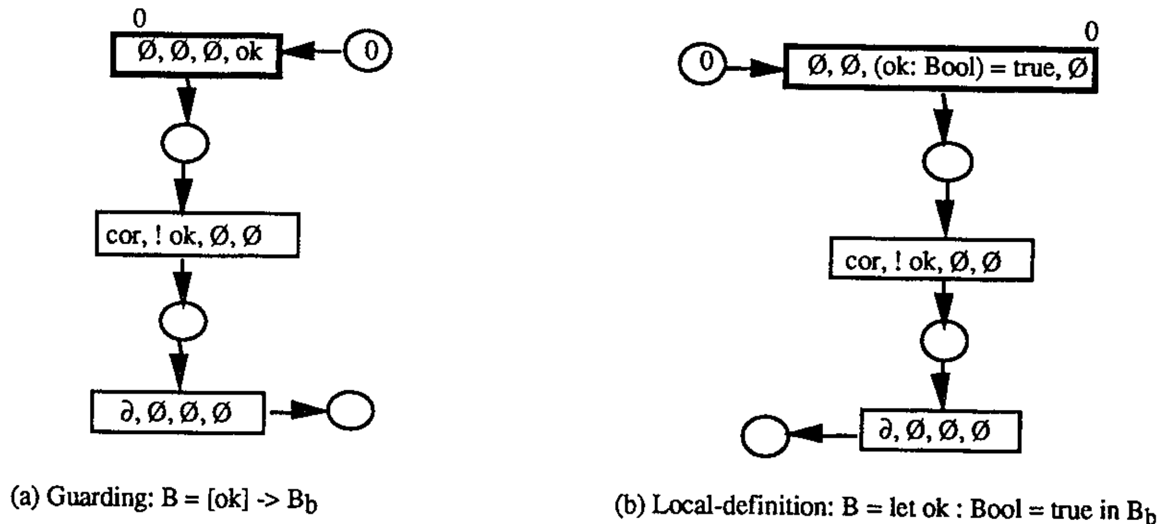


Figure 5.2 Transformation of the "unary" operations

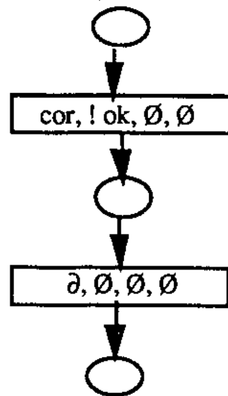
c) B is a "binary" operation in the form "B_a *op* B_b", where B_a and B_b are two behaviour expressions and *op* is a binary operator including action-prefix ";", enable ">> accept x_1:s_1, ..., x_k: s_k in", choice "[]", disable "[>" and parallel "|Gs|" where Gs is a set of user-defined gates.

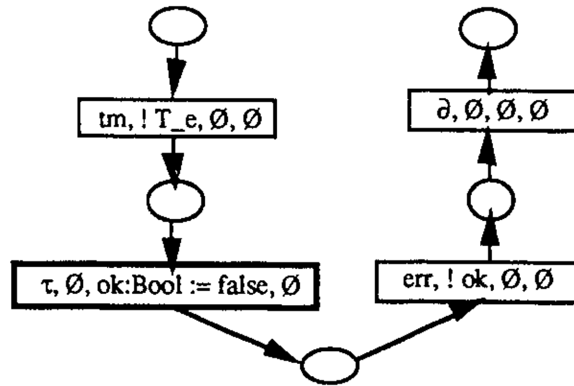
N is constructed in two steps: First, B_a and B_b are transformed to their DPNs N_a and N_b, respectively. Then, N_a and N_b are connected according to the contents of *op* as follows:

c.1) If *op* is an action-prefix operator, the output place of N_a is merged with the input place of N_b.

Example 5.5

The DPN for the action-prefix operation "B = B_a; B_b" is given in Figure 5.3 (a), where B_a = cor ! ok and B_b = exit.





(a) Action-prefix: B = B_a; B_b

(b) Enabling: B = B_a >> accept ok : Bool in B_b

Figure 5.3 Transformation of the action-prefix and enabling operations

c.2) If *op* is an enable operator, the output place of N_a is merged with the input place of N_b. Also, the do-clause of every transition connected to the output place of N_a is redefined for value passing.

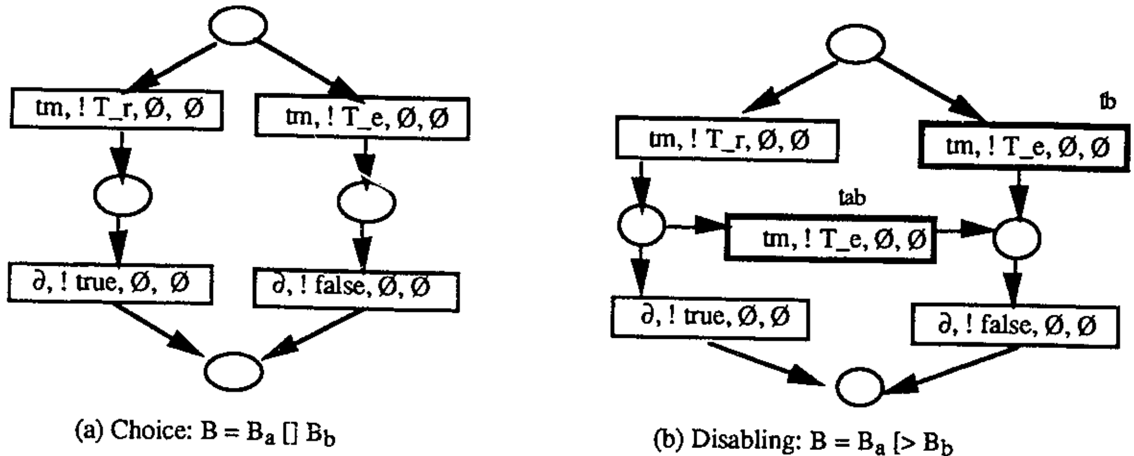
Example 5.6

The DPN for the enabling operation "B = B_a >> accept ok : Bool in B_b" is given in Figure 5.3 (b), where B_a = tm ! T_e; exit(false) and B_b = err ! ok ; exit.

c.3) If *op* is a choice operator, the input (resp., output) place of N_a is merged with the input (resp., output) place of N_b.

Example 5.7

The DPN for the choice operation "B = B_a [] B_b" is given in Figure 5.4 (a), where B_a = tm ! T_r ; exit(true) and B_b = tm ! T_e ; exit(false).



(a) Choice: B = B_a [] B_b

(b) Disabling: B = B_a [> B_b

Figure 5.4 Transformation of the choice and disabling operations

c.4) If *op* is a disable operator, the input (resp., output) place of N_a is merged with the input (resp., output) place of N_b. Also, for every place p_a in N_a and every transition t_b connected to the *input place* of N_b, a transition t_{ab} is created from p_a to the post-place(s) of t_b and label(t_{ab}) is set to label(t_b).

Example 5.8

The DPN for the disabling operation "B = B_a [> B_b" is given in Figure 5.4 (b), where B_a = tm ! T_r ; exit(true) and B_b = tm ! T_e ; exit(false).

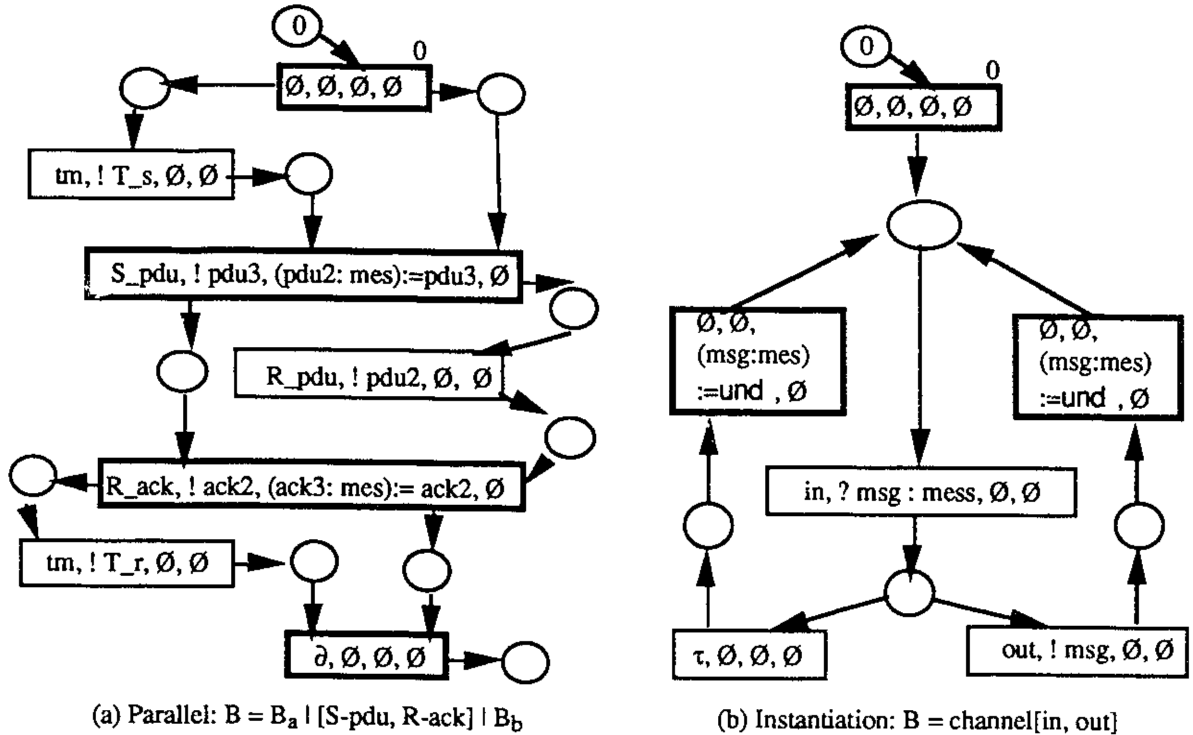
c.5) If *op* is a parallel operator, N_a and N_b are connected in two steps: First, a place p0 is created and connected to the input places of N_a and N_b by a transition t0. Then, every

transition t_a in Na in which gate(t_a) $\in$ Gs $\cup$ {∂} is merged with a transition t_b in Nb in which gate(t_a) = gate(t_b).

Example 5.9

The DPN for parallel operation "B = B_a | [S-pdu, R-ack] | B_b" is given in Figure 5.5(a), where B_a = tm ! T_s ; S_pdu ! pdu3 ; R_ack ? ack3 : mes ; tm ! T_r ; exit and

B_b = S_pdu ? pdu2 : mes ; R_pdu ! pdu2 ; R_ack ! ack2 ; exit.



(a) Parallel: B = B_a | [S-pdu, R-ack] | B_b

(b) Instantiation: B = channel[in, out]

Figure 5.5 Transformation of the parallel operations and process instantiations

d) B is a process instantiation , i.e., "P[g_1,...,g_m] (e_1, ..., e_k)".

N is constructed in two steps: First, a place p0 and a transition t0 from p0 are created and connected. Then, if the instantiation is recursive, i.e., a DPN N_a for the instantiated process P has been created, t0 is connected to the input place of N_a; otherwise, a DPN N_b is created according to the instantiated process P and t0 is connected to the input place of N_b.

Example 5.10

The DPN for process instantiation "B = channel[in, out]" is given in Figure 5.5 (b), where the instantiated process channel is defined below.

```
process channel [in, out] : noexit :=
    in ? msg : mes;
    (( i ; channel [in, out] ) []
     out ! msg ;
     channel [in, out] )
endproc
```

e) B is a hiding expression, i.e., "hide G_h in B_b", where G_h is a set of user-defined gates.

N is the same as the one for B_b with a new labelling function which sets every gate appearing in G_h to "τ".

The formal description of the transformation of LOTOS behaviour expressions is given in Subsection 5.2.2 by defining the sets P, T, F, V, L and function label.

Discussion (simplification of the transformed DPN, Figure 5.6):

In general, simplification of the transformed DPN is a difficult problem. In this thesis, we do not attempt to solve this problem completely. Instead, we consider only the following special structure transformed from expression "[Pt] -> a; B": Two transitions t1 (representing predicate *Pt*) and t2 (representing action *a*) are connected by place p. In this case, t1 can be merged with t2 because t2 is the only succeeding transition guarded by t1. An example is given in Figure 5.6. As an optional step of our transformation, a resulting DPN can be simplified as below.

For every place p, if there exist two transitions t1 and t2 such that gate(t1) = event(t1) = do(t1) = Ø, pre(p) ={t1} and post(p) = {t2}, then delete p and merge t1 and t2 into a new transition t such that gate(t) = gate(t2), event(t)= event(t2), do(t) = do(t2), and when(t) consists of when(t1) and when(t2).

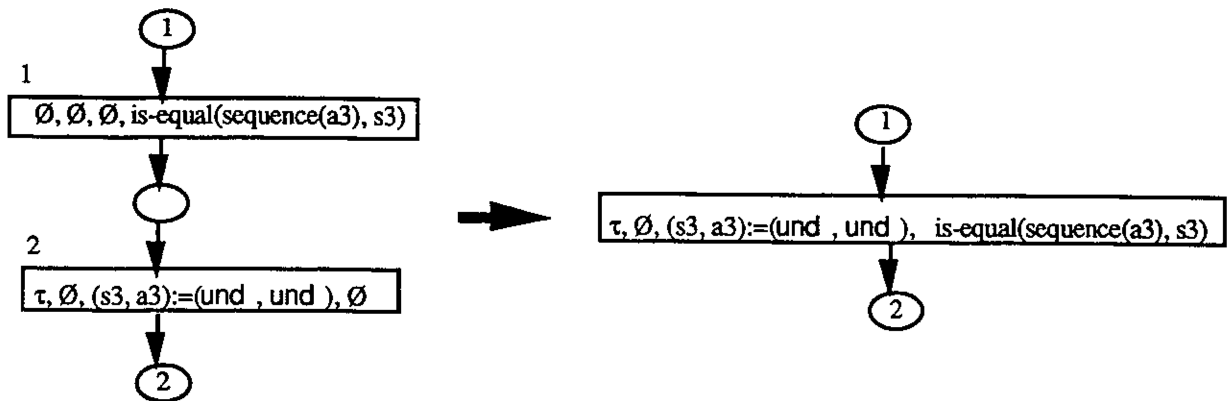


Figure 5.6 Simplification of the transformation

5.2.2 Formal Description of the Transformation Rules for Behaviour Expressions

This subsection presents formally the rules for transforming LOTOS expressions. The notations to be used are summarized in Table 5.2.

Symbols	Meaning
$g, g_1, \ldots, g_k$	user-defined gates
∂	the gate where the successful termination (exit) occurs
i	the internal action
a	a user-defined action consisting of a gate-name g and a list α of value/variable declarations
gate(a)	the gate where action a occurs $\text{gate (a)} = \begin{cases} \partial & \text{if a is a successful termination} \\ \tau & \text{if a is an internal action} \\ g & \text{if a uses g as its gate-name} \end{cases}$
event(a)	the value/variable declarations appearing in action a
P_t	a predicate
B, B_a, B_b	behaviour expressions

Table 5.2 Notation used in the transformation of behaviour expressions (to be continued)

Symbols	Meaning
variable(B)	the set of variables appearing in B
B - a -> B'	behaviour B, after performing action a, becomes behaviour B'
e1/e2	expression e2 is replaced by expression e1
N	the DPN representation for B, N = <P, T, F, DT, DS, V, C, L, f, label>
P_i	the set of the input places of N, $P_i = \{p \mid p \in P, pre(p) = \emptyset\}$
P_o	the set of the output places of N, $P_o = \{p \mid p \in P, post(p) = \emptyset\}$
H	the set of the labels of N, $H = \{(t, label(t)) \mid t \in T\}$
+	a label merge operator

Table 5.2 Notation used in the transformation of behaviour expressions (continued)

Note: 1) The merge operator (+) works in the following way: i) Let x_1: s_1 and x_2: s_2 be two variable identifiers, e_1 and e_2 be two expressions, "(x_1: s_1) := e_1" + "(x_2: s_2) := e_2" yields an assignment "(x_1: s_1, x_2: s_2) := (e_1, e_2)". ii) Let b_1 and b_2 be two Boolean expressions, "b_1" + "b_2" yields a Boolean expression consisting of b_1 and b_2.

2) Notations for DPN N_a (resp., N_b) are the same as the ones for N with a subscript *a* (resp., *b*).

Definition 5.1 (*transformation rules*)

(1) Inaction B = **stop**

LOTOS: No axiom or inference rule is associated with the behaviour expression **stop**, and it is thus impossible to derive any transition from it.

DPN: $P = \{p_0\}$; $P_i = \{p_0\}$; $P_o = \emptyset$; $T = \emptyset$; $F = \emptyset$; $V = \emptyset$; $H = \emptyset$.

(2) Successful-termination B = **exit** (e_1, ..., e_k)

LOTOS: $B - \partial <!e_1, ..., !e_k> \to$ stop

DPN: $P = \{p_0, p_1\}$; $P_i = \{p_0\}$; $P_o = \{p_1\}$;

$T = \{t_0\};$

$F = \{(p_0, t_0), (t_0, p_1)\};$

$V = \emptyset;$

$H = \{(t_0, <\partial, !e_1...!e_k, \emptyset, \emptyset>)\}.$

Note: If the parameter list is empty, i.e., B = **exit**, the resulting net is the same as the one with a new labelling function defined as $H = \{ (t_0, <\partial, \emptyset, \emptyset, \emptyset>) \}$.

(3) Guarding $B = [P_t]$ -> B_b

LOTOS: P_t = true implies $B = B_b$

P_t = false implies B = stop

DPN: $P = \{p_0\} \cup P_b; P_i = \{p_0\}; P_o = P_{ob};$

$T = \{t_0\} \cup T_b;$

$F = \{(p_0, t_0), (t_0, p) \mid p \in P_{ib}\} \cup F_b;$

$V = V_b \cup$ variable(P_t);

$H = \{(t_0, <\emptyset, \emptyset, \emptyset, P_t>)\} \cup H_b.$

(4) Local-definition B = **let** $x_1: s_1 = e_1, \ldots, x_k : s_k = e_k$ **in** B_b

LOTOS: $B_b[e_1/x_1, ..., e_n/x_n]$ - a -> B_b' implies B - a -> B_b'

DPN: $P = \{p_0\} \cup P_b; P_i = \{p_0\}; P_o = P_{ob};$

$T = \{t_0\} \cup T_b;$

$F = \{(p_0, t_0), (t_0, p) \mid p \in P_{ib}\} \cup F_b;$

$V = V_b \cup \{x_1, ..., x_k\};$

$H = \{(t0, <\emptyset, \emptyset, (x_1: s_1, ..., x_k : s_k) := (e_1, ..., e_k), \emptyset>)\} \cup H_b.$

(5) Action-prefix $B = a [P_t] ; B_b$

LOTOS: B - a -> B_b iff P_t = true.

DPN: $P = \{ p_0\} \cup P_b; P_i = \{p_0\}; P_o = P_{ob};$

$T = \{t_0\} \cup T_b;$

$F = \{(p_0, t_0), (t_0, p_b) \mid p_b \in P_{ib}\} \cup F_b;$

$V = (\text{variable}(a\ [P_t])) \cup V_b;$

$H = \{(t_0, \langle \text{gate}(a), \text{event}(a), \emptyset, P_t \rangle)\} \cup H_b.$

Note: If the selection predicate Pt is not specified, Ø will be taken as the default value.

(6) Enabling $B = B_a >> \textbf{accept } x_1{:}s_1, ..., x_k{:}\ s_k \textbf{ in } B_b$

LOTOS: B_a - a -> B_a' and gate(a) $\neq \partial$

implies B - a -> B_a' >> **accept** $x_1{:}\ s_1, ..., x_k{:}\ s_k$ **in** B_b;

B_a - ∂ <$!e_1, ..., !e_k$ > -> B_a'

implies B - i -> $[e_1/x_1, ..., e_k/x_k]\ B_b$.

DPN: $P = (P_a - P_{oa}) \cup P_b;\ P_i = P_{ia};\ P_o = P_{ob};$

$T = T_a \cup T_b;$

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

$\{(t_a, p_b) \mid t_a \in T_a, \text{post}(t_a) = P_{oa}, p_b \in P_{ib}\};$

$V = V_a \cup V_b;$

$H = H_{xa} \cup H_b$

where H_{xa} is transformed from Ha as follows,

for every $t_a \in T_a$

if $(t_a, p_a) \in F_a$ and $p_a \in P_{oa}$

then $H_{xa}(t_a) = \langle \tau, \emptyset, (x_1{:}\ s_1,...,x_k{:}\ s_k) := (e_1,...,e_k), \text{when}(t_a) \rangle$

else $H_{xa}(t_a) = H_a(t_a)$.

Note: The interpretation is that if B_a **exit**, the next behaviour is B_b where the variables x_1 to x_n are substituted for the result values e_1 to e_n from B_a (via. the **exit** parameters).

(7) Choice $B = B_a\ [\,]\ B_b$

LOTOS: B_a - a -> B_a' implies B - a -> B_a'

B_b - a -> B_b' implies B - a -> B_b'

DPN: $P = (P_a - P_{ia} - P_{oa}) \cup P_b$; $P_i = P_{ib}$; $P_o = P_{ob}$;

$T = T_a \cup T_b$,

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

$\{(p,t_a) \mid p \in P_i, t_a \in T_a, pre(t_a) = P_{ia}\} \cup$

$\{(t_a,p) \mid p \in P_o, t_a \in T_a, post(t_a) = P_{oa}\}$;

$V = V_a \cup V_b$;

$H = H_a \cup H_b$.

(8) Disabling $B = B_a$ [> B_b

LOTOS:	B_a - a -> B_a' and gate (a) $\neq \partial$	implies	B - a -> B_a' [> B_b
	B_a - a -> B_a' and gate (a) $= \partial$	implies	B - a -> B_a'
	B_b - a -> B_b'	implies	B - a -> B_b'

DPN: $P = (P_a - P_{oa}) \cup P_b$; $P_i = P_{ia}$; $P_o = P_{ob}$;

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

where reachable(N_a) = $\{ p \mid p \in P_a, gate(t) = \emptyset$ for every $t \in pre(p)\}$

$PT_a = \{p_a \mid p_a \in (P_a - reachable(N_a))\}$;

$Txb = \{t_b \mid P_{ib} \supseteq post(t_b)\}$

$T_{ab} = \{ (t_a,t_b) \mid PT_a \supseteq pre(t_a), t_b \in T_{xb}\}$;

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

$\{(t_a, p_b) \mid t_a \in T_a, post(t_a) = P_{oa}, p_b \in P_{ob}\} \cup$

$\{(p_a, t_{ab}) \mid p_a \in PT_a, t_{ab} \in T_{ab}\} \cup \{(t_{ab},p_b) \mid pre(p_b) \in T_{xb}, t_{ab} \in T_{ab}\}$;

$V = V_a \cup V_b$;

$H = H_a \cup (H_b - \{(t_b, label(t_b)) \mid t_b \in Txb\}) \cup \{(t_{ab}, label(t_b)) \mid t_{ab} = (t_a, t_b) \in T_{ab}\}$.

Note: If B_a includes process instantiations, every variable in these processes will be undefined when B_a is disabled. This is performed at every t_{ab} where $t_{ab} = (t_a, t_b) \in T_{ab}$ and t_a belongs to one of the disabled processes.

(9) Parallel-composition $B = B_a \mid Gs \mid B_b$

where $Gs = [g_1, .., g_n]$ is a set of user-defined synchronization gates.

LOTOS: B_a - a -> B_a' and gate(a) $\notin$ S implies $B_a \mid Gs \mid B_b$ - a -> $B_a' \mid Gs \mid B_b$

B_b - a -> B_b' and gate(a) $\notin$ S implies $B_a \mid Gs \mid B_b$ - a -> $B_a \mid Gs \mid B_b'$

B_a - a -> B_a' and B_b - a -> B_b' and gate(a) $\in Gs \cup \{\partial\}$

implies $B_a \mid Gs \mid B_b$ - a -> $B_a' \mid Gs \mid B_b'$

DPN: $P = \{p_0\} \cup (P_a - P_{oa}) \cup P_b$; $P_i = \{p_0\}$; $P_o = P_{ob}$;

$T = \{t_0\} \cup (T_a - T_{xa}) \cup (T_b - T_{xb}) \cup T_{ab}$

where possible$(N_a, N_b, Gs) = \{(t_a, t_b) \mid t_a \in T_a, t_b \in T_b, \text{gate}(t_a) = \text{gate}(t_b) \in Gs\}$

$T_{xa} = \{t_a \mid t_a \in T_a \text{ and gate}(t_a) \in Gs \cup \{\partial\}\}$;

$T_{xb} = \{t_b \mid t_b \in T_b \text{ and gate}(t_b) \in Gs \cup \{\partial\}\}$;

$T_{ab} = \{(t_a, t_b) \mid t_a \in T_{xa}, t_b \in T_{xb}, (t_a, t_b) \in \text{possible}(N_a, N_b, Gs)\}$;

$F = \{(p_0, t_0)\} \cup \{(t_0, p_a) \mid p_a \in P_{ia}\} \cup \{(t_0, p_b) \mid p_b \in P_{ib}\} \cup$

$(F_a - \{(t_a, p_a) \mid t_a \in T_a, p_a \in P_{oa}\} - \{(p_a, t_a), (t_a, p_a) \mid p_a \in P_a, t_a \in T_{xa}\}) \cup$

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

$\{(p_a, t_{ab}) \mid (p_a, t_a) \in F_a, t_a \in T_{xa}, t_{ab} \in T_{ab}\} \cup$

$\{(t_{ab}, p_a) \mid (t_a, p_a) \in F_a, t_a \in T_{xa}, t_{ab} \in T_{ab}\} \cup$

$\{(p_b, t_{ab}) \mid (p_b, t_b) \in F_b, t_b \in T_{xb}, t_{ab} \in T_{ab}\} \cup$

$\{(t_{ab}, p_b) \mid (t_b, p_b) \in F_b, t_b \in T_{xb}, t_{ab} \in T_{ab}\}$;

$V = V_a \cup V_b$;

$H = \{(t_0, \langle \varepsilon, \emptyset, \emptyset, \text{true} \rangle)\} \cup (H_a - H_{xa}) \cup (H_b - H_{xb}) \cup H_{ab}$

where $H_{xa} = \{(t_a, H(t_a)) \mid t_a \in T_{xa}\}$;

$H_{xb} = \{(t_b, H(t_b)) \mid t_b \in T_{xb}\}$;

H_{ab} is defined as following,

for every $(t_a, t_b) \in T_{ab}$

$t_{ab} = (t_a, t_b)$

$\text{gate}(t_{ab}) = \text{gate}(t_a)$

case 1: /* event(t_a) = " ! e_a " and event(t_b) = " ! e_b " */

event(t_{ab}) = event(t_a)

do(t_{ab}) = do(t_a) + do(t_b)

when(t_{ab}) =when(t_a)+when(t_b)+ "$e_a = e_b$"

case 2: /* event(t_a) = "! e_a " and event(t_b) = " ? x_b : s_x" */

event(t_{ab}) = event(t_a)

do(t_{ab}) = do(t_a) + do(t_b) + "x_b : s_x := e_a"

when(t_{ab}) = when(t_a) + when(t_b)

case 3: /* event(t_a) = " ? x_a : s_x" and event(t_b) = " ? x_b : s_x" */

event (t_{ab}) = event(t_a)

do(t_{ab}) = do(t_a) + do(t_b) + "x_b : s_x := x_a"

when(t_{ab}) = when(t_a) + when(t_b)

end /* cases 1-3 */

/* Hab(t_{ab}) = <gate(t_{ab}), event(t_{ab}), do(t_{ab}), when(t_{ab}) > */

Note: 1) In case of pure interleaving, i.e., '|Gs|' is written as '|||', Gs is empty;

2) In case of full synchronization, i.e., '|Gs|' is written as '||', Gs consists of all gates.

(10) Process-instantiation B = P[g_1,...,g_m] (e_1, ..., e_k)

LOTOS: if 'process P[g'_1, ..., g'_m](x_1:s_1, ..., x_k:s_k) := B_b endproc' is a process definition

then: B_b[g_1/g'_1, ..., g_m/g'_m,e_1/x_1, ..., e_k/x_k] - a -> B_b'

implies P[g_1,...,g_m] - a -> B_b'

DPN: Case 1: /* The process instantiation is nonrecursive. If the functionality of this process is exit, every variable of this process must be undefined before exiting from this process. This is performed by function undef(Bb) at the "exit" transition t where post(t) $\subset P_{ob}$. */

Let N_b be the DPN of B_b. The result net N is defined as follows:

$P = \{p_0\} \cup P_b$; $P_i = \{p_0\}$; $P_o = P_{ob}$;

$T = \{t_0\} \cup T_b$;

$F = \{(p_0,t_0),(t_0,p) \mid p \in P_{ib}\} \cup F_b$;

$H = \{(t_0, <\emptyset, \emptyset, (x_1: s_1, ..., x_k: s_k) := (e_1, ...,e_k), \emptyset >) \} \cup H_{xb}$

where Hxb is transformed from Hb as follows.

for every $t_b \in T_b$, let $Hb(t_b)=(gate(t_b),event(t_b),do(t_b),when(t_b))$

$H_{xb}(t_b) =$

$$\begin{cases} (g_i,event(t_b),do(t_b),when(t_b)) & post(t_b)\not\subset Pob \text{ and } gate(t_b)=g'_i \\ (g_i,event(t_b),do(t_b)+undef(B_b),when(t_b)) & post(t_b)\subset Pob \text{ and } gate(t_b)=g'_i \\ H_b(t_b) & post(t_b)\not\subset Pob \text{ and } gate(t_b)\neq g'_i. \end{cases}$$

Case 2: /* The process instantiation is recursive. Before entering B_b again, every variables which do not appear in $(x_1{:}s_1, ..., x_k{:}s_k)$ must be undefined. This is performed by function $undef(t_0)$.*/

Let N_b be the equivalent DPN of process B_b without recursion, p_x be the output recursive place. The result net N is defined as follows:

$P = P_b$; $P_i = P_{ib}$; $P_o = P_{ob}$;

$T = \{t_0\} \cup T_b$;

$F = \{(p_x,t_0),(t_0,p) \mid p \in P_{ib}\} \cup F_b$;

$H = \{(t_0, <\emptyset, \emptyset, ((x_1: s_1, ..., x_k: s_k):=(e_1, ...,e_k))+undef(t_0), \emptyset >)\}\cup H_b$

end /*cases 1-2*/

Note: Like a program, every variable name in a LOTOS specification has its local-meaning, i.e., it is unique only within the process where it occurs. In order to specify a LOTOS specification via a single DPN, every variable must be renamed to avoid the global conflicts. The task should be completed in Case 1.

(11) Hiding $B = \textbf{hide } G_h \textbf{ in } B_b$

where $G_h = \{g_1, ..., g_n\}$ is a set of user-defined gates.

LOTOS: B_b - a -> B_b' and gate (a) $\notin G_h$ implies B - a -> hide G_h in B_b'

B_b - a -> B_b' and gate (a) $\in G_h$ implies B - i -> hide G_h in B_b'

DPN: $P = P_b$; $P_i = P_{ib}$; $P_o = P_{ob}$;

$T = T_b$;

$F = F_b$;

$V = V_b$;

$$H = \begin{cases} (t_b, < \tau, \emptyset, do(t_b), when(t_b)>) & t_b \in T_b \text{ and } gate(t_b) \in G_h \\ (t_b, H_b(t_b)) & t_b \in T_b \text{ and } gate(t_b) \notin G_h. \end{cases}$$

Note: This transformation is ignored when a gate $g \in G_h$ must be tested.

Chapter 6
APPLICATION TO THE ALTERNATING BIT PROTOCOL

6.1 INTRODUCTION

This chapter contains a detailed example of applying the validation methods developed in Chapter 5 on the process *Sender* of the Alternating Bit Protocol (ABP). The LOTOS specification of ABP is originally given in [BER88]. To make it more independent from the other processes not described here, Sender is modified as follows: 1) Its functionality is changed to "exit". 2) The variable *seqsender* gets its value through the gate *sendgate*. The coverage criterion to be used is "all-defs", for covering all definitions existing in Sender.

The rest of this chapter is organized as follows. Section 6.2 transforms Sender to its DPN representation N.Sender. Section 6.3 detects data flow anomalies in N.Sender and generates its marking graph G.Sender. Section 6.4 generates test sequences for covering all definitions of Sender. The LOTOS specification of process Sender and three related processes, namely *safemesser*, *messer* and *timer*, are given in the appendix of this chapter.

6.2 REPRESENTATION OF SENDER IN DATA PETRI-NETS

By the method described in Section 5.2, the LOTOS specification of Sender is transformed to the DPN *N.Sender* shown in Figures 6.1 and 6.2. Explanation of the transitions of N.Sender is given in Table 6.1. To avoid the global conflicts, according to the transformation rules (Definition 5.2), variables of processes *Sender*, *safemesser* and *messer* are renamed by adding number "1", "2" and "3" to their original names, respectively. Besides, the "hide" operation on gate *tm* is ignored for the purpose of generating test sequences related to this gate.

In the rest of this chapter, for the convenience of our presentation, the identifiers of transitions, places and markings are written inside their graphical representations. Also, the data sorts for variables in do-clauses are omitted, i.e., "(datmsg2,seqsender2):=(datmsg1, seqsender1)" is written instead of "(datmsg2 : datamsg, seqsender2 : Bit) := (datmsg1, seqsender1)", etc.

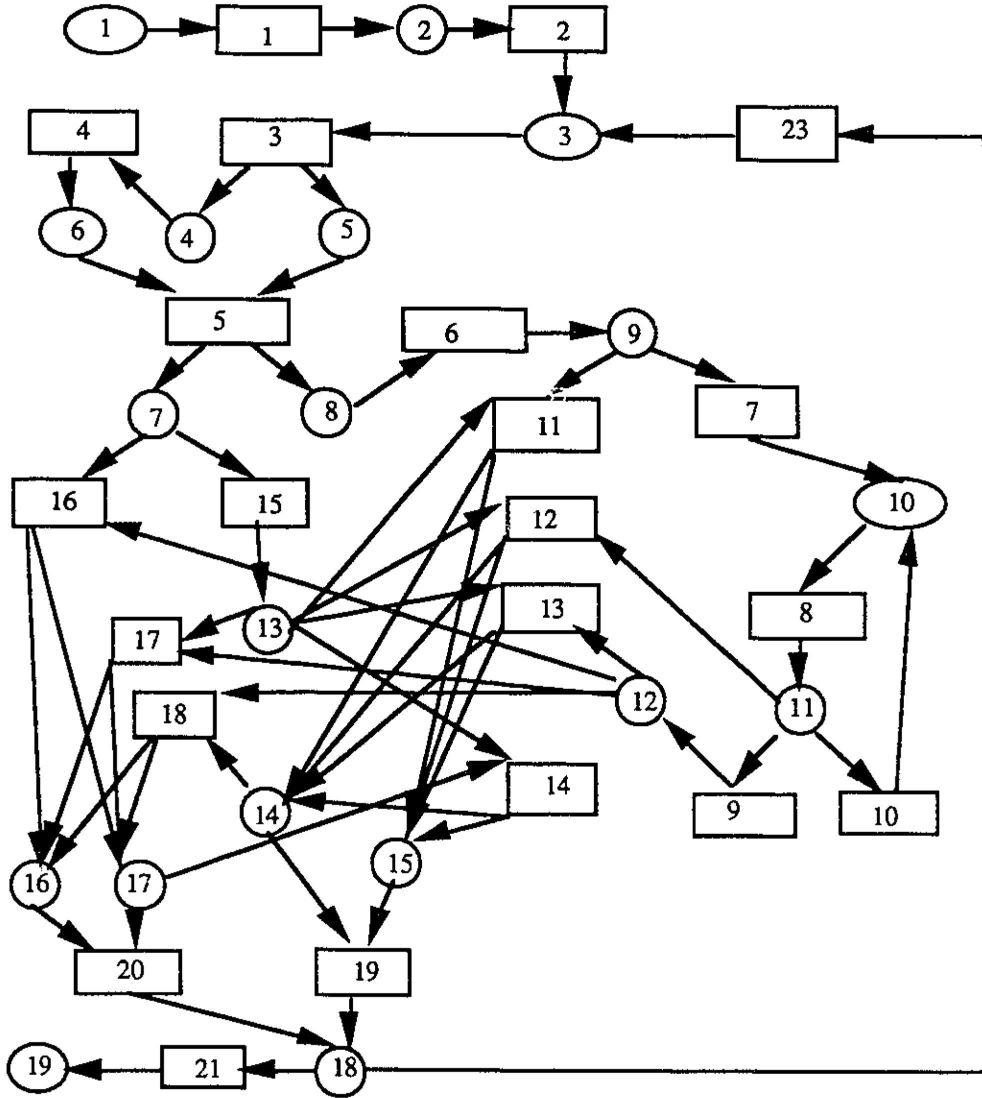


Figure 6.1 The Data Petri-Net N.Sender for the Sender of the Alternating Bit Protocol

DT := {Boolean, Bit, BitString, data, mess, time}; DS := {Bool, Nat, Bit, Bitstring, datamsg, message, tim};

V := {(ok2 : Bool), (seqsender1 : Bit), (seqsender2 : Bit), (seqsender3 : Bit), (datmsg1: datamsg),

(datmsg2 : datamsg), (ack3 : message)};

C := {(true : Bool), (false : Bool), (0 : Nat), (0 : Bit), (T_start : tim), (T_expired : tim), (T-reset : tim)};

f(Boolean) = {Bool}; f(Bit) = {Bool, Nat, Bit}; f(BitString) = {Bool, Nat, Bit, BitString};

f(data) = {Bool, Nat, Bit, datamsg}; f(mess) = {Bool, Nat, Bit, datamsg, message}; f(time) = {tim}.

Figure 6.2 The data types, data sorts, variables, constants and functions for N.Sender

k	four clauses of the label of transition k: gate-clause, event-clause, do-clause, when-clause
t1	sendgate, ? datmsg1 : datamsg ? seqsender1 : Bit, Ø, Ø
t2	Ø, Ø, (datmsg2, seqsender2) := (datmsg1, seqsender1), Ø
t3	Ø, Ø, Ø, Ø
t4	Ø, Ø, Ø, Ø
t5	tm, ! T_start, Ø, Ø
t6	sendpdu, ! makepdu(datmsg2,seqsender2), Ø, Ø
t7	Ø, Ø, seqsender3 :=seqsender2, Ø
t8	receiveack, ? ack3 : message, Ø, Ø
t9	τ, Ø, (seqsender3, ack3) := (und, und), is-equal(sequence(ack3),seqsender3)
t10	Ø, Ø, (seqsender3, ack3):= (seqsender3, und), is-not-equal(sequence(ack3),seqsender3)
t11	tm, ! T_expired, Ø, Ø
t12	tm, ! T_expired, (seqsender3, ack3) :=(und, und), Ø
t13	tm, ! T_expired, (seqsender3, ack3) :=(und, und), Ø
t14	tm, ! T_expired, Ø, Ø
t15	τ, Ø, Ø, Ø
t16	tm, ! T_reset, Ø, Ø
t17	tm, ! T_reset, Ø, Ø
t18	tm, ! T_reset, Ø, Ø
t19	Ø, Ø, ok2 := false, Ø
t20	Ø, Ø, ok2 := true, Ø
t21	∂, Ø, Ø, ok2
t23	Ø, Ø, (datmsg2, seqsender2, ok2) := (datmsg2,seqsender2,und), not(ok2)

Table 6.1 Labels for the transitions of the Data Petri-Net N.Sender

6.3 DETECTION OF DATA FLOW ANOMALIES FOR SENDER

In this section, the method DETANOM is applied on N.Sender for detecting any data flow anomaly. To illustrate the general cases, symbolic execution is adopted. In particular, variable ack3 is assigned with a symbolic value a when transition t8 is fired. Variables datamsg1 and seqsender1 are assigned respectively with symbolic values d and s when transition t1 is fired. Because no undef-use data flow anomaly is found in N.Sender and no error is found in N.Sender, the list of

anomalies is omitted. The marking graph *G.Sender* for N.Sender is shown through Figure 6.3 and Tables 6.2-6.3.

The columns in Tables 6.2-6.3 have the following meanings. In Table 6.2, for showing the labels of the transitions in G.Sender, the column "TinG" lists transitions in G.Sender. The column "TinN" lists the corresponding transition in N.Sender. The column "label$^{\mu}$" lists the four clauses of every label: gate-clause, event-clause, do-clause and when-clause. In Table 6.3, for showing the markings in G.Sender, column "places" lists the places containing tokens under the corresponding marking. The other columns list the values of the variables, where a blank indicates that the variable is currently undefined.

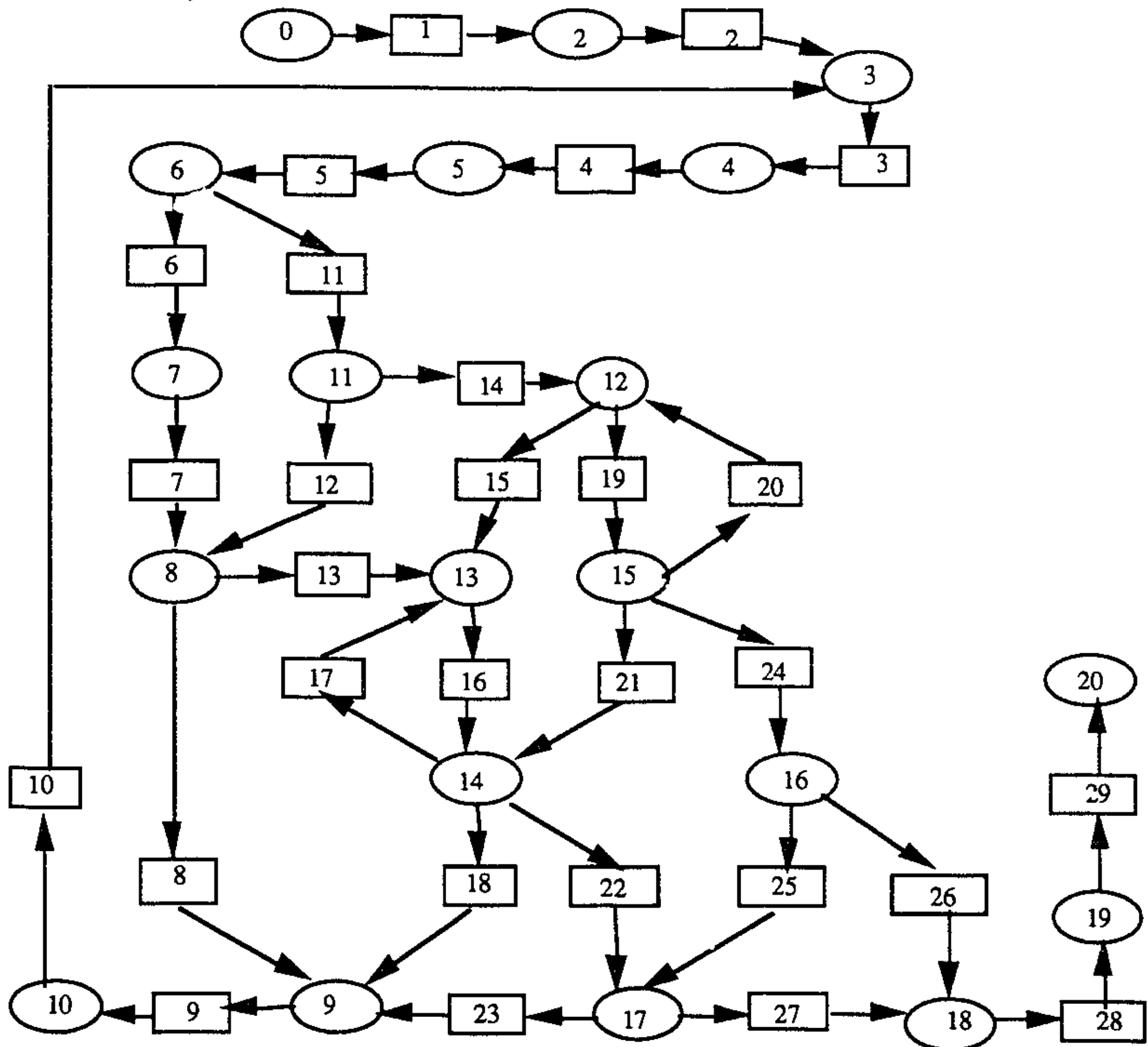


Figure 6.3 The marking graph G.Sender for the Data Petri-Net N.Sender

TinG	TinN	label$^{\mu}$
t1	t1	sendgate, ? datmsg1 : datamsg, Ø, Ø
t2	t2	Ø, Ø, (datmsg2, seqsender2) := (datmsg1, seqsender1), Ø
t3	t3	Ø, Ø, Ø, Ø
t4	t4	Ø, Ø, Ø, Ø
t5	t5	tm, ! T_start, Ø, Ø
t6	t15	τ, Ø, Ø, Ø
t7	t6	sendpdu, ! makepdu(datmsg2,seqsender2), Ø, Ø
t8	t11	tm, ! T_expired, Ø, Ø
t9	t19	Ø, Ø, ok2 := false, Ø
t10	t23	Ø, Ø, (datmsg2, seqsender2, ok2) := (datmsg2,seqsender2,und), not(ok2)
t11	t6	sendpdu, ! makepdu(datmsg2,seqsender2), Ø, Ø
t12	t15	τ, Ø, Ø, Ø
t13	t7	Ø, Ø, seqsender3 :=seqsender2, Ø
t14	t7	Ø, Ø, seqsender3 :=seqsender2, Ø
t15	t15	τ, Ø, Ø, Ø
t16	t8	receiveack, ? ack3 : message, Ø, Ø
t17	t10	Ø, Ø, (seqsender3, ack3) := (seqsender3, und), is-not-equal(sequence(ack3), seqsender3)
t18	t12	tm, ! T_expired, (seqsender3, ack3) :=(und, und), Ø
t19	t8	receiveack, ? ack3 : message, Ø, Ø
t20	t10	Ø, Ø, (seqsender3, ack3) := (seqsender3, und), is-not-equal(sequence(ack3), seqsender3)
t21	t15	τ, Ø, Ø, Ø
t22	t9	τ, Ø, (seqsender3, ack3) := (und, und), is-equal(sequence(ack3),seqsender3)
t23	t13	tm, ! T_expired, (seqsender3, ack3) :=(und, und), Ø
t24	t9	τ, Ø, (seqsender3, ack3) := (und, und), is-equal(sequence(ack3),seqsender3)
t25	t15	τ, Ø, Ø, Ø
t26	t16	tm, ! T_reset, Ø, Ø
t27	t17	tm, ! T_reset, Ø, Ø
t28	t20	Ø, Ø, ok2 := true, Ø
t29	t21	∂, Ø, Ø, ok2

Table 6.2 The label function for the marking graph G.Sender

μ	places	ack3	datmsg1	datmsg2	ok2	seqsender1	seqsender2	seqsender3
μ0	p1							
μ2	p2		d			s		
μ3	p3		d	d		s	s	
μ4	p4, p5		d	d		s	s	
μ5	p5, p6		d	d		s	s	
μ6	p7, p8		d	d		s	s	
μ7	p13, p8		d	d		s	s	
μ8	p13, p9		d	d		s	s	
μ9	p14, p15		d	d		s	s	
μ10	p18		d	d	false	s	s	
μ11	p7, p9		d	d		s	s	
μ12	p7, p10		d	d		s	s	s
μ13	p13, p10		d	d		s	s	s
μ14	p13, p11	a	d	d		s	s	s
μ15	p7, p11	a	d	d		s	s	s
μ16	p7, p12		d	d		s	s	
μ17	p13, p12		d	d		s	s	
μ18	p16, p17		d	d		s	s	
μ19	p18		d	d	true	s	s	
μ20	p19							

Table 6.3 The markings for the marking graph G.Sender

6.4 GENERATION OF TEST SEQUENCES FOR SENDER

In this section, the method GENTEST is applied on G.Sender with the criterion "all-defs". According to this criterion, all parameters of N.Sender are involved. The du-paths generated are listed in Table 6.4, where the column "du-no." lists the reference numbers of du-paths. Finally, Table 6.5 presents the complete paths and test sequences covering all definitions in Sender, where variables are recovered with their original names.

par.	def. at	used at	du-paths	du-no.
ack3	t16, t19	t20,t22,t24	t16-μ14-t22, t19-μ15-t20	1, 2
datamsg1	t1	t2	t1-μ2-t2	3
datamsg2	t2, t10	t7,t10 t11	t2-μ3-t3-μ4-t4-μ5-t5-μ6-t6-μ7-t7, t10-μ3-t3-μ4-t4-μ5-t5-μ6-t6-μ7-t7	4, 5
seqsender1	t1	t2	t1-μ2-t2	6
seqsender2	t2, t10	t7,t10,t11, t13,t14,t17	t2-μ3-t3-μ4-t4-μ5-t5-μ6-t6-μ7-t7, t10-μ3-t3-μ4-t4-μ5-t5-μ6-t6-μ7-t7	7, 8
seqsender3	t13,t14, t17,t20	t20,t22,t24	t13-μ13-t16-μ14-t22, t14-μ12-t19-μ15-t20, t17-μ13-t16-μ14-t22, t20-μ12-t19-μ15-t20	9, 10, 11, 12
ok2	t9,t28	t10,t29	t9-μ10-t10, t28-μ19-t29	13, 14
T_start	μ0	t5	μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5	15
T_reset	μ0	t26,t27	μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-μ6-t11-μ11-t14- -μ12-t19-μ15-t24--μ16-t26	16
T_expired	μ0	t8,t18,t23	μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-μ6-t6-μ7-t7-μ8-t8	17
true	μ0	t28	μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-μ6-t11-μ11-t14- -μ12-t19-μ15-t24-μ16-t26-μ18-t28	18
false	μ0	t9	μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-μ6-t6-μ7-t7-μ8-t8-μ9-t9	19

Table 6.4 The du-paths covering all definitions in G.Sender

complete paths	test sequences	du-no.

μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-
-μ6-t11-μ11-t14-μ12-t19-μ15-
-t20-μ12-t19-μ15-t20-μ12-t19-
-μ15-t24-μ16-t26-μ18-t28-μ19-
-t29-μ20;
	sendgate ? datmsg: datamsg ? seqsender: Bit; tm ! T_start;
	sendpdu ! makepdu(datmsg, seqsender);
	receiveack ? ack : message [is-not-equal(sequence(ack), seqsender)];
	receiveack ? ack : message [is-not-equal(sequence(ack), seqsender)];
	receiveack ? ack:message [is-equal(sequence(ack), seqsender)];
	tm ! T_reset; exit
		2, 3,
		6, 10,
		12,14,
		15

μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-
-μ6-t11-μ11-t14-μ12-t19-μ15-
-t24-μ16-t26-μ18-t28-μ19-t29-
-μ20;
	sendgate ? datmsg: datamsg ? seqsender: Bit; tm ! T_start;
	sendpdu ! makepdu(datmsg, seqsender);
	receiveack ? ack: message [is-equal(sequence(ack), seqsender)];
	tm ! T_reset; exit
		16,18

Table 6.5 The complete paths and test sequences covering all definitions in G.Sender (to be con't)

complete paths	test sequences	du-no.

μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-
-μ6-t6-μ7-t7-μ8-t8-μ9-t9-μ10-
-t10-μ3-t3-μ4-t4-μ5-t5-μ6-t6-
-μ7-t7-μ8-t13-μ13-t16-μ14-t22-
-μ17-t27-μ18-t28-μ19-t29-μ20;
	sendgate ? datmsg : datamsg ? seqsender : Bit; tm ! T_start;
	sendpdu ! makepdu(datmsg, seqsender); tm ! T_expired;
	tm ! T_start; sendpdu ! makepdu(datmsg, seqsender);
	receiveack ? ack : message [is-equal(sequence(ack), seqsender)];
	tm ! T_reset; exit
		1, 4,
		5, 7,
		8, 9,
		13,17,
		19

μ0-t1-μ2-t2-μ3-t3-μ4-t4-μ5-t5-
-μ6-t6-μ7-t7-μ8-t13-μ13-t16-
-μ14-t17-μ13-t16-μ14-t22-μ17-
-t27-μ18-t28-μ19-t29-μ20.
	sendgate ? datmsg: datamsg ? seqsender: Bit; tm ! T_start;
	sendpdu ! makepdu(datmsg, seqsender);
	receiveack ? ack : message [is-not-equal(sequence(ack),seqsender)];
	receiveack ? ack : message [is-equal(sequence(ack),seqsender)];
	tm ! T_reset; exit
		11

Table 6.5 The complete paths and test sequences covering all definitions in G.Sender (con't)

Chapter 7
SUMMARY AND FUTURE RESEARCH

In this thesis, we report our research on methods for the validation of distributed systems. We propose an extended Petri-net model called Data Petri-Net (DPN), focusing its capabilities for the handling of three data-related problems: abstract data types, data manipulation (especially synchronization) and data flow analysis. On the basis of data flow analysis, we develop a method for detecting data flow anomalies, such as undef-use, def-def and def-undef anomalies, and propose three families of coverage criteria: all-defs, all-uses and all-du-paths. Test sequences selected according to these criteria aim at checking whether an implementation under test possesses the desired associations among the values of the input and output parameters. We then develop a method for generating selective test sequences based on these criteria.

We also apply these DPN-based methods to the validation of LOTOS specifications. In the literature, there exist four Petri-net-based models for representing and validating LOTOS specifications. These models fall in two categories: 1) Those representing basic LOTOS, i.e., no data operations are involved. This includes the Place/Transition-net of Barbeau, et al. [BAR90] and the Galileo net of Marchena, et al. [MAR89]. 2) Those representing full LOTOS. This includes the Communicating System of Petri-Nets of Cheung, et al. [CHE88, 92] and the Network of Garavel, et al. [GAR90]. Although data and data operations are considered, no data flow analysis techniques have been applied in both models. Our application of data flow analysis to the detection of anomalies and generation of selective test sequences for LOTOS specifications seems to be the first attempt of this kind.

At present, our approach has several shortcomings which leave the door open for further research. Three of them are listed below:

I. <u>To alleviate the state explosion problem in the generation of the marking graph</u>

In our approach, *all* transitions are added to the marking graph if they are firable. It suffers from the traditional state explosion problem. Some strategies should be designed to reduce the number of transitions added so that the growth of a making graph can be limited and the explosion problem can be partially solved. Following are two possible approaches.

a) Apply selection criteria for limiting the firing of transitions. Generally, methods for generating selective test sequences fall in two categories: 1) Those that generate *all* test sequences and then select some sequences according a specific criterion. This requires the generation of the *entire* marking graph. Our method belongs to this category. 2) Those that generate only the selective test sequences according to a specific criterion. This requires the generation of a *partial* marking graph. The method proposed in [CHE92] (as reviewed in Chapter 3) belongs to this category. However, this method does not consider data and its partial marking graph can not be used for detecting data flow anomalies. For future research, it can be enhanced with data handling and can be applied to DPNs.

b) Design strategies for including fired transitions to a marking graph. According to the contents of their gate-clauses, transitions in a DPN fall in four categories: 1) Those whose gate-clauses are external gate-names. They represent external actions. 2) Those whose gate-clauses are "τ"s. They represent internal actions. 3) Those whose gate-clauses are "δ"s. They represents successful terminations. 4) Those whose gate-clauses are "Ø"s. They represent null actions. It is not necessary to include *all* these types of transitions even they are fired. Some strategies should be designed to ensure that *only* certain types of transitions can be added to the marking graph. For example, as in Garavel's method [GAR90], we may add *only* the first three types of transitions, i.e., ones whose gate-clauses are not empty. Garavel's method is based on the ε-closure algorithm developed originally for non-deterministic automata. However, the paper [GAR90] does not provide enough details to enable us to try their approach in this thesis. The development of similar strategies is left for future works.

II. To find other criteria for test sequence selection

In the traditional software testing methodology, such as Rapps and Weyuker's approach reviewed in Chapter 2, the use of a variable is further classified as *p-use* (a use within predicates) and c-use (a use outside predicates). In sequential software, covering all p-uses is sufficient for covering all branches because all branchings are generated by p-uses. However, in a distributed system, there exist operations other then p-use which generate branchings. Consider a system specified in LOTOS. There are four kinds of branching operations: choice, disabling, guarding and parallel operations. Consider the following four behaviour expressions: 1) "a1; B1 [] a2; B2"; 2) "B1 [> a2; B2"; 3) "[Pt] -> B"; and 4) "B1 | S | a2; B2". In expression 1, the executions of actions a1 and a2 will lead to two different branches. In expression 2, an additional branch will be created whenever action a2 is executed. In expression 3, a branch leading to B will be created when the condition specified by Pt is satisfied. And, in expression 4, every action of B1 and B2 will probably leads to an additional branch. For future research, the analysis of these kinds of operations may lead to some operation coverage, which play a similar role in distributed systems as the p-use coverage in sequential software.

III. To enhance net representations for recursive LOTOS process instantiations

Like other net representations reviewed in Chapter 3, our transformation rules are applicable only to a subset of LOTOS in which a recursive process instantiation may not be allowed in the left sub-expression of an enabling/disabling operator or the operands of a parallel operator. The enhancement of the Petri-net representation for recursions is still an open problem. Besides, the optimization for the transformed DPNs also needs more efforts.

REFERENCES

[BAR90] M. Barbeau and G.v. Bochmann, "Deriving analyzable Petri-nets from LOTOS specifications", Technical Report Publication #707, Dept. of Computer Science, Universite de Montreal, (1990).

[BER88] P. Berlinguette and D. Gueraichi, "The Alternating Bit Protocol in LOTOS: 'textual' and 'graphical' representations", Technical Report TR-88-25, Dept. of Computer Science, Univ. of Ottawa, (1988).

[BIE89] J. M. Bieman and J. L. Schultz, "Estimating the number of test cases required to satisfy the all-du-paths testing criterion", Software Engineering Notes, Vol. 14, No. 8 - Proc. ACM SIGSOFT'89, edited by R. A. Kemmerer, (1989), pp.179-186.

[BIL88] J. Billington, G. Wheeler and M. Wilbur-ham, "PROTEAN: a high-level Petri Net tool for the specification and verification of communication protocols", IEEE Trans. on Software Engineering, Vol. SE-14, No. 3, (1988), pp. 301-316.

[BOL89] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS", The Formal Description Technique LOTOS, edited by P.H.J. van Eijk, C.A. Vissers and M. Diaz, (1989), pp. 23-73.

[BRI89] E. Brinksma, "A theory for the derivation of tests", The Formal Description Techniques LOTOS, edited by P.H.J. van Eijk, C.A. Vissers and M. Diaz, (1989), pp.235-247.

[BUD87] S. Budkowski and P.Dembinski, "An introduction to Estelle: a specificatio language for distributed systems", Computer Networks and ISDN Systems 14 (1987), pp. 3-23.

[CCI87a] CCITT, "Specification and description language (SDL)", Recommendation Z.100, (Oct. 1987).

[CCI87b] CCITT, "Specification and description language (SDL)", Annex D to REC. Z.100, (Dec. 1987).

[CHE88] T.Y. Cheung and Y. Zhu, "A Petri-net-based method for specifying distributed systems and deriving executable graphical LOTOS and ESTELLE", Technical Report TR-88-04, Dept. of Computer Science, Univ. of Ottawa, (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", Formal Description Techniques, II - Proc. FORTE'89, edited by S.T. Voung, (1989), pp. 31-36.

[CHE90] T. Y. Cheung and Y. C. Ye, "An executor for graphical LOTOS", Formal Description Techniques, III - Proc. FORTE'90, edited by J. Quemada, J. Manas and E. Vazquez, pp.547-550.

[CHE91] T. Y. Cheung, Y. Wu and X. Ye, "Generating test sequences and their degrees of indeterminism for protocols", Proc. 11th IFIP International Symp. on Protocol Specification, Testing and Verification, edited by B. Johsson, J. Parrow and B. Pehrson, Stockholm, (1991), pp. 278-293.

[CHE92] T.Y. Cheung and S. Ren, "Operational coverage and selective test sequences obtained from LOTOS specifications", Technical Report TR-92-07, Dept. of Computer Science, Univ. of Ottawa, (1992).

[CLA89] L.A. Clarke, A. Podgurski, D. J. Richardson and S. J. Zeil, "A formal evaluation of data flow path selection criteria", IEEE Trans. on Software Engineering Vol. SE-15, No. 11, (1989), pp.1318-1332.

[DIA82] M. Diaz, "Modeling and analysis of communication and cooperation protocol using Petri Net based models", Computer Networks 6, (1982), pp. 419-444.

[DIA87] M. Diaz, "Petri net based models in the specification and verification of protocol", Lecture Notes in Computer Science 255, Springer-Verlag, (1987), pp.135-170.

[EHR85] H. Ehrig and B. Mahr, Fundamentals of Algebraic Specification 1: equations and initial semantics, Springer-Verlag, (1985).

[FAC91] M. Faci, L. Logrippo and B. Stepien, "Formal specification of telephone systems in LOTOS: the constraint-oriented style approach", Computer Networks and ISDN Systems 21 (1991), pp. 53-67.

[FOS76] L. D. Fosdick and L.J. Osterweil, "Data flow analysis in software reliability", ACM Computer Surveys, Vol. 8, No. 3, (1976), pp. 305-330.

[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.

[GUE89] D. Gueraichi and L. Logrippo, "Derivation of test cases for LAPB from a LOTOS specification", Formal Description Techniques, II - Proc. FORTE'89, edited by S.T. Voung, (1989), pp. 361-374.

[HOA85] C.A.R. Hoare, Communicating sequential processes, Prentice-hall Intl., (1985).

[ISO87] ISO/TC97/SC21/WG1/DIS9074 Estelle - a formal description technique based on an extended state transition model, (1987).

[ISO88] ISO8807, "Information processing systems - open systems interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behavior", ISO publication, (1988).

[ISO89] ISO/IEN JTC1/SC21 N3253, "G-LOTOS: a graphical syntax for LOTOS", (1989).

[KEL76] R. Keller, "Formal verification of parallel programs", Communications of the ACM, Vol. 19, No. 7, (1976), pp. 371-384.

[LAS83] J.W. Laski and B. Korel, "A data flow oriented program testing strategy", IEEE Trans. on Software Engineering Vol. SE-9, No. 3, (1983), pp. 347-354.

[MAR89] S. Marchena and G. Leon, "Transformation from LOTOS to Galileo nets", Formal Description Techniques, I - FORTE'88, edited by K.J. Turner, (1989), pp. 217-230.

[MIL80] R. Milner, "A calculus of communicating systems", Lecture Notes in Computer Science 92, Springer-Verlag, (1980).

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

[NTA84] S.C. Ntafos, "On required element testing," IEEE Trans. on Software Engineering, Vol. SE-10, No. 6, (1984), pp. 795-803.

[PIT90] D. H. Pitt and D. Freestone, “The derivation of conformance tests from LOTOS specification”, IEEE Trans. on Software Engineering, Vol. 16, No. 12, (1990), pp. 1337-1343.

[RAP85] S. Rapps and E. Weyuker, "Selecting software test data using data flow information", IEEE Trans. on Software Engineering, Vol. SE-11, No. 4, (1985), pp. 367-375.

[SAN86] C.Sanchez, "Galileo: model, language, and tolls", Electrical Communication, vol. 60, no.3/4, (1986), pp.216-224.

[SID89] D.P. Sidhu, T. Leung, "Formal methods for protocol testing: a detailed study", IEEE Trans. on 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, “Test sequence selection based on static data flow analysis”, Computer Communications, Vol. 10, No. 5. (1987), pp. 234-242.

[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.

[YAN88] B. Yang, “A test selection strategy based on input-output relation analysis”, Master Thesis, Univ. of Ottawa, (1988).

APPENDIX:
LOTOS SPECIFICATION OF SENDER

This appendix contains the LOTOS specification of the process Sender and three related processes: safemesser, messer and timer.

```
process  Sender [sendgate,sendpdu,receiveack] : exit :=
   library
     Bitstring, Boolean, Bit
   endlib
   type data is BitString
     renamedby
       sortnames datamsg for BitString
   endtype
   type mess is data
    sorts message
    opns
     sequence        : message         -> Bit
     makepdu         : datamsg,  Bit   -> message
     makeack         : Bit             -> message
     is-equal        : Bit, Bit        -> Bool
     is-not-equal    : Bit, Bit        -> Bool
     datamessage     : message         -> datamsg
     succ            : Bit             -> Bit
    eqns
     forall  n: Bit, msg : datamsg, m : Bit
     ofsort Bit
```

```
    sequence(makeack(n))                = n;
    sequence(makepdu(msg,n))            = n;
    succ(succ(n))                       = n;
  ofsort datamsg
    datamessage(makepdu(msg,n))         = msg;
  ofsort Bool
    is-equal (n, m)                     = n eq m;
    is-not-equal (n, m)                 = n ne m;
endtype
type time is
 sorts    tim
 opns
   T_start       :   -> tim
   T_expired   :   -> tim
   T_reset       :   -> tim
endtype


sendgate  ? datmsg : datamsg ? seqsender : Bit;
safemesser[sendpdu,receiveack] (datmsg, seqsender)
where
process  safemesser[sendpdu,receiveack](datmsg : datamsg, seqsender :  Bit)  :  exit  :=
 hide     tm     in
 ( (       tm  !  T_start;
       sendpdu  !   makepdu(datmsg, seqsender);
       ( (  messer[receiveack] (seqsender)
            >>
           tm  !  T_reset ;  exit(true)   )
```

```
        [>
        (tm ! T_expired ; exit(false) )
    | [tm] |
    timer[tm] )
  >> accept ok : Bool in
  ( [ ok ]          -> exit
    []
    [not (ok) ]     -> safemesser[sendpdu,receiveack](datmsg,seqsender) )
    where
        process  messer[receiveack](seqsender : Bit) : exit :=
        receiveack ? ack : message ;
        ([is-equal (sequence(ack), seqsender) ] -> exit
         [ ]
         [is-not-equal(sequence(ack), seqsender) ] -> messer[receiveack](seqsender) )
        endproc /*messer*/
        process  timer[tm] : exit (Bool) :=
        tm ! T_start ;
        (   i ; tm ! T_expired ; exit(false)
          [>
            ( tm ! T_reset ; exit(true) ) )
        endproc /*timer*/
  endproc /*safemesser*/
endproc /*sender*/
```