Finding Executable Paths

In Protocol Conformance Testing

by

Yuemin Wang

B.Sc, Jilin University, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the School
of
Computing Science

SIMON FRASER UNIVERSITY
April 1990

Approval

Name: Yuemin Wang

Degree: Master of Science

Title of Thesis: Finding Executable Paths in Protocol Conformance Testing

Examining Committee:

Dr. Lou J. Hafer
Chairman

Dr. Tiko Kameda
Senior Supervisor

Dr. Stella Atkins
Supervisor

Dr. Slawomir Pilarski
External Examiner

April 9, 1990
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Finding Executable Paths in Protocol Conformance Testing.

Author: _____
(signature)

Yuemin Wang
(name)

April 9, 1990
(date)

ABSTRACT

Communication protocol conformance testing aims at demonstrating the adherence of a protocol implementation to the protocol specification which is assumed to be correct. One of the most important issues in protocol testing is the generation of a small set of test sequences with large fault coverage. Since even the simplest protocol may require a very large number of test sequences to assure almost complete fault coverage, it is challenging to solve this problem. To derive efficient test sequences, a very fundamental and crucial problem is the **executable path (EP)** problem which consists of EP identification and EP selection. Because of its complexity, this problem has remained open so far.

This thesis is concerned with protocol test sequence generation. Particularly, the EP problem is studied in detail. Although this problem is *NP*-complete in general, we attempt to develop some efficient algorithms to solve it under certain reasonable restrictions. We first establish a formal graph model based on the extended finite state machine (EFSM) and the normal form specification (NFS) in Estelle to describe both control and data portions of the communication protocol. We then precisely define the EP problem and discuss its complexity. Two basic algorithms for the EP identification are developed and their complexity is analyzed. We also investigate the EP selection problem and propose several test path selection criteria. Finally, we apply our methods to a real communication protocol.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude and appreciation to Professor Tiko Kameda, my senior supervisor, for his invaluable guidance, support and encouragement during the past year for preparation of this thesis. In fact, our weekly discussions were something I looked forward to. He has been a constant source of inspiration, without which this thesis would not have been possible.

I would also like to thank other members of my examining committee, Dr. Stella Atkins, Dr. Slawomir Pilarski, and Dr. Lou J.Hafer for reading this thesis carefully and making thoughtful suggestions.

I am thankful to my fellow grad students, particularly to Sanjeev Mahanjan who gave me many quick and excellent suggestions and to Yandong Cai who helped me with troff macros.

I am grateful to Simon Fraser University and the School of Computing Science for the scholarship and financial support.

I owe sincere thanks to my dear parents and my elder sisters for their unfailing support and education.

Finally, my special thanks to my wife, Xiao-Yun Shao, for her understanding, patience and support throughout this entire effort.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A **communication protocol** is a set of precise rules governing the possible interactions among the components in a communication system. The **specification** of a protocol is, in general, given as a detailed document describing the interfaces and mechanisms of the protocol. An **implementation** of a protocol is a running version which realizes the various functions defined in the specification. Obviously, a protocol specification may lead to different implementations. Incorrect or incompatible implementations of a logically correct and completely specified protocol specification may not be able to communicate with each other. In order to make sure that the implementations conform to the specification and work reliably, some kind of certification is essential. Since the state of the art of the program verification is far from providing practical tools to verify large concurrent software such as a protocol implementation, **protocol conformance testing** has been widely advocated for ensuring that protocol implementations are consistent with the specifications [BoS83, ISO87a].

Protocol conformance testing, or protocol testing for short, aims at demonstrating the adherence of a protocol implementation, called **implementation under test (IUT)**, to the protocol specification that it implements. Typically, the source listing of an implementation cannot be assumed to be available and it is tested as a black box locally or remotely, based on different **test architectures** [Ray87]. The testing is carried out by applying a group of inputs to the implementation and verifying that the corresponding outputs are what is expected. These input sequences are called **test sequences** and the process of automatically deriving efficient

test sequences from the protocol specification is called **test sequence generation**. Since even the simplest protocols may require an astronomical number of different test sequences, the test sequence generation problem is combinatorially challenging [ADU88].

To generate protocol test sequences efficiently, a very fundamental and crucial issue which has to be solved is the **executable path problem**. Intuitively, an executable path is a sequence of states and transitions which the implementation can go through. Basically, the executable path problem consists of two parts: **executable path identification** and **executable path selection**; the former is concerned with finding a set of executable or feasible paths from the specification and the latter is concerned with selecting a subset of the executable paths to generate test sequences so that we can make the implementation take these paths during the testing. Because of its mathematical complexity, the executable path problem has remained as an open problem so far [Ura87].

The major topic of this thesis is the protocol test sequence generation in general. Particularly, the executable path problem is investigated in detail.

This thesis is organized as follows. In chapter 2, we give a brief survey of the previous research work on protocol testing and present some related background information. In chapter 3, an acyclic graph based on an extended finite state machine model is introduced and the executable path problem is formally defined in the context of this graph. Chapter 4 is devoted to the executable path identification problem and two basic algorithms and their variations are proposed. In chapter 5, the executable path selection problem is discussed and some new criteria for test path selection and test sequence generation are suggested. In chapter 6, we apply our algorithms and criteria to a real communication protocol. Finally, in chapter 7, we summarize our major contributions and conclude this thesis.

CHAPTER 2

PROTOCOL CONFORMANCE TESTING: AN OVERVIEW

A substantial amount of research has been devoted to protocol testing. Most previous work has centered around the topics of protocol test sequence generation, protocol test architecture and formal description techniques (FDTs). In this chapter, I give a brief survey of such work and provide the related background information.

2.1. The Fundamentals of Communications Protocols

The basic goal of computer networks is to provide interconnection and communication among the entities (e.g., processes) in different systems (e.g., computers). As computer networks have been growing more and more complex, an ad hoc or special-purpose approach to network software development is too costly to be acceptable; this is particularly true when communication is desired among heterogeneous systems. To reduce the cost, the only alternative is to develop a common set of conventions or protocols. International Organization for Standardization (ISO) took up this challenge and established the **Open System Interconnection (OSI) reference model** which is a framework for defining protocol standards to make the interconnection of heterogeneous computer systems possible [ISO84, Zim80]. Since OSI reference model has achieved nearly universal acceptance, the discussions in this thesis are based on this model.

The most important concept in OSI reference model is **layering**. The whole network is organized as a series of layers and layer (N) provides a set of capabilities or services to layer

(N+1) by enhancing those performed by layers 1~(N-1). Within a system, layer (N) is constituted by one or more protocol entities called N-entities which are capable of sending and receiving information. Externally, an N-entity interacts with other entities of the adjacent layers within the same system via the invocation of **abstract service primitives (ASP)** which, in an abstract manner, describe the operations and parameter exchanges at the layer interface. Meanwhile, an N-entity interacts with another N-entity, called a **peer entity**, in a remote system by exchanging messages called **protocol data units (PDU)**. The ASPs and PDUs are known as **external interactions** which define the external behavior of a protocol entity and are essential to protocol testing.

2.2. Formal Description Techniques

To avoid imprecision and ambiguity, **formal description techniques (FDTs)** are considered to be important tools for the design, verification, implementation and testing of communication protocols. A variety of general formalisms such as the state transition model, programming languages, temporal logic and some reasonable combinations thereof, can be used to describe a protocol. Presently, a number of FDTs have been or are being developed [BoS83, BoB87, BuD87].

A protocol specification should describe the external interactions and internally initiated operations (e.g., timeouts) of a protocol entity. Roughly, a protocol specification can be broken up into two portions: **control and data**. The control portion is concerned with the various states in which the protocol entity can be and the state transitions; the data portion deals with the values and their variations of parameter fields of external interactions.

The control portion of a protocol entity can be easily modeled as a **finite state machine (FSM)** [SaD88]. However, it is usually impractical to model the data portion of a protocol by a FSM; for example, to model a protocol using sequence numbers, there must be different states to represent every possible sequence number, which results in the *state space explosion* problem [Hai83].

The **extended finite state machine (EFSM)** model [Boc83] attempts to combine the advantages of state transition technique and programming language technique. It is called *extended* since variables (called context variables) are introduced to the basic FSM model for describing the data portion of a protocol. The EFSM is considered to be the most promising to model most practical protocols.

2.2.1. Estelle

Based on the EFSM model, a FDT called **Estelle** has been developed by ISO [ISO87b]. In Estelle, a protocol entity may be specified in terms of possibly more than one module. The behavior of each module is described by state transitions and the context variables of the module. A state transition from one state to another state may depend on some predicates on the context variables and input interactions. Associated with each transition is a sequence of operations to be executed as part of the transition. To specify these operations, Pascal executable statements can be used. Some procedure calls and the right-hand sides of some assignment statements may be undefined to leave the interpretation to the implementor. A concrete example is given in Figure 2-1.

```
FROM idle
TO wait_for_T_connect_resp
WHEN cr (source_ref, dest_ref, variable_part) /* 'cr' stands for Connection Request*/
PROVIDED (cr.variable_part.qts.req = ok)
```

```
BEGIN
    remote_ref := cr.source_ref;
    if (cr.variable_part.TPDU_size) <> undefined
    then
        TPDU_size := cr.variable_part.TPDU_size;
    else
        TPDU_size := 128;
    remote_add := cr.variable_part.calling_T_add;
    called_add := ... ; /* implementation dependent */
    calling_add := ... ;
    output T_connect_ind (called_add, calling_add, TPDU_size, ...);
END;
```

Figure 2-1: A transition in Estelle

In general, a protocol specification in Estelle may still contain certain constructs which make test sequence generation complicated.

2.3. Normal Form Specification (NFS) of Estelle

A protocol specification should be precise, well-defined, detailed and easy-to-analyze so that any implementation based on it will work with any other implementation. For testing purposes, the analysis of the dynamic behavior of a protocol entity based on the specification is very important.

A protocol specification in Estelle may contain many modules and there may exist some complex interactions among these modules. The major complications of any protocol specification in Estelle result from inter-module interactions, multiple control paths and local procedure/function. The inter-module interactions make the analysis and description of the behavior of a protocol entity complicated. Since these interactions are internal and cannot be observed by the tester, they are irrelevant to protocol testing.

In a protocol specification in Estelle, any Pascal statement can be used within the operation part of a transition. The conditional IF and CASE statements and iteration statements can constitute multiple control paths within a transition and the implementation may take different control paths when traversing one transition. Therefore, the execution of the subsequent transitions may depend not only on which transitions have been fired previously but also on which control paths have been taken within these transitions. Obviously, the multiple control paths within a module make the analysis of protocol behavior difficult.

The procedure/function calls can be used in a protocol specification in Estelle to achieve abstraction. But for protocol testing, it is desired to unfold these abstractions if they are defined and to make every specification detail directly available to the test sequence generation algorithm.

Based on the above discussions, a group of transformations to an Estelle specification are proposed in [Sar84, SaB86, SBG87]. The basic ideas behind these transformations are:

(1) combining modules and eliminating internal interactions by textual substitutions;

(2) creating a new transition for every distinct path in the operation part of an original transition and modifying the corresponding condition predicates to reflect the conditions imposed for taking these paths;

(3) unfolding the local procedure/function calls by symbolically executing the local procedure/function bodies if they are defined.

After these transformations, a single-module and single-path specification, called **Normal Form Specification (NFS)**, can be derived. Informally, a NFS describes the behavior of a protocol entity in terms of a group of **Normal Form Transitions (NFT)** [SaB86]. Each NFT consists of the following five components :

(1) an optional WHEN clause specifying the external input interactions of this transition. If this clause is absent, the transition is said to be *spontaneous*;

(2) a FROM clause indicating the source state of this transition;

(3) a TO clause indicating the target state of this transition;

(4) an optional PROVIDED clause specifying an enabling predicate which must be true for the transition to take place;

(5) a BEGIN-END clause block specifying a single path composed of assignment statements, undefined procedure/function calls, and possibly some output statements defining external output interactions.

The NFS of the example given in Figure 2-1 is shown in Figure 2-2.

```
FROM idle   /* first NFT */
TO wait_for_T_connect_resp
WHEN cr (source_ref, dest_ref, variable_part)
PROVIDED (cr.variable_part.qts.req = ok) /\ (cr.variable_part.TPDU_size <>
undefined)
BEGIN
    remote_ref := cr.source_ref;
    TPDU_size := cr.variable_part.max_TPDU_size;
    remote_add := cr.variable_part.calling_T_add;
    called_add := ... ; /* implementation dependent */
    calling_add := ... ;
    output T_connect_ind (called_add, calling_add, max_TPDU_size, ...);
END;


FROM idle   /* second NFT */
TO wait_for_T_connect_resp
WHEN cr (source_ref, dest_ref, variable_part)
PROVIDED (cr.variable_part.qts.req = ok) /\ (cr.variable_part.max_TPDU_size =
undefined)
BEGIN
    remote_ref := cr.source_ref;
    TPDU_size := 128;
    remote_add := cr.variable_part.calling_T_add;
    called_add := ... ; /* implementation dependent */
    calling_add := ... ;
```

```
        output T_connect_ind (called_add, calling_add, max_TPDU_size, ...);
END;
```

Figure 2-2: Two NFTs equivalent to the transition of Fig. 2-1.

2.4. PICS and PIXIT

In a typical protocol specification, many special features or options related to the specific implementation are left to be determined by the implementors so that certain abstraction can be achieved. Nevertheless, such implementation-related parameters might be important to protocol testing. For this purpose, ISO has defined two special documents to facilitate protocol testing [ISO87a].

The **Protocol Implementation Conformance Statement (PICS)** is a statement made by the implementor to state the capabilities and options which have been implemented, and any features which have been omitted. It is needed so that the implementation can be tested for conformance against relevant requirements, and against those requirements only.

In addition to the information provided by PICS, the tester might require further information to conduct testing. The **Protocol Implementation eXtra Information for Testing(PIXIT)** is for this purpose. PIXIT is a statement which may contain the following information:

(1) information needed by the tester in order to be able to run the appropriate test sequence on the specific system (e.g., addressing information);

(2) information already mentioned in the PICS and which needs to be made precise (e.g., a timer value which is declared as a parameter in the PICS should be specified in the PIXIT);

(3) information to help determine which capabilities stated in the PICS as being supported are testable and untestable;

(4) other administrative matters (e.g., the IUT identifier).

2.5. Test Architecture

The test architecture deals with the testing environment and configuration. Unlike ordinary program testing, protocol testing may be performed from a remote testing site or in distributed manner. Furthermore, since most networks are organized as a series of layers, the placement of testing modules in particular layers at the test and implementation sites according to some configuration criteria also gives rise to the problem of test architecture.

ISO has done pioneering work on protocol test architecture. [Ray87] proposes various test configurations for different environments and applications. The advantages and disadvantages of different architectures are also discussed in this paper. The major criteria by which to classify different test architectures depend on *where, what,* and *how* external interactions of the implementation under test (IUT) can be observed and controlled.

From the *where* point of view, the test architectures can be categorized as **local** or **external**, which indicates that testing is carried out within the implementation site (in-house testing) or in a real communications environment, respectively. The external approach can be further divided into: *distributed, coordinated* and *remote.*

From the *what* point of view, we have **single-layer, multi-layer** and **embedded-layer** testing. Single-layer methods are designed for testing a single layer without reference to the layers above it. Multi-layer methods are designed for testing a multi-layer IUT as a whole. Embedded methods are designed for testing a single layer within a multi-layer IUT, using the

knowledge of what protocols are implemented in the layers above the layer being tested.

From the *how* point of view, we may use a **lower tester (LT)** and an **upper tester (UT)** to control and observe the lower and upper boundary of an IUT, and **test coordination procedure (TCP)** to coordinate the UT and the LT.

A concrete test architecture is given in Figure 2-3. Currently, protocol testing architectures and methodology are still being refined and standardized by ISO [ISO87a].

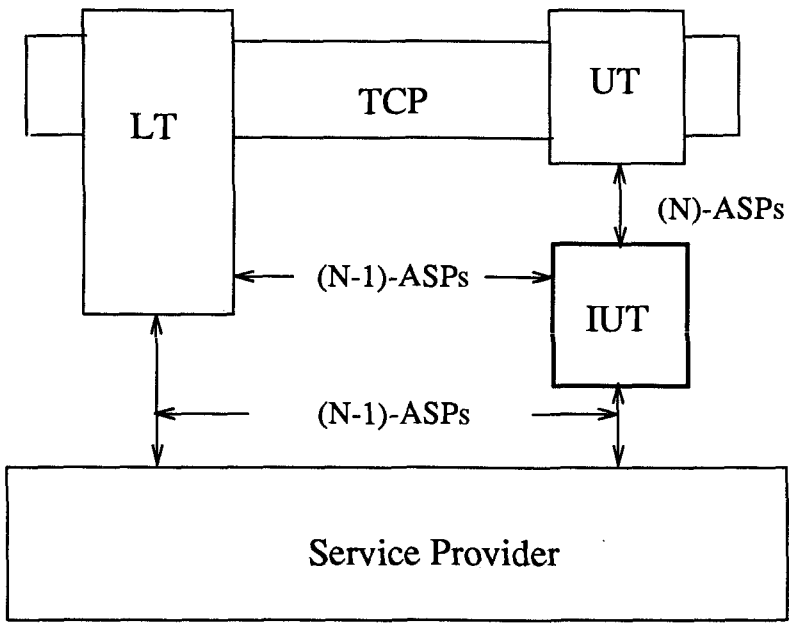


Figure 2-3: The distributed single-layer test architecture

The algorithms described in the following chapters are independent of any particular test architecture. We only assume that the tester can directly or indirectly observe and control the external interactions of an IUT. This assumption is valid for most test architectures.

2.6. Test Sequence Generation

Test sequence generation is a key step in protocol testing. A test sequence consists of input data used to exercise the implementation and the corresponding correct output responses. Since the complexity of most real protocols makes exhaustive testing both technically and economically impossible, the goal of test sequence generation is to derive a small set of tests from the protocol specification such that they have large fault coverage.

2.6.1. Test Sequence Generation Based on FSM Model

Much research work has been done on the test sequence generation based on the FSM model. The basic idea behind most existing testing techniques is *transition testing*, that is, putting the implementation at the source state of the tested transition, forcing it to undergo the transition and observing whether the outputs and the target state are correct. It is worth pointing out that the transition testing is complicated by the limited controllability by the external tester, which in most cases cannot directly place the implementation into a predetermined state, and by the limited observability by the external tester, which cannot directly observe the state of the implementation.

Four test sequence generation techniques (**T-, D-, W-, U-methods**) have been designed [Cho78, Gon70, NaT81, SaD88]. In a recent study, Sidiu and Leung investigate the efficiency and fault coverage of these techniques [SiL89]. In [ADU88] and [ShS89], an optimization technique for the test sequence generation based on U-method and the rural Chinese postman tour in graph theory is proposed to find a minimum-cost test sequence. However, because of the limitations of the FSM model, all these techniques can only be applied to the protocols with simple data portion or to the control portion of a protocol.

2.6.2. Test Sequence Generation Based on EFSM model

Test sequence generation becomes much more complicated and challenging when we attempt to test both control and data portions. The major complication results from the complex interactions between the data portion and the control portion. Since a state transition depends on the external interactions and the execution history of the previous state transitions and related operations, the idea of pure transition testing discussed in the last section may not work, simply because some transitions cannot be executed together. This is, in fact, the executable path problem mentioned earlier. Another complication lies in the parameter variation of the data portion. Choosing the effective testing data and variation is not straightforward when the executable path problem is considered. Based on the EFSM model, several test sequence generation schemes which take into account both control and data portion of a protocol have been proposed.

[SBG87] applies the idea of functional program testing [How80] to the generation of protocol test sequence. In this scheme, the formal specification of the protocol in Estelle is transformed into normal form specification which can be further decomposed and represented by the **control graph (CG)** and **dataflow graph (DFG)**. The CG and DFG aim at describing the control and data portions of a protocol, respectively. From the CG, subtours, which are paths starting and ending at the idle state, can be derived. From the DFG, dataflow functions which represent various real protocol functions can be obtained. For each dataflow function, a test sequence is designed by parameter variations and by simulating all of the executable subtours related to this dataflow function. However, the main disadvantage of this method is that it is quite complicated; especially, it is not clear how to mechanize the dataflow function decomposition. In this scheme, the executable path problem is not studied and it is assumed

that the executable subtours can, somehow, be found manually.

To improve the method described above, [Dat87] proposes a method in which the DFG is completely eliminated and only the condition part of each transition is considered. The basic idea behind this method is to execute all the transitions of the control part of a protocol at least once and to vary the corresponding parameter values of the data portion. The chief advantage of this method is that it is easier to understand and implement. In this scheme, the executable path problem is discussed only in the context of ISO class 2 transport protocol and no general algorithm is given.

The method in [Ura87] is based on the data flow analysis technique [FoO76] to generate a set of test sequences to cover all definition and usage pairs satisfying certain constraints given in [RaW81]. This method can be used to determine whether an implementation establishes the desired flow of data expressed in the given specification. The major drawback of this method is that its fault coverage is relatively limited and the executable path problem is totally ignored.

Another structural testing method which aims at testing the data portion of a protocol and improving the fault coverage is suggested in [UYP88]. This method is based on the identification of all inputs that influence each output from the point of view of syntactic structure of the specification. It is claimed that this method has a better fault coverage than the method in [Ura87]. However, like other purely structural test sequence generation methods, the test sequences derived by this scheme often contain non-executable paths since syntactic information is not sufficient to determine whether a particular path is executable or not.

Some research work has been devoted to the executable path problem. In [WaK88], a heuristic method of identifying the executable paths in the context of transport layer protocol

is proposed.

CHAPTER 3

MODELS, DEFINITIONS AND COMPLEXITY

In the previous chapters, we have given a brief description of a formal model EFSM and a formal protocol description technique called normal form specification (NFS) in Estelle, which is based on the EFSM model. We have also informally explained the concepts of protocol test sequence generation and the executable path problem. Such descriptions in natural language are adequate for presenting a general idea of these concepts, but when it comes to actually design algorithms, more precise definitions become essential.

In this chapter, we first precisely define the EFSM model and the NFS in Estelle. Based on these definitions, an acyclic graph which can be used to describe both the control and data portions of a protocol is proposed. The executable path problem and its complexity are also formally defined and studied in this chapter. Finally, the general assumptions that are used throughout this thesis are stated.

3.1. Formal Definitions of Models

Definition 3.1. An **Extended Finite State Machine**, or **EFSM** for short, is a 7-tuple:

$$\text{EFSM} = (S, I, O, V, A, C, T),$$

where

S is a finite set of states and one of them is called the *initial or idle state*;

I is a finite set of inputs;

O is a finite set of outputs;

V is a finite set of variables called *context variables*;

A is a finite set of actions or operations on inputs and context variables;

C is a finite set of predicates on inputs and context variables;

T is a finite set of state transitions and each state transition $t \in T$ is a 5-tuple defined below :

$$t = <q_t^s,\ q_t^t,\ I_t,\ C_t,\ A_t>,$$

where

$q_t^s \in S$ is the source state of t;

$q_t^t \in S$ is the target state of t;

$I_t \in I$ is the external input of t;

$C_t \in C$ is the enabling condition or predicate of t which must be true for t to take place;

$A_t \in A$ is the actions of t which are executed when t is fired. □

When an EFSM is used to model a communication protocol, the states are chosen to be those instants where the protocol entity is waiting for the next event to happen. One particular state is designated as the initial or idle state which is the state of the protocol entity when it begins running, or some convenient starting place thereafter. Typically, a state is used to represent the *status of connection* of a protocol entity, e.g., CLOSED, OPENING, IDLE, etc., while the context variables are used to store sequence numbers, quality of service, exchanged data, and the like. The above definition also illustrates how the control and data portions of a protocol interact with each other when a protocol is modeled by an EFSM. In fact, context variables play a very important role in such interactions. As the enabling predicate of each transition is a predicate on the context variables as well as the external inputs, the data portion (e.g., context variables) affects the control portion (e.g., state transitions) of a protocol. On the other hand, a state transition may alter the values of the context variables in addition to

producing outputs.

Obviously, the state in S of an EFSM does not represent the "*global state*" or "*complete context*" of a protocol entity modeled by this EFSM. Unlike a FSM, the global state of an EFSM is capable of describing both the control and data aspects of a protocol entity. The following definition further reflects the importance of the context variables in an EFSM model.

Definition 3.2. The **global state** of an extended finite state machine (EFSM) is an $(k+1)$-tuple:

$$(q, \dot{v}_1, \ldots, \dot{v}_k),$$

where $q \in S$ is the current state of the EFSM, $\dot{v}_i$ $(1 \le i \le k)$ is the current value of the context variable $v_i \in V$ and k is the total number of the context variables in the EFSM, i.e., $k = |V|$. □

Essentially , the global state of an EFSM reflects the execution history of the protocol entity in the sense that firing different state transitions usually results in different context variable values or different global states even if the protocol entity terminates in the same state after the execution.

Based on the EFSM model, the normal form specification in Estelle (NFS) is developed as a formal protocol description technique. The precise definition of NFS in Estelle is as follows.

Definition 3.3. A **Normal Form Specification (NFS)** in Estelle consists of a set of *normal form transitions* (NFT). A NFT t consists of the following five components:

WHEN $(t) = nil$ or $I_t(i_t^1, \ldots, i_t^m)$, where I_t stands for the external input of the transition t and $i_t^1, \ldots, i_t^m$, $(m>0)$, are the external input parameters.

FROM (t) is the source state of the transition t, q_t^s.

TO (t) is the target state of the transition t, q_t^t.

PROVIDED (t) = $C_t(i_t^1,...,i_t^m, v_t^1,...,v_t^k)$, where C_t is the enabling condition of the transition t; $i_t^1,...,i_t^m$, ($m \geq 0$), are input parameters and $v_t^1,...,v_t^k$ ($k \geq 0$) are context variables.

BEGIN-END (t) = A_t ($aa_t^1,...,aa_t^j, cc_t^1,...,cc_t^u, oo_t^1,...,oo_t^w$), where

A_t stands for a set of actions of the transition t;

aa_t^i, $i=1,...,j$, ($j \geq 0$), is an assignment statement of the form $y:=nil$ or $y:=aa$ ($i_t^1,...,i_t^m$, $v_t^1,...,v_t^k$), where $m \geq 0$, $k \geq 0$ and y is a context variable;

cc_t^i, $i=1,...,u$, ($u \geq 0$), is a procedure call of the form $cc(x_t^1,...,x_t^h)$, where $h \geq 0$, cc is an undefined procedure name and $x_t^1,...,x_t^h$ ($h \geq 0$) are procedure parameters of cc;

oo_t^i, $i=1,...,w$, ($w \geq 0$), is an output statement of the form $oo(y_t^1,...,y_t^r)$, where oo stands for the output interaction and $y_t^1,...,y_t^r$, ($r \geq 0$) are output interaction parameters. □

Obviously, the EFSM is the formal mathematical model behind the NFS in Estelle. The NFS gives more detail and at a low-level.

3.2. An Acyclic Graph Model

In order to develop an algorithm to generate efficient protocol testing sequences, it is useful to establish a graph model to describe the protocol. The main advantage of using a graph model is that many protocol testing problems, especially the executable path problem, can be conveniently stated and analyzed.

In this section, an acyclic graph model derived from the NFS in Estelle is proposed. This graph model can be used to describe both the control and data portions of a protocol, and it possesses some nice features which are useful for protocol test sequence generation, especially for executable path identification and selection. We present our graph model in two

stages. First, a graph named G_{NFS}, which can be directly derived from a NFS in Estelle, is introduced; then the cycles in this graph are removed according to certain criteria to derive an acyclic graph called **acyclic protocol description graph**, denoted by G_{PD}.

3.2.1. Graph G_{NFS}

Definition 3.4. Given a protocol specification in terms of the NFS in Estelle, a graph $G_{NFS} = (V_{NFS}, E_{NFS})$ is a directed graph with a vertex set V_{NFS} and an arc set E_{NFS}, where

$V_{NFS} = \{s \mid s$ is a state in the given NFS$\}$;

$s_0 \in V_{NFS}$ is called the *initial vertex*;

$E_{NFS} = \{t \mid t$ is a state transition in the given NFS$\}$;

For each $t \in E_{NFS}$, which stands for the transition t in the NFS, the following 3 components are attached to it :

I_t-part = the WHEN clause of the transition t in the NFS;

C_t-part = the PROVIDED clause of the transition t in the NFS;

A_t-part = the BEGIN-END operation block of the transition t in the NFS. □

G_{NFS} is easy to understand because it is quite similar to the ordinary state transition graph of a FSM except for the three new components, in place of an I/O label, attached to each arc.

An example of G_{NFS} is shown in Figure 6-1 and Appendix I.

For protocol testing, however, working directly with G_{NFS} has the following drawbacks:

(1) In the A-part of an arc, some procedure calls and the right-hand-sides of some assignment statements may be undefined, which are left to be decided or interpreted by the

protocol implementor. However, *some* of these undefined context variables or procedures may affect the C-part which cause uncertainties when we want to identify the executable path or analyze the behavior of a protocol entity from the specification.

(2) There may exist some cycles in G_{NFS}. A cycle is a sequence of arcs which start from and terminate at the same vertex. For protocol testing, these cycles may result in complications when the executable path problem is tackled.

To get around the first difficulty, we assume that the protocol implementor can provide the tester with the implementation-related definitions or choices for those undefined components or options in the protocol specification if they are important to executable path identification or protocol testing. In fact, the major purpose of the Protocol Implementation Conformance Statement (PICS) and the Protocol Implementation eXtra Information for Testing (PIXIT) described in Chapter 2 is to supply the additional information by the implementor when it is necessary.

As it is not straightforward to get rid of the second drawback, the next section is devoted to this problem.

3.2.2. Eliminating Cycles from G_{NFS}

Definition 3.5. Given graph G_{NFS}, a **path** in G_{NFS} is a finite, non-null sequence of distinct arcs: $P = (v_{i1},v_{i2}), (v_{i2}, v_{i3}),..., (v_{ir-1},v_{ir}), r\geq 2$. □

Definition 3.6. Given G_{NFS}, a **cycle** in G_{NFS} is a path that starts from and terminates at the same vertex: $C = (v_{i1},v_{i2}), (v_{i2}, v_{i3}),..., (v_{ir-1},v_{i1})$, r is called the **length** of C. □

Cycles in G_{NFS} can be categorized as follows :

(1) **homing-cycle** : a cycle including the initial vertex v_0;

(2) **self-loop** : a cycle of length 1 whose vertices do not include v_0;

(3) **intermediate-cycle** : a cycle of length greater than one whose vertices do not include v_0.

To understand the significance of these cycles in a protocol specification, it is necessary to introduce the concept of *protocol entity connection session*. As mentioned in Chapter 2, one of the basic purposes of a protocol specification is to define the behavior of a protocol entity when it interacts with another protocol entity, called a *peer entity*, in a remote system. The whole process of such interaction activities are called a **protocol entity connection session**. Basically, a protocol entity connection session may consist of the following 3 or more phases: (1) connection establishment phase; (2) data exchange phase; (3) connection release/termination phase. Each phase may consist of a number of states and transitions. Each state in a protocol specification should possess the RESET function or a transition directly going back to the initial state so that the protocol entity can never get stuck anywhere when the connection session is interrupted due to some errors.

Typically, most cycles in G_{NFS} are homing-cycles which are important to protocol testing. A homing-cycle can be intuitively interpreted as a sequence of operations which constitute either a protocol entity connection session or part of it interrupted by RESET. Obviously, every vertex in G_{NFS} must be within some homing-cycle.

Self-loops represent those operations which may change the values of the context variables or the global state of a protocol entity but do not alter the connection status of a protocol entity. For instance, after a connection between two protocol entities has been established suc-

cessfully, the connection status of the protocol entities is 'CONNECTED' and data exchanges start. During the process of data exchanges, the connection status of the protocol entities remains the same even through some context variables may be updated. Therefore, the data exchange operations can be modeled by some self-loops in G_{NFS}.

Besides homing-cycles and self-loops, there may exist some intermediate-cycles in a G_{NFS}. Usually, this kind of cycle represents a sequence of repeated intermediate state transfers and operations within a protocol entity connection session.

For protocol testing, the goal of cycle-elimination is to transform the given G_{NFS} into an acyclic graph such that the new graph is *semantically* the same as the original one as far as the test sequence generation is concerned. Put in another way, the test sequences generated from the new acyclic graph should be as valid, effective and powerful as those generated from the the original graph. In the following subsections, three cycle-elimination methods are proposed.

3.2.2.1. Homing-Cycle Elimination

Since a homing-cycle starts from and terminates at the initial vertex v_0, homing-cycle elimination is based on the special property of v_0. As we have pointed out previously, v_0 is both the starting and ending point of a group of protocol operations since v_0 represents both the initial and the idle state of a protocol entity. This implies that the global state of a protocol entity should be initialized once v_0 is reached.

Based on the above observation, the **image vertex** method is proposed to eliminate homing-cycles from a G_{NFS} [Dat87] [WaK88]. A new vertex $\bar{v}_0$ called the *image vertex* of v_0 is introduced into G_{NFS}. This image vertex functions as a sink and all of the arcs originally

entering v_0 are now redirected to $\bar{v}_0$. From another point of view, v_0 in a G_{NFS} is split into two vertices denoted as v_0 and $\bar{v}_0$, which represent the starting and ending point of a protocol entity connection session, respectively. We can imagine that there exists an *invisible* link from $\bar{v}_0$ to v_0 and it is always automatically traversed when $\bar{v}_0$ is reached so that a new protocol entity connection session can start. This link does not represent an ordinary state transition of the EFSM and it does not need to be tested. As far as protocol testing is concerned, the new graph derived by this method is clearly equivalent to the original one.

The algorithm for homing-cycle elimination is quite straightforward.

Algorithm 3.1 HOMING-CYCLE ELIMINATION

Input : A graph G_{NFS}.

Output : A graph G'_{NFS} without homing-cycles.

Step 1. Introduce a new vertex $\bar{v}_0$ into graph G_{NFS}.

Step 2. For any arc t whose target vertex is v_0 **Do**

Begin

Create a copy arc of t from the source vertex of t to $\bar{v}_0$;

Delete the original arc t;

End

□

Obviously, the complexity of the above algorithm is linear in the number of vertices in a G_{NFS}. By applying this algorithm to the G_{NFS} in Figure 6-1, the resulting graph is given in Figure 6-2.

3.2.2.2. Self-Loop Elimination

To eliminate self-loops or intermediate-cycles, the following observation is essential. For protocol testing, any transition in an EFSM model can be tested only a bounded number of times or any arc a G_{NFS} can be traversed only a bounded number of times. Thus, any cycle in a G_{NFS} can also be traversed only a bounded number of times.

Based on this observation, we propose a **cycle expansion** method to remove self-loops from a G_{NFS}. Suppose that there are m self-loops on a vertex and the *expansion constants* k_1, k_2, ..., k_m, which imposes the limits on how many times the corresponding self-loops cycles l_1, l_2, ..., l_m can be traversed, are given by the tester. According to these expansion constants, a bounded number of new vertices and arcs are introduced to expand or unfold these self-loops such that all and only possible paths containing at most k_j cycle-traversals on self-loop l_j (j=1, 2, ... , m) exist in the expanded graph.

When a vertex possesses more than one self-loop in a G_{NFS}, there might, in fact, exist a precedence order among them or certain valid loop combinations. If this kind of order or combinations can be derived directly from the G_{NFS} and used in the expansion, the number of the unfolded vertices and arcs can be greatly reduced. However, it is possible that no such constraint can be derived directly from G_{NFS} or they do not exist. Without loss of generality, we assume that no constraint on precedence order or valid self-loop traversal combinations exists.

Before describing the algorithm, let us study a concrete example. Suppose that there are 2 self-loops labeled as l_1, l_2 on vertex v and let $k_1 = k_2 = 2$ be the given expansion constants. To expand these self-loops, we have to consider the following possible self-loop traversal sequences :

$$l_1 l_1 l_2 l_2,\ \ l_1 l_2 l_1 l_2,\ \ l_1 l_2 l_2 l_1,$$

$$l_2 l_1 l_1 l_2,\ l_2 l_1 l_2 l_1,\ l_2 l_2 l_1 l_1.$$

This sequence includes all possible self-loop traversal sequences on l_1 and l_2 with the restriction that each loop can only be traversed at most twice.

In general, cycle expansion is essentially a permutation problem. The results of the expansion are premutations of the self-loop traversals with k_j (j=1,2,...,m) traversal repetitions for self-loop l_j (j=1,2,...,m). In [MKB83] and [Tuc84], algorithms and a formula for enumerating permutations with repetitions are given. The total number of possible self-loop traversal sequences = $(k_1 + k_2 + ... + k_m)! / k_1! \ k_2! \ ... \ k_m!$. In fact, we can combine some unfolded arcs if they have the same subsequent arcs.

Based on these intuitive discussions, we adopt the following self-loop elimination algorithm.

Algorithm 3.2 SELF-LOOP ELIMINATION

Input : A graph G_{NFS} with n self-loops and the expansion constant k_j (j=1,2,...,n) for each self-loop.

Output : A graph G'_{NFS} without self-loops.

Step 1. For each vertex v with self-loops l_1, l_2 ,...., l_m (m>0), repeat Step 2 to Step 4.

Step 2. For the self-loops l_j (j=1,2,..,m), enumerate the permutations of all possible traversal sequences with k_j repetitions for l_j according to the algorithm given in [Tuc84].

Step 3. For each self-loop permutation sequence derived in step 2, create a sequence of new vertices and copy the corresponding self-loop as the new arcs to connect these new vertices. The permutation sequence starts from v and

terminates at new vertex v', which has the same outgoing arcs as v except for the unfolded self-loops.

Step 4. Delete the original self-loops on vertex v.

□

The resulting graph of applying this algorithm to the graph in Figure 6-2 is given in Figure 6-3.

3.2.2.3. Intermediate-Cycle Elimination

Compared with homing-cycle or self-loop elimination, it is much more difficult to remove intermediate-cycles although the idea of *cycle expansion* can still be used here. The major difficulties are as follows :

(1) In order to remove or expand intermediate-cycles in a G_{NFS}, it is necessary to detect or identify these cycles in the first place. It is quite straightforward to detect self-loops. On the other hand, finding all the intermediate-cycles in a G_{NFS} is not easy because there may theoretically exist an exponential number of intermediate-cycles in a G_{NFS}, which implies that the complexity of an intermediate-cycle finding algorithm will also be exponential in the worst case.

(2) Supposing that there are n intermediate-cycles c_j $(j=1,2,..,n)$ in a G_{NFS} and the corresponding expansion constants k_j $(j=1,2,...,n)$ are given by the tester, similar to self-loop expansion, all of the up to k_j $(j = 1, 2,.., n)$ cycle-traversals for cycle c_j $(j = 1, 2,..., n)$ and their combinations have to be taken into account when these cycles are unfolded. The number of expanded vertices in the new acyclic graph may become exponential in the number of vertices even if there are only polynomial number of

cycles in the original graph.

Fortunately, the number of intermediate-cycles is usually quite small in a protocol specification. For instance, there is no intermediate-cycle in OSI Class 0 Transport Protocol (TP) Specification in Estelle and there are only 7 intermediate-cycles as opposed to 125 homing-cycles in the relatively complex OSI Class 2 TP Specification in Estelle. Therefore, the idea of cycle expansion is still effective in many practical cases.

In order to make cycle detection easier, we propose another method based on **regular expressions**. For cycle-elimination, an EFSM can be treated as a FSM or a finite automaton because the C-part and A-part of a transition are irrelevant in this situation. Thus, a G_{NFS} can also be treated as the state transition graph of a FSM in this case. It is well-known that the languages accepted by finite automata or finite state machines are precisely the languages denoted by regular expressions [HoU79]. In other words, every finite state machine represents a regular expression and conversely. The proof of this equivalence is given in [Arb69, HoU79].

For cycle elimination, the major benefit of using regular expressions rather than state transition graphs is that all cycles in a state transition graph can be represented as **Kleene closures** or **stars** in the corresponding regular expression. This fact can greatly simplify the cycle detection algorithm because finding all the stars in a regular expression is obviously easier than finding all the cycles in a graph. Furthermore, there are algorithms to transform a state transition graph to a regular expression and vice versa [Brz62, HoU79]. Regular expressions also facilitate the cycle expansion process because each star in the regular expression can be substituted by one of the given expansion constants and then the cycle expansion problem becomes a regular expression expansion problem.

Based on the above discussions, an intermediate-cycle elimination algorithm is given below :

Algorithm 3.3 INTERMEDIATE-CYCLE ELIMINATION

Input : A graph G_{NFS} without homing-cycles or self-loops, and the cycle expansion constant k_j (j=1,2,...,n) for each intermediate-cycle in the G_{NFS}.

Output : A graph G'_{NFS} without cycles.

Step 1. Derive a regular expression corresponding to the input state transition graph according to the algorithm given in [Brz62]

Step 2. Substitute the star * in the derived regular expression by the given expansion constant k_j denoted as k_j*.

Step 3. Expand each sub-expression in the form X^{k*} derived in step 2 as follows :

$$X^{k*} = \varepsilon + X + X^2 + ... + X^k$$

Step 4. Transform the derived regular expression into its corresponding state transition graph according to the algorithm given in [Brz62].

□

It should be noted that the worst-case time complexity of the above algorithm is still exponential because the stars in a regular expression are at least as many as the cycles in the corresponding state transition graph. In the worst case, we still have to deal with an exponential number of stars.

After running these algorithms, an acyclic graph can be derived. For protocol testing, the new acyclic graph is equivalent to the original graph G_{NFS} if cycle-traversals are bounded by the given expansion constants.

Definition 3.7. Given a graph G_{NFS} and a set of expansion constants, an acyclic graph is called **acyclic protocol description graph or** G_{PD} if it is derived by : (1) applying the cycle elimination algorithms described above to the G_{NFS}; and (2) redefining the incomplete assignment statements and the undefined procedure calls in the given G_{NFS} according to the information provided in PICS and PIXIT if these incomplete components affect at least one C-part in the G_{NFS}.

Discussions in the following sections make use of G_{PD} extensively.

3.3. the Executable Path Problem: Definitions and Complexity

In this section, the executable path problem will be formally defined and discussed in the context of G_{PD} and the EFSM model. The complexity of this problem will also be studied.

3.3.1. Definitions

In a FSM model, a transition t_i can always be executed or *fired* if the source state of t_i can be reached and the external input stimuli of t_i are exerted. The execution of a transition t_i has nothing to do with *how* the source state of t_i becomes the current state.

In an EFSM model, however, the enabling predicate (i.e., the c-part) of a transition t_i must be true before t_i can be executed. Since the enabling predicate of a transition t_i is a predicate on context variables as well as inputs, whether or not t_i can be fired might depend not only on whether the source state of t_i can be reached, but also on the history, namely, *how* it is reached or which transitions have previously been executed, because different transition execution histories may result in different context variable values. From the point of view of

G_{PD}, it is obvious that not every path to the source vertex of an arc e_i can make e_i be actually traversed, simply because the C-part of e_i may be false. Therefore, the traversal of a specific arc is closely related to the arc traversal or transition execution history of the EFSM. This is the significant difference between a FSM and an EFSM.

Like a FSM model, testing a specific transition t_i in an EFSM model requires the actual execution of t_i. Informally, an executable path to a transition t_i in an EFSM model is a sequence of transitions from the initial state to the source state of t_i such that these transitions can be executed sequentially and then t_i can also be executed. From the point of view of G_{PD}, an executable path to an arc e_i is a path from the initial vertex v_0 to the source vertex of this arc such that the C-parts of all arcs along this path are true when this path is actually traversed, and then the C-part of e_i is also true so that e_i can also be traversed.

To formalize these ideas, the executable path problem and some related concepts are now defined more formally.

Definition 3.8. In a G_{PD}, an arc e_i is **executable** if

(1) the current vertex is the source vertex of e_i,

(2) the I-part of e_i can be satisfied,

(3) the C-part of e_i is true. □

Definition 3.9. In G_{PD}, two arcs e_i and e_j are **compatible** with respect to an execution history if

(1) the target vertex of e_i is the source vertex of e_j,

(2) after e_i is traversed, e_j is executable. □

Definition 3.10. Given an arc e_i in G_{PD}, an arc sequence $e_0, e_1,..., e_i$ is an **executable path** if

(1) the source vertex of e_0 is the initial vertex v_0 in G_{PD},

(2) For $0 \leq k \leq i-1$, e_k and e_{k+1} are compatible with respect to an execution history.

Definition 3.11. Given an arc e_i in G_{PD}, the **executable path identification problem** is to determine whether there exists an executable path to e_i and, if so, to identify at least one of them.

Definition 3.12. Given G_{PD} and a set of executable paths in G_{PD}, the **executable path selection problem** is to select a subset of them as the test paths according to certain criteria. □

Definition 3.13. Given an arc e_i in G_{PD}, the **executable path problem** consists of both the executable path identification problem and the executable path selection problem. □

It is worth noting that the concept of the executable path is closely related to the actual execution of the EFSM or the actual traversal of the arcs in G_{PD}. In other words, one cannot, in general, determine that a path is executable or not by analyzing whether there exists a set of context variable values satisfying the enabling predicates along this path. Since the operations or the A-part are the integral part of each transition in an EFSM, whether the enabling predicate of a transition is true or not depends on the resulting values of the context variables after the actual execution of the previous transitions. Our definitions have reflected this point.

3.3.2. Complexity of the Executable Path Problem

The executable path problem has long been recognized as a very important issue in program testing [Che87, How76]. The complexity of a related problem described below is discussed in [GMU76].

In program testing, the conventional approach is to represent a program as a *program flow graph* - a directed graph in which each vertex represents a basic computation block, containing no conditional branches, of the program and each arc represents a possible control transfer among such blocks. Associated with each arc are the conditions under which this control transfer can take place. Without loss of generality, we can assume that the program flow graph has a single entry vertex v_0 which has no incoming arc and a single exit vertex v_e which has no outgoing arcs. The path condition is defined as the conjunction of the individual arc conditions along a specific path.

In [GMU76], a problem called **impossible pairs constrained program path (IPP) problem** is defined in the context of program flow graph. An *impossible pair* in a program flow graph is defined as two arcs which have mutually exclusive or contradictory conditions. An *impossible pairs constrained path* is a path in the program flow graph which does not contain any impossible pair. In other words, the path condition of an impossible pairs constrained path is always true. The IPP problem is to determine whether or not an impossible pairs constrained path exists. The formal description of this problem is as follows: Given a program flow graph $G_f=(V_f, E_f)$ and n pairs of arcs in G_f: $(a_i, b_i), 1 \le i \le n$, determine if there exists a path from the entry vertex v_0 to the exit vertex v_e, containing at most one arc from each of the n given pairs.

[GMU76] has proved, by polynomially transforming a well-known NP-complete problem 3SAT to IPP, that IPP is NP-complete [GaJ79] even when the underlying flow graph is acyclic, and all in- and out-degrees are at most two. This implies that there may exist no polynomial algorithm to solve the IPP problem even for a highly restricted class of digraphs.

When the underlying flow graph is acyclic, the IPP problem is, in fact, a special case of the executable path identification problem defined before. In G_{PD}, if the I-part and A-part of each arc are ignored and only the C-part of each transition is taken into account, then a path in G_{PD} is executable if and only if there exists no arcs with contradictory C-part pairs along that path. In this case, the problem of determining whether or not an executable path exists in G_{PD} is the same as the IPP problem. Since the IPP problem, which is a special case of the executable path identification problem, is NP-complete, we can conclude that determining whether an executable path exists or not is also NP-hard.

3.4. General Assumption

Protocol testing is usually based on certain assumptions. There are two kinds of assumptions: (1) *general assumptions* which are independent of the specific method, and (2) *algorithm-specific assumptions*.

In this section, we describe the general assumptions which are used throughout this thesis. It is obvious that the more restrictive the assumptions are, the easier may the algorithms become. But the assumptions should be as general as possible so that our methods will be applicable to most practical protocols. The general assumptions are summarized below :

(1) The protocol specification is logically correct and the protocol is specified in an Estelle-like language.

(2) The source listing of the implementation under test (IUT) is not available, but the implementor can, if necessary, provide the tester with the implementation-dependent definitions and parameter options, which are undefined in the specification.

(3) From the protocol specification, the domain of each context variable defined in the specification can be derived.

(4) The tester can directly or indirectly control and observe the external interactions (inputs/outputs) of the IUT.

CHAPTER 4

EXECUTABLE PATH IDENTIFICATION

In this chapter, we study the Executable Path (EP, for short) identification problem in detail. Since the EP identification problem is NP-hard, there may exist no general polynomial algorithm to solve it in the worst case [GaJ79]. However, this theoretical result does not rule out the possibility that we might be able to find some efficient algorithms when considering the average behavior of the algorithm or taking advantage of the properties of the protocol in question. This is the motivation for our further exploring this problem.

In Section 4.1, we describe two properties of graph G_{PD} which are important to EP identification. Some important strategies and general operations used in our algorithms are discussed in Section 4.2. In Section 4.3 and 4.4, two basic EP identification algorithms are developed and their complexities are analyzed. Finally, some variations of our basic algorithms are proposed in Section 4.5.

4.1. Two Properties of G_{PD}

Before trying to solve the EP identification problem, it is worthwhile to study some important properties of graph G_{PD}. These properties can be used either to design an EP identification algorithm or to explain the nature of the EP problem.

It is well known that one of the most effective strategies of solving a large problem is splitting its input set into some distinct subsets to yield a number of smaller subproblems such that the original problem can be solved once its subproblems are solved. The *divide-and-*

conquer paradigm is a good example of such a strategy. Since G_{PD} can be quite large and complex, it is desirable to divide its arc set into some disjoint subsets with different properties so that these subsets can be processed at different stages. In other words, we want to derive a partition of the arc set of G_{PD}.

Definition 4.1. Given $G_{PD}=(V_{PD}, E_{PD})$ and $v_i \in V_{PD}$, a path P is called the **longest path** to v_i if (1) P is from v_0 to v_i, and (2) the length of P is not less than that of any other path from v_0 to v_i. The **longest distance** of v_i is the length of the longest path to v_i. □

Definition 4.2. Given $G_{PD}=(V_{PD}, E_{PD})$, **level** l ($l \geq 0$) is subset of E_{PD} such that the longest distance to the source vertex of every arc in level l equals l. □

Property 4.1. Given $G_{PD}=(V_{PD}, E_{PD})$, the nonempty *levels* form a partition of E_{PD}. □

Proof. Since G_{PD} is an acyclic graph, the longest distance to any vertex in G_{PD} is uniquely determined. Thus, each arc in G_{PD} can only belong to one level. □

Obviously, the number of levels in a given G_{PD} is at most $(n-1)$, where n is the number of vertices in G_{PD}.

Given a vertex $v_i \in V_{PD}$, there might exist more than one path from the initial vertex v_0 to v_i. A question important to the EP identification is: how many paths can exist from v_0 to v_i? In order to answer this question, it is beneficial to study a concrete example in Figure 4-1. For G_{PD} in the figure, there are $(n+1)$ vertices. From vertex v_i to vertex v_{i+1}, $(0 \leq i \leq n-1)$, there exist k arcs, where $k(>1)$ is a constant. Obviously, the number of paths from v_0 to vertex $v_i = k^i$ $(1 \leq i \leq n)$ in this specific example.

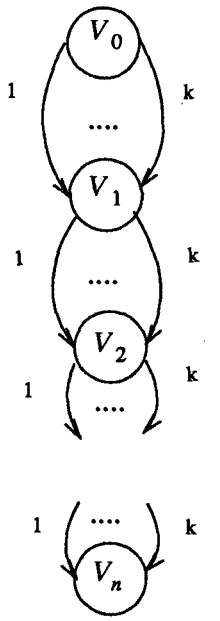


Figure 4-1: A G_{PD} with an exponential number of paths

Property 4.2. Given a vertex v_i in $G_{PD} = (V_{PD}, E_{PD})$, the number of paths from v_0 to v_i may be exponential in $|V_{PD}|$. □

As pointed out in Chapter 3, not every path in G_{PD} is executable since it is possible that two consecutive arcs in a path are not compatible. Thus, the EP identification problem is essentially to identify the executable paths from possibly very many paths in G_{PD}. In the worst case, there might exist only one executable path among the exponential number of paths. This intuitively explains why finding a general efficient solution to this problem is very hard.

Before concluding this section, we give the following definitions.

Definition 4.3. Given two arcs e_i and e_j in G_{PD}, if the target vertex of e_i is the source vertex of e_j, e_i is called a **parent arc** of e_j, and e_j is called a **child arc** of e_i. □

Definition 4.4. Given two arcs e_i and e_j in G_{PD}, e_j is called an **ancestor arc** of e_i or e_i is called a **descendent arc** of e_j if e_j is on at least one path from v_0 to e_i. □

4.2. Strategies and Operations

Since G_{PD} is derived from an EFSM, search and arc traversal on G_{PD} must satisfy certain constraints. In this section, we describe our basic strategies and operations in the context of G_{PD} and EP identification.

4.2.1. Context Information in G_{PD}

Since G_{PD} represents an EFSM which models a protocol specification, search or arc traversal on G_{PD} depends on the execution history of the EFSM as well as the external inputs. As stated previously, the *global state* of an EFSM is the most important information which reflects the execution history of an EFSM. The global state consists of two parts : (1) the current state of the EFSM, and (2) the current values of the context variables of the EFSM. For EP identification, only the second part needs to be recorded since the state is already represented as a vertex in G_{PD}.

Definition 4.5. Given $G_{PD}=(V_{PD}, E_{PD})$ and $e_i \in E_{PD}$, **pre-context** (e_i) is the values of the context variables before the execution of e_i. **Post-context** (e_i) is the values of the context variables after the execution of e_i. Pre-context (e_i) and post-context (e_i) are called **context**(e_i). □

The context of an arc depends on the previously traversed arcs and the external input interactions encountered on them. Basically, pre-context(e_i) can be derived by inheriting post-context(s) of the parent arc(s) of e_i according to certain criteria. Post-context(e_i) can be derived from the results of the A-part evaluation of e_i or by inheriting values from pre-context(e_i) if they are not modified by the A-part.

The context of each arc can be organized as a set of **context vectors** such that each component of a vector corresponds to a context variable and is a *single value*, a *value set* or a *value interval* represented by the corresponding context variable. Theoretically, it is sufficient to allow only single values. Then the context vectors can be considered as representing a relation. However, in order to reduce the number of vectors to be recorded, it is useful to allow a set of values and value intervals as a component. Formally, each vector is of the form:

$$(\ddot{v}_1, \ddot{v}_2, ..., \ddot{v}_n)$$

where n is the total number of context variables in G_{PD} and $\ddot{v}_i$ (i=1,2,..,n) is of one of the following three forms :

(1) b,

(2) $\{c_1, c_2, ..., c_m\}$ ($m>1$),

(3) $[d_l, d_u]$,

where b, c_j ($1 \le j \le m$), d_l and d_u are single values and $[d_l, d_u]$ represents a data interval from d_l to d_u, inclusive. The context vectors representing pre-context(e_i) or post-context(e_i) are called **pre-vectors**(e_i) *or* **post-vectors**(e_i), respectively.

It is natural to use a *value set* or a *value interval* as a component of a context vector when external input parameters are involved on the right-hand-side of an assignment statement of the form y:=aa(i_t^1 ,..., i_t^m, v_t^1 ,..., v_t^k), where y is a context variable. In our algorithms,

all the valid values of input parameter i_t^j ($1 \leq j \leq m$) are taken into account when such assignment statement is evaluated. Thus, the resulting "value" of y might be a value set or a value interval instead of a single value.

More than one vector might be used to represent the pre-context or the post-context of an arc e_i because more than one of the parent arcs of e_i might be considered at the same time in our algorithms. Even in the situation where just a single path is considered, only certain combinations of values, value subsets or value sub-intervals from different components of a post-vector of the parent arc can make the C-part of the current arc true, and it is convenient to record these different combinations separately by using more than one vector. We will discuss this problem in more detail later on.

4.2.2. Arc Traversal and the Related Operations

Arc traversal is fundamental in our algorithms. Since each arc in G_{PD} has C-, I-, and A-parts associated with it and arc traversal represents a state transition in the corresponding EFSM, arc traversal on G_{PD} involves several operations. From the standpoint of EP-identification, the following operations are important: (1) *compatibility check and context inheritance*, (2) *the analysis of the I-part* , and (3) *the evaluation of the A-part and context recording*.

4.2.2.1. Compatibility Check and Context Inheritance

A compatibility check operation, which determines whether the C-part of the current arc can be true or not under the post-context of its parent arc, has to be conducted before the current arc is actually traversed. Another closely related operation is context inheritance

which determines what kind of context should be inherited by the current arc from the parent arc if the current arc and the parent arc are compatible.

For the current arc, the compatibility check is performed by checking all of the post-vectors of its parent arc one by one. It is easier to perform a compatibility check if we transform the predicate in the C-part into the *disjunctive normal form*:

$$M_1 \vee .... \vee M_k,$$

where M_j $(1 \leq j \leq k)$, called a *conjunctive term*, is of the form :

$$R_1 \wedge R_2 ... \wedge R_n,$$

where R_i $(1 \leq i \leq n)$ is a relational function on context variables.

If every component in a post-vector is a single value, the compatibility check becomes quite straightforward since we only need to determine whether these values can make one of M_j $(1 \leq j \leq k)$ true. If a component of a post-vector is a value set or value interval, we have to find those combinations of value subsets or value sub-intervals from every componenet of this post-vector which can make one of M_j $(1 \leq j \leq k)$ true.

In order to identify an executable path, a context inheritance policy called **restrictive inheritance** is used in our algorithms. The basic idea behind this policy is that the current arc e_j inherits only those combinations of values, value-sets or value-intervals which can make the C-part of e_j true from the post-vectors of its parent arc e_i by establishing a pre-vector on e_j for each such combination. Intuitively, the value sets or intervals of a context variable in the context vectors along a path becomes smaller and smaller if this context variable does not appear on the left-hand-side of the intervening assignment statements. If it is assigned new values, clearly, the value set or interval may become larger.

Example 4.1. Suppose that e_i is a parent arc of e_j and there are 3 context variables x, y and z in G_{PD};

post-vector(e_i): $<x=1, y=\{F,T\}, z=[5,15]>$;

C-part(e_j): $\{(x=1) \wedge (y=F) \wedge (10 \le z \le 13)\} \vee \{(x=1) \wedge (y=T) \wedge (z \le 8)\}$,

where T(F) stands for TRUE(FALSE). The compatibility check for e_i and e_j will return TRUE and the results of context inheritance operation will be:

pre-vector-1 (e_j): $<x=1, y=F, z=[10,13]>$

pre-vector-2 (e_j): $<x=1, y=T, z=[5,8]>$

Obviously, these two vectors inherit those context combinations from e_i which can make the C-part of e_j true. □

4.2.2.2. External Input/Output Analysis and Determination

External input/output interaction analysis and determination is another important aspect when an arc is traversed. Since we assume that the tester can observe or control the external interactions directly or indirectly, the appropriate input/output parameter values should be determined by the EP identification algorithm. As the external input parameters can be used in both the A-part and C-part, the major functions of this operation are: (1) determining the valid domains of the input parameters when they are used on the right-hand-side of an assignment statement or a predicate in the C-part; and (2) determining the appropriate domains of input/output parameters which can make a *potentially* executable path in G_{PD} executable. We will describe this operation in more detail in later sections.

4.2.2.3. A-part Evaluation

This operation evaluates the assignment statements in the A-part by using the pre-context and the external input parameters of this arc. After the evaluation, the resulting values of the context variables that are on the left-hand-sides of the assignment statements are stored into the post-vectors.

Like compatibility check, complications arise when the components in a pre-vector are value sets or value intervals. For a value set, we have to evaluate the A-part by considering every value in this set. For value intervals, we can evaluate the starting and ending values of this interval to derive a new interval if the assignment function is monotone.

Example 4.2 Suppose that pre-vector(e_i) is $<x=\{1,30\}, y=\{T, F\}, z=[5,10]>$ and the A-part(e_i) is $x:=x+z$. After A-part evaluation, we have:

post-vector-1(e_i): $<x=[6,11], y=\{T, F\}, z=[5,10]>$

post-vector-2(e_i): $<x=[35,40], y=\{T, F\}, z=[5,10]>$ □

4.2.3. Search on G_{PD}

When it is desired to determine or identify a subset of arcs or vertices which possess a certain property in a graph, the determination process can be carried out by traversing or examining the arcs or vertices. A systematic *search* serves as a skeleton around which many efficient algorithms can be built. Many search schemes are described in [AHU74]. Our main concern in this section is to modify and improve these standard methods to adapt them to G_{PD} and EP identification.

4.2.3.1. Top-Down vs. Bottom-UP

Depending on which vertex is chosen as the starting vertex and in which direction arcs are traversed, a search on G_{PD} can be conducted in two ways: **top-down** or **bottom-up**. Top-down search starts with the initial vertex v_0, traverses forward in the directions of the arcs, and stops when certain termination conditions are satisfied, e.g., every arc in G_{PD} has been traversed or some specific arc has been reached.

As far as EP identification is concerned, the main benefit of top-down search is that it can be used to simulate actual state transitions or protocol operations because search starts with the initial state and paths in G_{PD} are traversed in the normal way. Furthermore, the contexts of the traversed arcs can be derived and the compatibility of two consecutive arcs can be checked. However, when top-down search is used to find some paths to a specific arc e_g, called the **goal arc** in G_{PD}, many arcs and paths which do not lead to e_g might be involved in the search process, because whether an arc or a path can lead to e_g is unknown until e_g or a vertex without any untraversed outgoing arc is reached. Therefore, certain pre-processing is necessary to make this scheme efficient.

The other approach is bottom-up search. In this method, search begins with a given arc e_g (or vertex v_g), traverses backwards and stops at the initial vertex v_0.

As the arc traversals in the above two methods are in the opposite directions, one's advantages often become the other's disadvantages. For example, bottom-up search cannot be used to simulate actual transition execution or protocol operations, but the arcs or paths involved in search all lead to the given arc e_g or vertex v_g. This method can also be used to collect certain information or to determine the appropriate external interactions for a potentially executable path, which will be discussed later.

4.2.3.2. Path-First vs. Level-First

Both top-down and bottom-up searches on G_{PD} can be conducted in two ways: *depth-first* or *breadth-first*.

For EP identification, we propose a search method based on depth-first search as follows. Given a G_{PD}, a starting-vertex v_s, and the search-termination-condition c_t, **path-first search (PFS)** starts with v_s and a path is followed as far as possible, traversing or processing all the arcs along the way as long as the consecutive arcs are compatible, until c_t becomes true or a so-called *dead end* is reached. A dead end is a vertex such that all of its outgoing traversable arcs, if any, have been traversed. At a dead end we backtrack along this path to the parent node of the dead end and select another path to traverse, if possible. After backtracking, if the current node is also a dead end, we backtrack one more arc, and so forth until another path is found or the search-termination-condition becomes true. The formal description of this method is given in Figure 4-2.

Procedure PFS (Input_Graph; Starting_Vertex; Initial_Op;
Compatibility_Check; Traversal_Op; Termination_Condition)

Output : A group of consecutive arcs marked *selected.*

Notation : *S* is a stack, initially empty. The vertex on the top of the stack at any time is referred to as *top*. Each arc in Input_Graph has a mark attached to it and the values of a mark can be: *untraversed, traversed* or *selected.*

```
Begin
Mark every arc in Input_Graph as untraversed;
Set the parent arc of Starting_Vertex as nil ;  /* nil is a fictitious arc marked selected */
Push Starting_Vertex into S ;
Perform Initial_Op;  /* other initialization operations */
While (S is nonempty) Do
    Begin
    While (there exists an arc marked untraversed from top ) Do
        Begin
        Current_Arc := an arc marked untraversed from top ;
        If (Compatibility_Check(Parent_Arc, Current_Arc) = true )Then
            Begin
            Mark Current_Arc as selected;
            Traversal_Op (Parent_Arc, Current_Arc);
            If (Termination_Condition = true ) Then Quit (PFS);
            Push the target node of Current_Arc into S ;
            Parent_Arc := Current_Arc;
            End
        Else Mark Current_Arc as traversed;
        End
    Mark all the arcs from top as untraversed and initialize these arcs;
    /* initialize the context vectors of these arcs */
    Pop S ;
    Change the mark of the arc from top marked selected to traversed;
    Parent_Arc := the arc to top marked selected;
    End
End
□
```

Figure 4-2: Path-first search on G_{PD}

Path-first search is straightforward and effective in many cases. Since only one path is handled at any moment, there is not much context necessary to be recorded or processed at intermediate stages. However, since this method is based on the path-by-path investigation and the principle of backtracking, an arc might be traversed many times because search might

go back and forth before it terminates. From Property 4.2, it is clear that the complexity of this search might become exponential if there are an exponential number of paths in G_{PD}.

An alternative to path-first search is level-first search which is a variation of the conventional breadth-first search. Breadth-first search is vertex-oriented and the vertices in a graph are visited in order of the increasing distance from the starting vertex, where the distance here is simply the number of arcs in a shortest path. For the EP identification problem, we propose an arc-oriented search based on the concept of *level* defined earlier as follows: Given G_{PD}, **level-first search (LFS)** traverses the arcs level by level. In other words, an arc of level k is not traversed until all of its parent arcs and all of the arcs of level $(k-1)$ have been traversed. A formal description of this method is given in Figure 4-3.

Since level-first search is carried out on increasing level, there is no backtracking involved. In other words, each arc in G_{PD} is traversed only once by this method. Level-first search thus handles many paths at the same time and the intermediate contexts of more than one relavant path have to be recorded for subsequent use. Since there might exist an exponential number of paths to an arc in G_{PD}, the major cost of this search method results from the amount of context which needs to be recorded.

4.2.3.3. Some Observations on Search

From the above two procedures, four search methods on G_{PD} can be derived: *top-down path-first, bottom-up path-first, top-down level-first* and *bottom-up level-first.* We have the following observations on these methods in the context of EP identification.

(1) As different search methods have different advanntages and disadvantages, all combinations of path-first and level-first with top-down and bottom-up may be useful depending

Procedure LFS (Input_Graph; Starting_Vertex; Initial_Op;
Compatibility_Check; Traversal_Op; Termination_Condition).

Output : A group of arcs marked *selected.*

Notation : Q is a queue, initially empty. We use *front* to represent the first element in Q. Each arc in Input_Graph has a mark attached to it and the values of a mark can be : *untraversed, traversed* or *selected.*

```
Begin
Mark every arc in Input_Graph as untraversed;
Set the arc to Starting_Vertex as nil and mark it as traversed;
/* nil is a fictitious arc marked selected */
Insert Starting_Vertex into Q;
Perform Initial_Op; /* other initialization */
While (Q is nonempty) Do
    Begin
    If (all the arcs to front have been marked selected or traversed)
    Then
        Begin
        For (each arc a marked untraversed from front) Do
            Begin
            For (each arc a ' to front marked selected)Do
                Begin
                If (Compatibility_Check(a ', a )=true )
                Then
                    Begin
                    Traversal_OP (a ', a );
                    Mark a as selected;
                    Insert the target vertex of a into Q if it is not in Q;
                    End
                End
            End
            If (a is marked as untraversed ) Then (mark a as traversed);
            If(Termination_Condition=true )Then Quit (LFS);
        End
    Remove front from Q;
    End
End
□
```

Figure 4-3: Level-Fisrt search on G_{PD}

on the situations.

(2) The major complexity of search on G_{PD} stems from either the number of paths to be traversed or the amount of intermediate contexts to be recorded. For level-first search, the main task is to control the size of the intermediate contexts. For path-first search, reducing the number of paths to be traversed is challenging.

(3) To design an efficient EP identification algorithm, one-phase search may not be enough. Multiple-phase search should perhaps be considered, using different search methods in different phases.

4.3. An EP Identification Algorithm Based on Path-First Search

Based on the previous discussions and observations, we propose our first EP identification algorithm in this section. The algorithm is described in two steps. First, the basic idea are presented, and then a formal description and complexity analysis are given.

4.3.1. Algorithm Overview

Given G_{PD} and the *goal arc* e_g, the objective of this algorithm is to identify an executable path, if any, from the initial vertex v_0 to e_g and to determine the appropriate external input/output interactions along this path. The algorithm consists of three phases and different search methods are used in different phases. Top-down path-first search is used in Phase 2 which is the major phase of this algorithm.

Phase 1 performs the initialization or pre-processing on the given G_{PD} so that more sophisticated search and manipulations can be conducted efficiently in the subsequent phases. The major tasks of phase 1 are as follows:

(1) Identifying, labeling and initializing all of the ancestor arcs of the goal arc e_g, because only these arcs need to be considered in the subsequent phases. These arcs constitute a subgraph G_{PD}^1 of G_{PD}.

(2) Determining which context variables should be recorded in the context vectors of each arc in G_{PD}^1. A protocol is usually organized as several protocol-phases and each protocol-phase is represented by a group of arcs in G_{PD}. Since different context variables tend to be used in different protocol-phases, it is unnecessary for an arc e_i to record the values of those context variables which are never used in either the C-parts or the right-hand sides of the assignment statements of the descendant arcs of e_i.

(3) Transforming the predicate in the C-part of each arc in G_{PD}^1 into the disjunctive normal form.

Clearly, every ancestor arc of e_g must be visited once in Phase 1. According to our previous discussions, bottom-up level-first search starting at e_g is ideal for this purpose.

The objective of Phase 2 is identifying and labeling one *potentially* executable path from v_0 to the given goal arc e_g. A path is called **potentially executable** if this path can become an executable path when the appropriate inputs are applied. Obviously, the direction of search in this phase should be top-down because our major task is to simulate actual transition along a path. We shall use path-first search in this phase. Before traversing an arc, the compatibility check and context inheritance are performed so that only compatible arcs are traversed. When an arc is traversed, the I-part is analyzed and the A-part is evaluated. This process is continued until one executable path from v_0 to e_g is found or the search fails.

Phase 3 is the last phase of our algorithm. The purpose of this phase is determining the appropriate external input interactions along the path identified by Phase 2 such that this path can be fired when these external interactions are applied. Since the *restrictive inheritance* policy is used in Phase 2, bottom-up path-first search is a natural choice for Phase 3, because we can start from the most restrictive context vectors and modify those less restrictive ones of the ancestor arcs. When an arc is visited, the following operations are performed: (1) modifying the value sets or intervals of the context variables in a context vector according to the more restrictive values in the corresponding vector of the child arc; and (2) determining the appropriate external input interactions of this arc.

4.3.2. Formal Description of the Algorithm

Our EP identification algorithm is formally described in Figure 4-4.

Algorithm 4.1 EP_IDENTIFICATION_1

Input : G_{PD}=(V_{PD}, E_{PD}); a goal arc $e_g \in E_{PD}$.

Output : an EP from v_0 to e_g and the inputs along the EP.

Phase 1. Reverse the directions of all arcs in G_{PD}. /* for bottom-up search */
Label the ancestor arcs of e_g to construct G^1_{PD}, determine the names of the context variables that need to be recorded in the context-vectors and transform the C-part of every arc into the disjunctive normal form by calling the procedure:

LEVEL_FIRST_SEARCH (Graph=G_{PD}; Starting_Vertex=the target vertex of e_g;
Initial_Op = create or initialize a *C_Variable_Name_Vector* for each arc
/* this vector records which context variables need to be recorded as the context of this arc */;
Compatibility_Check = nil;
Traversal_Op = PHASE_1_TRAVERSAL_OP /* defined in Fig. 4-5 */;
Termination_Condition = nil).

Phase 2. Reverse the directions of all arcs in the subgraph derived from Phase 1.
Identify and label a potentially executable path by calling the procedure:

PFS (Graph = G^1_{PD}
/* subgraph derived from Phase 1 with the arc directions reversed */;
Starting_Vertex = v_0;
Compatibility_Check = PHASE_2_COMPATIBILITY_CHECK
/* defined in Fig. 4-6 */;
Arc_Traversal_Op = PHASE_2_TAVERSAL_OP
/*defined in Fig. 4-7 */;
Termination_Condition = e_g is traversed).

Phase 3. If e_g is not maked *selected*, then Quit. /* no EP found */
Reverse the directions of all arcs in the subgraph derived from Phase 2.
Determine the input parameters of the path derived in Phase 2 by calling the procedure:

PFS (Graph = G^2_{PD}
/* Subgraph derived from Phase 2 with the arc directions reversed */;
Starting_Vertex = the target vertex of e_g;
Compatibility_Check = null;
Traversal_Op = PHASE_3_TRAVERSAL_OP /* defined in Fig.4-8 */;
Termination_Condition = v_0 is reached).

□

Figure 4-4: EP_Identification_1

Procedure PHASE_1_TRAVERSAL_OP (Arc_from_*front*, Arc_to_*front*)

Step 1. Label Arc_from_*front* as *ancestor*.

Step 2. Record the names of those context variables into C_Variable_Name_Vector of Arc_from_*front* if they are either already recorded in C_Variable_Name_Vector of Arc_to_*front* or used in the C-part or the left-hand-sides of the assignment statements of the A-part on Arc_from_*front*.

Step 3. Transform the predicate in the C-part of Arc_from_*front* into the disjunctive normal form:

$$M_1 \vee .... \vee M_k$$

where M_j $(1 \leq j \leq k)$ is of the form :

$$R_1 \wedge R_2 ... \wedge R_n$$

where R_i $(1 \leq i \leq n)$ is a relational function.

□

Figure 4-5: Procedure PHASE_1_TRAVERSAL_OP

Procedure PHASE_2_COMPATIBILITY_CHECK (Parent_Arc, Current_Arc)

Step 1. For each post-vector of Parent_Arc, perform Step 2 to Step 3.

Step 2. Find all combinations of values, value sets or value intervals of the components in the post-vector such that each combination can make at least one M_j $(1 \leq j \leq k)$ in the C-part of Current_Arc true.

Step 3. Inherit each combination derived from step 2 by constructing a corresponding pre-vector on Current_Arc according to C_Variable_Name_Vector of this arc if this combination is not a pre-vector of Current_Arc.

Step 4. Return *true* if at least one combination is found in Step 2. Otherwise, return *False*.

□

Figure 4-6: Procedure PHASE_2_COMPATIBILITY_CHECK

Procedure PHASE_2_TRAVERSAL_OP (Parent_Arc, Current_Arc)

Step 1. For each pre-vector of Current_Arc, perform Step 2 and Step 3.

Step 2. Evaluate each assignment statement and construct the post-vector.

Step 3. For context variable v_i, if it is on the left-hand-side of an assignment statement in the A-part, store the evaluation result of v_i into the corresponding post-vector. Otherwise, copy the value of v_i from the pre-vector to the corresponding post-vector.
□

Figure 4-7: Procedure PHASE_2_TRAVERSAL_OP

Procedure PHASE_3_TRAVERSAL_OP (Current_Arc, Parent_Arc)

Step 1. For each pre-vector of Current_Arc, if there is no corresponding post-vector on Parent_Arc, delete this pre-vector. Otherwise, modify the pre-vector of Current_Arc by copying the contents of the corresponding post-vector of Parent_Arc.

Step 2. For each post-vector of Current_Arc, if there is no corresponding pre-vector of Current_Arc, delete this post-vector. Otherwise, modify the post-vector by copying the more restrictive values, value sets or value intervals from the corresponding pre-vector if the context variable does not appear on the left-hand-side of any assignment statements in the A-part of the Current_Arc.

Step 3. Set the external inputs of the I-part of Current_Arc to those values which can be used to derive the values of the post-vectors of Current_Arc.
□

Figure 4-8: Procedure PHASE_3_TRAVERSAL_OP

4.3.3. Complexity Analysis

In this section, we analyze the time complexity of Algorithm 4.1. Both the worst-case and the average-case complexity are discussed. Since there is no backtracking involved in Phase 1 and Phase 3, the complexity of these two phases are bounded if the number of context variables and relational operators are bounded, which is usually true in a protocol

specification. Thus, we will focus our attention on Phase 2 in the following discussions.

4.3.3.1. Worst-Case Analysis

In Phase 2, the major cost is incurred by a possibly large number of paths traversed before an executable path is found. In our algorithm, the worst case will occur when there is only one executable path from v_0 to e_g in G_{PD} and this path is the last one selected to be traversed. This situation can happen because we randomly choose the next arc to traverse from those compatible arcs marked *untraversed* in Phase 2. As there might exist an exponential number of paths leading to the given goal arc e_g in G_{PD} according to Property 4.2, the number of paths traversed in Phase 2 or the complexity of this algorithm is exponential in the worst case.

4.3.3.2. Average-Case Analysis

Although the worst-case complexity is exponential, this algorithm may not necessarily be very costly on average. There are three situations in which the time complexity of Phase 2 is low.

(1) The goal arc e_g is not far away from the initial vertex v_0, in other words, e_g is on a very low level (the initial vertex v_0 is on the lowest level) in G_{PD}. In this situation, the total number of paths from v_0 to e_g is likely to be small.

(2) For some protocols, most paths in G_{PD} are executable so that only a small number of paths need to be traversed before an executable one can be found.

(3) Backtracking occurs only at a very low level before an executable path can be found. In this case, many paths need not be considered.

In order to take the above cases into account, it is necessary to do average-case complexity analysis. Practically, the average-case complexity analysis is more meaningful than worst-case analysis in many situations. However, this kind of analysis is usually much harder to perform because a model for describing the behavior of the algorithm has to be constructed and a reasonable probability distribution of input data or certain algorithm behavior has to be determined, which is often mathematically intractable [AHU74].

In the following discussions, we first use a tree to model path-first search; then we define the probability of certain behavior of path-first search and propose a group of formulas to calculate them; finally, we discuss the average-case complexity of our algorithm.

4.3.3.2.1. Path-First-Search-Tree (PFS-Tree) Model

As stated previously, an arc in G_{PD} might be traversed many times by path-first search in Phase 2. To analyze its complexity, it is desirable to model these arc traversals explicitly in some way. For this purpose, we propose a tree to model path-first search.

Definition 4.6. Given G_{PD}, an ordered tree is called **Path-First-Search-Tree** (PFS-tree, for short) of G_{PD} if (1) each arc in this tree represents a possible arc traversal by path-first-search on G_{PD}, and (2) the arc sequence traversed in the PRE-ORDER on PFS-tree is the same as the arc sequence traversed by path-first search on G_{PD}. In PFS-tree, if a and b are *sibling* (children of the same vertex) and a is the left sibling of b, then all of the descendants of a are drawn *to the left* of all the descendants of b. □

To facilitate our discussion to follow, we give the following definitions.

Definition 4.7. In a PFS-tree, an arc is called a *leaf arc* if the target vertex of this arc is a leaf vertex.

The following observations can be made on a PFS-tree according to the above definitions. Given two arcs e_i and e_j in a PFS-tree, if e_i is an ancestor of e_j, e_i represents the arc traversal on the corresponding G_{PD} which is performed earlier than that represented by e_j; the same is true if e_i is left to e_j. Each leaf arc in a PFS-tree represents a possible goal arc traversal in G_{PD}. An example of G_{PD} and a PFS-tree for G_{PD} is given in Figure 4-9.

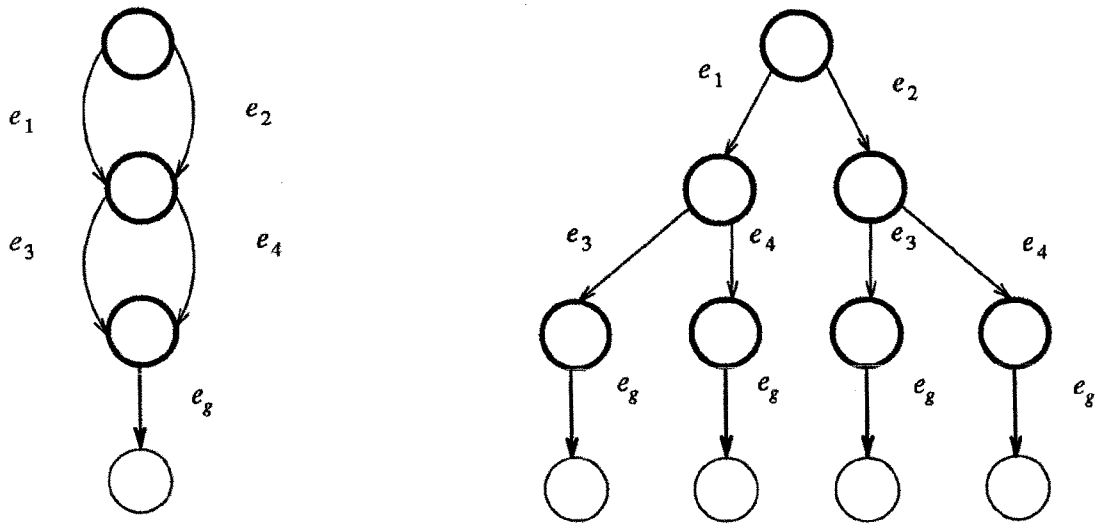


Figure 4-9: G_{PD} and a PFS-tree for G_{PD}

The main purpose of introducing the PFS-tree model is to transform the average-case complexity analysis of path-first search on G_{PD} to the computation of the expected "size" of the corresponding PFS-tree, which is much easier to handle. Moreover, it is more convenient to define, assign and compute certain probabilities on the PFS-tree to model the behavior of path-first search on G_{PD}.

4.3.3.2.2. Probability Definition and Computation

Given a PFS-tree $T_{pfs} = (V_{pfs}, E_{pfs})$, for an arc $e_i \in E_{pfs}$ and a subtree T_s of T_{pfs}, we define the following probabilities :

$P(e_i)$: Probability that e_i is traversed .

$P1(e_i)$: Probability that e_i becomes the current arc.

$P2(e_i)$: Probability that C-part of e_i is true.

$P3(e_i)$: Conditional probability that a leaf arc is not reached via e_i given that the parent arc of e_i is traversed.

$P3(T_s)$: Conditional probability that T_s is not traversed given that the arc to the root T_s is traversed. $P3(T_s)=0$ if T_s is a degenerate tree consisting only of the root.

For a PFS-tree, we have the following formulas according to probability theory.

$$P1(\textit{leftmost arc starting with Root}) = 1 \tag{4.1}$$

$$P(e_i) = P1(e_i) * P2(e_i) \tag{4.2}$$

$$P1(e_i) = P(\textit{parent\_arc of } e_i) * \prod_{e_j \textit{ is left-sibling of } e_i} P3(e_j) \tag{4.3}$$

These formulas can be explained as follows. The root of a PFS-tree corresponds to v_0 in G_{PD} and the leftmost arc of the root represents the first arc traversal in G_{PD}. Since path-first search on G_{PD} always begins with v_0, (4.1) obviously holds. (4.2) is true because an arc in G_{PD} can be traversed by path-first search only if it becomes the current arc and its C-part can be satisfied. (4.3) means that an arc in G_{PD} can become the current arc only if the source vertex of this arc is reached and all the paths starting from this vertex that are selected earlier cannot be traversed to reach the goal arc e_g in G_{PD}.

To simplify the probability computation, we assume that P2(e_i)=q, where q is a constant and independent of e_i, and let $\bar{q}$=(1–q). This assumption is valid for those protocol specifications in which the C-parts of most transitions have similiar structure. Now, we discuss how to calculate P3. First, let us consider a simple subtree T_0 given in Figure 4-10, which consists of one root, k leaf arcs and k leaf vertices.

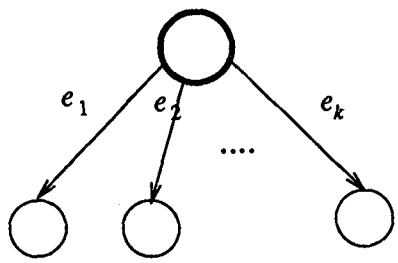


Figure 4-10: A subtree T_0

According to the definition of P3, the following formulas can easily be derived:

$$\begin{aligned} &P3(e_1)=\bar{q} \\ &P3(e_2)=\bar{q}*\bar{q}=\bar{q}^2 \\ &..... \\ &P3(e_k)=\bar{q}*\bar{q}*...*\bar{q}=\bar{q}^k \\ &P3(T_0)=P3(e_k) \end{aligned} \qquad (4.4)$$

In general, we consider a subtree T_s given in Figure 4.11.

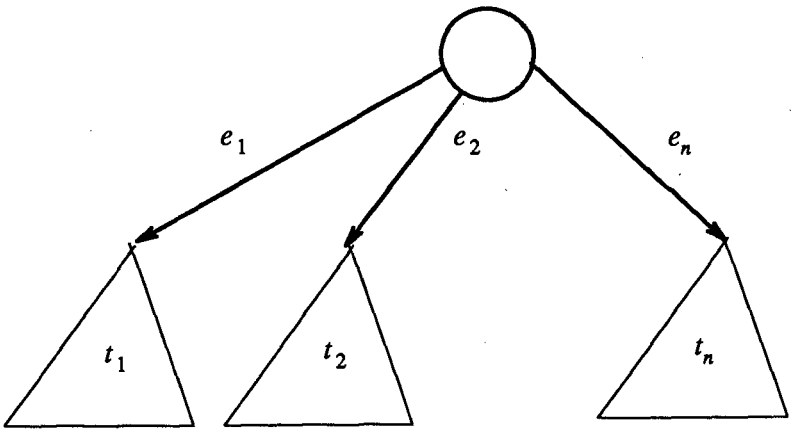


Figure 4-11: A general subtree T_s

This tree consists of one root, n arcs and subtrees $t_1, t_2, \ldots, t_n$. The following formulas can be used to compute P3 for arc e_1 to e_n and T_s.

$$P3(e_1)=\overline{q}+q * P3(t_1)$$

$$P3(e_2)=P3(e_1)* [\overline{q}+q * P3(t_2)]$$

$$P3(e_3)=P3(e_1)* P3(e_2)* [\overline{q}+q * P3(t_3)]$$

$$\ldots\ldots \tag{4.5}$$

$$P3(e_n)= \prod_{i=1}^{(n-1)} P3(e_i)* [\overline{q}+q* P3(t_n)]$$

$$P3(T_s)=P3(e_n)$$

Formulas (4.4) and (4.5) are derived from the following facts. An arc e_i and the subtree below it cannot be traversed to reach the goal arc e_g if either e_i cannot be traversed or e_i can be traversed but the subtree below it cannot be traversed to reach the goal arc. Now, we can easily compute $P(t_i)$ for every arc e_i in a PFS-tree by formulas (4.1) to (4.5).

4.3.3.2.3. Average Case Complexity

Given a graph G_{PD} and its corresponding PFS-tree, the average-case complexity of algorithm 4.1 can be computed as follows. Let arcs $l_1, l_2, ..., l_k$ be the leaf arcs of the PFS-tree and C_a stand for the average-case complexity of path-first-search of our algorithm, we have

$$C_a = \sum_{j=1}^{k} P(l_j)*H(l_j), \text{ where } H(l_j) \text{ is the height of } l_j \qquad (4.6)$$

For a give G_{PD}, the average complexity C_a can be easily computed according to formulas (4.1) to (4.6) and C_a is a function of P2.

From the above discussions and formulas, we have the following observations on the average-case complexity of path-first search.

(1) The average-case complexity C_a given by (4.6) is a function of the probability P2 = q. When q is very small, P will also be very small by (4.2). Intuitively, this corresponds to the situation that backtracking occurs only at a very low level because search can hardly move along a path when the C-parts of most arcs are very difficult to be satisfied.

(2) Another situation is that P2 = q is almost equal to one. In this case, $\bar{q}$ is very small. According to (4.4) and (4.5), P3 will also be very small, and therefore P1, P and C_a are also small. Intuitively, this means that almost every path in G_{PD} is executable so that very few backtrackings are necessary before an executable path can be found.

(3) C_a also depends on the size of the PFS-tree by (4.6). The smaller the size of the PFS-tree is, the smaller will $H(l_{j)}$ be. Intuitively, this corresponds to the situation that the goal arc in G_{PD} is at a very low level and the subgraph generated by Phase 1 is small. So, the total number of paths in this subgraph and the cost of search process are also small.

4.4. An EP Identification Algorithm Based on Level-First Search

The major cost of Algorithm 4.1 stems from the large number of traversed paths because an arc in G_{PD} might be traversed many times before an executable path can be found. To reduce the complexity, it may be useful to abandon the idea of path-by-path processing.

In this section, we propose another algorithm to solve the EP identification problem based on *level-first search*. The major advantage of this scheme is that backtracking can be avoided or each arc is traversed at most once before an EP can be found. However, the complexity of this scheme comes from the amount of recorded intermediate contexts for each traversed path because many paths are handled at the same time.

4.4.1. Algorithm Overview

Similar to Algorithm 4.1, our new algorithm is organized as three phases and each phase has different purposes.

Phase 1 in this algorithm is the same as Phase 1 in Algorithm 4.1.

Phase 2 is the major phase of this algorithm. The purpose of this phase is to identify at least one potentially executable path from v_0 to the given goal arc e_g. Top-down level-first search is used. This phase starts with the initial vertex v_0 or level 0, traverses and examines the arcs level by level using level-first search. Before traversing an arc, compatibility check is conducted. For adjacent compatible arcs, the context is inherited according to the *restrictive inheritance* criterion. When an arc is traversed, the arc traversal operations are performed. The above process is continued until the goal arc e_g is traversed or search fails.

Phase 3 finds and labels one executable path and determines the appropriate external input interactions along this path. Bottom-up path-first search is used in this phase. However, unlike Algorithm 4.1, we have to choose one arc to traverse from possibly more than one compatible parent arcs at each level.

4.4.2. Formal Description of the Algorithm

Algorithm 4.2 EP_IDENTIFICATION_2

Input : $G_{PD}=(V_{PD}, E_{PD})$; a goal arc $e_g \in E_{PD}$.

Output : an EP from v_0 to e_g and the input interactions along the EP.

Phase 1. Same as Phase 1 of Algorithm 4.1.

Phase 2. Reverse the directions of all the arcs in the subgraph derived in Phase 1.
Identify and label all executable paths from v_0 to e_g by calling the procedure :
LEVEL_FIRST_SEARCH (Graph = G^1_{PD}
/* subgraph derived in Phase 1 with the arc directions reversed */;
starting_Vertex = v_0; Compatibility_Check = PHASE_2_COMPATIBILITY_CHECK;
Traversal_Op = PHASE_2_TRAVERSAL_OP;
Termination_Condition = e_g is traversed).

Phase 3. If e_g is not maked *selected*, then Quit. /* no EP found */
Reverse the directions of all the arcs in the subgraph derived from Phase 2.
Determine the inputs of one path by calling the procedure:
PATH_FIRST_SEARCH (Graph = G^2_{PD}
/* subgraph derived in Phase 2 with the arc directions reversed */;
Starting_Vertex = the target vertex of e_g;
Initial_Op = select a post-vector of e_g as the *chosen* vector;
Compatibility_Check = two arcs must have the corresponding vectors;
Traversal_Op = PHASE_3_TRAVERSAL_OP_2 /* defined in Fig. 4-13 */;
Termination_Condition = v_0 is reached).
□

Figure 4-12: EP_IDENTIFICATION_2

Procedure PHASE_3_TRAVERSAL_OP_2 (Parents_Arc, Current_Arc)

Step 1. Repeat Step 2 to Step 4.

Step 2. For the *chosen* post-vector of Parent_Arc, select the corresponding pre-vector of Current_Arc and modify this pre-vector by copying the contents of the post-vector and mark this pre-vector as *chosen.*

Step 3. For the *chosen* pre_vector of Current_Arc, modify the corresponding post-vector of Current_Arc by copying the more restrictive values if the context variable does not appear on the left-hand-side of any assignment statement of the Current_Arc. Mark this post-vector as *chosen.*

Step 4. Set the external inputs of I-part of Current_Arc to those values which can be used to derive the values of the *chosen* post-vector of Current_Arc.

□

Figure 4-13: Procedure PHASE_3_TRAVERSAL_OP_2

4.4.3. The Complexity Analysis

In this section, we analyze the worst-case complexity of Algorithm 4.2. Similar to Algorithm 4.1, we will focus our attention on Phase 2 because the major cost of this algorithm results from this phase.

Since there might exist an exponential number of paths to the goal arc e_g in G_{PD}, the worst case occurs when all the paths from v_0 to e_g are executable and the contexts of different paths are different so that very many context vectors have to be constructed. This is a possible situation because no restriction is imposed on the context variables or A-part in G_{PD}.

If we make a close observation about real protocol specifications and their G_{PD}, the following facts can be found.

(1) A context variable only represents a constant number of values. Practically, the domains of many context variables (e.g., a boolean variable) in a protocol specification

are quite small. The exceptions are those context variables involved in cycles, like a *sequence number*. However, the cycles have been eliminated or unfolded in G_{PD} and only a small portions of them need to be considered in our algorithm. Thus, the domains of these context variables cannot be very large in G_{PD} either.

(2) The number of context variables used to describe a protocol depends on how this protocol is formally specified. In general, the fewer the context variables used, the larger will G_{PD} become. If no context variable is used, G_{PD} essentially becomes a transition diagram of a FSM. In this case, G_{PD} will be huge because of the state space explosion problem discussed in Chapter 2.

From the above discussions, we have the following theorem.

Theorem 4.1: Given G_{PD}, if the size of the domain of each context variable is a constant which is independent of the size of G_{PD}, the complexity of Algorithm 4.2 is linear in the number of arcs in G_{PD}.

Proof. Let $v_1, v_2, ..., v_k$ be the context variables in G_{PD} and these context variables represent $n_1, n_2, ..., n_k$ possible values, respectively. We define a constant: $J = n_1 * n_2 * ... * n_k$. It is easy to see that the total number of all possible different context vectors in G_{PD} is no more than J. Therefore, the size of the intermediate context necessary to be recorded in Phase 2 can not exceed $J*k$. Assuming J and k are independent of the size of G_{PD}, since each arc is traversed only at most once in our algorithm, the worst-case complexity is linear in the number of arcs in G_{PD}. □

Intuitively, there are two situations in which the amount of the recorded context in Algorithm 4.2 can be reduced : (1) if two consecutive arcs are not compatible, it is unneces-

sary for a child arc to inherit the context from its parent arc; and (2) the duplicated context vectors will not be recorded on the same arc.

Although the worst-case complexity of Algorithm 4.2 is linear in the size of G_{PD}, it still depends on the the sizes of the context variable domains. In some cases, the algorithm might be costly or the constant J is large when the protocol is specified in such a way that the sizes of many context variable domains are much larger than the size of graph G_{PD}.

4.4.4. Comparisons

Given G_{PD} and a goal arc e_g, we now have two basic algorithms to identify an executable path from the initial vertex v_0 to e_g. A very natural question is: which one is better ? In this section, we discuss the pros and cons of these two algorithms.

(1) For some protocol, most paths in its G_{PD} are executable. Path-first search algorithm will be efficient in such a situation according to our previous average-case analysis because backtracking will seldom occur and very few paths need to be traversed before an EP can be found. Level-first search is not quite suitable in this situation because the intermediate context has to be recorded for very many traversed paths, especially when many paths in G_{PD} are very long.

(2) Some protocol might be modeled in such a way that the size of G_{PD} is quite large but either the number of the context variables or the sizes of the domains of the context variables are very small. In this case, level-first search is a natural choice according to Theorem 4.2.

(3) If we want to find more than one executable path to e_g, level-first search algorithm is better because all the potential executable paths to e_g have already been found after the execution of Phase 2 of Algorithm 4.2 and only Phase 3 needs to be repeated to determine the external input interactions for different executable paths.

4.4.5. Variations of the Two Basic Algorithms

For protocol testing, we may need the following variations of our two basic algorithms :

(1) An algorithm which can identify an executable path to every arc in G_{PD}.

(2) An algorithm which can identify all the executable paths to a goal arc in G_{PD}.

(3) An algorithm which can identify all the executable paths to every arc in G_{PD}.

Obviously, these variations can be constructed based on our basic algorithms. In order to design an algorithm which can be used to identify an executable path to every arc in G_{PD}, we can simply repeat one of the basic algorithms until an executable path to every arc in G_{PD} has been found. In the worst case, the basic algorithm is used for m times where m is the number of arcs in G_{PD}.

In order to find all the executable paths to the goal arc, level-first search algorithm is preferred. As we discussed previously, all potential executable paths have already been implicitly derived after Phase 2. We can modify our algorithm by repeating Phase 3 such that each traversal of Phase 3 yields a different executable path and its I/O interactions.

When we want to find all the executable paths to every arc in a G_{PD}, generating a *complete compatible search-tree* is the most straightforward method. A complete compatible search-tree is a tree in which every consecutive arcs are compatible and every executable path

is in this tree. A complete compatible tree can be generated based on path-first search or level-first search principle. In this section, we only briefly describe the generation of **complete compatible level-first-search tree (CCLFS-tree).**

Algorithm 4.3 CCLFS-TREE GENERATION

Input : a G_{PD} with L levels.

Output : a tree T_{lfs} such that every path in T_{lfs} is an EP in G_{PD} and all the EPs of G_{PD} are in T_{lfs}.

Step 1. From level 1 to level L, repeat Step 2 to Step 4.

Step 2. For each vertex on one level, repeat step 3 to step 4 on each vertex and the newly generated vertices.

Step 3. If there are i ($i>1$) incoming arcs to a vertex and j outgoing arcs from the same vertex, split this vertex into i_1 new vertices ($i_1 \leq i$) such that (1) each new vertex has only one incoming arc and m *compatible* outgoing arcs ($0 \leq m \leq j$), and (2) each new vertex has a different incoming arc.

Step 4. For a vertex which has one compatible parent arc, inherit the context from the parent arc by creating pre-vectors according to *restrictive inherence policy*, perform the arc traversal operations described in the basic algorithms and create new post-vectors.

□

Figure 4-14: CCLFS-tree generation

From our previous discussions, it is not difficult to observe that each incoming arc is compatible with all the outgoing arcs for a vertex in the resulting CCLFS-tree. Thus, every path from the root to a vertex in CCLFS-tree is an executable path of G_{PD}. On the other hand, since we have considered and expanded every possible path in G_{PD} when we generate CCLFS-tree, all the executable paths in G_{PD} constitute the corresponding CCLFS-tree. An example is given in Figure 6.6.

CHAPTER 5

EXECUTABLE PATH SELECTION

Besides executable path identification, another important aspect of the executable path problem is **executable path (EP) selection.** Given a set of identified executable paths in a G_{PD}, EP selection is concerned with choosing a subset of them as the test paths such that the generated test sequences have a large fault coverage and a low cost.

This chapter is devoted to the EP selection problem. First, we briefly review the conventional test path selection in program testing. Then we propose some EP selection criteria for protocol testing.

5.1. Conventional Test Path Selection Criteria

One of the major concerns of both program and protocol testing is to generate a group of test sequences that adequately exercise the implementation such that most faults can be exposed by these test sequences. Since the research work on program testing or protocol testing is usually based on certain graph models, test sequences consist of two parts : *test paths* and the *test data* along the test paths. Research has indicated that, when left to their own devices, implementors do a poor job of selecting test sequences of good fault coverage [Stu73]. This has led to the development of a number of test path selection criteria [ABC82]. The purpose of these criteria is to provide a guideline for choosing certain test paths which have a large fault coverage and a low cost.

Given a program flowgraph G_f, the well-known *control-flow-based* test path selection criteria in program testing are as follows.

(1) **Node coverage criterion** : Choose a set of test paths in G_f such that every *node* in G_f is covered at least once by these test paths.

(2) **Branch coverage criterion** : Choose a set of test paths in G_f such that every *branch* out of each node in G_f is covered at least once by these test paths.

(3) **Path coverage criterion** : Choose a set of test paths in G_f as the test paths such that every *path* in G_f is covered at least once by these test paths.

The node coverage criterion is rather weak, representing a necessary but by no means sufficient condition for obtaining a reasonable test sequence [CPR89]. Path coverage criterion is the strongest one but it is difficult to achieve for a program flowgraph of more than moderate complexity. Branch coverage implies node coverage and is generally regarded as a basic and important requirement in program testing [Whi87].

In many situations, even the branch coverage criterion is not effective enough to detect certain faults in a program [CPR89]. Thus, there have been a number of more thorough path selection criteria based on **data flow (DF) analysis**, some of which bridge the gap between the branch coverage criterion and the path coverage criterion, have been proposed and studied [LaK83, Nta81, RaW81, RaW85].

DF analysis aims at detecting questionable coding practices (or data flow anomalies) in a given implementation. The most common data flow anomalies are: referencing undefined variables, defining variables without subsequent usage, etc. DF-based test path selection criteria are based on the intuition that one should not feel confident that a variable has been

assigned the correct value at some point in the implementation if no test data cause the execution of a path from the assignment to a point where the variable's value is subsequently used.

In DF analysis, each variable occurrence is generally classified as being a **definition**, in which a value is assigned to the variable, or a **use**, in which the value of this variable is accessed. Two different types of *use* are also distinguished. The first type of *use* directly affects the computation being performed or outputs the result of some earlier definition; such a use is called a **computation use**, or a **c-use**. The second type of *use* is called **predicate use** or **p-use** if the variable is used in a predicate.

The test path selection criteria based on DF analysis focus on tracing the flow of data through the association between the *definitions* and the subsequent *uses*. [RaW81] proposes the following DF test path selection criteria: **all-defs, all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, all-uses, and all-du-paths**. The test paths satisfy, for example, the *all-p-uses criterion* if they cover all *p-uses* in the given program flowgraph. The comparisons of various control-flow-based and dataflow-based test path selection criteria in program testing can be found in [CPR89].

Since there exist many similarities between program testing and protocol testing, some research work has been done on adapting the existing test path selection criteria in program testing to protocol testing [Ura87, UYP88].

It should be pointed out that all of the existing test path selection criteria in both program testing and protocol testing belong to the so-called **structural** or **syntactic** selection approachs which rely only on the syntactic structure of the given graph model and make no attempt to deal with the executable path problem. Thus, all of these criteria suffer from the weakness that the selected paths might not be usable as the test paths.

5.2. Executable Path Selection Criteria in Protocol Testing

In Chapter 4, we developed a group of algorithms which are capable of identifying some or all of the executable paths in a G_{PD}. Since the number of identified executable paths can be very large, we attempt to establish certain criteria or guidelines for executable path selection in this section.

The most straightforward method to come up with executable path selection criteria is to adopt the existing test path selection criteria from program testing. However, because of the differences between protocol testing and program testing and the differences between the conventional test path selection and the executable path selection, the existing syntax-based criteria in program testing are not necessarily effective and efficient for the executable path selection in protocol testing. In order to establish new criteria, it is beneficial to first study the special properties of protocol testing and the executable path selection.

Protocol testing is usually conducted by considering the IUT as a black-box and it is widely believed that the **input/output (I/O) behavior** are most important [ADU88, ShS89, Wu89]. As far as the formal protocol specification or G_{PD} is concerned, I/O behavior consists of the following two parts: (1) **external I/O interactions**, that is, input/output statements and the parameters of these statements; and (2) **internal I/O associations**, that is, the associations between the input and output parameters of I/O statements through the assignment statements in the A-parts or the predicates in the C-parts.

The input and output statements in a G_{PD} can interact via one or more assignment statements on one or more transitions. In a protocol specification, the values of many context variables are defined directly or indirectly by the external input interactions because the right-hand sides of many assignment statements are computations on either the input parameters or

some context variables which are affected by some input parameters through a sequence of assignment statements. The left-hand sides of some assignment statements either directly become the output parameters or appear on the right-hand sides of some other assignment statements which might affect certain output parameters.

Since a predicate in the C-part of a transition can be a boolean function on input parameters as well as the context variables which might be affected by some input parameters, the input and output statements can also interact through the C-part.

As the only way to determine the correctness of the IUT is to make the observations on the external I/O interactions when the IUT is exercised by the test sequences, the fault coverage of the selected test sequences depends on how well they test the external I/O interactions and the internal I/O associations in a G_{PD}. Moreover, the ultimate goal of a communication protocol is to correctly govern the possible communication activities among the different components or protocol entities in a complex communication system and such activities are modeled by the I/O behavior in the protocol specification. Therefore, it is natural and beneficial to establish the executable path selection criteria around the I/O behavior of a protocol.

There are also some differences between the traditional test path selection in program testing and the executable path selection in protocol testing. Since no intensive research work on the executable path identification has been done in program testing, the traditional test path selection is usually based only on the syntactic or structural information of the program flowgraph and the major concern is how many syntactic components, i.e., statements (nodes), conditions (branches), can be covered by the selected paths. However, since we can identify the executable paths from a protocol specification and we can take advantage of the fact that

that every selected test path is executable, our main concern here is choosing a group of executable paths which are more semantically meaningful, that is, more semantic components in G_{PD}, e.g., the external I/O interactions and the internal I/O associations, can be properly exercised.

To facilitate our subsequent discussions, we give the following definitions.

Definition 5.1. Given a G_{PD}, a *data item* is a context variable or an input/output parameter. We say that data item d_1 *affects* data item d_2 if different values of d_1 can result in different values of d_2. □

Definition 5.2. Given a G_{PD}, an input statement I_k in the I-part *influences* a predicate P_j in the C-part if either (1) at least one input parameter of I_k occurs in P_j or (2) a context variable v occurs in P_j and v is affected by at least one input parameter of I_k. □

Definition 5.3. Given a G_{PD}, an input statement I_k in I-part *influences* an assignment statement S_j in the A-part if either (1) at least one input parameter of I_k is on the right-hand-side of S_j or (2) a context variable v is on the right-hand-side of S_j and and v is affected by at least one input parameter of I_k. □

Definition 5.4. Given a G_{PD}, an assignment statement S_j *influences* an output statement O_k if either (1) the left-hand-side of S_j becomes the output parameter of O_k or (2) a context variable v becomes the output parameter of O_k and v is affected by the left-hand side of S_j. □

Definition 5.5. Given a G_{PD}, a predicate P_j *influences* an output statement O_k if different truth values of P_j can result in different output parameter values of O_k. □

Definition 5.6. Given a G_{PD}, an input statement I_k of I-part *influences* an output statement O_k in A-part if either (1) I_k influences an assignment statement S_j and S_j influences O_k,

or (2) I_k influences a predicate P_j in the C-part and P_j influences O_k. In the above two situations, I_k and O_k are called **computation-related I/O pair** and **predicate-related I/O pair**, respectively. These two *pairs* are called **I/O pairs.**□

We can generalize the above definitions by generalizing the concept of input interaction.

Definition 5.7. Given a G_{PD}, the **extended input interaction** of an arc includes both the input statements and the assignment statements in which the right-hand sides contain only the constants. □

Analogically, we can easily define the following concepts: **extended computation-related I/O pair, extended predicate-related I/O pair,** and **extended I/O pair.**

Example 5.1. Supposing that e_2, e_3 are two child-arcs of e_1 as shown in Figure 5-1. The I-, C-, and A-parts of these arcs are defined as follows.

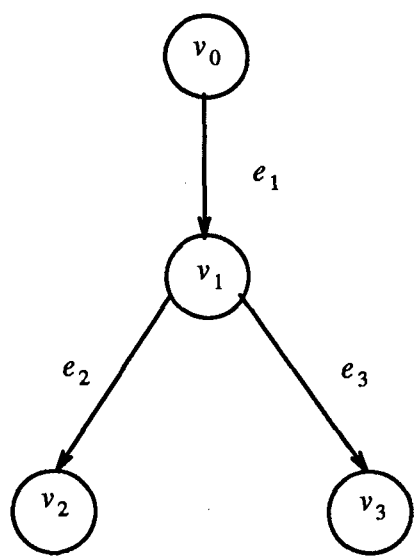


Figure 5-1: Three compatible arcs in a G_{PD}

e_1: FROM v_0, TO v_1

I-part: IN_1 (i_1, i_2)
C-part: ..... /*irrelavant to our discussions*/
A-part:

```
BEGIN
x := i1 + i2;
y := i2;
z := 100
o1 := x;
o2 := y + z;
OUT1 (o1, o2);
END;
```

e_2 FROM v_1, TO v_2

I-part: .....
C-part: .....
A-part:

```
BEGIN
w := x+y ;
o3 := w + 1;
OUT2 (o3);
END;
```

e_3: FROM v_1, TO v_3

I-part: .....
C-part: (x+y+z) > 100
A-part:

```
BEGIN
o4 := true;
OUT3 (o4);
END;
```

In this example, input statement IN_1 of e_1 *influences* the C-part of e_3 because different values of i_1 and i_2 can result in different values of x and y which can affect the truth value of the predicate of e_3. Input statement IN_1 of e_1 *influences* the assignment statements of e_2 because i_1 and i_2 *affect* x and y which are used on the right-hand-side of an assignment statement of e_2. Similarly, IN_1 and OUT_2 are *computation-related I/O pair* and IN_1 and OUT_3 are

predicate-related I/O pair.

Based on above concepts, we propose the following executable path selection criteria for protocol testing:

(1) **(Extended) Input-Computation-Output (I-C-O) Coverage:** Given a G_{PD} and a group of executable paths, we choose a minimum set of executable paths as the test paths such that every *(extended) computation-related I/O pairs* on the given executable paths can be tested at least once by these test paths.

(2) **(Extended) Input-Predicate-Output (I-P-O) Coverage:** Given a G_{PD} and a group of executable paths, we choose a minimal set of the executable paths as the test paths such that every *(extended) predicate-related I/O pairs* on the given executable paths can be tested at least once by these test paths.

(3) **(Extended) Input-Output (I-O) Coverage:** Given a G_{PD} and a group of executable paths, we choose a minimal set of the executable paths as the test paths such that every *(extended) I/O pairs* on the given executable paths can be tested at least once by these test paths.

The above six EP selection criteria are based on the intuition that most errors in an IUT can be detected by certain tests on I/O behavior and that one should not feel confident that an output interaction is correct if no test causes the execution of the path from each input interaction that influences this output interaction. These criteria also take into account the fact that the goal of a communication protocol is to provide correct I/O interactions and therefore most assignment statements in the A-parts and predicates in the C-parts are for the purpose of correctly bridging the relevant input and output statements. In summary, these criteria focus on tracing the control and data flow among the relevant input and output statements.

In order to compare the thoroughness of these criteria, we establish the following relations. Given a G_{PD} and two EP selection criteria C_1 and C_2, we say that C_1 is more **thorough** or has a larger fault coverge than C_2 if all of the executable paths selected by C_2 can also be selected by C_1, and the reverse is false. Two executable path selection criteria are called **incompatible** if neither is more thorough than the other.

It is not difficult to observe that three *extended* criteria are more thorough than their non-extended counterparts because the *extended* (predicate-related, computation-related) I/O pairs is the *superset* of the corresponding (predicate-related, computation-related) I/O pairs. Similarly, the (extended) I-O coverage criterion is more thorough than both the (extended) I-C-O coverage and the (extended) I-P-O coverage critera. However, the (extended) I-C-O coverage criterion and the (extended) I-P-O criterion are incompatible. In Example 5.1, e_1 and e_2 will be tested together according to the I-C-O coverage criterion but e_1 and e_3 can only be tested together based on the I-P-O coverage criterion.

5.3. Input Test Data Selection

Since an executable path can be activated or traversed by applying different external input parameters which may reveal different errors in the IUT, we have to choose the proper input test data along the selected executable paths such that they have a large fault coverage.

As discussed in Chapter 4, an EP-identification algorithm can be used to determine the valid domains of the input parameters along an executable path such that the values of these domains can be used to activate this EP. Therefore, the major task of the input test data selection here is to choose certain values from these domains.

Basically, there are two types of input parameter domains: **enumeration data domain** and **continuous data domain.** Examples of the input parameters of the first type are: *T_Connect_req, class, max_TPDU_size*, etc. Examples of the input parameters of the second type are: *user data, destination reference values*, etc. Similar to program testing [MiH81], certain guidelines for test data selection in protocol testing have been proposed [Dat87, SBG87]. The basic idea of these guidelines is as follows. An input parameter of enumeration type can be tested exhaustively, that is, every possible value in the data domain is tested. For an input parameter of continuous domain type, three specific values can be selected: the two end point and some interior point of the valid data domain. The detailed discussions on the input test parameter selection in protocol testing can be found in [Dat87].

CHAPTER 6

APPLICATIONS

The formal protocol description model, the executable path identification algorithms and the executable path selection criteria introduced in the previous chapters will be illustrated with a concrete communication protocol in this chapter. We demonstrate our methodologies by studying OSI Class 0 Transport Protocol (Class_0 TP) since this protocol has become a standard example in the protocol testing field. The example is centered around the following aspects: (1) establishing the acyclic protocol graph G_{PD} from the Estelle normal form specification (NFS) of the protocol, (2) identifying the executable paths and determining the external I/O interactions along these paths, and (3) selecting a group of test paths from the identified executable paths.

6.1. Application to Class_0 TP

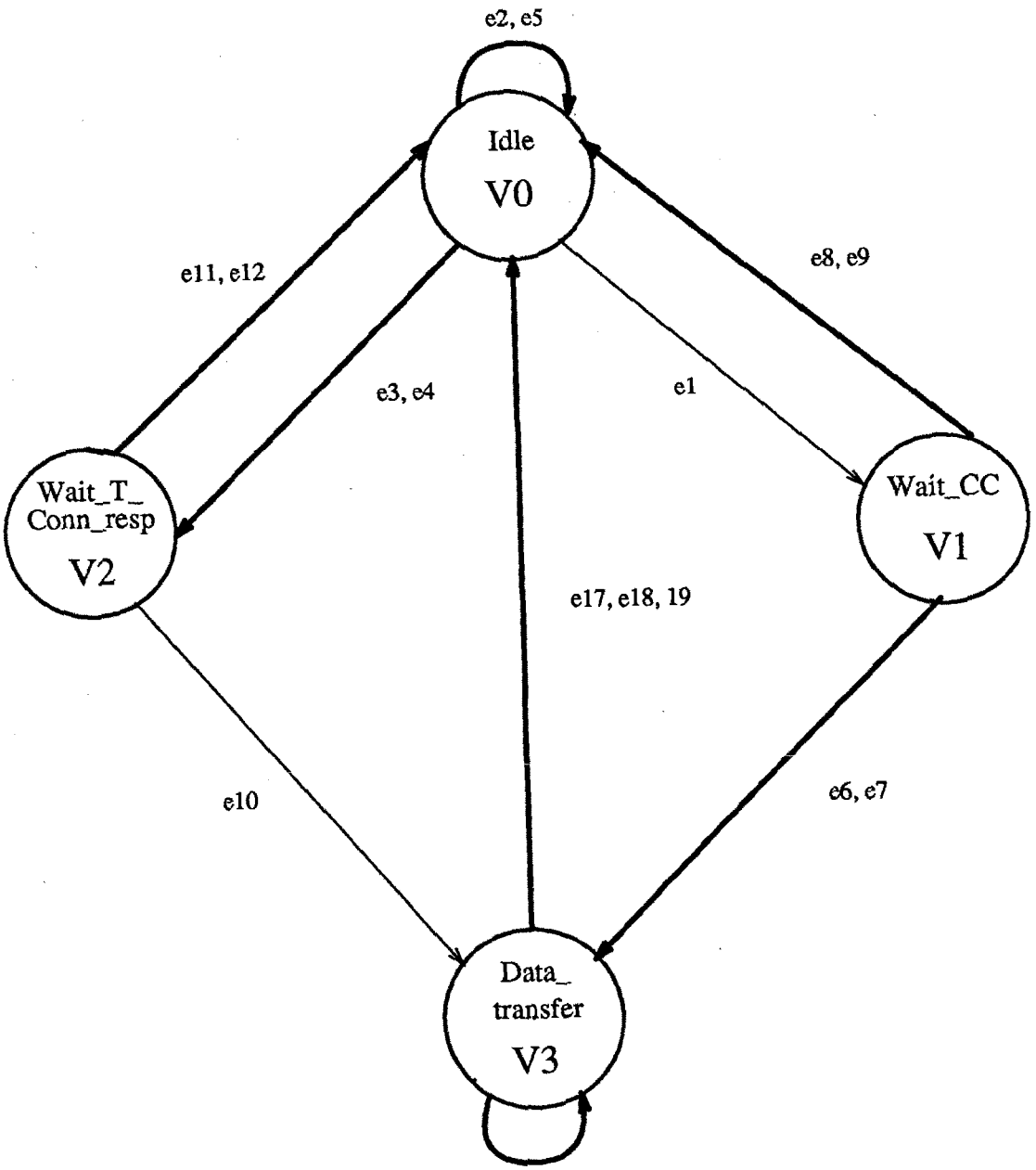
The transport protocol layer has received considerable attention since it is the one and only layer in the OSI architecture with overall responsibility for controlling the transportation of data between the source end-system and the destination end-system. Form the point of view of protocol testing, transport protocols have been intensively studied because of the complexity and the importance of such protocols.

In order to handle a variety of user service requirements and available network services, ISO has defined five classes of transport protocols [ISO82, Kni82]. Class_0 TP is based on the network services which can provide network connection with acceptable residual error rate

and acceptable rate of signaled failures. Class_0 TP is designed to provide a transport connection with data flow control based on network-level flow control and connection release based on the release of the network connection.

6.1.1. Constructing G_a for Class_0 TP

The complete specification of Class_0 TP is in [ISO82], and the normal form specification (NFS) in Estelle of this protocol is given in [SBG87]. From the NFS of this protocol, the graph G_{NFS} can be derived and is shown in Figure 6-1. The definitions of each transitions can be found in Appendix 2.



Single Arc

Multiple Arcs

ei = <Ii, Ci, Ai> i=1,2, ...,19

Figure 6-1: G_{NFS} of Class_0 Transport Protocol

In the G_{NFS} of Class_0 TP, there is no *intermediate-cycle* and there are 4 *self-loops*: e13, e14, e15 and e16. There are the following 20 *homing-cycles*:

e2;
e5;
e1, (e8 + e9);
e1, (e6 + e7), (e17 + e18 + e19);
(e3 + e4), (e11 + e12);
(e3 + e4), e10, (e17 + e18 + e19).

By introducing a new *image vertex* V0' for the initial vertex V0 and applying Algorithm 3.1, the graph G_{NFS} without homing-cycles is derived and shown in Figure 6-2.

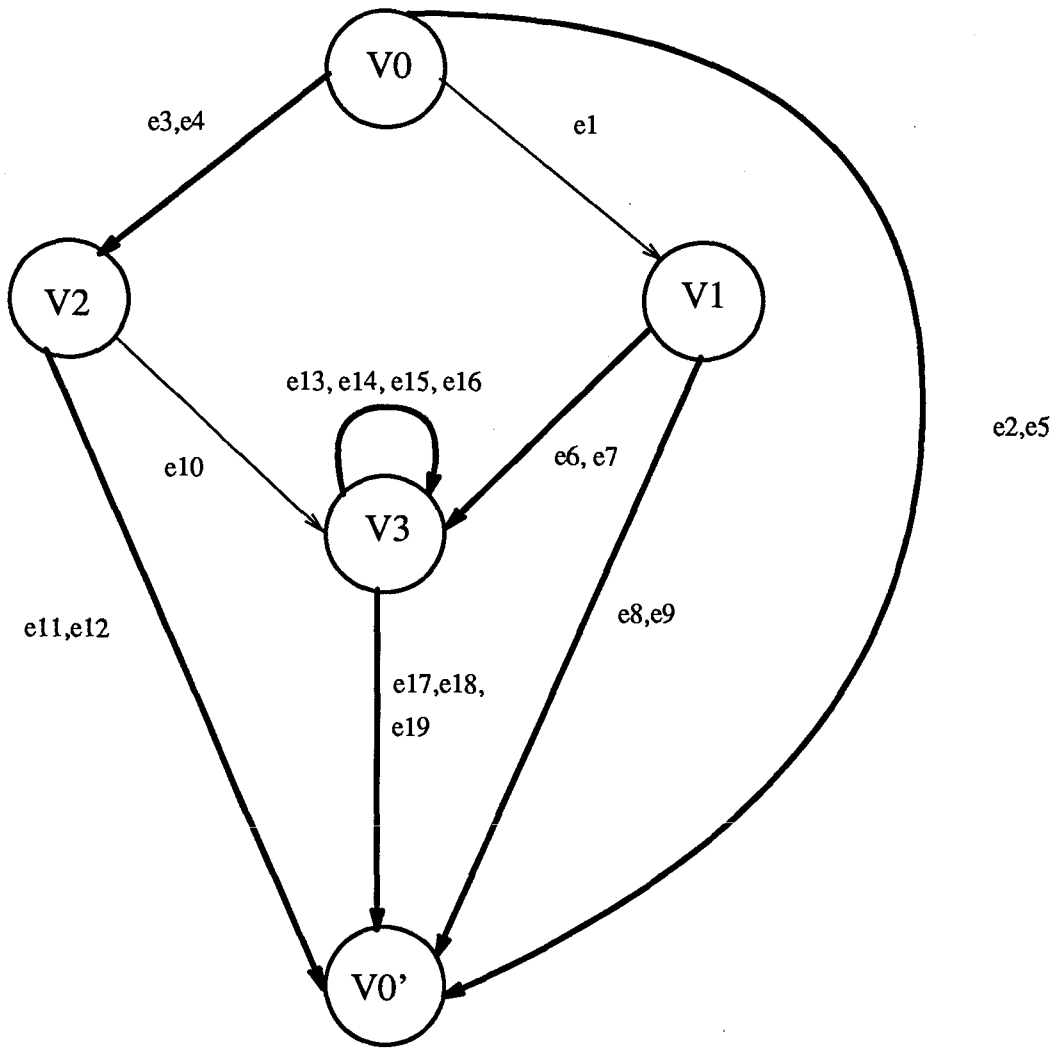


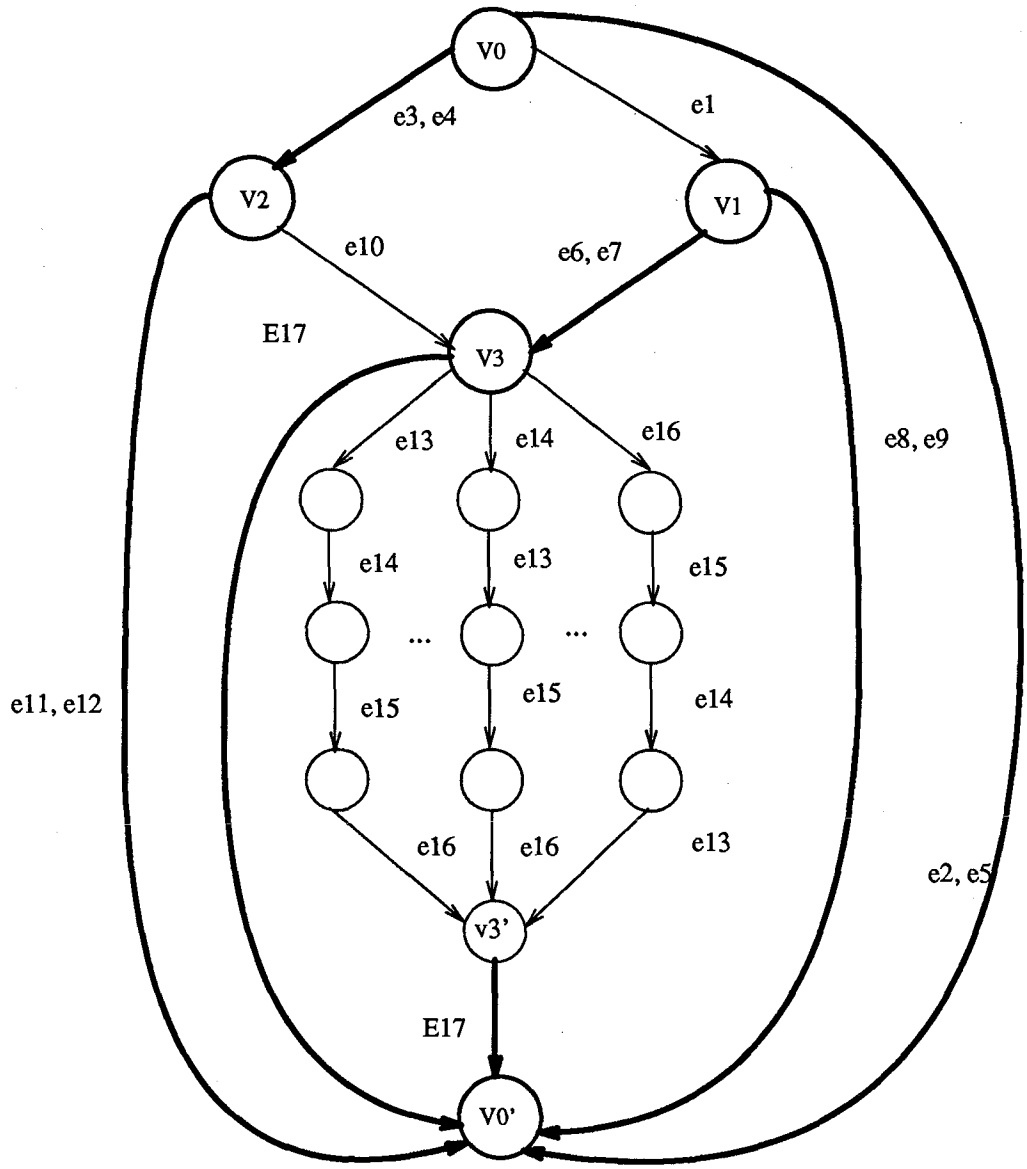
Figure 6-2: G_{NFS} without homing-cycles

To unfold the self-loops in the above G_{NFS}, we assume that each self-loop is tested only once and there is no heuristics about the valid arc traversal combinations or precedence order on these self-loops available. By applying Algorithm 3.2, we unfold the self-loops e13, e14, e15, and e16 into the following 4!=24 possible arc traversal or permutation sequenceses:

(e13 e14 e15 e16), (e13 e14 e16 e15), (e13 e15 e14 e16),
(e13 e15 e16 e14), (e13 e16 e14 e15), (e13 e16 e15 e14),
(e14 e13 e15 e16), (e14 e13 e16 e15), (e14 e15 e13 e16),
(e14 e15 e16 e13), (e14 e16 e13 e15), (e14 e16 e15 e13),

(e15 e13 e14 e16), (e15 e13 e16 e14), (e15 e14 e13 e16),
(e15 e14 e16 e13), (e15 e16 e13 e14), (e15 e16 e14 e13),
(e16 e13 e14 e15), (e16 e13 e15 e14), (e16 e14 e13 e15),
(e16 e14 e15 e13), (e16 e15 e13 e14), (e16 e15 e14 e13).

To simplify our discussion, only 3 of them are given in Figure 6-3. There is a RESET arc from each new vertex to V0' which is not shown in Figure 6-3.



E17=e17,e18,e19

Figure 6-3: G_{PD} of Class_0 transport protocol

In the NFS of Class_0 TP, there exist undefined right-hand-sides in some assignment statements and undefined procedure calls. However, these undefined components do not affect

the C-parts of the transitions in this protocol and it is unnecessary to unfold or redefine them. Thus, the I-part, C-part and A-part of each arc in G_{PD} are the same as those in G_{NFS} which are in Appendix 1.

6.1.2. EP identification for Class_0 TP

Now, we apply our EP identification algorithms described in Chapter 4 to the G_{PD} of Class_0 TP. First, we use Algorithm 4.1 to find an executable path from the initial vertex V0 to a specific arc and to determine the external valid input interactions along this path. Then, we apply Algorithm 4.2 to G_{PD} to do the same thing. Finally, we use Algorithm 4.3 to find all the executable paths in this G_{PD}.

Example 6.1. In this example, we assume that the target arc is **e17** which appears in the path: V3, e13, e14, e15, e16, e17, V0' in Figure 6-2. After the execution of *Phase One* of Algorithm 4.1, the ancestor arcs of this e17 are identified and the graph G_{PD}^{1} is shown in Figure 6-4.

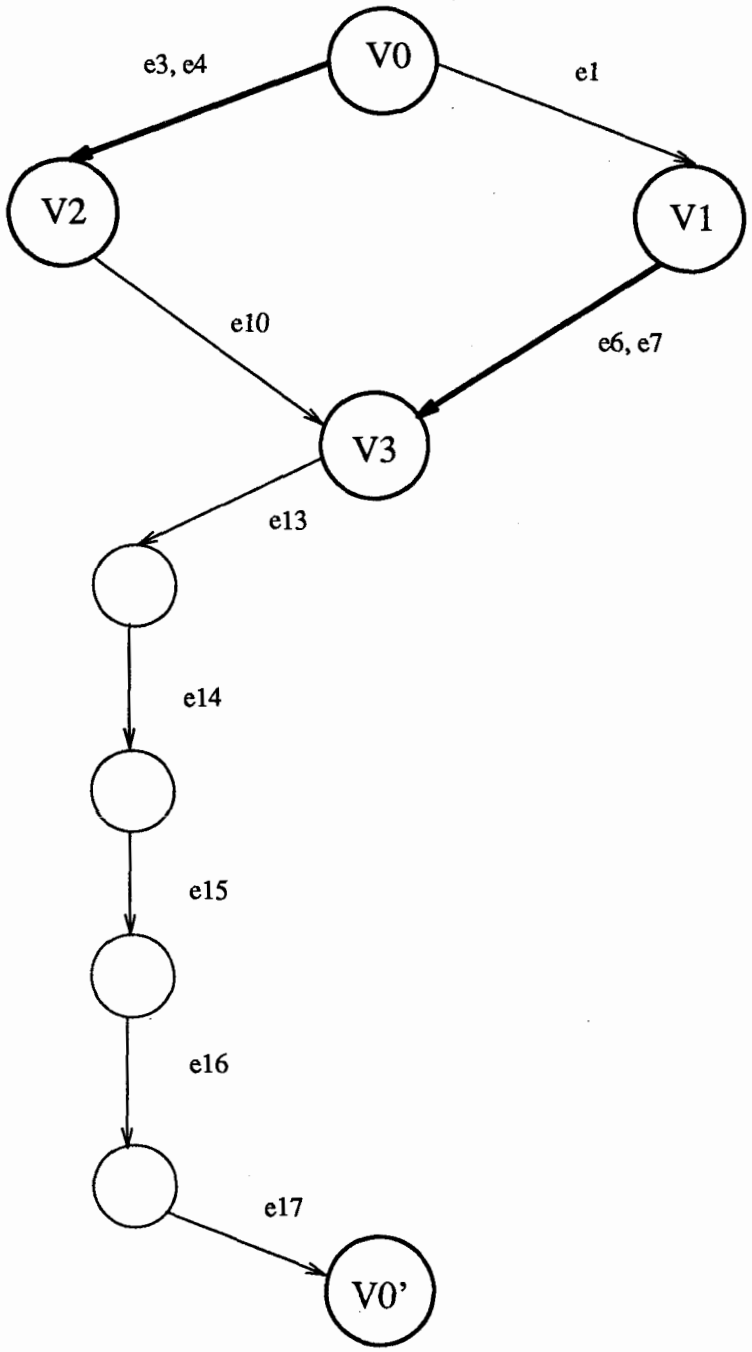


Figure 6-4: G_{PD}^{1} wrt e17

The context variables which need to be recorded in the contex-vectors are in the following *Context_Variable_Name_Vectors.*

Name_Vector 17 = <nil>;
Name_Vector 16 (15) = <in.buffer.mark>;
Name_Vector 14 (13) = <in.buffer.mark, out.buffer.mark>;
Name_Vector 10 (3, 4) = <in.buffer.mark, out.buffer.mark, qts.estimate, remote.refer,
calling.t.addr, called.t.addr, tpdu.size>;
Name_Vector 6 (7, 1) = <in.buffer.mark, out.buffer.mark>;

After the execution of phase 2, we can identify an executable path, for example, (V0, e1, e6, e13, e14, e15, d16, e17 and V0') from G_{PD}^{1}. The important external input interactions along this path can be derived by phase 3 and are listed below:

e1 : **tcreq** with qts.req = ok (Primitive from user);
e6 : **cc** with.max.tpdu.size <> nill (PDU from peer);
e13 : **tdatr** (Primitive from user);
e14 : nil;
e15 : **dt** (PDU from peer entity);
e16 : nil;
e17 : **tdreq** (Primitive from user)

Example 6.2. Now, we assume that the target arc is **e18** which appears in the path: V3, e14, e15, e16, e18, V0' and we attempt to find an executale path from V0' to e18 by applying Algorithm 4.2. After the execution of the *Phase One* of Algorithm 4.2, the ancestor arcs of e18 are identified and the graph G_{PD}^{1} is shown in Figure 6-5.

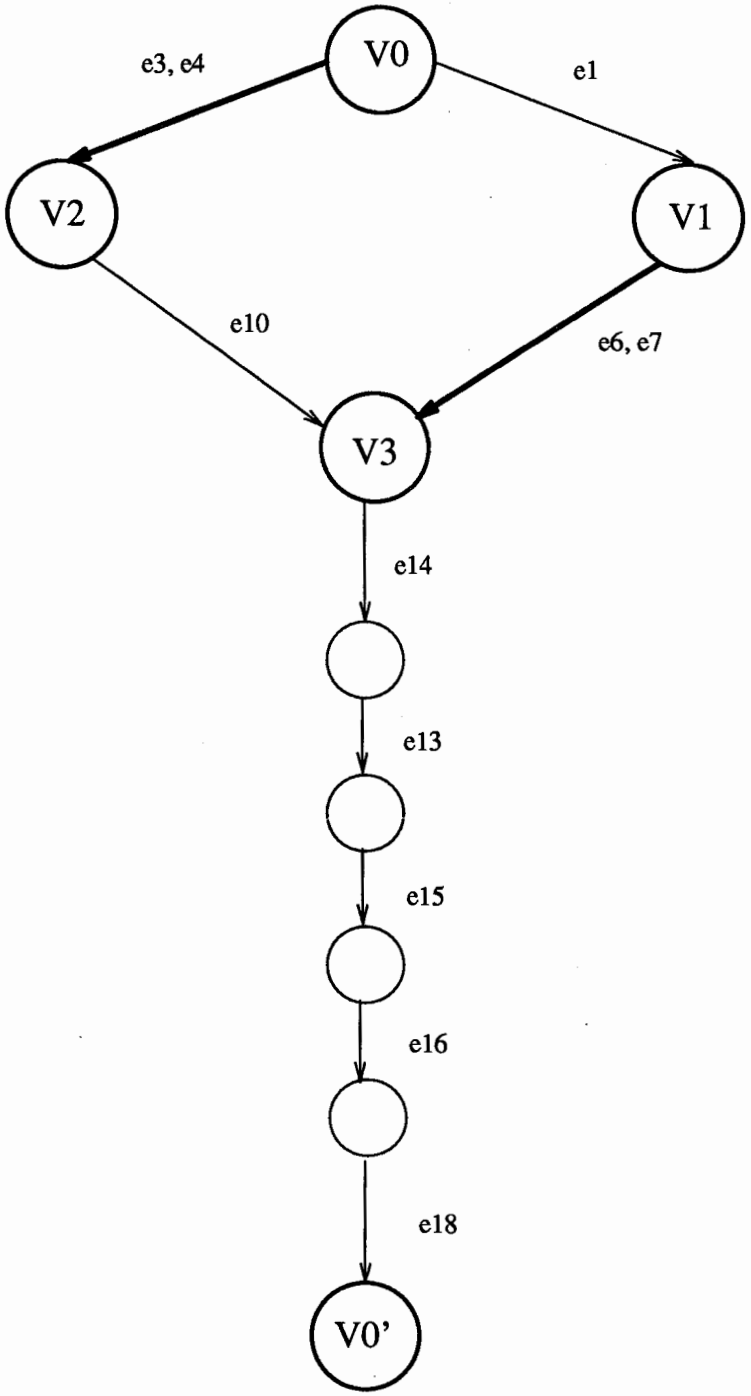


Figure 6-5: G_{PD}^{1} wrt e18

The *Context_Variable_Name_Vectors* are as follows:

Name_Vector 18 = <nil>;
Name_Vector 16 (15) = <in.buffer.mark>;
Name_Vector 14 (13) = <in.buffer.mark, out.buffer.mark>;
Name_Vector 10 (3, 4) = <in.buffer.mark, out.buffer.mark, qts.estimate, remote.refer,
calling.t.addr, called.t.addr, tpdu.size>;
Name_Vector 6 (7, 1) = <in.buffer.mark, out.buffer.mark>;

After traversing {e1, e3, e4} and {e10, e6, e7} by the level-first seach in Phase 2 of Algorithm 4.2, the post vectors of e6, e7 and e10 are listed as follows.

Post-Vector (e10) = <in.buffer.mark = empty, out.buffer.mark = empty; ...>;
Post-Vector (e6) = <in.buffer.mark = empty, out.buffer.mark = empty>;
Post-Vector (e7) = <in.buffer.mark = empty, out.buffer.mark = empty>.

It is obvious that the C-part of arc e14 cannot be true under any of these context vectors because out.buffer.marker = empty. Intuitively, there should be some data received from the local transport service user via a transport data transfer request primitive (tdatr) before it can be sent to the peer protocol entity via a data transfer protocol data unit. In other words, e14 can be traversed only when at least one occurence e13 precedes the first occurence of e14. Therefore, there exists no executable path from V0 to e18 in this situation.

Example 6.3. Now, we apply our Algorithm 4.3 to the G_{PD} of Class_0 TP. The purpose of this algorithm is to generate the **complete compatible level-first-search tree** (CCLFS-tree). After the execution of Algorithm 4.3, a part of the generated CCLFS-tree is shown in Figure 6-6. In this tree, every path is an executable path from V0 to V0' and all the executable paths from V0 to V0' are in this tree. It is clear that this tree has a much smaller size than the complete path tree for G_{PD} because only 6 of 24 newly expanded paths including e13, e14, e15 and e16 are executable.

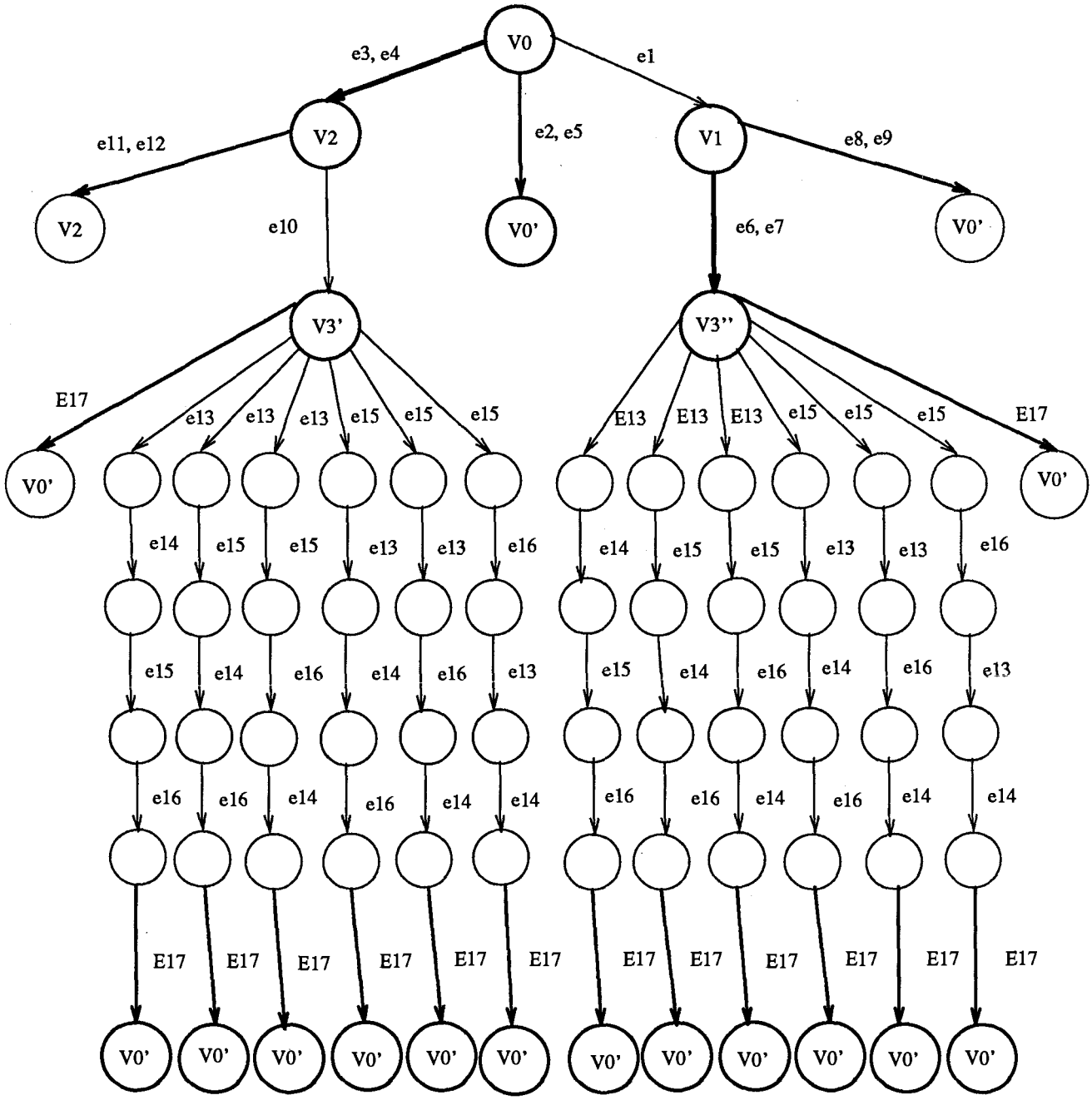


Figure 6-6: CCLFS-tree of Class_0 Transport Protocol

6.1.3. EP selection for Class_0 TP

In Class_0 TP, each input and output statement pair within one transition is obviously an I/O pair. Some I/O statements on different transitions are also I/O pairs. For example, the I-part of e1 and the output of e6 are predicate-selected I/O pair because the truth value of the C-part of e1 affects the output of e6. There are also some computation-selected I/O pairs. For example, the input statement in e3 and the outtput statements in form computational-selected I/O pair because the input parameter *source.ref* from a peer protocol entity eventually becomes output parameter *dest.refer* to the protocol service user. It is not difficult to find other I/O pairs.

In this protocol, there is no difference among the test paths derived by different criteria discussed in Chapter 5. By applying one of them to the derived EPs, a possible test path set and the meaning of these test paths are as follows:

```
e2;              /* user-initiated-call refused by protocol entity */
e5;              /* peer-initiated-call refused by protocol entity */
e1, e8;          /* user-initiated-call refused by peer */
e1, e9;          /* user-initiated-call refused by peer */
e1, e6, e17;     /* user-initiated-connection, freeing */
e1, e7, e13, e14, e15, e16, e18; /* user-ini-connection, data-trans, freeing */
e3, e11;         /* peer-initiated-call refused by protocol entity */
e4, e12;         /* peer-initiated-call refused by user */
e3, e10, e17;    /* peer-initiated-connction, freeing */
e4, e10, e15, e16, e13, e14, e19; /* peer-ini-connction, data-trans, freeing */
```

Compared with the test paths derived by the schemes in [SBG87] and [UYP88], the major advantages of the above test paths are: (1) all of them are executable while the *subtours* or the *test path* proposed in those papers contain many non-executable paths, and (2) the above test paths take the input/output interactions and the extended input/output associations

into account which reflect the nature of protocol testing.

CHAPTER 7

CONCLUSIONS

Communication protocol conformance testing aims at demonstrating the adherence of an implementation under test (IUT) to the protocol specification. It is both attractive and challenging to automatically generate test sequences from a formal protocol specification. To generate efficient and powerful test sequences, a very fundamental and crucial problem is the executable path (EP) problem which consists of executable path identification and selection. In this thesis, we have investigated the protocol test sequence generation in general. Particularly, the executable path problem was studied in detail.

We have established a formal acyclic graph G_{PD} based on the extended finite state machine and the normal form specification in Estelle which was developed by ISO. This graph model can be used to describe both the control and the data portions of a communication protocol. Particularly, this model makes executable path identification and protocol test sequence generation easier. Based on this model, the executable path problem is precisely defined and its complexity is analyzed.

We have developed two basic algorithms for the executable path identification. These multi-phase algorithms use different search schemes. The first algorithm is based on path-first search while the second one is based on level-first search. The complexity analyses have shown that these two algorithms possess different advantages and disadvantages in different situations. Under certain reasonable restrictions, these algorithms can be used to identify executable paths in a G_{PD} efficiently. Some variations of two basic algorithms are also sug-

gested.

In order to generate efficient test sequences, the executable path selection problem has also been explored in this thesis. We have proposed several EP selection criteria based on the external I/O interactions and internal I/O associations. Our criteria reflect the difference between the ordinary program testing and protocol testing. The selected test paths can be used to cover more meaningful semantic components in a protocol specification and to reveal more faults in an IUT.

We have applied our methodologies to a concrete communication protocol. For this protocol, our algorithms work well.

Since the executable path problem is *NP*-hard in general, further research work is needed to improve our algorithms and make them as practical as possible. We identify the following limitations of our solutions.

(1) The complexity of cycle elimination, especially intermediate-cycle elimination, is still very high when we deal with complex protocols. As mentioned in Chapter 3, the cost of intermediate-cycle detection is exponential in the number of nodes in general since there may be exponential number of such cycles. It is also worth noting that the number of stars generated by our algorithm based on regular expression might be larger than the absolute number of the cycles in the corresponding G_{NFS}.

(2) Although the major cost of our two basic EP identification algorithms in Chapter 4 is incurred by Phase 2, the time complexity of the other two phases might also be exponential in the worst case. For example, the complexity of Phase 1 is not linear unless the size of the C-part can be bounded by a constant. Some research work on systematically classifying the C-parts for protocol testing has been done in [Dat87].

(3) As pointed out in Chapter 4, the complexity of Algorithm 4.2 can be very high when the sizes of the context variable domains are very large, because the number of context vectors can be very large in this case. Therefore, further research on selecting a smart set of test data becomes essential to reduce the cost in this case.

APPENDIX I: Estelle NFS of Class_0 Transport Protocol

Service Primitives:

tcreq = T-CONNECT request
tdind = T-DISCONNECT indication
tccon = T-CONNECT confirm
tdatr = T-DATA request
tdati = T-DATA indication
tdreq = T-DISCONNECT request
ndreq = N-DISCONNECT request
ndind = N-DISCONNECT indication
nrind = N-RESET indication

Protocol Data Units (PDUs):

cr = connection request
cc = connection confirm
dr = disconnect request
dt = data

e1: FROM *V0*, TO *V1*;

I-part: tcreq(to.t.addr, from.t.addr, qts.req) /* PRIMITIVE from user */
C-part: tcreq.in.qts.req=ok /* transport entity able to provide the quality asked for */
A-part:

```
BEGIN
local.refer :=...;  /* implementation dependent */
tpdu.size :=...;
calling.t.addr := tcreq.in.from.t.addr;
called.t.addr := tcreq.in.to.t.addr;
cr.out.source.ref :=local.refer;
cr.out.option :='normal';
cr.out.calling.t.addr := calling.t.addr;
cr.out.called.t.addr := called.t.addr;
cr.out.max.tpdu.size := tpdu.size;
out cr(source.ref, option, calling.t.addr, called.t.addr, max.tpdu.size);  /* PDU to peer
entity */
END;
```

e2: FROM *V0*, TO *V0*;

I-part: tcreq(qts.ref)

C-part: tcreq.in.qts.ref<>ok
A-part:

```
BEGIN
tdind.out.ts.disc.reason :=...;
tdind.out.ts.user.reason :=...;
out tdind(ts.disc.reason, ts.user.reason);  /* to user */
END;
```

e3: FROM *V0*, TO *V2*;

I-part: cr(source.ref, option, calling.t.addr, called.t.addr, max.tpdu.size) /* from peer */
C-part: (cr.in.max.tpdu.size<>nil) /\ (cr.in.option=ok)
A-part:

```
BEGIN
remote.refer := cr.in.source.ref;
tpdu.size :=cr.in.max.tpdu.size;
calling.t.addr := cr.in. calling.t.addr;
called.t.addr  := cr.in.called.t.addr;
qts.estimate := ...;
tcind.out.to.t.addr := called.t.addr;
tcind.out.from.t.addr :=calling.t.addr;
tcind.out.qts.pro := qts.estimate;
out tcind(to.t.addr,from.t.addr,qts.pro);   /* to user */
END;
```

e4: FROM *V0*, TO *V2*;

I-part: cr(source.ref, option, calling.t.addr, called.t.addr, max.tpdu.size)
C-part: cr.in.max.tpdu.size = nil and cr.in.option = ok
A-part:

```
BEGIN
remote.refer := cr.in.source.ref;
tpdu.size   :=...;
calling.t.addr := cr.in.calling.t.addr;
called.t.addr  := cr.in.called.t.addr;
qts.estimate   :=...;
tcind.out.to.t.addr   := called.t.addr;
tcind.out.from.t.addr := calling.t.addr;
tcind.out.qts.pro    :=qts.estimate;
out tcind(to.t.addr, from.t.addr, qts.pro);
END;
```

e5: FROM *V0*, TO *V0*

I-part: cr(source.ref, option)
C-part: cr.in.option <> ok

A-part:

```
BEGIN
dr.out.dest.refer :=cr.in.source.ref;
dr.out.disconnect.reason :=...;
out dr(dest.refer, disconnect.reason);
END;
```

e6: FROM *V1*, TO *V3*;

I-part: cc(max.tpdu.size) /* from peer entity */
C-part: cc.in.max.tpdu.size <> nil
A-part:

```
BEGIN
qts.estimate :=...;
tccon.out.qts.res := qts.estimate;
in.buffer.mark := empty;
out.buffer.mark := empty;
out tccon(qts.res);   /* to user */
END;
```

e7: FROM *V1*, TO *V3*;

I-part: cc(max.tpdu.size)
C-part: cc.in.max.tpdu.size = nil
A-part:

```
BEGIN
qts.estimate :=...;
tccon.out.qts.res :=qts.estimate;
in.buffer.mark := empty;
out.buffer.mark := empty;
out tccon(qts.res);
END;
```

e8: FROM *V1*, TO *V0*;

I-part: dr(disconnect.reason, add.clear.reason) /* from peer entity */
C-part: dr.in.disconnect.reason ="user.init"
A-part:

```
BEGIN
ndreq.out.disc.reason  := dr.in.disconnect.reason;
tdind.out.ts.disc.reason := dr.in.disconnect.reason;
tdind.out.ts.user.reason := dr.in.add. clear.reason;
out ndreq(disc.reason);   /* to network level */
out tdind(ts.user.reason,ts.disc.reason);   /* to user */
END;
```

e9: FROM *V1*, TO *V0*;

I-part: dr(disconnect.reason)
C-part: dr.in.disconnect.reason <> "user.init";
A-part:

```
BEGIN
ndreq.out.disc.reason   := dr.in.disconnect.reason;
tdind.out.ts.disc.reason := dr.in.disconnect.reason;
out ndreq(disc.reason);
out tdind(ts.disc.reason);
END;
```

e10: FROM *V2*, TO *V3*;

I-part: tcres(qts.req) /* from user */
C-part: tcres.in.qts.req <=qts.estimate
A-part:

```
BEGIN
in.buffer.mark := empty;
out.buffer.mark := empty;
local.refer    :=...;
cc.out.dest.refer := remote.refer;
cc.out.source.ref  := local.refer;
cc.out.calling.t.addr  := calling.t.addr;
cc.out.called.t.addr  := called.t.addr;
cc.out.max.tqdu.size  := tqdu.size;
out cc(dest.refer, source.ref, calling.t.addr, called.t.addr, max.tqdu.size);   /* to peer */
END;
```

e11: FROM *V2*, TO *V0*;

I-part: tcres(qts.req) /* from user */
C-part: tcres.in.qts.req > qts.estimate
A-part:

```
BEGIN
dr.out.dest.refer := remote.refer;
dr.out.disconnect.reason := ...;
dr.out.add.clear.reason := ...;
tdind.out.ts.disc.reason := ...;
out dr(dest.refer, disconnect.reason, add.clear.reason);   /* to peer */
out tdind(ts.disc.reason);   /* to user */
END;
```

e12: FROM *V2*, TO *V0*;

I-part: tdreq(qts.req) /*from user */
A-part:
BEGIN
dr.out.disconnect.reason := ...;
dr.out.add.clear.reason := tdreq.in.ts.user.reason;
dr.out.dest.refer := remote.refer;
out dr(dest.refer, disconnect.reason, add.clear.reason); /* to peer */
END;

e13: FROM *V3*, TO *V3*;
I-part : tdatr(tsdu.fragment) /* from user */

A-part:
BEGIN
insert(out.buffer, tdatr.in.tsdu.fragment); /* put into out.buffer */
out.buffer.mark := empty
END;

e14: FROM *V3*, TO *V3*;
C-part: out.buffer.mark <> empty
A-part:
BEGIN
remove (out.buffer, dt.out.user.data);
out dt(user.data); /* to peer entity */
out.buffer.mark := empty;
END;

e15: FROM *V3*, TO *V3*; /* from peer entity */
I-part: dt(user.data);
A-part:
BEGIN
insert(in.buffer, dt.in.user.data); /* put into in.buffer */
in.buffer.mark := full;
END;

e16: FROM *V3*, TO *V3*;
C-part: in.buffer.mark <> empty
A-part:
BEGIN
remove (in.buffer, tdati.out.tsdu.fragment);
out tdati(tsdu.fragment); /*to user */
in.buffer.mark := empty
END;

e17: FROM *V3*, TO *V0*;

I-part: tdreq(ts.user.reason); /* from user */
A-part:

```
BEGIN
ndreq.out.disc.reason := tdreq.in.ts.user.reason;
out ndreq (disc.reason);  /* to network */
END;
```

e18: FROM *V3*, TO *V0*;

I-part: ndind( ) /* from network */
A-part:

```
BEGIN
tdind.out.ts.disc.reason := ...;
out tdind (ts.disc..reason);  /* to user */
END;
```

e19: FROM *V3*, TO *V0*; /* network RESET */

I-part: nrind( )
A-part:

```
BEGIN
tdind.out.ts.disc.reason := ...;
out tdind (ts.disc..reason);  /* to user */
END;
```

REFERENCES

References

[ABC82] W. R. Adrion, M. A. Branstad and J. C. Cherniavsky, Validation, Verification, and Testing of Computer Software, *Computer Serveys 14*, 2 (June 1982), 159-192.

[AHU74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[ADU88] A. V. Aho, A. T. Dahbura and M. U. Uyar, An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours, Protocol Specification, Testing, and Verification, VIII, S. Aggarwal and K.K. Sabnani(Eds.), New York: Elsevier North-Holland, 1988.

[Arb69] M. A. Arbib, *Theories of Abstract Automata* , Prentice-Hall Inc. , 1969.

[BoS83] G. V. Bochmann and C. A. Sunshine, A Survey of Formal Methods, Computer Networks and Protocols, P.E. Green(Eds.) , New York: Plenum Press, 1983.

[Boc83] G. V. Bochmann, A Hybrid Model and the Representation of Communication Services , Computer Networks and Protocols, P.E. Green(Eds.) , New York: Plenum Press, 1983.

[BoB87] T. Bolognesi and E. Brinksma, Introduction to the ISO Specification language LOTOS , *Computer Network and ISDN System 14*, 1 (1987), 25-29.

[Brz62] J. A. Brzozowski, A Servey of Regular Expressions and Their Applications, *IRE Trans. on Electronic Computers*, , Jun. 1962, 324-335.

[BuD87] S. Budkowski and P. Dembinski, An Introduction to Estelle: A Specification Language for Distributed System, *Computer Network and ISDN System 14*, 1 (1987), 3-32.

[Che87] M. Chellappa, Nontraversible Paths in a Program, *IEEE Trans. Software Engineering SE-13*, 6 (Jun. 87), 751-756.

[Cho78] T. Chow, Testing Software Design Modeled by Finite State Machines , *IEEE Trans. on Software Engineering SE4*, 3 (1978), 178-187.

[CPR89] 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 15*, 11 (1989), 1318-1332.

[Dat87] R. R. Datar, Test Sequence Generation for Network Protocol, M.Sc. Thesis, Simon Fraser University, May 1987.

[FoO76] L. D. Fosdick and L. J. Osterweil, Data Flow Analysis in Software Reliability, *ACM Computer Serveys 8*, 3 (1976), 305-330.

[GMU76] H. N. Gabow, S. N. Maheshwari and L. J. Usternell, On Two Problems in the Generation of Program Test Paths, *IEEE Trans. Software Engineering SE-2*, 3 (Sept. 1976), 227-231.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability-A guide to the theory of NP-completeness*, Freeman, San Francisco, 1979.

[Gon70] G. Gonenc, A method for the Design of Detection Experiments , *IEEE Trans. Comput., C-19*, 6 (Jun. 1970), 551-558.

[Hai83] B. T. Hailpern, Specifying and Verifying Protocols Represented as Abstract Program , Computer Networks and Protocols, P.E. Green(Eds.) , New York: Plenum Press, 1983.

[HoU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[How76] W. E. Howden, Reliability of the Path Analysis Testing Strategy , *IEEE Trans. on Software Engineering SE-2*, 3 (Sept. 1976), 208-215.

[How80] W. E. Howden, Functional Program Testing, *IEEE Trans. on Software Engineering SE-6*, 2 (Jul. 1980), 21-29.

[ISO82] ISO/TC97/SC16, Transport Protocol Specification, ISO N1169, Jun. 1982.

[ISO84] ISO/TC97/SC21, Information Processing System-Open Systems Interconnection-Basic Reference model, ISO 7498 , 1984.

[ISO87a] ISO/TC97/SC21, OSI Conformance Testing Methodology and Framework-Part 1: General Concept, ISO 2nd DP Revised Text, Vancouver, Dec. 1987.

[ISO87b] ISO/TC97/SC21, A Formal Description Technique Based on an Extended State transition Model , ISO 9074 , 1987.

[Kni82] K. G. Knisghtston, The Transport Layer, *Computer Coomunication Review 3-4*, 12 (Jul. 1982), 14-67.

[LaK83] J. W. Laski and B. Korel, A Data Flow Oriented Program Testing Strategy, *IEEE Trans. Software Engineering SE-9*, 3 (May 1983), 347-354.

[MiH81] E. Miller and W. E. Howden, *Software Testing and Validation Techniques, IEEE Tutorial*, IEEE Computer Society Press, 1981.

[MKB83] J. L. Mott, A. Kandel and T. P. Baker, *Discret Mathematics for Computer Scientists* , Reston Publishing Company, Inc. , 1983.

[NaT81] S. Natio and M. Tsunoyana, Fault Detection for Sqquential Machine by Transition Tours, Proc. 11th IEEE Tault Tolerant Computing Symposium, 1981.

[Nta81] S. C. Ntafos, On testing with Required Elements , Proc. COMPSAC '81, IEEE Compt. Soc. , Nov. 1981.

[RaW81] S. Rapps and E. J. Weyuker, Dataflow Analysis Technique for Test Data Selection , Tech. Rept. No. 023, Dept. of Computer Science, New York University, 1981.

[RaW85] S. Rapps and E. J. Weyuker, Selecting Software Test Data Using Data Flow Information , *IEEE Trans. Software Engineering SE-11*, 4 (Apr. 1985), 367-375.

[Ray87] D. Rayner, OSI Conformance Testing , *Computer Network and ISDN System 14*, 1 (1987), 79-89.

[SaD88] K. Sabnani and A. Dahbura, A Protocol Test Generation Procodure , *Computer Network and ISDN System 15*, 4 (1988), 258-297.

[Sar84] B. Sarikaya, Test Design for Computer Network Protocol , Ph.D. Dissertation, McGill University , Mar. 1984.

[SaB86] B. Sarikaya and G. Bochmann, Obtaining Normal Form Specifications for Protocols , COMPUTER NETWORK USAGE: Recent Experiences , Elsevier Science Publishers B.V. (North-Holland) , 1986.

[SBG87] B. Sarikaya, G. Bochmann and E. Gcerny, A Test Design Methodology for Protocol Testing , *IEEE Transaction On Software Engineering SE-13*, 5 (May 1987), 518-531.

[ShS89] Y. N. Shen and F. Sabnani, Protocol Conformance Testing by Multiple UIO Sequences, Proc. 9th symp. on Protocol Specification, Testing and Verification, Twente, Jun. 1989.

[SiL89] D. Sidhu and T. K. Leung, Formal Methods for Protocol Testing: A detailed Study , *IEEE Trans. on Software Engineering SE15*, 4 (1989), 413-426.

[Stu73] L. G. Stucki, Automatic Generation of Self-metric Software , Rec. 1973 IEEE Symp. Software Reliability, IEEE Comput. Soc., 1973.

[Tuc84] A. Tucker, *Applied Combinatorics*, John Wiley & Sons Inc, 1984.

[Ura87] H. Ural, A Test Derivation Method for Protocol Conformance Testing , Protocol Specification, Testing, and Verification, VII, H. Rudin, C. West (Eds.), New York: Elsevier North-Holland, 1987.

[UYP88] H. Ural, B. Yang and R. L. Probert, A Test Selection Method for Protocol Specified in Estelle, Tech. Rept. No. 88-1, Dept. of Computer Science, University of Ottawa, Jun. 1988.

[WaK88] L. Wang and T. Kameda, A heuristic Method for Executable Path Problem , Internal Report, School of Computing Science, Simon Fraser University , March 1988.

[Whi87] L. J. White, Software Testing and Verification, Advances in Computers, M.C. Yovits (Eds.), 1987.

[Wu89] J. P. Wu and S. T. Chanson, A New Approach to Test Sequence Derivation Based on External Behaviour Expression, Tech. Rept. 89-3, Dept. of Computer Science, University of British Columbia, Jan. 1989.

[Zim80] H. Zimmermann, OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, *IEEE Trans. on Communication Comm-28*, 4 (Apr. 1980), 425-432.