# Test Sequence Generation For Network Protocols

by

Rajendra R. Datar

B.E. University of Jabalpur, M.Tech. I.I.T. Bombay

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Rajendra R. Datar 1987

SIMON FRASER UNIVERSITY

May 1987

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

# Approval

Name: Rajendra R. Datar

Degree: Master of Science

Title of Thesis: Test Sequence Generation For Network Protocols

Examining Committee:

Chairperson: Dr. Joseph Peters

Dr. Tiko Kameda
Senior Supervisor

Dr. Louis Hafer

Dr. Wo-Shun Luk

Dr. Son Vuong
External Examinar

May 28, 1987
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~~

Test Sequence Generation

For Network Protocols.

Author: ______________________

(signature)

Rajendra Dalal

(name)

8/13/87.

(date)

# Abstract

To establish confidence in software, some kind of testing is normally carried out. It involves presenting a set of test cases to the software and evaluating the output generated for appropriateness. Since it is not practical to present every possible input to the software under consideration, different methods have been developed to reduce the total number of test cases required for effective testing.

This thesis is concerned with testing products that implement computer communication protocols. We investigate methods to systematically generate test cases directly from their formal specifications in *Estelle*, a formal description technique which is based on the extended state transition model, being developed by the International Standards Organization (ISO).

We first discuss the transformations used in our algorithms. We then develop a general framework for generating different test cases. We also present another algorithm, based on network flow theory, to obtain "optimal" test cases. Conditions which must be satisfied for using this algorithm are discussed.

# Table of Contents


Approval ii
Abstract iii
Table of Contents iv
List of Figures vi
1. Introduction 1
 1.1. Importance of testing 2
 1.2. Previous work 3
 1.3. Motivation and goals of this thesis 4
 1.4. Assumptions 5
 1.5. Organization of the thesis 6
2. Background 7
 2.1. Extended State Transition Model 8
 2.2. Introduction to the Formal Description Technique 8
  2.2.1. System architecture 9
  2.2.2. Channel specification 9
  2.2.3. Module specification 10
 2.3. Test architecture using replacement scheme 12
 2.4. Test architecture using derail scheme 14
3. NFTs and Graphical Representations 17
 3.1. Normal Form Transitions 17
  3.1.1. Phase I 18
  3.1.2. Phase II 22
 3.2. The Control Graph 25
 3.3. The Dual Control Graph 25
4. Algorithm 28
 4.1. Parameter and value variations 29
 4.2. Parameter and value combinations 33
 4.3. Obtaining message instances 34
 4.4. Overview of the algorithm 35
 4.5. Major steps of the algorithm 36
 4.6. Procedure New_subtour 37
 4.7. Termination proof for New_subtour 38
 4.8. Procedure One_transition 38

5. Optimal algorithm 51
5.1. Overview of the algorithm 51
5.2. Weighted Dual Control Graph 52
5.3. Network 53
5.4. Listing all the required subtours in a network 53
5.4.1. Analysis of the algorithm 54
5.5. Network flow based algorithm 55
5.6. Comments 55
5.7. Example 56
6. Conclusion and Discussion 64
6.1. Contributions of this thesis 64
6.2. Comparison with Sarikaya's algorithm 64
6.3. Future work 66
6.3.1. Formal models 66
6.3.2. Test architectures 67
6.3.3. Program testing 67
Appendix A. Sarikaya's Algorithm 68
A.1 The Data Flow Graph 68
A.2. Partitioning of the Data Flow Graph 69
A.2.1 Stage 1 69
A.2.2. Stage 2 70
A.3. Sarikaya's Algorithm for generating test sequences 70
Appendix B. List of NFTs for Class 0 TP 72
Appendix C. Repetitive Executions of Self-loops 79
C.1. Connection establishment phase 84
C.2. Data transfer phase 84
C.3. Connection termination phase 84
Appendix D. Property of ISO Class 2 TP 85
D.1. Introduction 85
D.2. Connection establishment in C2TP 87
D.3. Proof 88
References 93

# List of Figures


| | | |
|---|---|---|
| **Figure 2-1:** | Test architecture using replacement scheme | 13 |
| **Figure 2-2:** | Test architecture using derail scheme | 15 |
| **Figure 3-1:** | A protocol entity with three modules | 20 |
| **Figure 4-1:** | Control Graph CG00 | 44 |
| **Figure 4-2:** | Control Graph CG11 | 45 |
| **Figure 4-3:** | Control Graph CG12 | 46 |
| **Figure 4-4:** | Control Graph CG21 | 47 |
| **Figure 4-5:** | Control Graph CG22 | 48 |
| **Figure 5-1:** | Control Graph for ISO Class 0 TP | 58 |
| **Figure 5-2:** | Dual Control Graph for ISO Class 0 TP | 59 |
| **Figure 5-3:** | Control Graph for COTP with I/O labels | 61 |

# Chapter 1
# Introduction

As computer networks proliferate, their protocols become diverse and complex. Problems of

- designing **logically correct** protocols,
- **specifying** them precisely so they can be implemented as intended by the designer,
- **testing** protocol implementations for compliance with a specification or a standard, etc.,

are becoming important. For designing a *logically correct* protocol, there are a number of general conditions, applicable to nearly all protocols. These include [Suns 79]:

- Freedom from **deadlock**,
- **Completeness** of the protocol to handle all conditions that may arise,
- **Stability** or **self synchronization**, namely the ability to return to "normal" behavior after an initial or temporary aberration [MeFa 76],
- **Progress**, or the absence of cyclic behavior in which no useful activity takes place,
- **Termination**, or arrival at the desired final state.

There are a number of ways in which protocols can be *specified*. They include flow charts, programming languages, state diagrams, and Petri nets. Generally, they may be based either on the transition technique or on the programming language technique. These techniques and their merits and demerits are discussed elsewhere [Suns 79]. In any case, formal specification is preferred to informal specification (e.g., in natural languages) for many reasons. For instance, it facilitates precise specification and verification of the protocol. The International Standards Organization

(ISO) is developing a hybrid technique, called the **Formal Description Technique** (**FDT**, for short), alias **Estelle**. A brief description of the FDT is included in this thesis.

In *protocol testing*, an actual implementation, called **implementation under test** (**IUT**, for short), of the protocol is exercised with selected sequences of inputs (called **test sequences**), for the purpose of comparing its behavior with the desired behavior specified in the specification.

This thesis is concerned with systematically generating test sequences from a formal specification of a protocol. It is assumed that the underlying system has a layered architecture and protocols are specified using Estelle. Initially we perform certain transformations on the original specification to obtain an equivalent specification which is more convenient for generating test sequences. Then we obtain a global finite state model with auxiliary variables. Using this model, we generate test sequences needed to cover all representative situations.

## 1.1. Importance of testing

Different approaches have been and are still being investigated to verify the correctness of computer programs. For example, efforts have been made to make "formal proof of correctness" a viable approach to software validation. Unfortunately, ingenuity required and the amount of work involved make this approach impractical at the present time.

Currently, testing is the most common, widely accepted, and practical method of establishing confidence in software. A **test** (or **test suite**) of software is defined to be a set of **test cases**. A test case is an instance of presenting test data to the

software. The resulting output is then compared with the expected output. Testing may find some flaws, but it cannot guarantee that all errors are detected, since it is not practical to apply every possible input. For example, exhaustive testing of a program containing two inputs, each 16 bits long, and requiring 1 millisecond to produce the output, would require more than 13 years of CPU time. Therefore, it is important to select a small set of test cases, capable of revealing most of the possible errors [Chan 81].

In this thesis we first propose guidelines for selecting test cases and "cost" associated with testing. We then propose two schemes to generate test sequences. The first scheme is more general, but the test it generates is not very "efficient", in the sense that it may send many redundant messages. The second scheme generates a minimum "cost" test, but is applicable to only those protocols which satisfy some restrictive set of assumptions.

## 1.2. Previous work

In this section we describe previous work on protocol testing and comment on each method.

1. Early efforts on test design were all based on Finite State Machine (FSM) models of the specification. In the testing scheme of Hanley and Rayner [HeRa 81] the objective was to "cover" each transition of the FSM [NaTs 81]. A sequence of transitions that includes all the transitions of an FSM is called a *transition tour* [NaTs 81]. The implementation is driven into a desired state and an input is applied and the output is observed. Merely checking all the transitions will reveal very few errors. In fact most current methods execute each transition with different values (for the fields of a message) for improved fault detection capability.

2. Ural and Probert [UrPr 83, UrPr 84] first derive a grammar describing the actions of the specification. They have written a program in Prolog that accepts the grammar and generates test sequences. In this method,

the problem of writing a Prolog program to accept any grammar (or a grammar derived from an arbitrary underlying FSM) to derive test sequences remains to be investigated.

3. If a correct implementation (called *reference implementation*) of layer N is available, then a method developed by Linn and Nightingale [LiNa 83] is applicable. A transition tour consisting of user (i.e., layer N+1) interactions is generated by considering the FSM for the service specification of layer N.

4. A variant of the above method is described by Linn and McCoy [LiMc 83]. A service specification is converted into two grammars (called user entity grammars) for the two peer entities. Then a composite grammar is derived which consists of 4-tuples (requests and indications of both entities). A test sequence is then generated starting with the initial state and terminating when the final state is reached. In this method the robustness of the IUT to handle the received peer level errors cannot be checked. Therefore, some modifications are required at the test site. The modifications to the peer level messages are carried out by an "exception generator" which interfaces with the reference implementation. Another problem is that some method must be devised to remove duplicate test sequences. Typically, 1/3 of the test sequences are duplicate sequences [LiMc 83].

5. ISO Transport Layer Protocols (Class 0 and Class 2), initially specified in a natural language, were subsequently converted into a formal specification in Estelle by ISO [ISOa 82]. Sarikaya [Sari 84] developed a scheme for test sequence generation from a specification in Estelle [see Appendix A].

Our point of departure is Sarikaya's work mentioned above.

## 1.3. Motivation and goals of this thesis

Initial work on test sequence generation for computer communication protocols was reported in the literature as early as 1980 [Kawa 80]. At that time, the specifications were described in natural languages. With the growing complexity of protocols, a need to formally describe them was recognized and different formal models began to appear in the literature. ISO took up the challenge and developed a

model. The model is still undergoing revisions, and therefore in this thesis the model given in [ISOb 84] is used. Initial work on automatically generating test sequences based on this model was carried out by Sarikaya [Sari 84]. His method decomposes a protocol into control and data flow functions (represented as the *control graph* and the *data flow graph*, respectively). Test sequences are then generated using these graphs. The major goal of this thesis is to devise a scheme which has some advantages over Sarikaya's scheme. We will considerably simplify his method and present two algorithms for generating test sequences.

## 1.4. Assumptions

In general, testing is based on certain assumptions. Some assumptions are independent of the method used in testing. We will refer to these as *common assumptions*. Additional assumptions are also made, depending on the specific method used. We list below these two types of assumptions.

*Common Assumptions:*

1. The specification is logically correct.[1]
2. The source listing of the IUT is not available.
3. Correctness of the implementation is not dependent on the system load at the product site.

*Method-specific assumptions:* All the additional assumptions are listed below. Some technical terms in the list not defined at this point, will be defined later in this thesis when appropriate. We invite the reader to return to this section after reading Chapter 3.

---

[1]The task of proving a protocol logically correct is called protocol *validation* or *verification*. It is an important task, but is separate from protocol testing, which is the topic of this thesis.

1. The type of queue discipline between modules is of *rendezvous* type. (See Section 2.2.3.)
2. No BEGIN block in the protocol contains loops with variable bounds. (See section 3.1.1.)
3. The functions and the procedures in the specification are not recursive. (See section 3.1.1.)

## 1.5. Organization of the thesis

In this chapter (Chapter 1), we have already described the motivation for testing products that implement communication protocols, and presented a brief description of the previous work carried out in this field. We have also stated assumptions that we make in generating test sequences. In Chapter 2, the formal model to be used throughout this thesis is introduced. Two architectures that are useful for testing are described, and their advantages and disadvantages are compared. Chapter 3 discusses the transformations which are applied to the original formal specification, to obtain an equivalent specification in terms of **Normal Form Transitions**. The actual algorithms for test sequence generation are described in Chapter 4 and Chapter 5. For illustration, we apply the second algorithm to generate a set of test sequences for ISO Class 0 Transport Protocol. In Chapter 6, we summarize the major contributions of this thesis. We also point out those areas in test sequence generation where further research work is needed.

In Appendix A, we present a brief description of Sarikaya's algorithm, and in Appendix B, we list the Normal Form Transitions of ISO Class 0 Transport Protocol. These are used in Chapter 5 to derive test sequences. In Appendices C and D, we prove that the algorithm presented in Chapter 4 is applicable to ISO Class 2 Transport Protocol.

# Chapter 2
# Background

In order to make interconnection of (heterogeneous) computers possible and convenient, it is necessary to provide a reliable data transportation service between end processes and to support a meaningful communication between them. In order to provide these services at a reasonable cost, and at the same time reduce the design complexity, most networks are organized as a series of **layers**. Typically, a layer (N) provides a layer (N) service to the (next higher) layer (N+1) using the services of (next lower) layer (N-1). A concrete example of such a layered architecture is the **Reference Model** developed by ISO [Zimm 80].

To test an IUT which is the layer (N) in a product, (assuming that lower layers are already tested and determined to be correct), at least two interacting and co-operating modules are required, one located at the tester site and the other located at the product site. Placement of the testing modules at particular layers at the test site and the product site gives rise to different **test configurations** or **test architectures**. In this chapter, we describe the formal description technique and two well known test architectures. The advantages and disadvantages of the two architectures are then discussed. A detailed discussion on merits and demerits of different architectures can be found in [Kawa 80].

## 2.1. Extended State Transition Model

In the past, Finite State Machine (FSM) models have been used quite successfully in the specification of simple protocols. When applied to more complex protocols, however, the FSM model becomes impractical because of a very large number of states required. This is called the "state space explosion" problem [Boch 80]. For example, the use of sequence numbers introduces a different state for each possible sequence number.

The **Extended Finite State Machine (EFSM) model** attempts to combine the advantages of state transition technique and programming language technique [Simo 82]. It is called *extended*, since it may contain variables (called minor variables) in addition to a finite state machine. In a (conventional) FSM, the occurrence of an event causes a state transition. For a transition to take place in an EFSM, some conditions associated with minor variables must be satisfied, in addition to the occurrence of an appropriate event. So, depending on the values of the minor variables, different transitions could occur in response to the same event. Moreover, a state transition may alter the values of minor variables as well as producing external outputs. This concept of minor variables has been borrowed from programming languages.

## 2.2. Introduction to the Formal Description Technique

This section briefly describes the FDT called Estelle, being developed by ISO [ISOa 82, ISOb 84], for the specification of communication protocols and services. The formal specification is based on an extended finite state transition model and the Pascal programming language.

### 2.2.1. System architecture

A system is defined by a set of **interacting modules.** Interacting modules need

- to be able to receive and send interactions (i.e., inputs/outputs) from/to its environment, and
- to be interconnected to other modules to/from which they may send/receive interactions.

The point at which a module interacts with its environment is called an **interaction point.** Each interaction of the specified module with its environment can be considered as an **atomic event.** The reception of an interaction (input) from the environment produces, in general, a state transition of the specified module, which may also give rise to other interactions (outputs).

If two modules are interconnected through a pair of interaction points (one for each module), they may exchange only a given set of interactions. Therefore, the concept of **channel type** is introduced, which defines a set of interactions to be exchanged. The channel type also defines the **role(s)** (**server** or **user**) that modules using this channel type are to play with respect to these interactions.

### 2.2.2. Channel specification

In the specification of a module, each interaction point of a module is characterized by making reference to a channel type and a role within this channel type. As an example of a channel type definition, consider a channel that connects some module to a timer service module. A possible definition might be:

```
CHANNEL
   timer_interface (user, server)

        BY user:
                start (period: integer);
                stop;
        BY server:
                time_out;
```

### 2.2.3. Module specification

A protocol is defined in terms of a collection of modules. The purpose of a module specification is to define the behavior of the module as *observable at the interaction points* to which it is connected.

The model allows output interaction of a module to be queued before it is considered as an input to other module. It is possible to have queues of infinite, finite or zero length. If the queue length is zero, then the interaction is said to be of **rendezvous type**. Moreover, it is also possible to choose different queue disciplines for different interaction points. An interaction point may use its own INDIVIDUAL QUEUE, or interaction points may share a COMMON QUEUE. All interaction points that specify COMMON QUEUE share the same queue.

The actions of each module are defined by the transitions of an EFSM, called a **protocol machine.** The complete "state" of the protocol machine is characterized by the values of the "major state variable", STATE, and the minor variables. The variable STATE indicates the state of the underlying finite state machine.

Each transition is characterized by:

- **enabling condition:** It consists of three parts. A transition is executed only when the condition associated with each part is satisfied. The three parts are:

- The interaction specified in the WHEN clause is received.
- The underlying FSM is in one of a set of states specified by the FROM clause.
- A boolean expression specified in the PROVIDED clause and consisting of minor variables is true.

Each part is optional and if unspecified it is assumed to be satisfied.

- **operation:** It may change the values of the variables. It consists of two parts.

  - The TO clause specifies the new state of the underlying FSM (i.e., the value of STATE is changed).
  - A BEGIN block (delimited by BEGIN ... END) specifies a set of statements in Pascal language to be executed. (These statements change the values of the minor variables.) The BEGIN block may also contain OUT clauses that generate interactions with the environment.

As an example, consider a possible description of a timer module. The timer module is connected to some module (that requires the service of the timer module) by a channel "timer_interface" whose description was given in section 2.2.2. Only one transition is listed in the specification given below.

```
MODULE timer (s: timer_interface (server) INDIVIDUAL QUEUE)
var
      :
      STATE             : (timer_on, timer_off);
      no_of_timers, max: integer;
      :

 STATESET
      either = [timer_on, timer_off];

 INITIALIZE
      begin
           :
           no_of_timers := 0;
           max := ....;     /* Implementation Dependent */
           STATE TO timer_off;
           :
      end

TRANS
        :
        :

        FROM     either
        TO       timer_on
        WHEN     s.start
        PROVIDED no_of_timers < max
          begin

                 no_of_timers  = no_of_timers + 1;
                 :
                 :        (* Some statements. *)
                 :
          end;
        :
        :                 (* Other transitions. *)
        :
end;
```

Currently, Estelle is being refined jointly by ISO and International Telegraph and Telephone Consultative Committee (CCITT).

## 2.3. Test architecture using replacement scheme

Let the IUT be at layer (N). In this architecture [Rayn 81], at the tester site all layers above layer (N-1) are removed and replaced by two modules. A module, called an **active tester (AT),** that can generate not only normal protocol messages but also abnormal messages, operates at layer (N). Another module, called a **test driver (TD),** that provides the data to the AT operates at layer (N+1).

At the product site, all layers above layer (N) are also removed. A special module, called a **test responder (TR)**, that co-operates with the AT and TD is implemented and operates as layer (N+1).

This architecture is illustrated in Figure 2-1. In some implementations, functions of the AT and TD are combined into one module and the resulting module operates as layer N, simulating layer (N+1).

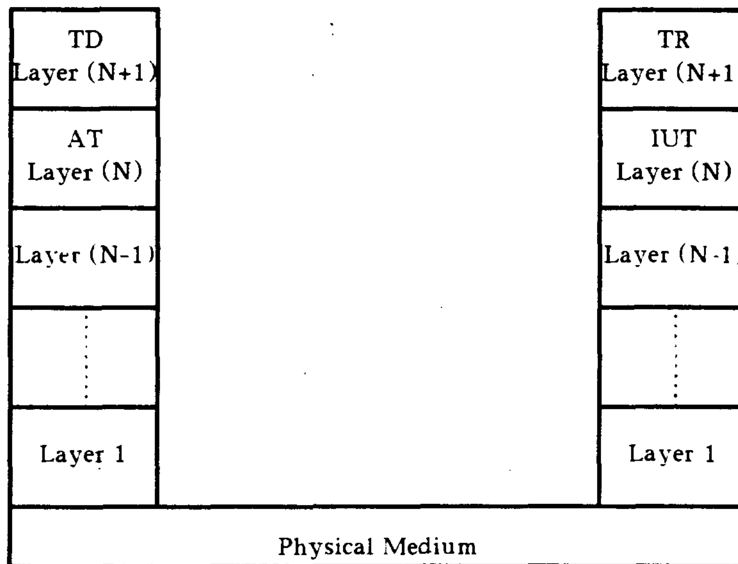


**Figure 2-1:** Test architecture using replacement scheme

As all layers above the IUT are absent in a product, it is possible to develop and test one layer at a time in a bottom up fashion. This is specially important when a product is being developed, since products are typically developed one layer at a time. At the product site, the TR operates as layer (N+1) directly interacting with the IUT. Thus it can initiate any interaction with the IUT. Therefore, it can easily and efficiently generate abnormal sequences.

The advantages of this architecture are:

- Abnormal sequences can be easily generated and, therefore, robustness of the IUT can be easily checked.
- It is efficient.
- It is possible to use test site facilities during product development stage.

The disadvantages of this architecture are:

- For each product, the TR must be reliably implemented for each layer.
- It is difficult to construct a test system since
    - For each layer under test, a new TD must be used.
    - For each layer under test, a new AT must be used.
- Concurrent testing of several layers is impossible.

## 2.4. Test architecture using derail scheme

In this architecture [Kawa 80, Yosh 82], there are two kinds of modules at the tester site. Modules that implement normal protocol processing (called **PP modules**), hence operating as normal layer implementation, and modules that generate abnormal sequences (called **derail modules**) and are executed at the (inter layer) interfaces.

This architecture is shown in Figure 2-2. We show only a few layers and the derail modules at the tester site. At the tester site, all communication between each pair of adjacent layers passes through a derail module. Depending on the type of testing, a derail module can be **passive** (the communication between the adjacent layers is unaltered) or **active** (the communication between the adjacent layers may be altered). Active modules could alter the communication (control information or data being exchanged) in one of the following ways:

- Insert new information (in either direction).
- Delete some passing information.
- Change some part of the passing information.

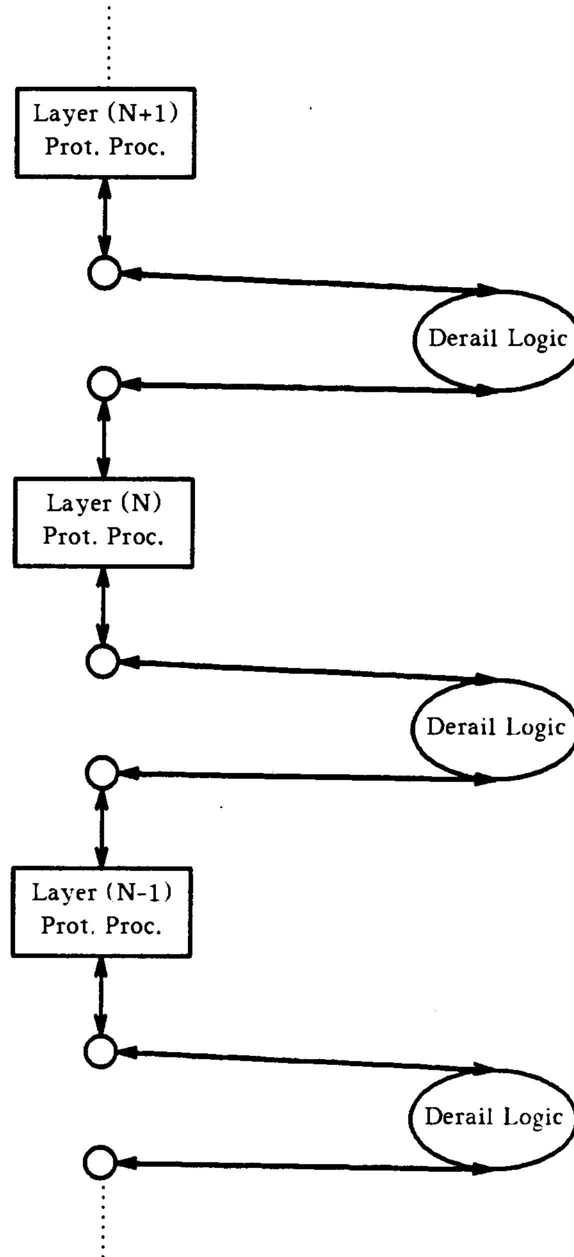


**Figure 2-2:** Test architecture using derail scheme

At the product site, no derail module is used and the TR operates as an application layer (highest layer) entity co-operateing with the tester in testing a product.

For testing a single layer (N), the derail module between layer (N) and layer (N-1)

is made active. If several (N) connections are multiplexed into one (N-1) connection, several derail modules (one per (N) connection) could be activated.

Since the TR cannot directly initiate interactions with the IUT, it uses the services provided by the IUT indirectly through intervening layers. Thus it is difficult and therefore inefficient to generate service errors. However, this scheme does not require any particular extension (either hardware or software) in a product.

The advantages of this architecture are:

- It is easy to construct a test system.
- It is easy to construct the TR since it is an application layer entity at the product site.
- Concurrent testing of several layers is possible.

The disadvantages of this architecture are:

- It is not possible to generate all abnormal sequences.
- It is less efficient compared to replacement scheme.

It may be noted that the algorithms to be described in Chapters 4 and 5 do not assume any particular architecture. Since each architecture has certain advantages, what really changes with the architecture is the number of messages and the number of ways a message can be exchanged between the product and the tester. Therefore, our algorithm will be applicable to both the architectures described above.

# Chapter 3

# NFTs and Graphical Representations

For testing purposes, it is useful to have a global representation of "control flow". In this chapter we develop some transformations on specifications in Estelle. These transformations are then used to obtain a single control graph (see Section 3.2 below), representing the major state transitions in a given formal specification in Estelle.

## 3.1. Normal Form Transitions

Working with an original specification in Estelle has three major drawbacks:

- A formal specification may contain many modules. Representing the global state space as the cross product of the state spaces of the modules will give rise to a large number of unnecessary global states that are never reached.
- These modules may generate local interactions, i.e., interactions that are not received by the environment. These internal interactions are not "visible". Therefore, the tester cannot know the specific action taken by the IUT.
- The BEGIN block of a transition may contain Pascal conditional statements (if-then, if-then-else, case). These conditional statements cause different parts of the BEGIN block of the transition to be executed. Thus a transition may assign different values to minor variables depending on the condition. Subsequent transitions may also be affected, since the PROVIDED clause of a transition contains minor variables. These conditional statements must somehow be removed from the BEGIN block, if the tester must know the exact values of the minor variables.

To get around the above difficulties, we apply certain transformations to the original specification, to obtain the **normal form transitions** (NFTs) [Sari 84]. The advantages of the NFT are that the specification of each transition has no conditional

statement in the BEGIN block, and that only one control graph is obtained for the entire specification.

An NFT has (optional) inputs from external interaction points and a BEGIN block with a single control path containing one or more OUT clauses specifying outputs to the external interaction points. All conditions appear in the PROVIDED clause.

We use the **symbolic execution** technique [King 76, ClRi 81] to determine all transition paths. In symbolic execution, symbolic values (e.g., x1, x2, ...) are assigned to the input variables of a program, and the program is then scanned, generating an "execution tree". Each node in the execution tree corresponds to a statement in a program, and arcs indicate the sequence of execution. For example, a condition, "if x1", generates two arcs labeled by "x1 = T (true)" and "x1 = F (false)". The condition which must be satisfied for a particular path to be executed is expressed as an expression using symbolic values of the input variables.

The transformations leading to NFTs are performed in two phases. An example is included to illustrate how such transformations can be applied. More details are available in [Sari 84].

### 3.1.1. Phase I

In phase I, two syntactic transformations are applied to constructs in Estelle. In the first transformation given below, we create a new transition for every distinct path in the BEGIN block of a transition and modify the PROVIDED clause to reflect the conditions for taking these paths.

**Transformation 1 :** For each transition, apply symbolic execution to enumerate the

paths in the BEGIN block and their associated path conditions (i.e., the conditions under which each path will be executed). Replace each procedure (function) call in the BEGIN block by the body of the procedure (function) itself with appropriate changes to the parameters of the procedure (function).

**Transformation 2 :** If the FROM clause has more than one state as the parameter, then create a new transition for each state. These newly created transitions are identical, except for the FROM clause, and they each have only one state in its FROM clause.

As an example, consider a layer with three modules shown in Figure 3-1 (viz., Module VC, Module DG and Module LL). Module VC and Module DG are connected to Module LL via two internal channels, called Int_chan1 and Int_chan2, respectively

The specifications for the transitions of the modules are given below. (Here S1, S2, and S3 are assignment statements and A, B, C, and D are boolean expressions).

```
                  (* Module VC *)
                  (* --------- *)

WHEN      Chan1.CONNECT_req
FROM      idle
TO        connecting
PROVIDED  A and (B or C)

BEGIN
          if D then S1 else S2;
          OUT Int_chan1.eventVC (CONNECT_req, x+2)
END;
```

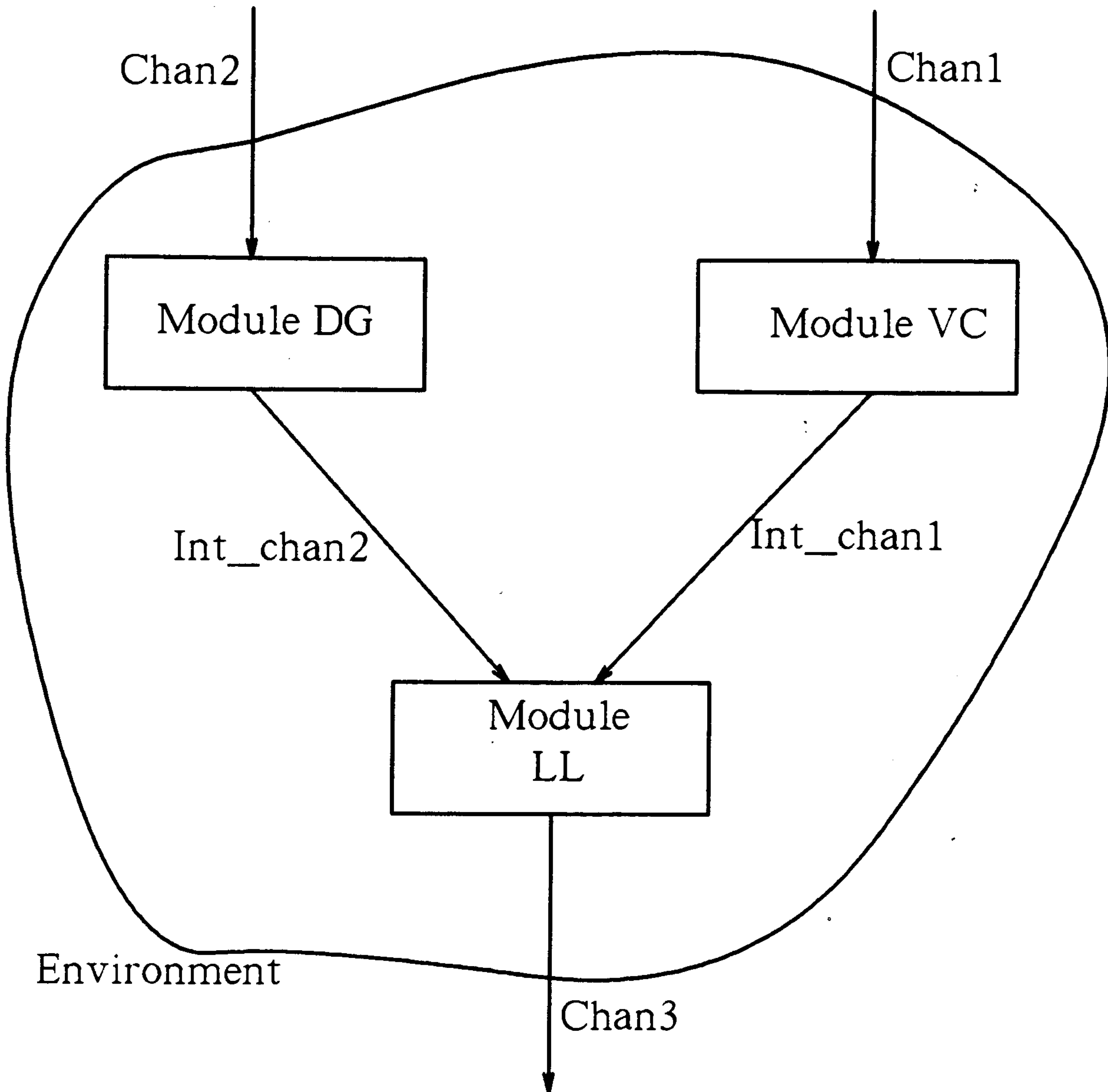


**Figure 3-1:** A protocol entity with three modules

```
            (* Module DG *)
            (* ---------- *)

WHEN      Chan2.CONNECT_req
FROM      idle
TO        connecting
PROVIDED A

BEGIN
          OUT Int_chan2.eventDG (CONNECT_req, DGlength)
END;
```

```
                    (* Module LL *)
                    (* ————————— *)

WHEN     Int_chan1.eventVC (data_unit, length)
PROVIDED (data_unit = CONNECT_req) and (length <= 10)

BEGIN
         S3;
         OUT Chan3.CONNECT_indVC
END;
WHEN     Int_chan2.eventDG (data_unit, length)
PROVIDED (data_unit = CONNECT_req) and (length <= MAXDG)

BEGIN
         OUT Chan3.CONNECT_indDG
END;
```

Module LL has only one state, hence FROM and TO clauses are missing. We apply phase I transformations to the modules of Figure 3-1 to obtain equivalent transitions. Since only Module VC has a conditional statement, we show only the application of the phase I transformations to Module VC. Two transitions are generated as a result. Other modules are unaltered.

```
                    (* Module VC *)
                    (* ————————— *)

WHEN     Chan1.CONNECT_req
FROM     idle
TO       connecting
PROVIDED A and (B or C) and D

BEGIN
         S1;
         OUT Int_chan1.eventVC (CONNECT_req, x+2)
END;

WHEN     Chan1.CONNECT_req
FROM     idle
TO       connecting
PROVIDED A and (B or C) and (not D)

BEGIN
         S2;
         OUT Int_chan1.eventVC (CONNECT_req, x+2)
END;
```

The transitions resulting from phase I transformations are called the *intermediate transitions* (i_trans, for short).

### 3.1.2. Phase II

A module may receive input interactions and generate output interactions. If an interaction is received or sent via an interaction point defined locally within a specification, i.e., if it is an intermodule interaction, then it is called an *internal interaction*. On the other hand, if an interaction is received/sent from/to the environment (via an external interaction point), it is called an *external interaction*. Thus depending on the type of interaction, we can classify i_trans into four different groups.

*out-out i_trans:* This type of transition receives interactions from external interaction point(s) and sends interactions to external interaction point(s).

*out-in i_trans:* This type of transition receives interactions from external interaction point(s) and sends interactions to internal interaction point(s).

*in-out i_trans* This type of transition receives interactions from internal interaction point(s) and sends interactions to external interaction point(s).

*in-in i_trans:* This type of transition receives interactions from internal interaction point(s) and sends interactions to internal interaction point(s).

Those i_trans which do not receive any interaction can be assumed to receive a dummy external interaction, and they are called **spontaneous transitions** [ISOb 84].

Since we are interested in test sequence generation, and since only the interactions at the external interaction points can be observed, the internal interactions are removed as follows.

Each in-out i_trans is removed using **in-line expansion technique**[2], i.e., by substituting its BEGIN block in all the out-in and in-in i_trans (which we call *recipients*) that output the internal interactions to it. This (transformations 3 and 4 given below) is repeated till the specification consists entirely of out-out i_trans. The out-out i_trans thus obtained are called **normal form transitions** (or NFTs).

**Transformation 3 :** AND the state in the FROM clause (TO clause) of an in-out i_trans with the state in the FROM clause (TO clause) of the recipient in-in or out-in i_trans, and rename it suitably to represent a new combined state.

A recipient i_tran will generate k i_trans, if there are k i_trans that have the same input interaction.

**Transformation 4 :** AND the PROVIDED clause of the recipient in-in or out-in i_trans with the PROVIDED clause of the in-out i_trans. Merge the BEGIN block of the in-out i_trans with the BEGIN block of the recipient in-in and out-in i_trans using the in-line expansion technique. Then delete the in-out i_trans.

The effect of applying transformations 3 and 4 is that some of the in-in i_trans become in-out i_trans and some of the out-in i_trans become out-out i_trans. Some of the in-out i_trans are eliminated. When the transformations 3 and 4 cannot be applied any more, all the i_trans are of type out-out.

Note that the maximum number of states generated due to transformation 3 and 4 equals the product of the numbers of the states of the modules. However, for

---

[2]This refers to the replacement of a call to a procedure by the body of the procedure itself.

practical protocols the number of states generated is far less than the maximum number.

The result of applying the phase II transformations to the above example is shown below:

```
                (* Module VC + Module DG + MODULE LL *)
                (* ------------------------------------ *)

WHEN     Chan1.CONNECT_req
FROM     idleVC
TO       connectingVC
PROVIDED A and (B or C) and D and (x <= 8)

BEGIN
         S1;
         S3;
         OUT Chan3.CONNECT_indVC
END;

WHEN     Chan1.CONNECT_req
FROM     idleVC
TO       connectingVC
PROVIDED A and (B or C) and not D and (x <= 8)

BEGIN
         S2;
         S3;
         OUT Chan3.CONNECT_indVC
END;

WHEN     Chan2.CONNECT_req
FROM     idleDG
TO       connectingDG
PROVIDED A and DGlength <=MAXDG

BEGIN
         OUT Chan3.CONNECT_indDG
END;
```

The only difference between the NFTs discussed in this thesis and in [Sari 84] is that in Sarikaya's work

- the FROM clause is also removed and the PROVIDED clause is modified to reflect this condition, and
- the TO clause is also removed and in the BEGIN block an assignment statement is added to reflect this condition.

Since we do not remove FROM and TO clauses, no extra effort is required to construct the control graph (discussed below).

## 3.2. The Control Graph

Informally, the **control graph (CG)** [Sari 84] depicts the transitions among the "control states". The nodes of a CG represent the values the major state variable STATE can take. The arcs of the CG represent the NFTs. For each NFT specified, the FROM clause gives the tail node of the arc and the TO clause gives the head node of the arc.

It is possible to construct CG's, one for each module, from a specification in Estelle. However, we construct a single CG for the entire collection of modules from the NFTs for the reasons mentioned in Section 3.1.

## 3.3. The Dual Control Graph

The protocol specification, and hence the CG, specifies the action to be taken when there is a protocol error, but does not specify how such an error could be generated. It is possible to transform a CG into another finite state machine called the **dual control graph (DCG)** [Arak 83]. The DCG is a *dual* of the CG in the sense that it imitates the behavior of the CG, generating an output whenever the CG expects an input. In particular, it generates errors if the specification of the CG has provisions for them. Therefore, a tester can make use of the DCG to communicate with an IUT, sending some errors as appropriate.

Normally, a protocol specification has transitions to handle unexpected input messages. An unexpected input message may, for example, be received due to delayed duplicates or the other peer entity not confirming to the protocol specification. From

the procedure that constructs the DCG corresponding to a given CG, it will be seen that input and output symbols of the transitions are interchanged. Thus the unexpected input message now appears as an output message of the DCG that is to be transmitted to the IUT.

A formal procedure to obtain a DCG from a CG is given below.

The input and output symbols of the CG can be divided into two groups.

- **External Symbols :** The messages sent or received between peer entities at different sites.
- **Internal Symbols :** The messages that are exchanged between layers at one site. All internal symbols are denoted by (a common symbol) *.

Let i and o denote an external input symbol and an external output symbol, respectively. The transitions of the CG can be classified into four types, depending on the group their inputs/outputs belong to, namely, */*, i/*, */o, i/o

Let a CG be denoted by $(Q_1, I_1, O_1, \delta_1, \omega_1, q_{10})$, where

- $Q_1$ is the set of states.
- $I_1$ is the set of external input symbols $\cup$ {*}.
- $O_1$ is the set of external output symbols $\cup$ {*}.
- $\delta_1$ is a mapping $Q_1 \times I_1 \rightarrow Q_1$.
- $\omega_1$ is a mapping $Q_1 \times I_1 \rightarrow O_1$.
- $q_{10}$ belongs to $Q_1$ and is the initial state.

Let a DCG be denoted by $(Q_2, I_2, O_2, \delta_2, \omega_2, q_{20})$, where

- $Q_2$ is the set of states.
- $I_2$ is the set of external input symbols $\cup$ {*}.
- $O_2$ is the set of external output symbols $\cup$ {*}.
- $\delta_2$ is a mapping $Q_2 \times I_2 \rightarrow Q_2$.

- $\omega_2$ is a mapping $Q_2 \times I_2 \rightarrow O_2$.
- $q_{20}$ belongs to $Q_2$ and is the initial state.

Now, from a CG, the corresponding DCG is obtained as follows [Arak 83].

1. For each q in Q1, introduce $\overline{q}$ in Q2.
2. For each transition in the CG, say from p to q, labeled $i_k/o_l$, one or two transitions in the DCG are obtained as follows:

   a. if $i_k/o_l$ is of type */* then DCG has a transition from $\overline{p}$ to $\overline{q}$ labeled */*.
   b. if $i_k/o_l$ is of type i/* then DCG has a transition from $\overline{p}$ to $\overline{q}$ labeled $*/i_k$.
   c. if $i_k/o_l$ is of type */o then DCG has a transition from $\overline{p}$ to $\overline{q}$ labeled $o_l/*$.
   d. if $i_k/o_l$ is of type i/o then a new state, say $q'$, is added to Q2. DCG has two transitions
      (i) from $\overline{p}$ to $q'$ labeled $*/i_k$ and
      (ii) from $q'$ to $\overline{q}$ labeled $o_l/*$.

3. The DCG contains only those states and transitions that are obtained in the above steps.

We note the following.

1. The set of input symbols (output symbols) of a CG and the set of output symbols (input symbols) of the corresponding DCG are the same.
2. There exists a one-to-one function from the state set of a CG into the state set of the corresponding DCG. (However, the DCG may have more states than the CG, due to the step 2d above.)

It is noted that, in case the robustness of the protocol to errors is not being tested, one can use the CG instead of the DCG. The example in chapter 5 shows how to obtain the DCG from a CG. (See Figures 5.2 and 5.3.)

# Chapter 4
# Algorithm

In this chapter we propose a new algorithm for test sequence generation, which makes use of the DCG (or CG) and NFTs.

Since protocol specification often uses a *finite state machine* (FSM) model, the testing of sequential circuits is relevant to protocol testing. A sequential circuit can be specified by means of a *state table* or *transition diagram* [Koha 78]. For the purpose of testing it is commonly assumed that the state table is reduced, completely specified, and strongly connected [Gone 70]. It has been claimed that if every transition is traversed at least once by the input test sequence, then typically more than 90% of the errors can be detected [NaTs 81]. However, for 100% fault detection, one must apply the distinguishing sequence [Koha 78] twice[3] for every state [Gone 70]. Executing all the transitions is the basis of many protocol testing strategies, including Sarikaya's [Sari 84] and ours.

In order to test a transition corresponding to an arc of the DCG, we must start from an idle state. Let a **subtour** denote an executable sequence of transitions that starts and ends in an idle state. Briefly, our objective is to select a set of subtours such that each arc of the DCG is on at least one of the subtours. Each transition,

---

[3]Testing by means of just traversing each transition has reduced fault detection capability, mainly because after each transition it is not verified if the transition terminated in a the correct state.

represented by an arc in the DCG, is usually caused by the reception of a message from, or causes the transmission of a message to, the peer layer or adjacent layer. Each message associated with a transition has several **fields**, and each field can be assigned several values (called **field values**). To increase the fault detection capability of our algorithm, we exchange similar messages with different field values. There are different **types** of messages. Each type of message is specified by a particular structure of its fields. A particular set of field values for a message of certain type is called a **message instance** (**MI**, for short).

The algorithm to be presented in this chapter does not optimize the "cost" (e.g., the total number of transitions executed) of testing, since to do so is computationally intractable

## 4.1. Parameter and value variations

We consider each field of a message to be of **scalar data type** [JeWi 74]. In this section, we propose guidelines as to how the contents of the fields in test messages might be varied (parameter or value variation). Using these guidelines, a test designer can generate MIs (in the algorithm to be presented below) or assign a weight to each arc of a DCG (in the second algorithm to be described in Chapter 5).

Initially, we try to classify the fields according to various criteria. An example is included for each type of field. Examples pertain to ISO's transport layer protocols. This is followed by possible variations in a boolean expression.

1. Fields are either:

- Always present (e.g., destination reference), or
- May be present (e.g., subsequence field in ISO Class 4 Transport Protocol).

2. Fields are either of:

   - Fixed length (e.g., length indicator field), or
   - Variable length (e.g., data field in a data message).

3. Content of a field may be:

   - History dependent (e.g., sequence number of the message), or
   - History independent (e.g., checksum).

4. The value of a particular field may be either:

   - Fixed permanently (e.g., class_of_protocol = 0 in Class 0 Protocol), or
   - Fixed for this connection only (e.g., source reference).

5. The field may have:

   - default values (e.g., maximum length of the TPDU), or
   - no default value at all (e.g., format of the TP, *normal* or *extended*).

6. A field may be sent:

   - only once (said to be of *F1 type*) (e.g., TPDU size field), or
   - more then once (said to be of *Fr type*) (e.g., data field in a data message).

Now consider the PROVIDED clause of the form "a *relop* b", where *relop* denotes a relational operator, which is one of <, <=, =, >=, >, <>. (Relops > and >= are handled similarly to *relops* < and <= respectively.)

1. If *relop* is <, then we should try the following variations:

   a. a < b: This should invoke normal protocol processing.
   b. a = b: This should cause an error indication.
   c. a > b: This should cause an error indication.

2. If *relop* is <=, then we should try the following variations:

   a. a < b: This should invoke normal protocol processing.
   b. a = b: This should invoke normal protocol processing.

c. a > b: This should cause an error indication.

3. If *relop* is =, then we should try the following variations:

   a. a = b: This should invoke normal protocol processing.
   b. a = b+1: This should cause an error indication.
   c. a+1 = b: This should cause an error indication.

4. If *relop* is <>, then we should try the following variations:

   a. a+1 = b, This should invoke normal processing.
   b. a = b+1, This should invoke normal processing.
   c. a = b, This should cause an error indication.

Now we will try to combine the various types of fields and possible variations in *relop*. In the above guidelines for *relop* variation, we have proposed to invoke normal processing and error indication at least once (for any *relop* there are either two cases that have normal protocol processing or two cases that involve error indication). In general, we observe the following:

1. In NFTs there are no conditional statements. All the conditions in the transitions of the original specification are visible in the PROVIDED clause. Therefore, observing the PROVIDED clause can give us a fairly good idea as to what should be varied.
2. The variables that appear in a PROVIDED clause appear to form groups. For example, the variables that appear in the NFTs during data transfer phase are different from the variables used in the connection establishment phase. This is probably the reason why the PROVIDED clause is simple and does not involve a large number of variables. So a very simple form having at most three variables is considered below. The form we consider is $P \land Q \land R$, where each of P, Q and R is of the form: "var *relop* var" or "[NOT] boolean variable".

It is possible that a product may not use the correct *relop* or logical operator. We have already seen how to detect an error when an incorrect *relop* is used. Now we consider the use of incorrect logical operator. Since the form we consider is $(P \land Q \land R)$, we have to try at most eight combinations.

The question now is how does one set a boolean expression *True* or *False*? This obviously depends on the type of the field involved in the boolean expression. Therefore, we suggest how to assign values to different fields. The values that should be sent may make the boolean expression either *True* or *False*. Additional values that must also be used are indicated. These boolean values must make the boolean expression *True*. In what follows we examine different types of fields and suggest how these fields could be varied. In brackets we indicate the value of the boolean expression, *True* [T], *False* [F] or additional parameter or field value variations [AV].

As stated earlier, messages contain two types of fields, F1 and Fr. Fields that are sent only once (F1), and those which can can be sent more than once (Fr), usually as many times as desired.

Consider the fields of type F1 first.

1. If they have default values, then the default values should be used at least once [T].
2. If invalid values can be assigned to them at all, they should be assigned invalid values (for error generation) [F]. Note that for some fields invalid values cannot be assigned, for example, the field that indicates the format of the messages (normal or extended).
3. From the ordered set of permissible values, the range of allowable values for the product being tested should be determined (by using binary search technique on the possible values for the field).

Consider the fields of type Fr.

1. If the value of a field is based on history, then modify it so that unexpected values are assigned [F]. (For example, in case of a sequence number, add 2 instead of 1.)
2. If the field can be intermittently present (ON), then select the sequence ON-OFF-ON-OFF or OFF-ON-OFF-ON. Example of such a field is the subsequence field in an ACK message in ISO Class 4 Transport Protocol [AV].

3. If the value of a field is constant for a connection, assign some invalid value [F].
4. If the length of a field can be varied, assign values to the field so that field length has following values: maximum (allowed) value, minimum value, some intermediate values, greater than maximum value and less than minimum value.

Now consider any message. Assigning different values to the fields (of which the message is composed) will generate different message instances. A value assinged to an individual field could be valid or invalid, generating a valid message instance or an invalid message instance. A valid (invalid) message instance causes normal processing (abnormal processing).

For a message, the number of valid message instances generated equals the maximum of the number of different valid values for any field and the number of invalid message instances generated equals the sum of the numbers of invalid values for all fields.

## 4.2. Parameter and value combinations

Using the guidelines given above, different values should be chosen for each field of a message. However, specific combinations of these possible field values must be used to obtain MIs that can be exchanged in testing. In the section given above *relops* indicate some field value combinations. In addition, if it is assumed that processing of each field is independent of the others, then we can construct messages as follows:

- Choose a field value which would generate an error for only one field. For all the other fields choose values that would generate normal processing.
- Choose parameter or field values that would generate normal processing. Clearly, the number of such messages is limited by a field which has the maximum number of possible (normal) field values.

## 4.3. Obtaining message instances

In the last two sections we have presented guidelines for generating MIs for a single message type, based on the fields the message type is composed of. We have also discussed above how MIs can be generated that would invoke normal processing and error processing. In addition, we have also seen how relational operators can influence the generation of MIs. We summerize below the guidelines in terms of a three-step algorithm for generating MIs.

**Step 1 :** For each message *M* repeat Step 2 and 3.

**Step 2 :** Observe the fields that message *M* is composed of. Obtain MIs corresponding to *M*, using the guidelines presented earlier.

**Step 3 :** Observe the WHEN clause of each NFT. If message *M* is specified in its WHEN clause, then obtain MIs, using the guidelines in Section 4.1 on relational operators.

Note that some redundant MIs may be generated. An example of how redundant MIs could be generated is given below.

Consider the Connection Request (CR) message of the ISO Class 0 Transport Protocol. The field "maximum size of the message" has a default value. Therefore, based on the type of field, Step 2 generates MIs. One of the MIs so generated will have this field unspecified (default value) and the others will contain some values specified for this field.

Also this field is specified in the WHEN clause of NFTs, P3 and P4, of Class 0 TP (listed in Appendix B). NFT P3 requires that the field value be specified and NFT P4 requires that this field value be unspecified. Therefore, Step 3 will generate two MIs.

Thus two identical messages will be generated by Step 2 and Step 3.

## 4.4. Overview of the algorithm

As stated earlier, the objective of the algorithm is to find a set of subtours such that (1) each NFT is on some subtour, and (2) each field value to be tested appears in a message associated with some NFT. The input to the algorithm is a matrix indicating which NFTs (columns) can send which MIs (rows). Note that, in general, a MI may be sent by more than one NFT, and more than one MI may be exchanged with the same NFT (not simultaneously). Using this matrix, the algorithm will select a set of subtours.

To implement the approach described above, we first obtain NFTs from the formal specification of the protocol. Then, from the NFTs thus obtained, we construct a two dimensional boolean matrix called **Message-Transition matrix** (or **M-T matrix**, for short) with columns representing the NFTs. If I instances of message type T are to be tested, then I rows are needed for this type in the M-T matrix. To facilitate construction, the rows are ordered as follows: All messages that establish a connection are placed first. These are followed by the messages that transfer data. Lastly the messages that terminate a connection are placed.

The algorithm selects all the MIs and NFTs in two stages. First it generates some possible subtours. If some MIs and/or NFTs remain that have not been used in any of the subtours in the first stage, it then generates additional subtours to exercise those NFTs and MIs. The algorithms given below do not explicitly consider the problem of terminating a partial path. The reason is that the terminating node can be reached from every node by traversing at most two arcs. So for every node we can fix a terminating path. If required, this path can then be used to terminate a

path. In what follows, the starting (terminating) node of an NFT is called its *tail-node* (*head-node*).

We introduce a column vector called the *Covered Instances (CI) vector*. The number of elements in the CI vector equals the number of rows in the M-T matrix. Initially, all the elements in the CI vector are set to *False*. For each MI selected, the corresponding element in the CI vector is set to *True*. We also introduce a row vector called the *Covered Transitions (CT) vector*. The number of elements in the CT vector equals the number of columns in the M-T matrix. Initially, all the elements in the CT vector are set to *False*. For each NFT selected, the corresponding element in the CI vector is set to *True*.

## 4.5. Major steps of the algorithm

**Step 1 :** Obtain the NFTs from the formal protocol specification.

**Step 2 :** Construct the M-T matrix (as described above), and set all elements in the CT and CI vectors to *False*.

**Step 3 :** Call procedure *New_subtour*, which will return a set of subtours.

**Step 4 :** For each entry in CI and CT that is *False* (i.e., MI and NFT not on any subtour generated in Step 3 above) call procedure *One_transition* to obtain a partial path. The partial path is completed (to obtain a subtour) by appending the NFT (or the NFT carrying the MI under consideration) and the terminating path from the *head_node* of the NFT (or the NFT carrying the MI under consideration).

## 4.6. Procedure New_subtour

This procedure selects a set of executable subtours. Beginning at the initial node, we do a search on the CG by systematically examining alternate MIs and NFTs that can be selected to extend a path to form a subtour. The search is done depth-first, but unlike the usual depth-first search, the same node may be visited many times. A MI (NFT) is said to be *feasible* if the field values (conditions) do not contradict the conditions associated with the partial path which has already been constructed. In the procedure given below, *at-least_one* is a boolean variable such that when it is *True*, it means that the subtour being constructed contains at least one new MI and/or a new NFT that has not been used on any other subtour generated so far.

**Step 1 :** *at_least_one = False*; Push the initial node on stack.

**Step 2 :** If stack is empty then return.

**Step 3 :** Pop a node from stack and assign it to *current_node* (Note that current_node is no longer in stack.)

**Step 4 :** Choose a feasible MI and an associated feasible NFT out of the *current node* such that at least one of them has not yet been selected (i.e., the corresponding entry in CI/CT is *False*). If no such NFT exists then go to step 7.

**Step 5 :** *at_least_one = True*; Push *current_node* and then the head-node of the chosen NFT on stack. Set the corresponding entry in CI and CT to *True*. If this head-node is not the terminal node then go to step 2.

**Step 6 :** (One more subtour has been found that has selected some more MIs or NFTs). Let *at_least_one = False*, pop a node (the terminal node) from stack, and go to step 2.

**Step 7 :** If *at_least_one = False* then go to step 2 else select the terminating path for the node to generate a new subtour. Let *at_least_one = False* and go to step 2.

## 4.7. Termination proof for New_subtour

We first observe that whenever a node is pushed into stack, at least one entry, in either CI or CT matrix, is set to *True*. Therefore, the total number of times nodes are pushed into stack is bounded by a constant.

We now prove that the procedure does not loop indefinitely. In the procedure there is one loop. A new "pass" through the loop begins with the execution of Step 3. In every "pass" the number of nodes in the stack can

- (Step 5) increase,
- (Step 7) decrease, or
- (Step 6) remain the same.

Note that only a fixed number of consecutive "passes" can stay in Step 6, since there are only so many executable NFTs/MIs going out of the node. Therefore, if the procedure loops indefinitely, then either the stack size must grow without bound, or it must grow and shrink indefinitely. In either case, there must be a node that is pushed in stack an unbounded number of times, which is impossible as we argued above.

The following procedure is needed to execute those NFTs/MIs which were not selected in any subtours generated by procedure New_subtour.

## 4.8. Procedure One_transition

This procedure will select a subtour that includes the NFT (and MI) given as the parameter. Obviously, the simplest solution would be to generate all the possible subtours and select a subtour that includes the given NFT and MI. Due to the exponential number of possible subtours, this is not practical.

In order to obtain a practical algorithm we must somehow restrict the scope of search. This is possible if some assumptions are made. Clearly, if the assumptions are too restrictive then the algorithm will be applicable only to trivial protocols. The ISO **Class 2 Transport Protocol** (**C2TP**, for short) is a widely known protocol and is considered sufficiently complex to serve as a model. Therefore, we have examined it carefully.

The CG[4] of the ISO C2TP is shown in Figure 4-1. It has two potential initial nodes and two potential terminal nodes, a total of four combinations, depending on whether a connection already exists or not. Therefore, we have constructed and examined all the four possible CGs. These four CGs, named CG11, CG12, CG21, and CG22, are included at the end of this chapter. At any given time, one of these will be applicable. It is observed that of these CGs only CG11 and CG21 have loops which are not self loops. The algorithm can handle loops (which are not self-loops) in the CG if the following condition is satisfied: in any loop there exists an arc $A$ such that if the arc $A$ is deleted then there exists a path from the initial node to the terminating node that has all the nodes of the loop in sequence. Such loops will be called *simple loops*.

We have also carefully examined the NFTs of ISO **Class 0 Transport Protocol** (**C0TP**, for short). These observations have led us to the following algorithm based on the three simplifying assumptions stated below.

Our approach is to traverse the CG backward, starting from the tail-node of the

---

[4]We obtain a DCG from a given CG to test robustness of the IUT as described in Chapter 3. However, to make the discussion that follows conceptually simple, we will refer to the CG instead of DCG.

NFT, t, under consideration to the initial node of the applicable CG. In other words, we construct a path backwards from t to the initial node. Suppose we have partially constructed such a backward path from t to an intermediate NFT, t'. We call the tail-node of t' the *current_node*. We want to extend the partial path by prepending to it an NFT, t'', whose head-node is the *current_node*.

Note that not every NFT terminating at the current node can be chosen as t'', since the NFTs that are already on the partial path may impose certain conditions in terms of their PROVIDED clauses. Thus, the chosen NFT must have a PROVIDED clause that is compatible with all of them, so that the completed path will be executable. (Such a NFT is called a **feasible** NFT.) Repeating the above process, we will eventually reach the initial node, at which point the construction completes.

Note that once a NFT, t'', is chosen it cannot be deleted Therefore, we have not allowed arbitrary loops in the CG. Since if arbitrary loops are allowed backtracking becomes necessary. However, we have allowed simple loops. The idea is that when partial path is being extended the path should lead "towards" the initial node and not "towards" the end node. In order to allow simple loops the algorithm given below maintains a list of "visited" nodes and allows "visiting" a node only once.

To see why backward traversal is more efficient, consider any NFT that is traversed only when call establishment is initiated by the IUT side. If forward traversing is used, all subtours selected that correspond to connection establishment by the tester site will have to be discarded. It is noted that in most cases there is only one path from the NFT under consideration back to the initial node in the CG. In cases where two paths are possible either one is acceptable.

We now clearly state our assumptions:

1. All loops of the CG other than self-loops are *simple loops*.
2. Repetitive executions of self-loops do not modify the conditions tested in the PROVIDED clause of any NFT. Therefore, they need to be chosen at most once in subtour selection.
3. A backward path to the initial node and the NFTs and MIs for the path can be chosen incrementally with available information. Therefore, the algorithm need not backtrack.[5]

The assumptions stated above are satisfied by C2TP. Consider our first assumption. For C2TP, of the four possible CGs (viz., CG11, CG12, CG21 and CG22) only two CGs, CG11 and CG21, have a loop (involving nodes numbered 2 and 6). The loops in CG11 and CG21 are simple loops.

The second assumption deals with self-loops. In general, for an arbitrary CG, it may be necessary to repetitively execute a self-loop before another NFT, which is not a self-loop, is chosen. Appendix C contains a proof that for C2TP this is not the case.

Appendix D contains a proof that our last assumption is also satisfied by C2TP.

The actual algorithm now follows:

**Step 1 :** Initially, the set of conditions include only those conditions that are associated with the MI and NFT under consideration and *current_node* is set to the *tail-node* of the NFT. *selected_nodes* is a set of nodes and initially contains only the *current_node*.

**Step 2 :** Choose self-loop arcs so that any specific condition that can be satisfied only by self-loop arcs can be chosen. (Refer to the discussion of "local variables" in NFT S in Appendix C.)

---

[5]Note that backtracking has nothing to do with a backward path.

**Step 3 :** Choose any feasible NFT whose *head-node* is the *current_node*, and whose *tail-node* does not belong to *selected_nodes*. Add the *tail-node* of the chosen NFT to the *selected_nodes* set.

**Step 4 :** Add the conditions of the chosen NFT to the existing conditions, and set the *current_node* to the *tail-node* of the *current_node*.

**Step 5 :** If *current_node* is the initial node then stop, else go to step 2.

We now list the labels used in the control graphs, CG00 to CG22, that follow.

<u>Node Numbers</u>
<u>State Value</u>.

1, 1′ NC_state=closed, state=closed.

2, 2′ NC_state=open, state=closed.

3 NC_state=closed, state=open_in_progress_calling.

4 NC_state=open_in_progress,
state=open_in_progress_calling.

5 NC_state=open, state=open_in_progress_calling.

6 NC_state=open, state=open_in_progress_called.

7 NC_state=open, state=open.

8 NC_state=open, state=closing.

9 NC_state=open, state=T_Err_sent.

10 NC_state=open, state=wait_before_closing.

The correspondence between the arc labels and the NFTs for C2TP is given below (labels are as defined by Sarikaya [Sari 84]).

| Arc Label | List of NFTs represented |
|---|---|
| A | PL1 |
| B | PJ1 |
| C | PD1, PN1 |
| D | PH1 |
| E | PD2, PD3 |
| F | PI1 |
| G | P502, PD2, PD3, PF02, PG02, PG06, PG12, PN2 |
| H | P507, P508, P510, P601...P604, PD1, PF06, PN1 |
| I | P509, P510 |
| J | PN3 |
| K | P101, P102, P117, P2, P3, PE1, PE2, PR2 |
| L | P505, PG09, PG15 |
| M | P605, PF09 |
| N | P2, P3 |
| O | PF04, PG04, PG08, PG14, PN6 |
| P | P607...P610, PF04 |
| Q | PN7 |
| R | P91, P92 |
| S | P103, 104, 110, 112, 113, 115, 116, 118, P2, P3, P82, P84, P93, PA2, PA3, PB1, PB2, PO1, PP1, PQ1, PR4, PS1, PT1, PU1 |
| T | P612 |
| U | PG18, PG19 |
| V | P116, P2, P3 |
| W | P606, P71, PF10 |
| X | P506, PG10, PG16 |
| Y | PM1, PM2 |
| Z | PF07, PN4 |
| a | P2, P3, PR3 |
| b | P414 |
| c | PG17 |
| d | PL1 |
| e | P106, P109, P116, P2, P3, P407, P413, P611 |
| f | P2, P3, P108 |

In order to make the understanding of the CGs for the Class 2 TP easier we list below all the NFTs and their input/output messages. We have used the same

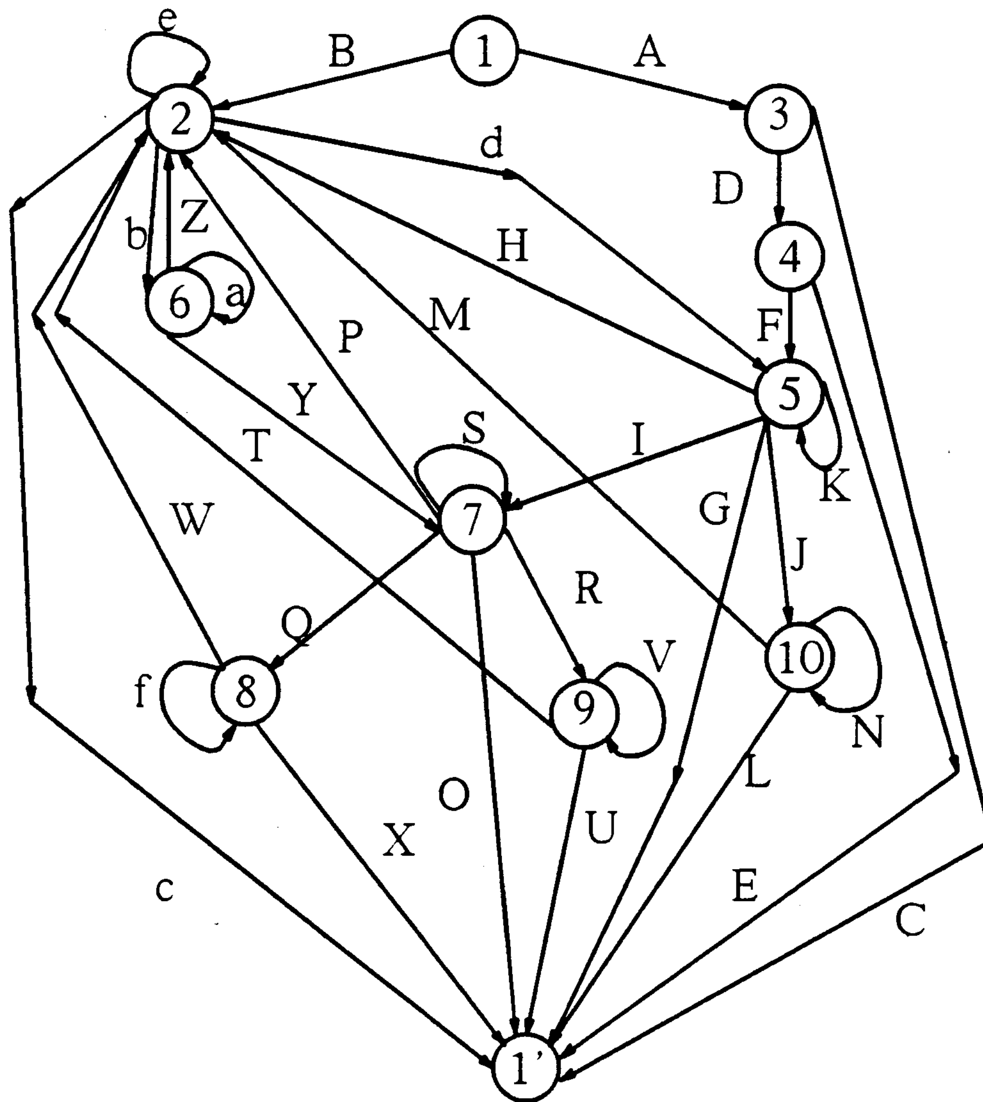


**Figure 4-1:** Control Graph CG00

terminology as in [Sari 84] which should be very clear to any reader with exposure to ISO protocols. Otherwise, the reader is referred to [ISOb 84].

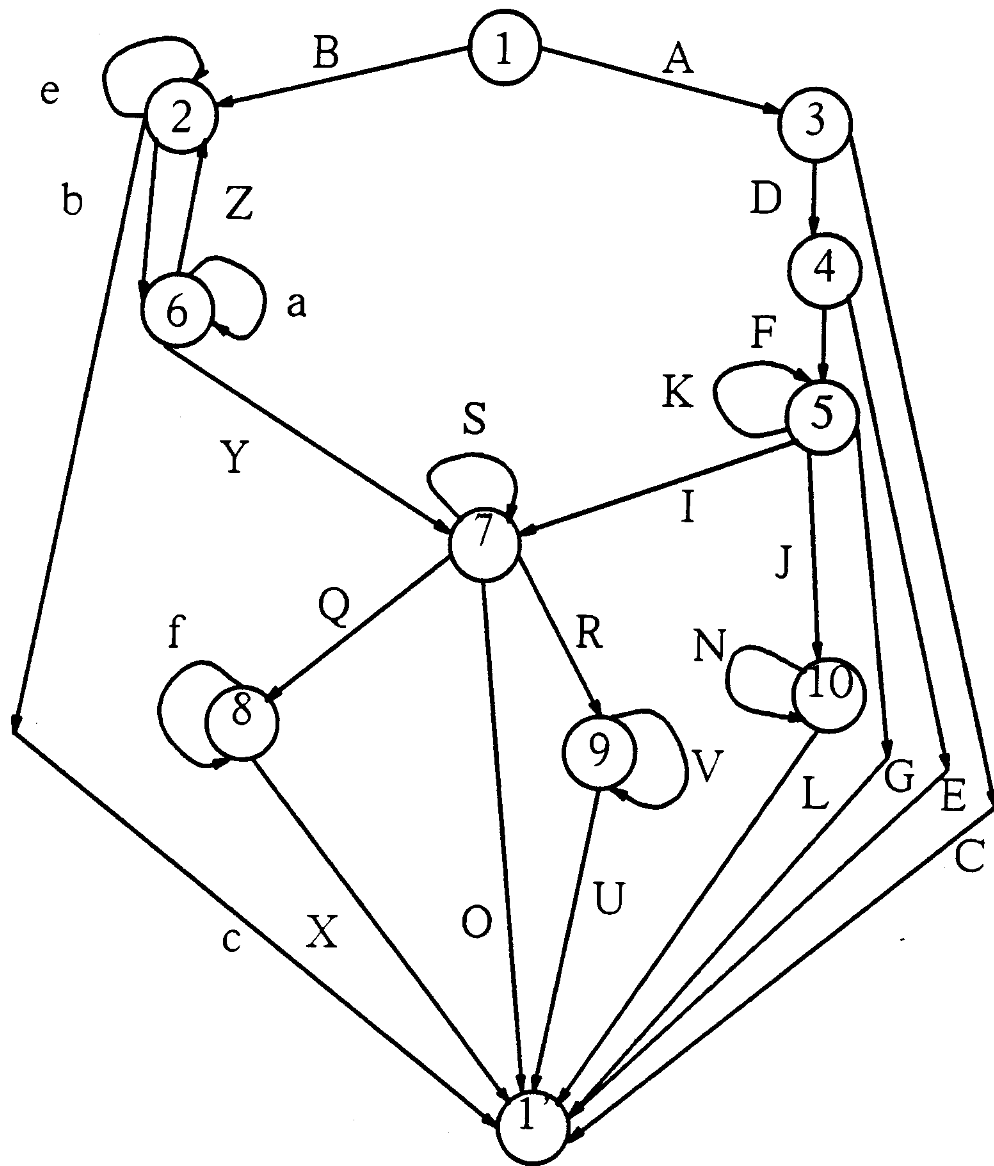


**Figure 4-2:** Control Graph CG11

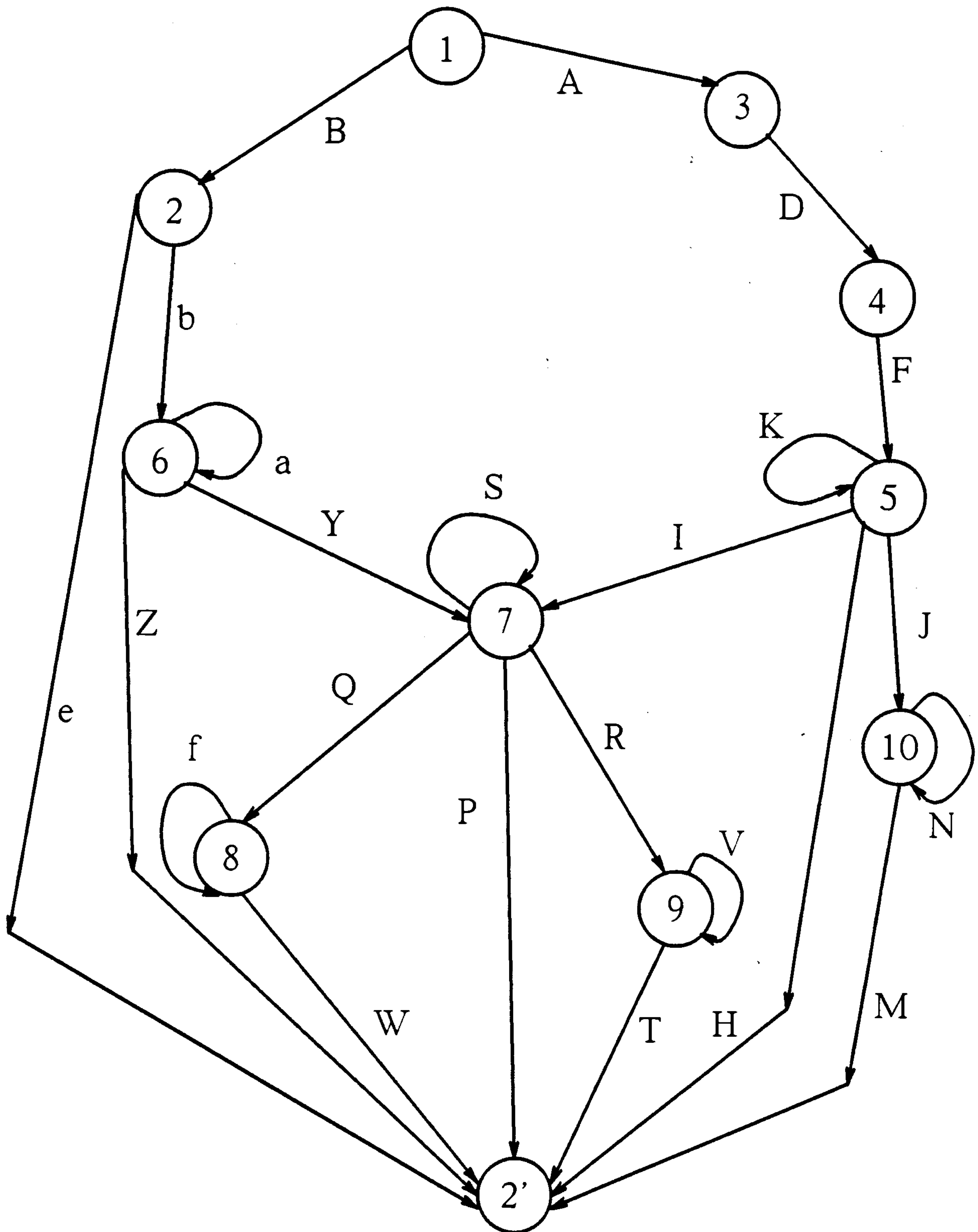


**Figure 4-3:** Control Graph CG12

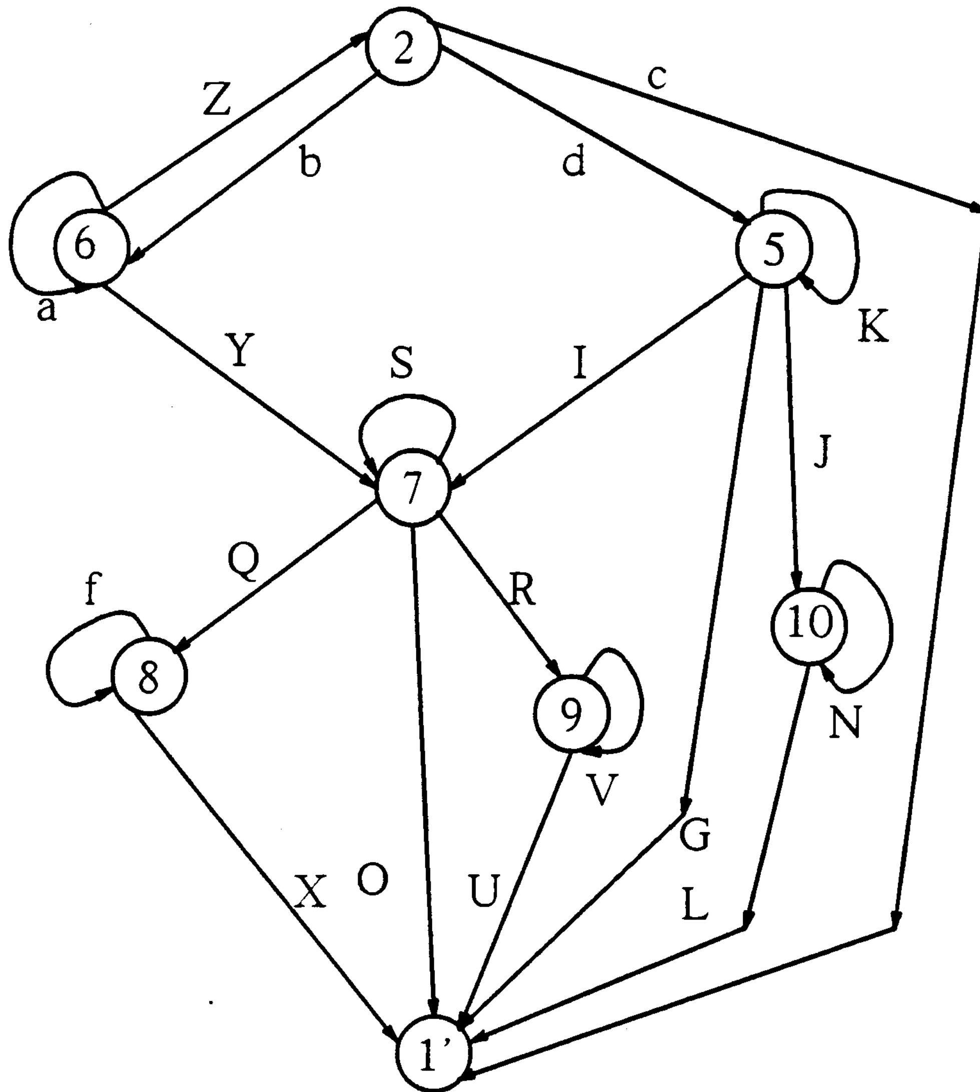


**Figure 4-4:** Control Graph CG21

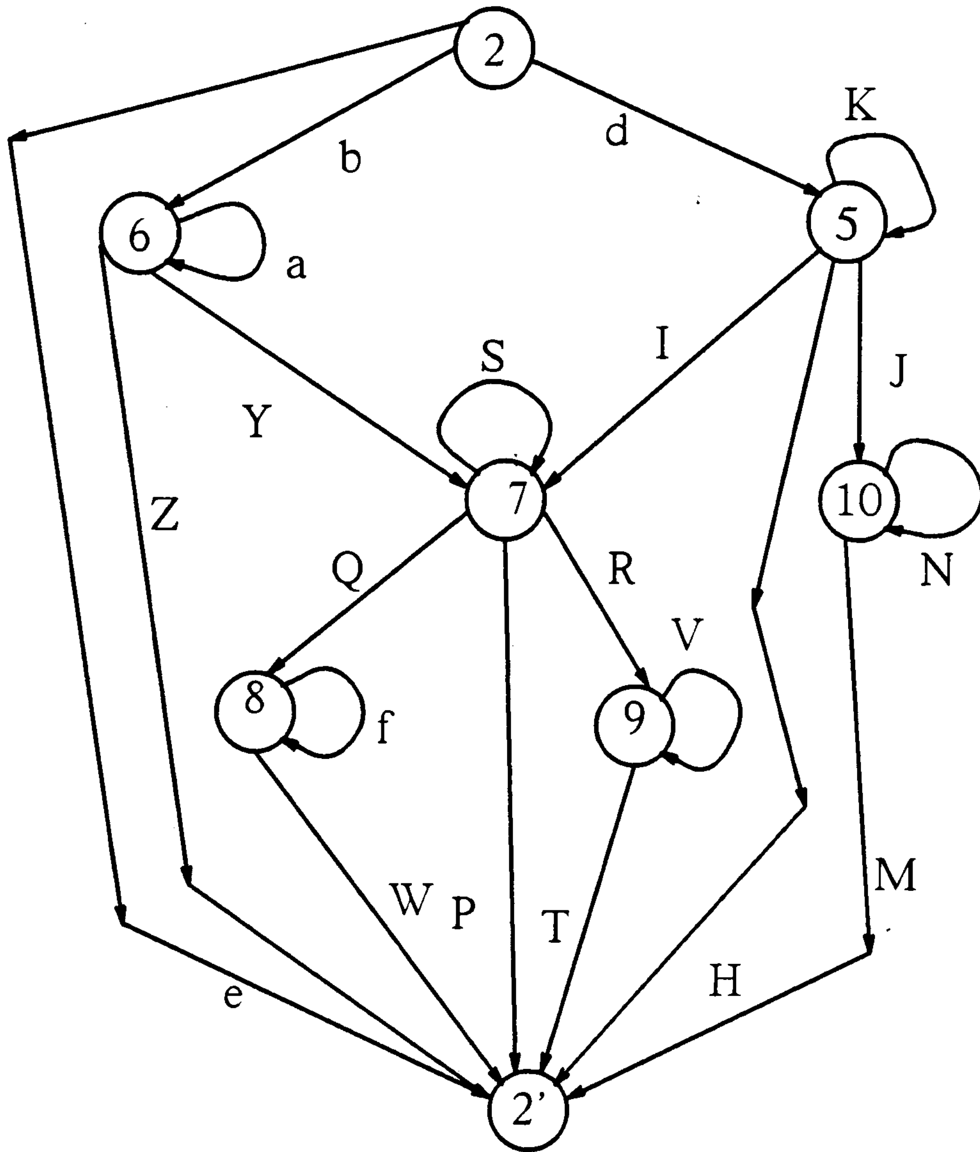


**Figure 4-5:** Control Graph CG22

| NFTs | Input or output message |
|---|---|
| P101, 102, 117 | Spontaneous transitions which output CR message. |
| P103, 104, 118 | Spontaneous transitions which output CC message. |
| P105-108 | Spontaneous transitions which output DR message. |
| P109 | Spontaneous transition which outputs DC message. |
| P110, 111 | Spontaneous transitions which output DT message. |
| P112 | Spontaneous transition which outputs AK message. |
| P113, 114 | Spontaneous transitions which output EDT message. |
| P115 | Spontaneous transition which outputs EAK message. |
| P116 | Spontaneous transition which outputs ERR message. |
| P2 | Spontaneous transition which outputs N_DATA_req. |
| P3 | Spontaneous transition which outputs N_DATA_ind. |
| P401-415 | Spontaneous transitions which decode N_DATA_ind into a CR message. |
| P501-513 | Spontaneous transitions which decode N_DATA_ind into a CC message. |
| P601-613 | Spontaneous transitions which decode N_DATA_ind into a DC message. |
| P71-72 | Spontaneous transitions which decode N_DATA_ind into a DT message. |
| P81-84 | Spontaneous transitions which decode N_DATA_ind into a AK message. |
| P91-93 | Spontaneous transitions which decode N_DATA_ind into a EDT message. |
| PB1-2 | Spontaneous transitions which decode N_DATA_ind into a EAK message. |
| PC1 | Spontaneous transition which decodes N_DATA_ind into a ERR message. |
| PD1-D3 | Spontaneous transition which outputs T_DISCONNECT_ind message. |
| PE1-E2 | Spontaneous transition that sets class and max_PDU_size. |
| PF01-10 | Received N_RESET_ind message. |
| PG01-18 | Received N_DISCONNECT_ind message. |
| PH1 | Spontaneous transition which outputs N_CONNECT_req. |
| PI1 | Output N_CONNECT_conf. |
| PJ1 | Received N_CONNECT_ind. |
| PK1 | Spontaneous transition which outputs N_DISCONNECT_req. |
| PL1 | Received T_CONNECT_req. |
| PM1-2 | Received T_CONNECT_res. |

| | |
|---|---|
| *PN1-7* | *Output T_DISCONNECT_req.* |
| *PO1* | *Output T-DATA_req.* |
| *PP1-2* | *Spontaneous transitions which output DT message.* |
| *PQ1* | *Spontaneous transition which outputs T_DATA_ind.* |
| *PR1-4* | *Spontaneous transitions which update R_credit.* |
| *PS1* | *Spontaneous transition which outputs AK message.* |
| *PT1* | *Output T_EX_DATA_req.* |
| *PU1* | *Output T_EX_D_READY_req.* |

# Chapter 5
# Optimal algorithm

In this chapter we propose another algorithm for test sequence generation. It finds a "minimum-cost" set of test sequences, but it has a limited applicability since it applies only to those protocols which satisfy a somewhat restrictive assumption: it will be assumed that the value of a minor variable cannot enable or disable any NFT, although it may affect the output associated with the NFT.

## 5.1. Overview of the algorithm

Each message exchanged between the peer entities has several fields. As commented earlier, in order to test a product, it is necessary to send the same message (traverse the corresponding arc of the CG or DCG) with different values (value combinations for the fields which the message is composed of). We attach an integer weight to each arc that indicates the number of different combinations to be tested for this arc. Unlike in Chapter 4, we can disregard the values of the fields and only count the number of distinct value combinations, since the values are assumed to have no effect on which transitions are possible. The weight of an arc is thus the lower bound on the number of times the arc must be traversed. Since an arc can be a part of many subtours, what is required is to make sure that the total number of times an arc is traversed by all the subtours is not less than the lower bound.

The method we present in this chapter can easily be generalized to deal with more

general "costs" to appropriately reflect the cost of processing a message, transmission cost, etc. A possible cost function could be $C = c1 \times l + c2 \times n$, where $c1$ and $c2$ are constants, and

- $l$ is the length of a message in bits (to account for transmission cost).
- $n$ is the number of fields in a message (to account for message processing time, since each field of the received message must be processed).

For simplicity, we use the total number of times that NFTs are fired as the cost. We make use of network flow algorithms [Tard 84, Tarj 83] to find a minimum cost solution.

## 5.2. Weighted Dual Control Graph

In the following sections we present a brief review of network flow algorithms. Each arc of a DCG will first be labeled to obtain a *weighted digraph*, where a label is an integer indicating the number of times the corresponding NFT should be tested.

A weighted digraph WG is denoted by (V, A, $s$, $w$), where

- (V, A) is the underlying digraph,
- $s \in$ V is a distinguished node, and
- $w$ is a (weight) function from A to the set of nonnegative integers $\{0, 1, 2, ...\}$.

If $a \in$ A then, $w(a)$ indicates the minimum number of times arc $a$ is to be traversed, each time with a different value combination. For a method of obtaining combinations, see Section 4.1.

## 5.3. Network

A network is denoted by N = (V, A, $s$, $t$, $w$) where

- $t$ is the "sink" node (outdegree of $t$ = 0).
- $s$ is the "source" node (indegree of $s$ = 0).

A WG G = (V, A, $s$, $w$) can be converted into a network N = (V', A', $s$, $t$, $w'$) by redirecting the incoming arcs of $s$ to the new node $t$. Here V' = V ∪ {$t$}, $t$ does not belong to V, and A' is obtained as follows:

For each $[a, s] \in$ A, let $[a, t] \in$ A' with $w'([a, t]) = w([a, s])$.
For each $[a, b] \in$ A such that $b \in$ V - {$s$}, let $[a, b] \in$ A' with $w'([a, b]) = w([a, b])$.

## 5.4. Listing all the required subtours in a network

Given a *flow function* $f$[6], the algorithm given below will list all subtours. It first finds subtours by traversing an arbitrary path beginning and terminating in the initial node. The flow of each arc in the subtour is reduced by 1. An arc with weight 0 is deleted to indicate that it need not be traversed any further. To account for those arcs whose weight is non zero when the outdegree of initial node becomes 0, the algorithm finds isolated cycles, starting at a node of each cycle.

**Input:** A network N = (V, A, $s$, $t$, $f$), where $f$ is a flow function satisfying the lower bounds.

---

[6]For each $v \in$ V - {$s$, $t$}, $f$ satisfies $\Sigma_{(v,w) \in A} f([v,w]) = \Sigma_{(v,u) \in A} f([u,v])$ and $\Sigma_{(s,a) \in A} f([s,a]) = \Sigma_{(b,t) \in A} f([b,t])$

**Output:** All the subtours of N. Each arc $a \in$ A is traversed by $f(a)$ subtours.

The algorithm is given below.

**Step 1 :** If A = {} then go to Step 9.

**Step 2 :** If $s \in$ V then *curr_node* := $s$ and *end_node* := $t$ else *end_node* := *curr_node* := $\forall\, v \in$ V.
/* A new subtour or cycle begins. */

**Step 3 :** Select any arc $a \in$ A whose tail is *curr_node*.

**Step 4 :** *new_node* := head of the arc $a$. Output the arc.

**Step 5 :** If $f(a)$ = 1 then A = A - $\{a\}$ else $f(a) = f(a)$ -1.

**Step 6 :** If there are no outgoing arcs from the *curr_node* then V = V - {*curr_node*}.

**Step 7 :** If *new_node* = *end_node* then go to Step 1. /* Current subtour or cycle ends */

**Step 8 :** *curr_node* := *new_node* and go to Step 3. /* Current subtour or cycle continues */

**Step 9 :** Merge the isolated cycles, if any, with any subtour beginning with $s$ which contains a node of the isolated cycle to obtain all the subtours of network N.

### 5.4.1. Analysis of the algorithm

We observe that an arc $a$, with flow $f(a)$, will be scanned $f(a)$ times. Thus the complexity of the algorithm is $O(\Sigma_{a \in A} f(a))$.

## 5.5. Network flow based algorithm

An algorithm for generating the minimum number of subtours is presented below.

**Step 1 :** Obtain the NFTs from the formal protocol specification.

**Step 2 :** Obtain the CG from the NFTs resulting from Step 1.

**Step 3 :** Obtain the DCG from the CG.

**Step 4 :** Obtain the weighted DCG (called WDCG) by assigning weight to each arc in the DCG using the guidelines given in Section 4.1.

**Step 5 :** Obtain a network N from the WDCG using the procedure given in Section 5.3.

**Step 6 :** Set the upper bound of each arc of N to the sum of all the lower bounds. [This upper bound ensures that a feasible flow can always be determined [Lawl 76] (pp. 140-159).]

**Step 7 :** Find a minimum cost flow through N using any of the known methods (e.g., the out-of-kilter method [Lawl 76] or the method given by [Tard 84]).

**Step 8 :** Obtain all the required subtours decomposing the flow obtained in Step 7 using the procedure given in Section 5.4.

## 5.6. Comments

An arbitrary subtour is, in general, not executable because of the PROVIDED clauses. As stated earlier, the above algorithm is applicable only if the CG has the property that any arbitrary subtour is executable.

Usually it is possible to set up multiple connections. When multiple connections are established, more than one CG is "active" at the same time. Each "active" CG may have different initial or terminating nodes. We cannot apply the network flow

algorithms to obtain minimum cost flow for each CG since the arcs are shared by the CGs.

## 5.7. Example

In this section we demonstrate the use of the above algorithms, to generate test sequences for ISO's Class 0 Transport Protocol (C0TP) [Knis 82]. The formal specification document for C0TP [ISOa 82] does not contain any transitions to handle messages that are not expected in a given state. To demonstrate the concept of DCG more effectively, we have added an extra NFT, named P20, which is fired if the current state (of the underlying FSM) is *data_transfer* and a CR message is received, sending an ERROR message and terminating the connection (the new state is idle). The ERROR message is a new message that we have added to the formal specification for illustration. It indicates to the peer entity that a procedure error (i.e., reception/transmission of the message not expected in the given state) has occurred and, therefore, the connection is being terminated.

All the NFTs, labeled P1 to P20, of the ISO C0TP are listed in Appendix B. The CG for C0TP is shown in Figure 5-1. Each arc in this figure represents a set of NFTs. The forward arcs are labeled A to D, and backward and self loop arcs are labeled a to e. The correspondence between the labels used in the CG and the NFTs is listed below. For easy reference, we also indicate the input/output for each NFT and, in parentheses, the type of input/output symbol (external/internal). A set of NFTs (e.g., P3 and P4) having the same input/output are grouped together.

| Arc | NFT | Input/Output (Type) |
|---|---|---|
| A | P3, P4: | CR/T_CONNECT_ind (CR/*). |
| B | P10: | T_CONNECT_resp/CC (*/CC). |
| C | P1: | T_CONNECT_req/CR (*/CR). |
| D | P6, P7: | CC/T_CONNECT_conf (CC/*). |
| a | P5: | CR/DR (CR/DR), |
| | P2: | T_CONNECT_req/T_DISCON_ind (*/*). |
| b | P11: | T_CONNECT_resp/(DR, T_DISCON_ind) (*/{DR, *}), |
| | P12: | T_DISCON_req/DR (*/DR). |
| c | P8, P9: | DR/(N_DISCON_req, T_DISCON_ind) (DR/{*, *}). |
| d | P13: | T_DATA_req/- (*/-), |
| | P14: | -/DT (-/DT), |
| | P15: | DT/- (DT/-), |
| | P16: | -/T_DATA_ind (-/*). |
| e | P17: | T_DISCON_req/N_DISCON_req (*/*), |
| | P18: | N_DISCON_ind/T_DISCON_ind (*/*), |
| | P19: | reset/T_DISCON_ind (*/*), |
| | P20: | CR/ERROR (CR/ERROR). |

From the CG shown in Figure 5-1, we obtain the corresponding DCG, shown in Figure 5-2, using the procedure given in Section 3.3. The two square states (q1 and q2) are new states. Typically, for each arc of the CG (e.g., A), we obtain the corresponding arc of the DCG (e.g., A') by adding an apostrophe. (The additional labels of the form "<m,n>", where m and n are integers will be explained later.) In drawing the CG, we have made minor changes to the input/output symbols of the arc d. This is purely to make things simple. The arc labels of the DCG and their input/output symbols are listed below. We also list the correspondence between the states of the CG and the DCG.

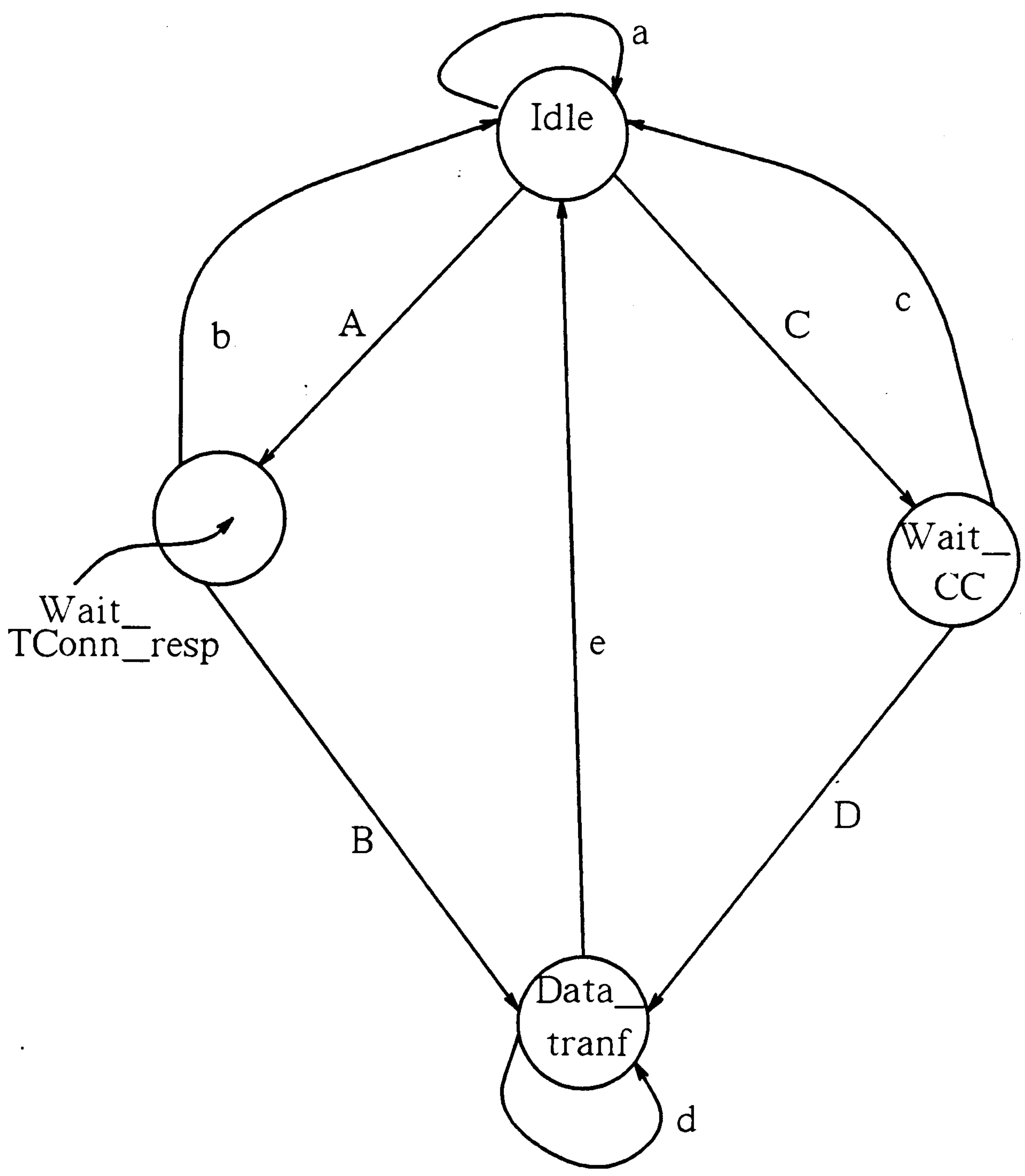


**Figure 5-1:** Control Graph for ISO Class 0 TP

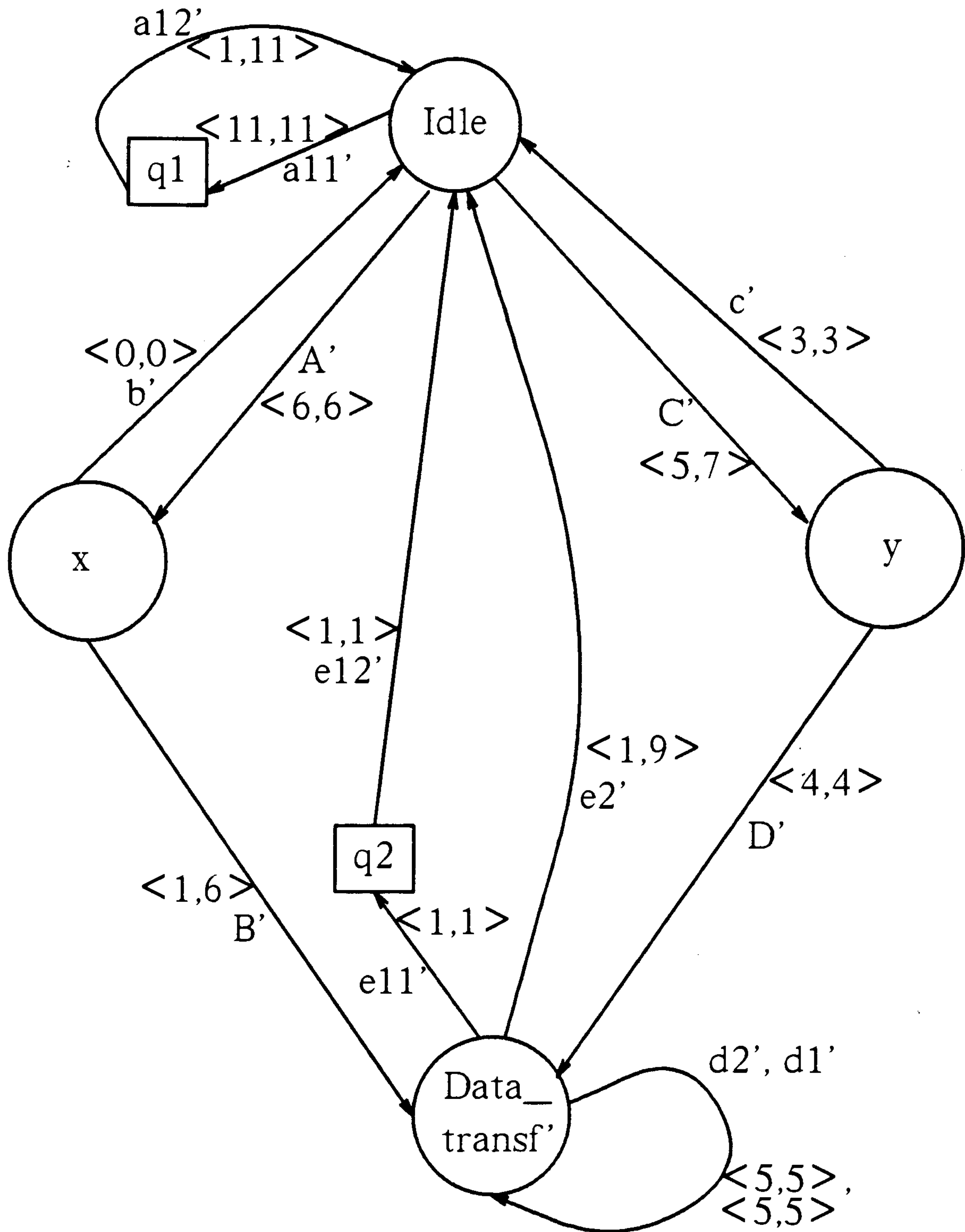


**Figure 5-2:** Dual Control Graph for ISO Class 0 TP

| Arc Labels | Input/output symbols |
|---|---|
| A' | */CR. |
| B' | CC/*. |
| C' | CR/*. |
| D' | */CC. |
| a11' | */CR. |
| a12' | DR/*. |
| b' | DR/*. |
| c' | */DR. |
| d1' | DT/*. |
| d2' | */DT. |
| e11' | */CR. |
| e12' | ERROR/*. |
| e2' | */*. |

| States of CG | Corresponding states of DCG |
|---|---|
| idle | idle' |
| data_transfer | data_transfer' |
| wait_for_T_CONNECT_res | x |
| wait _for_CC | y |

For comparison, we present in Figure 5-3 the CG for COTP with I/O labels.

Now we have to assign a weight to each arc of the DCG using the guidelines given in Section 4.1. As an example, consider arc A' which outputs a CR message to the peer entity. We will describe how a weight is assigned to the A'. Other arcs can be handled in a similar fashion. It is assumed that the product under test has implemented COTP only. Using the same terminology as in [Knis 82], we will examine each field of the CR message and see in how many different ways (valid or invalid) values can be assigned to it.

A CR message has following fields.

1. Length Indicator.

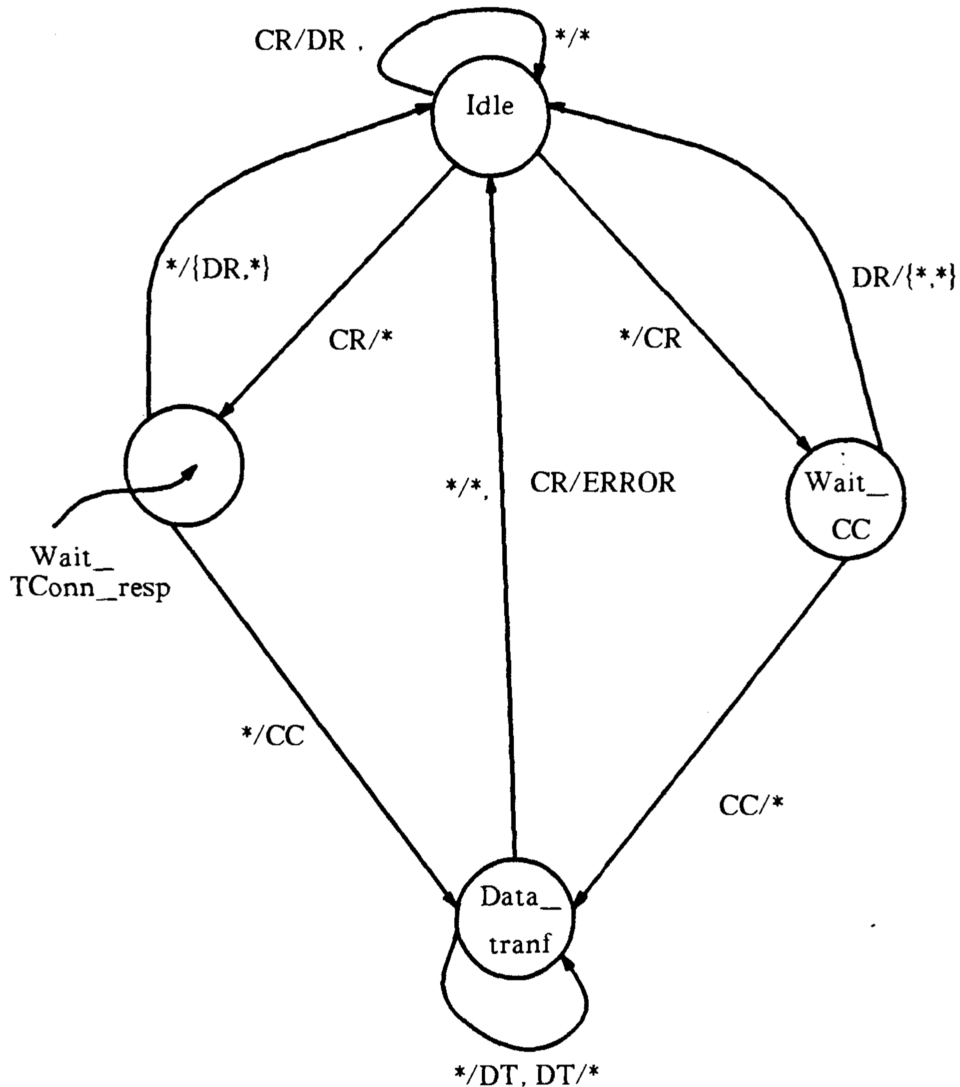


**Figure 5-3:** Control Graph for COTP with I/O labels

Number of invalid values = 1 (incorrect value), otherwise valid values only.

2. Credit
Number of invalid values = 1 (non zero value), otherwise valid values only (i.e., 0).
3. Source reference
Number of invalid values = 1 (zero value), otherwise valid value only (non zero value)
4. Class options. This field has three subfields.

a. Class of TP (0..4): Number of invalid values = 1 (<> 0), otherwise valid value only (0).

b. Format of the message (normal or extended): No invalid values. Both formats to be used.
c. Two bits always zero: Number of invalid values = 1 (set either one or both of them to 1), otherwise valid values only (both bits 0).

5. Variable part. This field has several subfields.

a. Called, calling TSAP-ID: Number of invalid values = 2 (once for calling address and once for called address), otherwise valid values only.
b. TPDU size: Only the following values are permitted in COTP: 128, 256, 512, 1024, 2048. Number of invalid values = 2 (>2048 and <128), Number of valid values = 6 (default and all 5 values listed above).
c. Acknowledge Time, Residual error rate, Priority, Transmit delay: For all these fields let us assume that only fixed known valid values are assigned.

6. Let us also add (say) 2 invalid values for those parameters in the variable part that are not allowed in the COTP, e.g., checksum.

Thus we will have to send 11 invalid CR messages (indicated by a label on a11' and used as the lower bound *lb(a11')* = 11), and at least 6 valid CR messages (*lb(A')* = 6) on all valid calling TSAP-IDs. Here *lb(a)* indicates the lower bound for an arc *a*. Similarly we can assign weights to other remaining arcs. However, we will not go into the same level of details for the remaining arcs as we have done with CR message. Instead, we will assign some "reasonable" lower bounds as follows: CC messages are sent in 4 different ways (*lb*(D') = 4); the IUT is forced to send CR messages in 5 different ways (*lb*(C') = 5); connection is denied at least 3 times (*lb*(c') = 3); 5 different data messages are sent and expected (*lb*(d2') = *lb*(d1') = 5); a CR message is sent once in data transfer state (*lb*(e11') = 1).

The meaning of additional arc labels in Figure 5-2 of the form <m,n> is as follows. "m" represents the lower bound. "n" represents the actual flow (i.e., the number of times the arc will be executed), so that at every state the flow is conserved and total input flow equals total output flow.

Once a minimum flow is computed, we obtain the different subtours using the algorithm presented in section 5.4. A possible list of subtours using this algorithm is given below. A sequence of arcs in parentheses with a superscript indicates that those arcs will be executed repeatedly in sequence: the number of times it will be repeated is indicated by the superscript.

1. $(a11', a12')^{11}$
2. A', B', d2', d1', e11', e12'
3. A', B', $(d1')^{3}$, e2'
4. $(A', B', d2', e2')^{4}$
5. $(C', c')^{3}$
6. C', D', d1', e2'
7. $(C', D', e2')^{4}$

# Chapter 6
# Conclusion and Discussion

In this thesis, we have proposed two algorithms for automatically generating test sequences for products implementing a layer of layered communication protocols. In this chapter we highlight the contributions of this thesis and indicate future directions.

## 6.1. Contributions of this thesis

We have proposed two algorithms for generating test sequences directly from the formal specification in Estelle. The first method is general and can be used for a broad class of protocols but does not generate optimal test sequences. The second method is based on network flow algorithms and obtains minimum cost test sequences. However, it cannot be used for complex protocols. Since the methods are independent of any particular test architecture, they are widely applicable.

## 6.2. Comparison with Sarikaya's algorithm

We now compare the new algorithms proposed in this thesis and Sarikaya's algorithm, described in Appendix A.3. For the definitions of the terms specific to Sarikaya's algorithm, the reader is referred to Appendix A.

The most important difference between our approach and Sarikaya's is that in our method the Data Flow Graph (DFG) is completely eliminated while the DFG plays the most important role in Sarikaya's algorithm.

The procedure to construct a DFG is described in detail in Appendix A. Briefly, a DFG depicts the flow of information (due to BEGIN blocks of NFTs) from the vertices that represent values (of different fields) of the received messages (called I-nodes) to the vertices that represent values (of different fields) of the message being sent (called O-nodes). For example, there will be an arc from vertex X to vertex Y labeled P, if, in an NFT labeled P, there is an assignment statement that assigns the value of a variable represented by vertex X to a variable represented by vertex Y. For real protocols (like ISO's TP), the DFG has a rather large number of vertices and arcs, since each variable is represented by a vertex and every assignment statement (of the BEGIN block) generates an arc (informally -- a DFG is very complex).

In Sarikaya's algorithm, the DFG is partitioned and each block of the DFG is tested independently. Heuristics are used to partition a DFG. Choosing an optimal solution is difficult, since to obtain an optimal solution, one must partition the DFG intelligently. No algorithm is suggested for the above problems and it is not clear how this step can be mechanized.

In Sarikaya's algorithm, parameters are varied by assigning different values to every node. This is achieved by assigning appropriate values to the I-nodes. Clearly, a very large number of values must be assigned to the I-nodes, to achieve the parameter variations as suggested by Sarikaya.

In our algorithm, for parameter variations, we observe the PROVIDED clause. Note that all the conditional statements are visible in the PROVIDED clause. PROVIDED clauses are completely ignored in Sarikaya's algorithm. It may also be noted that our test executes each statement of the BEGIN block since they execute all the NFTs at

least once. Furthermore, our first algorithm is powerful enough to handle complex protocols and our second algorithm obtains an optimal solution in a special case.

In summary, it can be said that our algorithms have two main advantages over Sarikaya's algorithm. They are:

- It is easier to generate test sequences by our algorithms, since we observe the PROVIDED clause to vary the parameters, doing away with the DFG.
- It is easier to implement our algorithm since partitioning DFG and choosing an executable subtour to traverse a particular arc in DFG is not required.

## 6.3. Future work

We expect significant research in the following areas in the near future:

- Formal model for protocol specification.
- Test architectures for testing protocol implementation.
- Program testing in general.

A brief comment on each area is given below.

### 6.3.1. Formal models

A formal specification is preferred to specification in a natural language because, since the specification becomes unambiguous, it can be verified for correctness, and it lends itself to automatic implementation.

A standard model for protocol specification, that is accepted universally, would be desirable. Currently only a draft version of the FDT model is available. It is not clear how soon the ISO will be able to standardize Estelle. The main difficulty in standardization is that many different models are available (see chapter 1) and none of them seems to be the best.

### 6.3.2. Test architectures

In chapter 2, two architectures for testing protocol implementations were described. Unfortunately, each architecture has advantages and disadvantages. Recently, yet another architecture has been proposed by Rafiq [Rafi 85]. This architecture is a variant of the replacement scheme described in Section 2.3. The TR of the replacement scheme is called an *astride responder* (AR) in [Rafi 85]. The AR operates at two levels, (N) (i.e., the same as that of IUT) and (N+1), simultaneously. Unfortunately, the principle of hierarchical or layered architecture (which is used to keep the overall complexity within manageable limits and) which requires building each layer using services of the immediately lower layer, is violated. Clearly more research leading to an architecture that combines the advantages of both the replacement scheme and derail scheme would be highly useful.

### 6.3.3. Program testing

Research in this field is also vital for effectively testing communication protocols. (In our scheme, we use it to obtain message instances and to assign integer weights to the arcs.) Although tremendous efforts are being made, no strategy that makes only a few assumptions and is yet powerful enough to detect most of the errors (and therefore acceptable to most applications) is available. Testing will continue to remain an "art" unless some break-through is made in the program testing field.

The research presented in this thesis could be used in program testing area if programs can be modeled as extended finite state machines.

# Appendix A
# Sarikaya's Algorithm

In this appendix the theory that Sarikaya's algorithm is based on and the algorithm itself are described. The concept of Data Flow Graph (DFG) due to Sarikaya [Sari 84] is introduced. The CG and DFG are first derived from the specification, and the DFG is then partitioned and each block of the partition is tested independently. To test each block, different combinations of input parameter values must be applied and output of each combination must be verified for correctness.

## A.1. The Data Flow Graph

The DFG is constructed from a specification in terms of NTFs. It models the path of information flow in the specification. A DFG contains four types of nodes:

- **I-nodes** to represent input interactions,
- **D-nodes** to represent protocol variables and constants,
- **O-nodes** to represent output interactions, and
- **F-nodes** to represent certain functions on data.

The arcs in the DFG are used to represent the information flow as derived from the actions (specified in the BEGIN blocks) of the NFTs. Simple assignment statements[7] in a BEGIN block are shown by arcs directed from source nodes to the destination nodes. The output values directly computed from the input values are modeled as simple assignment statements.

---

[7] An assignment statement in which a variable (represented by a node in a DFG) is assigned a value of other variable is referred to as a simple assignment statement.

Each arc in a DFG is labeled with the identifier of the corresponding NFT. This identifier represents the pre-condition of the NFT (viz. (1) boolean expression specified in the PROVIDED clause and (2) received interaction specified in the WHEN clause) which must be satisfied if the indicated flow of information should occur. A given BEGIN block can contain many statements, hence there can exist more than one arc labeled with the same identifier. Also, the same assignment used in more than one NFT is represented as a single arc carrying more than one label.

## A.2. Partitioning of the Data Flow Graph

Sarikaya [Sari 84] partitions the DFG in two stages. After the first stage, a large number of blocks are obtained. Some of the blocks are then combined in the next stage to obtain the final partition. The procedure is rather involved, so we briefly outline the procedure for each stage. Complete details are given in [Sari 84].

### A.2.1. Stage 1

Initially all D-nodes are considered *unprocessed*. An unprocessed D-node is selected for processing and a new block is created for this D-node. For this D-node, all input I-nodes, D-nodes and F-nodes and output O-nodes are placed in the same block. These O-nodes may have other input D-nodes. All such D-nodes are also placed in the same block. For any input F-node, all D-nodes that represent constants and input I-nodes are included in the same block. If all the input nodes of the F-nodes are in the same block then the F-node is placed in the same block. When no more nodes can be added to the block by the above procedure, some unprocessed D-node is selected and the operations are repeated. When an O-node is assigned a value directly by an I-nodes (F-nodes), additional blocks are created containing only the I-nodes (F-nodes) and the O-node.

### A.2.2. Stage 2

Using the heuristic procedure given below, some blocks are combined to form a larger block.

- Two blocks $B_i$ and $B_j$ are combined if their O-nodes contain parameters of the same type. Examples of parameters that are of the same type are:
  - Source reference and destination reference.
  - Called address and calling address.
- Two independent blocks $B_i$ and $B_j$ are combined if the types of all the I-nodes of $B_i$ are the same as those of the O-nodes of $B_j$.
- Two blocks $B_i$ and $B_j$ are combined if their O-nodes contain different but related parameters of the same message. Which parameters are related is determined by the test designer.
- If block $B_i$ contains only O-nodes and F-nodes and if F-nodes have incoming arcs from D-nodes of different blocks, then $B_i$ is combined with one of the blocks. If there are more than one block, then the particular block is chosen by the test designer.
- Two blocks $B_i$ and $B_j$ are combined if they contain related D-nodes. Which D-nodes are related is determined by the test designer.
- If block $B_i$ is an independent block and it has some D-nodes that are not assigned to any other node, then $B_i$ can be combined with some block $B_j$ if NFTs in $B_i$ form a subset of NFTs in $B_j$.

## A.3. Sarikaya's Algorithm for generating test sequences

(01) From the CG, a subtour is selected. The subtour indicates the sequence of input interactions to be applied.

(02) The I-nodes of the block under test that are involved in the input interactions of Step 01 are determined.

(03) Parameters of the I-node of Step 02 that are not involved in the current subtour are assigned suitable constant values.

(04) From the DFG, the expected parameter values of each O-node, corresponding to the input values of Step 03, are determined.

(05) Input interactions are applied and the Output interactions are observed.

(06) If the parameter values of the output interactions agree with the values computed in Step 04, then Step 07 is executed, else it is inferred that there is an error in the implementation. More details about the error are provided and the testing procedure terminates.

(07) The subtour is repeated (Step 05) for different combinations of values for the I-nodes.

(08) Steps 01 to 07 are repeated till a complete block is tested. (A subtour may test only a part of a block.)

# Appendix B

# List of NFTs for Class 0 TP

In this appendix we list the NFTs of the ISO Class 0 TP. These NFTs are given in [Sari 84]. Since there are some minor differences in ours and Sarikaya's NFTs (see section 3.1.2), corresponding changes are reflected in the NFTs listed below. We also use "add" for "address", "var" for "variable", and "DISCON" for "DISCONNECT". The NFT labeled P20 is our own addition.

```
P1:
WHEN       TSAP.T_CONNECT_req
FROM       idle
PROVIDED   (/Trans. entity able to provide required QOS /)
TO         wait_for_CC
BEGIN
           local_ref := ....;
           TPDU_size := ....;
           var_part_to_send := ....;
           CR (0, local_ref, class_0, normal,
                                   var_part_to_send);
END

P2:
WHEN       TSAP.T_CONNECT_req
FROM       idle
PROVIDED   (/Trans. entity NOT able to provide required QOS /)
TO         idle
BEGIN
           T_DISCON_ind (TCEPI, inability_to_provide_QOS);
END
```

```
P3:
WHEN       mapping.CR
FROM       idle
PROVIDED   var_part.max_TPDU_size <> undefined AND
           (/ able to provide QOS /)
TO         wait_for_T_CONNECT_resp
BEGIN
           remote_ref := source_ref;
           TPDU_size := var_part.max_TPDU_size;
           remote_add := var_part.calling_add;
           TCEP := ....;
           called_add := ....;
           calling_add := ....;
           QOTS_estimate := ....;
           T_CONNECT_ind (TCEP, called_add, calling_add,
                          QOTS_estimate, normal);
END

P4:
WHEN       mapping.CR
FROM       idle
PROVIDED   (var_part.max_TPDU_size = undefined) and
           (/Trans. entity able to provide required QOS /)
TO         wait_for_T_CONNECT_resp
BEGIN
           remote_ref := source_ref;
           TPDU_size := 128;
           remote_add := var_part.calling_add;
           TCEP := ....;
           called_add := ....;
           calling_add := ....;
           QOTS_estimate := ....;
           T_CONNECT_ind (TCEP, called_add, calling_add,
                          QOTS_estimate, normal);

END
```

```
P5:
WHEN        mapping.CR
FROM        idle
PROVIDED    NOT (/Trans. entity able to provide required QOS /)
TO          idle
BEGIN
            var_part_to_send.add_clear_reason := ....;
            DR (source_ref, 0, connection_negotiation_failed,
                var_part_to_send);
END

P6:
WHEN        mapping.CC
FROM        wait_for_CC
PROVIDED    var_part.max_TPDU_size <> undefined
TO          data_transfer
BEGIN
            remote_ref := source_ref;
            TPDU_size := var_part.max_TPDU_size;
            QOTS_estimate := ....;
            T_CONNECT_conf (TECP, QOTS_estimate, normal);
            in_buffer.clear;
            out_buffer.clear;
            out_buffer.set_max_get_size (TPDU_size);
END

P7:
WHEN        mapping.CC
FROM        wait_for_CC
PROVIDED    var_part.max_TPDU_size = undefined
TO          data_transfer
BEGIN
            remote_ref := source_ref;
            TPDU_size := ....;
            QOTS_estimate := ....;
            T_CONNECT_conf (TECP, QOTS_estimate, normal);
            in_buffer.clear;
            out_buffer.clear;
            out_buffer.set_max_get_size (TPDU_size);
END
```

```
P8:
WHEN        mapping.DR
FROM        wait_for_CC
PROVIDED    disconnect_reason = TS_user_initiated_termination
TO          idle
BEGIN
            disc_reason := disconnect_reason;
            user_reason := var_part.additional_clear_reason;
            N_DISCON_req (..., disc_reason);
            T_DISCON_ind (TCEP, disc_reason, user_reason);
END

P9:
WHEN        mapping.DR
FROM        wait_for_CC
PROVIDED    disconnect_reason <> TS_user_initiated_termination
TO          idle
BEGIN
          disc_reason := disconnect_reason;
          N_DISCON_req (...., disc_reason);
          T_DISCON_ind (TCEP, disc_reason, user_reason);
END

P10:
WHEN        TSAP.T_CONNECT_resp
FROM        wait_for_T_CONNECT_resp
PROVIDED    (/QOS requested <= QOS proposed in T_CONNECT_ind /)
TO          data_transfer
BEGIN
            local_ref := ....;
            TPDU_size := ....;
            var_part_to_send.called_add := remote_add;
            var_part_to_send.calling_add := ....;
            var_part_to_send.max_TPDU_size := TPDU_size;
            CC (remote_ref, local_ref, class_0, normal,
                var_part_to_send);
            inbuffer.clear;
            outbuffer.clear;
            outbuffer.set_max_get_size (TPSU_size);

END
```

```
P11:
WHEN      TSAP.T_CONNECT_resp
FROM      wait_for_T_CONNECT_resp
PROVIDED  (/QOS requested › QOS proposed in T_CONNECT_ind /)
TO        idle
BEGIN
          var_part_to_send.add_clear_reason := ....;
          DR (remote_ref, 0, connection_negotiation_failed,
              var_part_to_send);
          T_DISCON_ind (TCEP,
                             inability_to_provide_the_QOS);
END

P12:
WHEN      TSAP.T_DISCON_req
FROM      wait_for_T_CONNECT_resp
PROVIDED
TO        idle
BEGIN
          var_part_to_send.add_clear_reason := ....;
          DR (remote_ref, 0, TS_user_initiated_termination,
              var_part_to_send);
END

P13:
WHEN      TSAP.T_DATA_req
FROM      data_transfer
PROVIDED  (/flow control from the user is ready /)
TO        data_transfer
BEGIN
          out_buffer.append (TSDU_fragment);
END

P14:
WHEN
FROM      data_transfer
PROVIDED  (/flow control to the user is ready /)
TO        data_transfer
BEGIN
          mapping.DT (get_next_fragment (out_buffer));
END
```

```
P15:
WHEN        mapping.DT
FROM        data_transfer
PROVIDED    (/flow from the network layer is ready /)
TO          data_transfer
BEGIN
            inbuffer.append (user_data);
END

P16:
WHEN
FROM        data_transfer
PROVIDED    (/flow control to the user is ready /)
TO          data_transfer
BEGIN
            TSAP.T_DATA_ind (get_next_fragment (inbuffer));
END

P17:
WHEN        TSAP.T_DISCON_req
FROM        data_transfer
PROVIDED
TO          idle
BEGIN
            N_DISCON_req (disconnect_reason, user_reason);
END

P18:
WHEN        N_DISCON_ind
FROM        data_transfer
PROVIDED
TO          idle
BEGIN
            disc_reason := ....;
            T_DISCON_ind (TCEP, disc_reason, user_reason);
END
```

```
P19:
WHEN        mapping.network_reset
FROM        data_transfer
PROVIDED
TO          idle
BEGIN
            disc_reason := ....;
            T_DISCON_ind (TCEP, disc_reason, ... );
END

P20:
WHEN        mapping.CR
FROM        data_transfer
PROVIDED
TO          idle
BEGIN
            disc_reason := ....;
            T_DISCON_ind (TCEP, disc_reason, ... );
            ERROR;
END
```

# Appendix C
# Repetitive Executions of Self-loops

The ISO model, described in Chapter 2, uses the concepts of major and minor variables. Execution of a NFT may change the values of major and/or minor variables. The change in the major variable of course represents a state transition. The change in minor variables may cause a change in the enabling conditions (PROVIDED clauses) of some NFTs from TRUE to FALSE or vise versa.

In this appendix we show that repetitive executions of self-loop NFTs either do not alter the condition in any PROVIDED clause or are equivalent to a single execution (as for NFT PR3 mentioned below) with some exceptions, e.g., NFT P84, which are discussed at length later in this appendix.

Arcs N, a, f, V, K, e and S in the control graph CG00 (shown in Chapter 4) are self-loops. We will now briefly examine and discuss the NFTs that each of the above arcs represents. The list of NFTs that these arcs represent is included at the end of Chapter 4.

We use the same terminology as in [Sari 84], which in turn conforms to [ISOb 84]. It is noted that when some messages, e.g., ERR, DR etc., are sent to the IUT, there is no change in any condition since only a message is constructed (i.e., appropriate values are placed in various fields of the message) and the message is "handed over" to the network layer.

Arc N represents two NFTs, i.e., P2 and P3. In fact, NFTs P2 and P3, which are spontaneous transitions (i.e., without WHEN clause), appear in every self-loop. These two NFTs send/receive data messages to/from the network layer. (Recall that any message sent to or received from the peer transport entity must be sent via the network layer). No change in any condition is caused by these NFTs. NFTs P2 and P3 will not be discussed again with the other remaining self-loops.

Arc a represents three NFTs, viz., PR3, P2 and P3. NFT PR3 is used to update the *R_credit* value. No other values are changed. Repetitive executions of NFT PR3 is equivalent to executing NFT PR3 only once with an appropriate value (so that *R_credit* has the same final value). The value of *R_credit* variable is examined when a data message is received. If the value is greater than 0 the received data message is accepted (assuming other conditions e.g., sequence number, are satisfied) and the *R_credit* value is decremented by 1. However, a value of 0 indicates that remote site has sent a data message when it should not have. This is an error and hence the connection is terminated (see NFT P81).

Arc f represents three NFTs, viz., P108, P2 and P3. NFT P108 is used to send a DR message to the IUT, after which a DC message is expected from the IUT. When a DC message is received, the state is changed to the initial state. There is no change in any condition since only a DR message is sent out.

Consider the arc V now. It represents three NFTs, viz., P116, P2 and P3. NFT P116 sends an error message (ERR message) to the IUT. Repetitive executions of NFT P116 (sending ERR messages) does not change any condition.

Arc e represents eight NFTs. NFTs P106 and P109 send a DR message to the IUT.

NFT P116, which sends an ERR message to the IUT, was discussed above. NFT P611 is executed when a duplicate DR message is received. A DC message is sent to the IUT in response. NFTs P407 and P413 are used to ignore previous CR messages from the IUT. NFT P407 sets the type of message to be sent as ERR (on the same network connection with the same destination reference). NFT P413 is very similar to NFT P407 with only minor difference. In all of the above cases the connection is terminated and the state is set to the initial state.

Arc K represents eight NFTs, viz., P101, P102, P117, PE1, PE2, PR2, P2 and P3. NFT PR2 is used to update the *R_credit* value only and is very similar to NFT PR3. NFTs PE1 and PE2 are used to indicate that ISO Class 2 Transport Protocol (Class 2 TP) will be used and set the maximum size of data messages. Repetitive executions of NFTs PE1 and PE2 will have the same effect if the NFT is executed only once with a suitable value for the maximum size of data messages. NFTs P101, P102, and P117 construct and send a CR message to the IUT. Note that state 5, in which arc K can be executed, represents a state where no connection has yet been established with the IUT. Only one CR message is sent to the the IUT.

Arc S represents 24 NFTs which are executed mainly to exchange data messages between the peer entities. Out of the 24 NFTs which arc S represents, the following 10 NFTs send a message to the IUT and hence do not change any conditions: P103(CC), P104(CC), P118(CC), P110(DT), P112(AK), P113(EDT), P115(EAK), P116(ERR), PP1(DT) and PS1(AK). The corresponding message type sent to the IUT is indicated above in parentheses.

NFT PU1 also sends an EAK message to the IUT. However, it does set variable *EX_D_sent* to FALSE. NFT PR4 only updates the *R_credit* value (see discussion of

the arc a given above). The remaining NFTs are executed when different data messages and different acknowledgement messages are received from the IUT. They also include spontaneous transitions which send messages to the network layer and the session layer. In the table given below we list each NFT and indicate the type of message that is received and the action taken at the tester site. A discussion of how the enabling conditions of NFTs are affected, due to changes in the values of the variables by each NFT is also included.

P82 Reception of a DT message. Received data is appended to the data buffer.

P84 Reception of a DT message. Received data is appended to the data buffer. *R_credit* value is decremented by 1, and the send sequence number is incremented by 1.

P93 Reception of an AK message. The *S_credit*[8] value is updated.

PA2 Reception of an EDT message. An ERR message is sent to the IUT, since an EDT message is not allowed when previous EDT message is "unacknowledged" (i.e., *EX_D_received*[9] is TRUE).

PA3 Reception of an EDT message. The variable *EX_D_received* is set to TRUE.

PB1 Reception of an EAK message. Send an ERR message to the IUT, since the use of an expedited data message was not chosen but an EAK message is received from the IUT.

---

[8]The value of *S_credit* variable is examined when a data message is sent. If the value is greater than 0 the data message is sent and *S_credit* value is decremented by 1. However, a value of 0 indicates that a data message cannot be sent now.

[9]Variable *EX_D_received* is set to TRUE when an EDT message is received and is set to FALSE when an EAK message is sent.

PB2 Reception of an EAK message. Set the variable *EX_D_sent*[10] to FALSE.

PO1 Reception of a *T_DATA_req* message. Append the data in the message to the data buffer to be sent to the IUT.

PQ1 Reception of a *T_DATA_ind* message. Append the data in the message to the data buffer to be sent to the session layer.

PT1 Reception of an *EX_DATA_req* message. The expedited data message is constructed in a buffer and the variable *EX_D_sent* is set to TRUE.

The variables that change due to the NFTs represented by arc S include the send sequence number and the receive sequence number. Here we will focus our attention on these two variables only. For *EX_D_sent*, *EX_D_received*, *R_credit* and *S_credit* variables a similar argument applies. The sequence numbers change when data messages (DT messages) are sent and received (see NFT P84). However, these (minor) variables can be considered "local" to the data transfer phase, since they do not appear in the PROVIDED clause or BEGIN blocks of NFTs during the connection establishment or connection termination phase.

Now we will examine both (a) the problem of partial path extension and (b) the effect of the "local" variables on the enabling conditions of NFTs when the NFT under consideration is in one of the three phases, i.e., connection establishment, data transfer and connection termination.

[10] Variable *EX_D_sent* is set to TRUE when a EDT message is sent and set to FALSE when a EAK message is received.

## C.1. Connection establishment phase

The NFTs in the connection establishment phase do not refer to the send or receive sequence numbers. Clearly, their enabling condition is independent of the values of the "local" variables.

## C.2. Data transfer phase

If it is required that send and receive sequence numbers have specific values, then appropriate number of data messages should be sent and received to and from the IUT (by the tester site).

## C.3. Connection termination phase

In this phase specific value of send or receive sequence number is not required Only a specific condition may need to be satisfied (e.g., sending out of sequence data message). In such a case repetitive executions of self-loop NFTs that send and receive data messages are required.

We have examined each NFT that appears as a self-loop in the control graph CG00. We have also discussed the effect of minor variables, that are modified by self-loop NFTs, on the enabling conditions of other NFTs. From the discussion above, it is clear that in all the cases we can choose NFTs to extend the partial path by the algorithm given in Chapter 4.

# Appendix D
# Property of ISO Class 2 TP

## D.1. Introduction

In our algorithm (more specifically procedure *One_transition* presented in Chapter 4), we made the assumption that no backtracking is necessary to find an executable path *to* the start node *backward from* the node from which a given NFT carrying a MI originates. By definition, all the NFTs on an executable path have PROVIDFD clauses that are *compatible* or *simultaneously satisfiable*. In this appendix we demonstrate that the above assumption is valid for the ISO Class 2 Transport Protocol (C2TP).

In what follows we consider a field of a message received by the IUT as an *input variable* if it can be assigned different values. Note that not all the fields can be assigned different values. As an example consider the *dest_reference* field in C2TP. This field is not considered as an input variable in data messages since its value is fixed. However, in the connection-confirm messages it is an input variable.

The **supported values** for an input variable mean those values which are allowed by the IUT. For example, if the IUT allows the maximum buffer size of 4096 bytes, then any value lower than 4096 for buffer size is termed as supported. Other values will be termed as **unsupported** values. During backward path construction, supported values are used by the tester, unless the conditions force us to use a

different value. (E.g., to make the IUT traverse one of the NFTs represented by arc C, we need to use an unsupported value with an NFT represented by arc A.)

In the analysis below, we will examine the PROVIDED clause of the current NFT. It may force an input variable in a message to the IUT to have a certain value. Those NFTs which do not require specific values for any input variables are referred to as NSIVR (No Specific Input Value Required) NFTs.

We will also examine the BEGIN block of each NFT and describe the possible message instances (MIs) that are output. Then we will show that for every MI we can extend the backward partial path. Those NFTs which can have only one MI will be referred to as NSOMI (No Specific Output Message Instance) NFTs.

Consider the data transfer state (state 7 in the CGs) Independent of how state 7 is reached, the connection can be disconnected by NFTs O or P. Therefore, MIs for NFTs O and P are completely independent of connection establishment parameters. Such NFTs will be called **Totally Independent** NFTs, or **TINFT**, for short. More formally, a TINFT is an NFT whose output message is independent of history (i.e., how this state was reached and the values of input messages that led to this state). Therefore, the MI output by a TINFT is a function only of the input message. In general, the field values of an output message depend on both the input MI and the history. If some output message is independent of its history, then clearly once the state is reached the desired MI can be sent (to the IUT) and its response (i.e., the output message) can be observed. All the NFTs that terminate connection in the CG (i.e., their *to_state* is the initial node) are TINFTs.

We will first show that if a given NFT t is in the connection establishment phase

then the (backward) extension of a partial path by one more NFT is always possible. Then proof will follow by induction. The other cases will then be examined. We will refer to the module at the IUT site that is one layer "above" the IUT as the **Co-operative module** (CM). To give the reader a clear idea about the connection establishment phase we first describe the connection establishment by the CM, referring to the control graph CG11. (The control graphs referred to in this appendix are shown at the end of Chapter 4). We then present the proof considering the complete control graph CG00.

## D.2. Connection establishment in C2TP

**Step 1:** The CM sends a *T_conn_request* to the IUT. This in turn generates a *CR* message (arc A). The IUT is now in state 3.

**Step 2:** A spontaneous NFT (no WHEN clause present in the NFT) corresponding to arc D then sends an *N_conn_request* to the tester site. The IUT is now in state 4.

**Step 3:** At the tester site, the reception of an *N_conn_request* causes arc B to send an *N_conn_response* to the IUT. The tester site is now in state 2.

**Step 4:** At the IUT site, the *N_conn_response* causes transition from state 4 to 5 due to arc F.

**Step 5:** At the IUT spontaneous arc K causes transmission of a *CR* to the tester site. No change in state.

**Step 6:** At the tester site, the reception of a *CR* causes the sending of a *T_conn_ind* to the "above" layer (arc b). State changes from 2 to 6.

**Step 7:** At the tester site, the reception of a *T_conn_res* from the "above" layer causes transition from state 6 to 7 and the transmission of a *CC* to the IUT (arc Y).

**Step 8:** At the IUT, the reception of a *CC* causes the sending of a *T_conn_conf* to the CM (arc I). The IUT changes state from 5 to 7.

At this stage both the CGs are in state 7, i.e., in the data transfer state and data can be exchanged. The arcs traversed at the IUT are A, D, F, K, I and the arcs traversed at the tester site are B, b, Y. When the arcs are interchanged we obtain the scenario where a call is established by the tester site.

## D.3. Proof

It is noted that

1. Values can be assigned only to messages that are sent out to the IUT and not to the messages received from the IUT.
2. NFTs that represent error conditions due to incorrect implementation of protocol cannot be guaranteed to be traversed since the IUT may be error-free. For example, consider NFT PG17 in state 2. To execute NFT PG17 in state 2, the remote site must send *N_conn_ind* immediately followed by *N_disc_ind*. The reception of two (rather conflicting) messages, one requesting connection establishment immediately followed by another requesting the termination of the connection (being established) is not expected. We also list these NFTs below.
3. The tester site or the CM can send any message with either supported or unsupported values. In addition, if the CG (of the IUT, if the CM is sending a message to the IUT, else the CG at the tester site) is in a certain state, the execution of specific NFTs at the IUT may always be possible. NFTs described as "always possible" below refer to such NFTs. It is noted that, depending on the state of the CG, the same input message may give rise to the traversal of different NFTs.

We first consider the case where the CM initiates connection establishment. The MIs for arc C are generated due to unsupported QOS (quality of service) options in

the *T_conn_req* message from the CM. The CM can send any values in a *T_conn_req* message (arc A). So backward traversing from C is always possible. The NFTs corresponding to arcs D and F are traversed only when supported values are used.

Arc E is traversed when the tester sends a *T_disc_ind*. Clearly, using supported values for all other arcs and tester sending a *T_disc_ind* is always possible.

The MIs for arc I are due to the reception of a *CC* message from the tester site, which in turn depends on the parameters of *T_conn_res*. Only supported values must be used in arcs F, D and A.

The MIs for arc J arise due to the reception of a *T_disc_req* from the CM, which is always possible. Arcs F, D and A can use any supported values to reach state 5.

Arcs G and H are traversed because either the *CC* message received from the tester site has unsupported values or a DR message or *N_reset_ind* or *N_disc_ind* message is received from tester site (always possible). Arcs F, D and A use any supported values to reach state 5.

Now consider the case where the tester site establishes a connection. In this case, arcs B, c, e, b, Z, a, Y will be traversed by the IUT.

Arc c is traversed when the tester site sends an *N_conn_req* with supported values followed by a *N_disc_req* (we cannot execute NFT PG17), which is always possible.

When an *N_reset_ind* is received from the tester site, arc Z is traversed. So first an *N_conn_req* is sent with supported values, followed by an *N_reset_ind* (we cannot execute NFT PN4), which is always possible.

The CM can send a *T_conn_rep* to the IUT with the IUT already in state 6 using supported values. Clearly always possible.

Arc a is traversed when data messages are received from the CM or *R_credits* are updated. Clearly always possible with only supported values used to reach state 6.

Arc e is to ignore the messages like DT and DR received when the states of both network and transport protocol are "closed". The messages are simply ignored since they are not expected.

In state 7 self loop S exchanges data and expedited data messages. It also handles delayed duplicate messages *CC* and *CR*. MIs are possible only if supported values are used in connection establishment. Also they are not dependent on which side established the connection.

Other arcs are TINFTs. States 8 and 9 can be reached by first reaching the data transfer state using supported values only. Then the connection can be terminated with NFTs in either Q, R, P or O. Arcs Q, P, and O are TINFTs. Consider arc R, which represents NFTs P91 and P92. They are not TINFTs. However, they represent an error condition. More specifically, a message received from the remote site has invalid values for some fields. What values are invalid depends on history. For example, a data message numbered 6 is out of sequence when only 3 data messages have previously been sent out, but is expected when 5 data messages have previously been sent out.

The most important point here is that the invalid values can be generated by the tester site very easily, e.g., by sending a data message out of sequence. This clearly

depends on the history, however, only supported values need be used to reach the data transfer state. Therefore, arc R is always executable.[11]

We now consider NFTs which originate from nodes 7, 8, 9 and 10 and which terminate on nodes 2 and 1 (note that nodes 1 and 1' are equivalent). These NFTs terminate a connection and are independent of how the originating state (node) is reached. Now also consider the remaining NFTs viz., f, V and N which are mainly responsible for obtaining data messages that may be in the network before terminating a connection. The table given below shows why a backward path is always possible for these NFTs.

| Arc Label | Applicable CGs | NFTs | How it is possible. |
|---|---|---|---|
| U' | CG11, CG21 | PG18-19 | NSIVR and NSOMI. TINFT. |
| T | CG12, CG22 | P612 | NSIVR and MIs based on N_disc_ind. TINFTs. |
| W | CG12, CG22 | P606 | NSIVR and NSOMI. TINFT. |
| | | P71 | NSIVR and NSOMI. TINFT. |
| | | PF10 | NSIVR and NSOMI. TINFT. |
| X | CG11, CG21 | P506 | NSIVR and NSOMI. TINFT. |
| | | PG10 | TINFT. |
| | | PG16 | NSIVR and NSOMI. TINFT. |
| M | CG12, CG22 | P605 | NSIVR and NSOMI. TINFT. |
| | | PF09 | NSIVR and NSOMI. TINFT. |
| L | CG11, CG21 | P505 | NSIVR and NSOMI. TINFT. |
| | | PG09 | NSIVR and NSOMI. TINFT. |
| | | PG15 | NSIVR and NSOMI. TINFT. |
| O | CG11, CG21 | PF04 | NSIVR. NSOMI. TINFT. |
| | | PG04 | TINFT. |

[11] Recall that an arc is a set of NFTs. When we use an arc instead of an NFT, the description is applicable to all NFTs that are represented by the arc.

| | | | |
|---|---|---|---|
| | | PG08 | TINFT. |
| | | PG14 | TINFT. |
| | | PN6 | NSIVR and NSOMI. TINFT. |
| P | CG12, CG22 | P607-610 | NSIVR and NSOMI. TINFT. |
| | | PF04 | NSIVR. NSOMI. TINFT. |
| f | all | P2-3 | Spontaneous NFTs that communicate with network layer to send and receive data. TINFTs. MIs due to different data message numbers. |
| | | P108 | Spontaneous NFT that sends DR. NSIVR. MIs due to remaining data bytes. TINFT. |
| V | all | P2-3 | Spontaneous NFTs that communicate with network layer to send and receive data. TINFTs. MIs due to different data message numbers. |
| | | P116 | Spontaneous NFT that sends an ERR message. TINFT. |
| N | all | P2-3 | Spontaneous NFTs that communicate with network layer to send and receive data. TINFTs. MIs due to different data message numbers. |

# References


[Arak 83] Araki T., Takada K., Yoshitake S.
A test logic generation method for layered protocol implementations.
In *Proc. of the IEEE COMPSAC*, pages 356-365. IEEE, November, 1983.

[Boch 80] Bochmann G.
A general transition model for protocols and communication services.
*IEEE Trans. on Comm.* Comm-28:624-631, April, 1980.

[Chan 81] Chandrasekaran B., Radicchi S. (Editor).
*Computer Program Testing.*
North-Holland Publishing Company, 1981.

[ClRi 81] Clarke L.A., Richardson D.J.
Symbolic Evaluation methods for program analysis.
*Prentice-Hall Software Series. Program Flow Analysis Theory and Applications.*
Prentice-Hall, 1981, pages 264-301, Chapter 9.
Editors are Muchnick S. S., Jones N.D.

[Gone 70] Gonenec G.
A method for the design of fault detection experiments.
*IEEE Trans. on computers* 19(6):551-558, June, 1970.

[HeRa 81] Henley R. F. L, Rayner D.
*Implementation assessment of transport and network services: An informal description of tests for public comment..*
Technical Report DNACS TM 5/81, National physical laboratory, 1981.

[ISOa 82] Bochmann G. V.
Examples of transport protocol specification.
*ISO /TC 96/ SC 16/ WG 1* , July, 1982.

[lSOb 84] International Organization for Standardization.
A formal description technique based on extended state transition model.
*ISO /TC 96/ SC 16/WG 1 Subgroup B* , March, 1984.

[JeWi 74] Jensen K., Wirth N.
*Pascal user manual and report.*
Springer-Verlag, New York, 1974.

[Kawa 80] Kawaoka T., Yoshitake S., Morino K.
A method for verifying layered protocol products and its application to data communication network architecture products.
In *Proc. of the ICCC*, pages 379-384. IEEE, 1980.

[King 76] King J.C.
Symbolic execution and program testing.
*CACM* 19(7):385-394, July, 1976.

[Knis 82] Knisghtston K. G.
The transport layer.
*Computer Communication Review* 12(3 & 4):14-67, July/October, 1982.
It contains entire ISO/TC 97/SC 16 N1169 document.

[Koha 78] Kohavi Z.
*McGraw-Hill Computer Science Series: Switching and finite automata theory.*
McGraw-Hill Book Company, 1978.

[Lawl 76] Lawler E.L.
*Combinatorial optimization Networks and matroids.*
Holt Rinehart Winston, 1976.

[LiMc 83] Linn R. J., McCoy W. H.
Producing tests for implementations of OSI protocols.
*INWG* , 1983.

[LiNa 83] Linn R. J., Nightingale S. J.
Some experience with testing tools for OSI protocol implementations.
*INWG* , 1983.

[MeFa 76] Merlin P.M., Farber D.J.
Recoverability of communication protocols- Implications of a theoretical study.
*IEEE Transactions on Communications* :1036-1043, September, 1976.

[NaTs 81] Naito S., Tsunoyama M.
Fault detection for sequential machines by transition tours.
*Proc. of FTCS* :238-243, 1981.

[Rafi 85] Rafiq O.
Tools and methodology for testing OSI protocol entities.
In *The Fifth Annual International Symposium on Fault-Tolerant Computing*, pages 184-189. IEEE Computer Society, June, 1985.

[Rayn 81] Rayner D. [Editor].
Protocol implementation assessment: Philosophy and Architecture.
*NPL Report* DNACS 44/81, April, 1981.

[Sari 84] Sarikaya B.
*Test design for computer network protocols.*
PhD thesis, McGill University, March, 1984.

[Simo 82] Simon G., Kaufman D.
An extended finite state machine approach to protocol specification.
In Sunshine C. (editor), *Protocol specification, testing, and verification,* pages 113-134. IFIP WG 6.1, North-Holland Publishing Company, May, 1982.

[Suns 79] Sunshine C.
Formal technique for protocol specification and verification.
*IEEE Computer* 9:20-27, September, 1979.

[Tard 84] Tardos E.
*A strongly polynomial minimum cost circulation algorithm.*
Technical Report 84356 - OR, Research Institute for Telecommunication, October, 1984.
Also in Combinatorica, May 1985.

[Tarj 83] Tarjan R. E.
*Data Structures and Network Algorithms.*
SIAM, Philadelphia, Pennsylvania, 1983.

[UrPr 83] Ural H., Probert R. L.
User guided test sequence generation.
In Rudin H., West C. H. (editor), *Protocol Specification, Testing and Verification, III,* pages 421-436. IFIP, 1983.

[UrPr 84] Ural H., Probert R. L.
Automated testing of protocol specifications and their implementations.
*ACM SIGCOMM* :149-155, 1984.

[Yosh 82] Yoshitake S., Mashio M., Ideguchi S., Katsumata M.
Method for testing data communication products that implement standard Protocols.
In *3rd International Conference on Distributed Computing Systems,* pages 742-747. IEEE, 1982.

[Zimm 80] Zimmermann H.
OSI reference model - The ISO model of architecture for open systems interconnection.
*IEEE Trans. on Communication* Comm-28(4):425-432, April, 1980.