TEST DESIGN FOR COMPUTER NETWORK PROTOCOLS

by

BEHÇET SARIKAYA

March 1984

A thesis submitted to the
Faculty of Graduate Studies and Research in partial
fulfillment of the requirement for the degree of
Doctor of Philosophy.

School of Computer Science,
McGill University,
Montreal.

ABSTRACT

Communication protocol testing can be done with a test architecture consisting of remote Tester and local Responder processes. By ignoring interaction primitive parameters and additional state variables, it is possible to adapt test sequence generation techniques for finite state machines (FSM) to generate sequences for protocols specified as incomplete finite state machines.

For real protocols, tests can be designed based on the formal specification of the protocol which uses an extended FSM model in specifying the transition types. The transition types are transformed into a simpler form called normal form transitions which can be modelled by a control and a data flow graph. Furthermore, the data flow graph is partitioned to obtain disjoint blocks representing the different functions of the protocol. Tests are designed by considering parameter variations of the input primitives of each data flow function and determining the expected outputs. This methodology gives complete test coverage of all data flow functions and tests for unspecified cases can be designed using the control and data flow graphs. The methodology is applied to two real protocols: Transport protocols Classes 0 and 2.

RESUME

Pour tester une implantation de protocole de communication on peut utiliser une architecture en deux parties: Un "testeur" et un "répondeur". Si on ignore les parametres des interactions et variables d'états additionnelles, on peut adapter des techniques de génération de séquences de test dévéloppées pour les automates d'états finis pour générer des séquences de test pour les protocoles spécifiés comme les automates d'états finis incomplèts.

La specification formelle du protocole est utile pour trouver les interactions à appliquer. Les "types de transitions" d'une specification peuvent être transformés en des formes plus simples applelées transitions de forme normal. Les transitions de forme normal peuvent être modélisées avec deux graphes: un pour le contrôle et un pour le flux de données. Le graph de flux de données se divise en blocs disjoints, chaque bloc indique une fonction de flux de données. La conception des tests pour les fonctions de flux de données se fait par variations des paramètres des entrées et la determination des sortiés attendues avec l'aide de ces deux modèles de graphes. Des tests pour les cas inattendus peuvent être aussi basé sur les deux graphes. La methodologie est appliquée à deux protocoles: Les protocoles de Transport Classes 0 et 2.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude and appreciation to my thesis supervisor, Professor G. v. Bochmann of Université de Montréal. He has provided an excellent research environment and invaluable advice in all stages of this thesis. I also thank him for many hours of illuminating discussions and his careful reading of the manuscript and providing excellent criticisms on writing.

I would also like to thank Professor E. Cerny of Université de Montréal for many pleasant and helpful discussions and corrections on the manuscript. He has originated the ideas that are explained in Part 2 of this thesis. I thank him for letting me report on the two Class 0 TP test programs that were designed by him.

Many thanks go to M. Maksud and J.M. Serre of Université de Montréal. Michel helped me debug the Class 0 test programs and my discussions with Jean Marc on Class 2 TP have been very helpful.

I would like to thank my wife, Nursen for drawing the flow graphs and for moral support. Without that support I would never be able to finish this work.

Many people have provided financial support to me, I would like to thank them all: My late father which I would like to pay hommage to his memory, G.v. Bochmann, R.N.S. Horspool (formerly of McGill). I would like to thank Professor M. Newborn for providing funds for printing this thesis by the Computer Center of McGill University.

TABLE OF CONTENTS

LIST OF FIGURES, TABLES AND ALGORITHMS

1.Introduction

1.1.Motivation

The idea of testing a protocol may probably be traced back to the 1950s where the rules of communication between a central processing unit of a computer and its input/output peripherals have first been established. However, the idea of protocol implementation assessment is quite new. It is originated in 1981 with a few reports from the National Physical Laboratory in England [Rayner 81], [HeRa 81], and [Henley 81] and the RHIN project in France [Ansart 81]. Of course, this is not coincidental since the beginning of the 1980s marks also the beginning of wide range use of public data networks linking computers and terminals. It also marks the beginning of implementing higher-level protocols in line with the Reference Model defined by the International Standards Organization (ISO), thus the research groups in NPL and RHIN project came up with the idea of establishing a national assessment, or even certification center of protocol implementations.

Source listings of implementations by various institutions cannot be assumed to be available to the test center. Also, the state of the art of the program verification is far from providing practical tools to verify large concurrent software such as a protocol implementation.

Therefore, testing remains the main tool of the assessment activity and any implementation of a higher-level protocol can be assessed by the test center through a physical connection of the public data network, with the application of a certain number of tests on the implementation.

The selection of interaction sequences for testing is a major problem. This thesis concentrates on the problem of selecting test sequences for protocol testing. We first show in the following chapters that by ignoring interaction parameters, test sequences can be generated using various test methods designed for finite state machines (FSMs) once the FSM model of the protocol is obtained from the protocol specification. When parameter variations are considered, it becomes important to decompose a given protocol. In this thesis, we base the decomposition on the formal specification of the protocol. From a transformed form of the specification we obtain a decomposition into several functions and design tests for each function.

1.2. Test Configuration

ISO standardization activity centers around a model which structures the design of distributed systems into a number of hierarchical layers. Each layer consists of a number of components, using the service of the layer below, it provides the "level N" service to the layer above. This

model is called the Reference Model of Open System Interconnection (OSI) [Zimmermann 80]. The protocol hierarchy of the OSI reference model is shown in Figure 1.1. Protocols up to level 3 in this figure are **lower-level,** above level 3 are **higher-level.**

The OSI reference model serves as a framework for the definition of standard protocols which make interconnection of heterogeneous computer systems possible. A system which implements standard protocols of all layers will be "open" to communicate with any other system, thus heterogeneous systems can be interconnected for the purpose of distributed applications such as data base access, file transfer and terminal access.

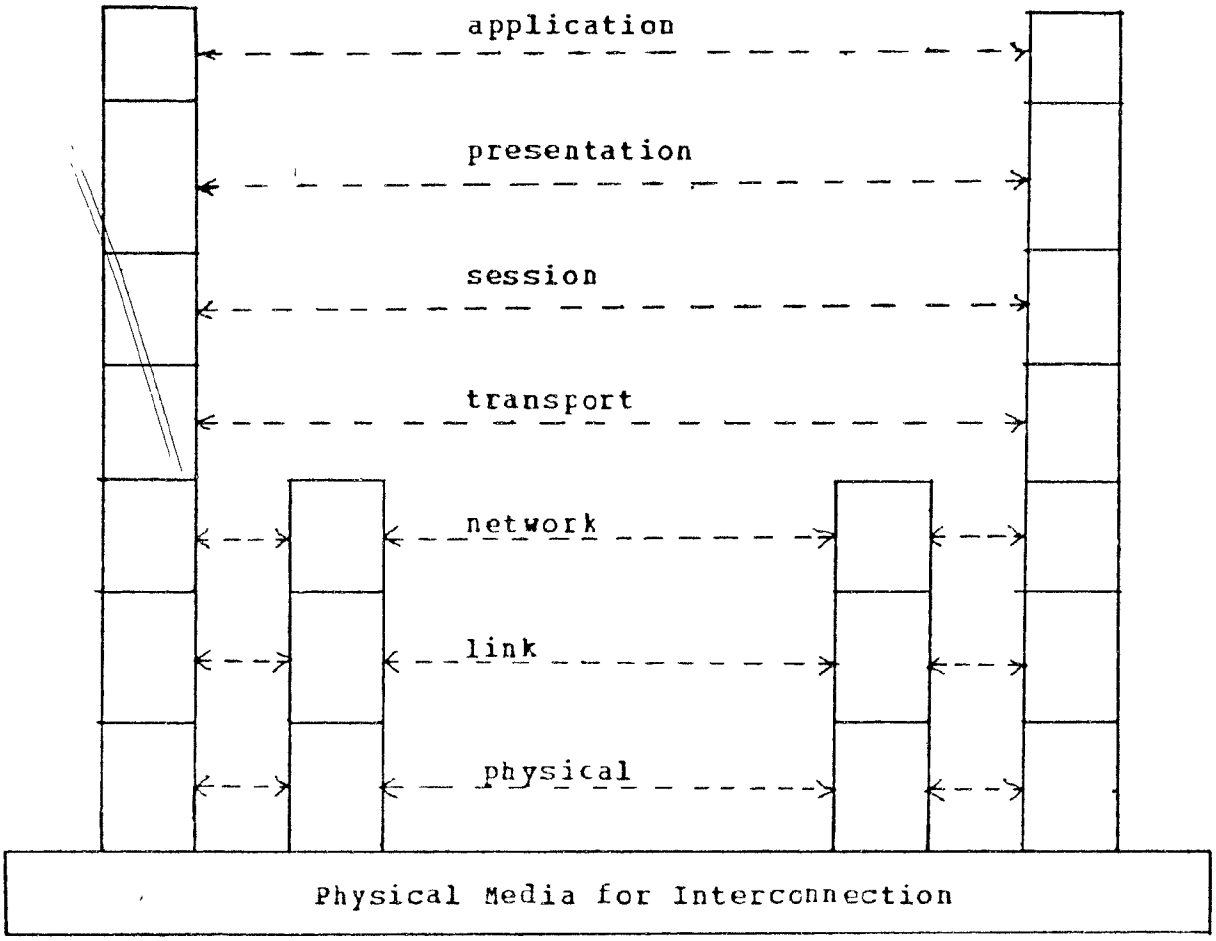


Figure 1.1.Protocol Hierarchy of the OSI Reference Model

[Rayner 81] proposes a basic architecture to be used in testing protocol implementations of level N in the OSI Reference Model. The protocol upper layer interface is assumed to be accessible through a user task which provides stimuli to the implementation. This task is called the Test Responder (we shorten it as R). Since the lower layer is not directly accessible, access to the interface at the physical level is assumed through the underlying data network. At the test center, a task called the Active Tester (we shorten it as T) takes the responsibilty of most of the testing

activity. The active tester behaves like a general entity at level N but may stimulate the implementation with unexpected stimulations, i.e., it may not follow the protocol in some cases, in order to test the robustness of the implementation. The basic architecture is given in Figure 1.2.

[Henley 81] describes a proposal for the test responder. One important design criteria for R is that it should be machine independent. A protocol between T and R is proposed. Using that protocol, T can dictate how R should behave in the next test. The protocol uses state and parameter tables that are interpreted during execution.

Using the same architecture, [BoCeMaSa 83] describes a slightly different approach to the designs of the two modules T and R. Each test is assumed to be stored as an executable task at the test center site and the responder site, therefore, only the test name needs to be transfered from T to R using a fault tolerant loading protocol. Both modules are designed to support the initial connection establishment to load the test tasks. Detailed designs of T and R are given in [CeBo 83].

In the basic architecture of Figure 1.2., the lower layer interface of the implementation is not directly accessible, therefore the implementation cannot be tested for the cases of errors arising from malfunctioning of the lower layer. This is especially important when testing for

an implementation of a network service where it is desirable to introduce X.25 errors to the implementation. Proposals [Rayner 81], [Ansart 81b] have been made to modify the basic architecture such that on the implementation site, a black box (possibly implemented in hardware) is introduced between the public data network and the equipment of the implementation site. Using this black box, it is possible to introduce X.25 errors to the implementation, such as giving a Reset to the implementation whenever desired.

The use of this modification is minimal when testing the implementations of transport or higher-level protocols of Figure 1.1. Therefore, in [BoCeMaSa 83] when testing an implementation of a level N protocol, the level N-1 service is assumed to be correct. An implication of this assumption is that the testing activity progresses from the lowest level to the highest level and the basic architecture is sufficient for this purpose.

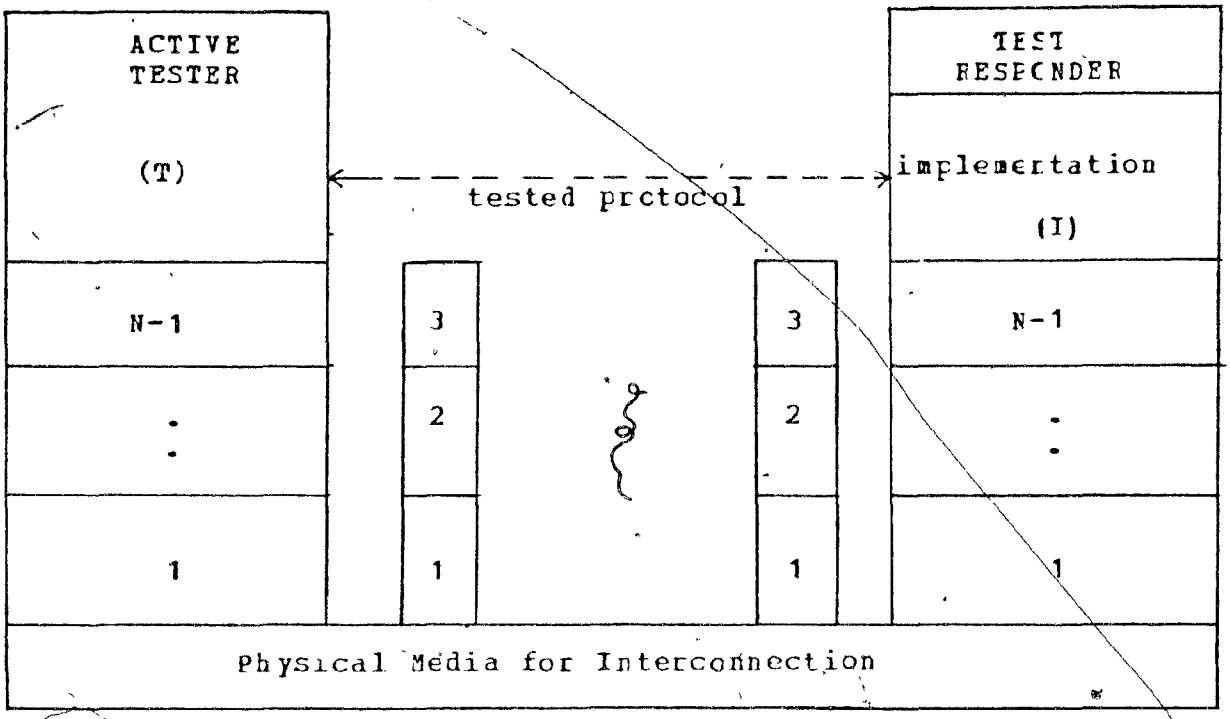


Figure 1.2.Test Architecture for Testing Protocol Implementations

1.3.Formal Specifications of the Protocols

International organizations such as ISO, CCITT provide natural language descriptions of the standard protocols. In what follows we call them **informal specifications.** The need for specifications given in a formal manner is appreciated by these organizations. A transition based formal language described in [FDT 84] has been accepted as a tool for formal specifications of protocols and services.

For a protocol of layer N, there are two levels of specification [Bochmann 83]:
-the **service specification** which describes the overall behavior of the subsystem of level N, and
-the **protocol specification** which defines the behavior of each component of a local system corresponding to the given layer.

The advantages of formally specifying the protocols are numerous: Protocols can be verified for correct operation (absence of deadlocks, etc.) before the implementation phase; if verified, an automatic implementation can be obtained from the formal specification. In this thesis, we show that test design can be based on the formal specification, thus extending the use of protocol specifications to the testing phase.

1.3.1. Structure of a Protocol Entity

A protocol entity is composed of one or more modules and channels connecting the different modules. Figure 1.3. shows an example substructure of a protocol entity consisting of a mapping module and an arbitrary number of abstract protocol modules.

Channels that connect the entity modules to other modules of the same entity or external entities (these channels are called service access points (SAPs)) represent

connections over which interactions are received and sent. In particular, if the protocol entity is a protocol of layer N in the OSI Reference Model, service access points are used to receive/send the interaction primitives from/to the peer entities on the same layer (NSAP in Figure 1.3). These primitives are called protocol data units (PDUs). PDUs are sent/received using the service of the layer N-1. Over other SAPs, the protocol entity provides the layer N services to the layer N+1, thus these SAPs are used to send/receive user interactions (TSAP in Figure 1.3). Internal communication between entity modules are done similarly on channels connecting them (PDU_and_control in Figure 1.3).

In Figure 1.3. the mapping module handles the connection establishment with level N-1 and decodes and checks the parameters of the PDUs received for possible errors and passes the correct PDUs to the proper abstract protocol module. The abstract protocol module provides the level N protocol services to its users over TSAPs. It passes the PDUs to be sent to the peer entities to the mapping module, and the mapping module in turn encodes the PDUs and sends them over the NSAPs.

The Subgroup B FDT supports two modes of communication between modules [FDT 84]:

a)via an infinite FIFO queue, and

b)the "rendezvous" communication which suspends the sending

process until the receiving process acknowledges the exchange. We assume that the communication between the modules of the same entity (internal communication) is of rendezvous type and the external communication can be either

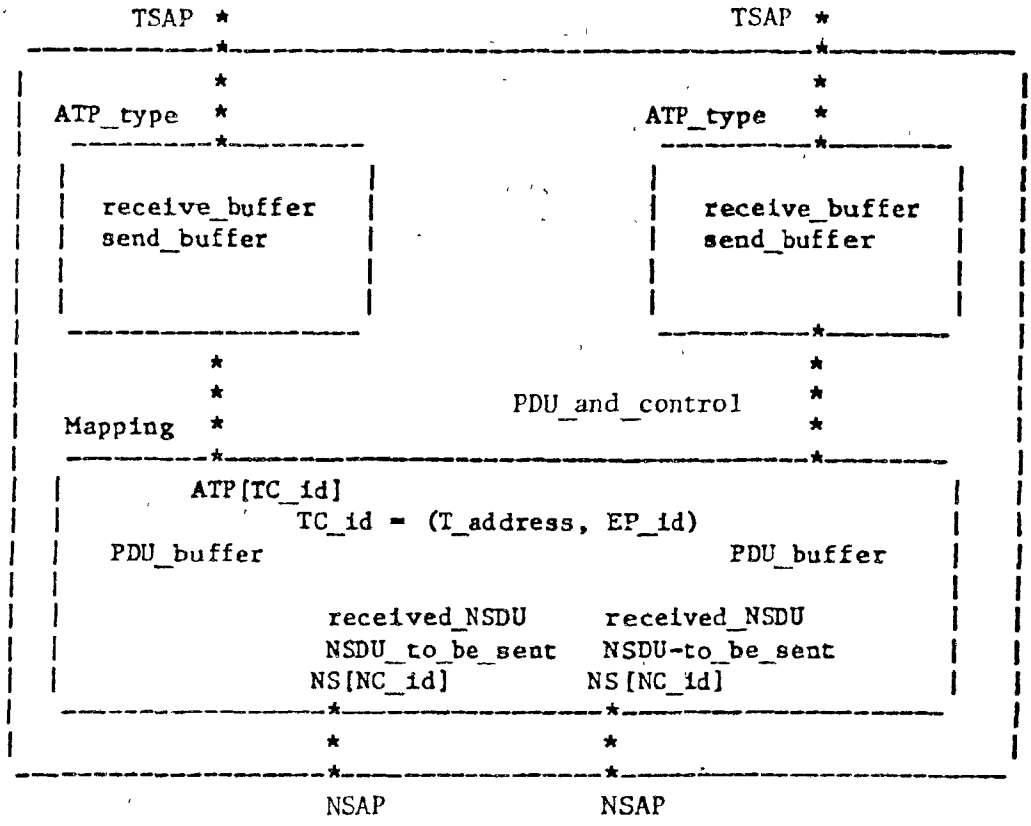


Figure 1.3.Subtructure of a Transport Protocol Entity

1.3.2.FDT

A formal description technique (FDT) for the specification of communication protocols and services [FDT 84] can be used to specify a protocol entity. This language, called Subgroup B FDT, is based on an extended finite state

transition model and the Pascal programming language.

Subgroup B FDT considers each interaction of the specified module with its environment as an atomic event and distinguishes between the interactions received by the module (inputs), and interactions initiated by the modules (outputs). The possible order of interactions of a module (or entity) is described by a formalism which combines finite state machines (FSMs) with the power of a programming language. The state space of a module is specified by a set of variables including a variable called "state" which represents the "major state" of the module.

All Pascal data types (records, arrays, enumeration types, integers, etc.) are supported in Subgroup B FDT. The type of interactions that may occur over the channels are defined as part of the type declarations which precedes the specification of each module. A module (like a Pascal program) is specified by listing type declarations, list of variables, definitions of interface predicates, local procedures followed by **transition types.**

Each transition type is characterized by an **enabling condition** and an **operation.** An enabling condition consists of a boolean expression depending on some of the variables defining the module state (the enabling condition is expressed using special Subgroup B FDT constructs called PROVIDED and FROM clauses) and (possibly) the specification of an input (the input is specified using the WHEN clause).

The PROVIDED clause may include predicates on variables of other modules of the same entity. These predicates are called **interface predicates.** A transition type without input is called a **spontaneous transition.**

The operation of a transition type is to be executed as part of the transition. All Pascal executable language constructs (assignment statement, IF statement, FOR, WHILE, CASE constructs, etc.) are allowed, the operation may also specify the initiation of output interactions with the environment as well as a next major state, using the TO clause.

The Subgroup B FDT has a special construct called ANY clause which is used either to introduce variables local to a transition type or in spontaneous transitions to introduce index variables which makes the spontaneous transition available for any valid value of the index variables. The index variables are used to identify the connections of the module with its environment. More discussion on Subgroup B FDT will be given in Chapter 4.

1.4.Survey of Methods for Test Sequence Selection

Selection of interaction sequences to be applied to a protocol implementation is a major problem in protocol testing. To select test sequences for protocols, we have applied the methods from the areas of finite state machine

testing, software testing, microprocessor testing and control system verification. The existing work in these areas are surveyed in what follows.

1.4.1.FSM Test Techniques

There have been three major approaches to test sequence selection for FSMs: transition tours [NaTs 81], checking sequences [Kohavi 78, Gonenc 70] and characterization sequences [Chow 78].

The above three methods will be used to select test sequences for protocols modeled as FSMs in Chapter 2.

1.4.2.Software Testing

Symbolic execution [King 76], an extension of normal execution of a program can be used in finding individual program paths and then selecting test data to experience the path. Assuming a finite number of paths in a program, global symbolic evaluation [ClRi 81] generalizes symbolic execution to enumerate all the paths. In Chapter 4, symbolic execution will be used to transform a protocol specification into a simpler form.

Functional program testing which views a program as an integrated collection of functions and bases the selection

of test data on the value spaces (domains) over which the functions are defined is shown to be the most reliable technique for discovering the errors [Howden 80]. In particular [Howden 80] identifies two main functions in scientific programs: computational and control. Functional testing requires that in the test data selection, the domain of each program variable be considered and functionally important values are used such as extremal values from domain boundaries or other special values.

Selecting test cases for even a small program is a tedious process. To make test case construction automatic, [PrSkUr 83] proposes a three step methodology consisting of first constructing an English-like test case specification, then implementing this specification via logic programming [Kowalski 79], and finally running the Prolog specification to generate actual test cases. [PrSkUr 83] parametrizes the Prolog program to introduce testing strategies in the test case selection.

It is possible to generate test cases for software in its earlier phases such as intentions, natural language assertions and formal specification. [PrUr 83] develops a methodology to reduce inconsistencies and ambiguities in the software development process by continuously comparing and evaluating the test case reference sets (TCREFs) obtained from intentions, natural language assertions and formal specification. Again, the use of Prolog programs as a high

level representation to generate test cases is emphasized.

Application of functional program testing and test case selection based on formal specification to protocol testing will be discussed in Chapters 5 and 6.

1.4.3.Microprocessor Testing and Control System Verification

In [ThAb 78, ThAb 79, ThAb 79b] microprocessors are tested with inputs which are valid machine instructions and the outputs are compared with the expected responses which are also stored in the machine memory or in the memory of an external tester.

Test sequences are derived based on certain fault assumptions called **fault models** which are derived functionally. For instance, there exists fault models for functions of a microprocessor such as register decoding/encoding, control and data storage (faults in various registers) and data transfer (faults in buses). Allowable faults are formalised for each of the above functions and instruction sequences are derived to guarantee the detection of these faults.

Fault models are helpful in designing tests for primitive decoding/encoding function of the protocol as will be discussed in Chapter 7.

[ThAb 79] introduces a graph model of a

microprocessor. The graph models the data flow during execution of each instruction.

[VaDi 78] proposes a model based on a control and data graph of a control system to formally specify and verify large parallel control systems. Petri nets are used to describe the scheduling of events and conditions and a data graph to describe the primitives defining the data part of the system. Verification can be done in two levels: the control level where the Petri net is analyzed to verify that it is safe, proper, live and well formed, and the schema level where both the control and data graphs are analyzed together to verify that the schema is deterministic and determinate.

We used these ideas in modelling data flow in a protocol specification with a graph to visualize the effects of parameter variations. This topic will be discussed in Chapter 5.

1.5.Survey of Existing Work on Protocol Testing

In this section we describe previous and ongoing work on protocol assessment.

Early efforts on test design for protocols were based on finite state machine (FSM) models of the protocols. For example, the main part of the tests proposed in [HeRa 81],

called protocol tests, were designed to test each transition in the FSM model as follows:

- the implementation is driven into the present state of the transition, and
- the input (either from T or R) is applied, and
- the output is observed.

Another approach has been the use of a logic programming language called Prolog [Kowalski 79] to generate test sequences for protocols [UrPr 83]. A grammar describing the actions of the protocol or service is derived from the specification and a Prolog program is written for the grammar. The output of the program is the test sequences (only transition tours) that involve protocol actions and/or user interactions depending on the grammar used. An advantage of this method is that, using attribute grammars, it is possible to generate test sequences that involve parameter variations [UrPr 83]. The question of writing a Prolog program to accept any grammar (i.e., any FSM) and derive the test sequences accordingly needs to be investigated.

1.5.1. Testing with Reference Implementations

There is an alternative approach to the test configuration described in Section 1.2. A correct implementation called **reference implementation** of the

protocol to be tested replaces the active tester (T) in Figure 1.2. With this configuration, the test programs on both test center and responder sites become a set of user interactions [LiNi 83]. Test sequence generation can be done using the service specification of the protocol. An example service specification for the transport service with Subgroup B FDT is reported in [BoCeLa 81]. Taking a transition tour of the FSM describing the service gives the test sequences for the two communicating users.

In [LiMc 83] a different approach is taken to generate test sequences. A service specification is converted into two grammars (called user-entity grammars) for the two peer entities. Then a composite grammar which gives the sequences consisting of one or more 4-tuples (requests and indications of both entities) is derived. Since the states of the FSM are the nonterminals in the grammar, test sequence is generated starting with the initial state and terminating when the final state is reached. [LiMc 83] describes how the composite grammar can be used as a Markov chain by assigning weights (stationary probabilities) to the productions. These weights are utilized for functional decomposition of the test sequences (connection establishment/ freeing, data transfer, etc.) and/ or generating test sequences for different service classes of a given protocol (for example the transport protocol has 5 service classes) without a separate generator for each service class.

Test sequences generated from a service specification may contain synchronization problems [SaBo 83] since it is difficult to synchronize the testing sides only through the IUT. This is due to the fact that the protocol service is nondeterministic (a protocol may accept or reject a connection request by one of its users) and the architecture of [LiMc 83] involves two implementations. [LiMc 83] mentions the need for modifying some of the automatically obtained sequences in order to obtain synchronizable test sequences. Another problem with the test sequence generation method of [LiMc 83] is that about 1/3 of the resulting sequences are duplicate sequences.

Protocol errors, i.e., PDUs introduced in unexpected states, cannot be introduced to the implementations if the tests involve only user interactions. The test configuration can be modified to include an exception generator on the Tester site which modifies the PDUs generated by the reference implementation. The exception generator interfaces with the test program (called scenario file in [LiNi 83]) in order to introduce the errors required by a given test.

Using a reference implementation at the test center has the disadvantage of the difficulty in finding such an implementation. But once a protocol implementation of level N is tested and all the errors removed, it can be used by the test center when testing protocol implementations of

level N+1.

1.6. Original Contributions of the Thesis

The thesis has the following original contributions to the state of the art of protocol testing:

- Systematic test sequence selection for protocols using the three identified methods from finite state machine test theory, i.e., transition tours, characterization and checking sequences.
- Treatment of the synchronization problem which occurs for certain test sequences obtained using the methods above, and the modifications to the basic algorithms of the above three methods such that only synchronizable test sequences are generated.
- A normalized representation for protocols which is obtained from the formal specification of the protocol by enumerating all control paths and combining submodules.
- The control and data flow graph models of normal form transitions.
- Use of the graph models of the normal form transitions in protocol design validation, i.e., for detecting certain syntactic and semantic errors in the specification.
-A test design methodology for protocols. Deriving control and data flow functions from the flow graphs, the methodology can be used to design tests for all of these functions using parameter variations of the input

primitives. For functions that are not covered in the specification, fault models are proposed to obtain test sequences. For protocol error cases, tests can be designed using the two flow graphs.

-Test design for two real protocols applying the test methodology, namely the transport protocols Classes 0 and 2.

1.7.Organization of the Thesis

The thesis is divided into three parts:

Part 1 establishes the basic theory, in Chapter 2 we discuss test sequence generation for protocol specifications modeled as finite state machines, and in Chapter 3, the software developed to generate test sequences automatically from a given FSM model is discussed. Basic algorithms from the literature are modified in order to generate synchronizable test sequences.

Part 2 describes test design for protocols based on a formal specification in Subgroup B FDT. Chapter 4 establishes the basis for specification based test design: The specification is transformed into single path transitions called normal form transitions. Chapter 5 discusses graphic representations of certain aspects of the normal form transitions: A control graph which models the changes in the major state variable (i.e. FSM aspects), and a data flow graph which models the flow of other protocol

data. Based on these graphs, Chapter 6 defines a methodology for test design.

Part 3 applies the test design methodology to two protocols: The transport protocol classes 0 and 2. In Chapter 7, the tests for an implementation of the Class 0 TP are outlined and Chapter 8 describes a test design for a more complicated protocol, i.e., the Class 2 TP. Finally, in Chapter 9 we state our conclusions.

2.Test Sequence Generation

In this chapter we assume that the protocol to be tested (or its specification) can be modelled as a finite state machine (FSM). A FSM for a protocol can be defined as a quintuple [Kohavi 78]:

M=(I,O,S,D,L) where

i) The input set (I) is the set of the request/ response primitives from the user of the protocol and the set of PDUs from the peer entity.

ii) The output set (O) is the union of the set of indication/ confirmation primitives to the user and the PDUs to be sent to the peer entity.

iii) The set of states (S) is usually selected to distinguish different phases of a connection such as idle, i.e., no connection, waiting for a response from the user or open, i.e., connection is established. One of the states in S represents the state of the machine before any input is applied, this state is called initial state.

iv) D is the state transition function defined as:

D: I x S -> S,

v)L is the output function defined as:

L: I x S -> O.

If the specification does not define any transition for some pairs of (input, state), the resulting FSM model is called **incomplete.** We assume that a FSM of a protocol specification is strongly connected (i.e., any state can be

reached starting with the initial state) and reduced (i.e., there exists no superfluous states) [Kohavi 78].

A FSM model of a protocol can be obtained from its formal specification in Subgroup B FDT by ignoring interaction parameters, predicates of transitions and the effects of parameters on the protocol variables other than the major state variable. We will discuss how to obtain a FSM model from the transformed specification in Chapter 5.

Using the FSM model described above, it is possible to apply the test techniques developed for FSMs to protocols [SaBo 82]. These techniques are used for algorithmically deriving interaction sequences to be applied to test an implementation under test (IUT). According to the distributed character of the test architecture described in Chapter 1, i.e., Tester and Responder, the two sites involved in testing a protocol are only synchronized through the IUT. Therefore some of the interaction sequences generated may contain synchronization problems. The test sequence generation methods and synchronization problems are discussed in more detail in [SaBo 84]. A short review of these issues is given in the remaining part of this chapter.

2.1.Transition Tour Method

A sequence of input symbols which starts with the initial state and includes, at least once, all the transitions defined in the protocol specification is called **a transition tour.** Transition tours can be obtained for any connected FSM. Two algorithms that obtain transition tours of a FSM using different techniques will be discussed in Chapter 3.

2.2.W-Method

This method selects test sequences found from concatenation of sequences from the following two sets of input sequences:

- P, the set containing all partial paths in the testing tree. The testing tree has the transitions (each transition used exactly once) as its branches and states as its nodes.
- W, the characterization set, a set of input sequences which contains for every pair of states a sequence that can distinguish them.

A testing tree exists for every connected FSM and every reduced, completely defined FSM possesses a W-set.

Each sequence in the concatenation of P and W (represented as P.W) is applied starting with the initial state, since the root of the testing tree is the initial

state, and possibly followed by a transfer sequence back to the initial state. This transfer sequence is called **reset.**

In Chapter 3 we give some algorithms to obtain W-sequences for protocols.

2.3.D-Method

This method applies to FSMs that possess a sequence of inputs called distinguishing sequence (DS). A DS is a W-set that has only one member. A D-sequence is selected as follows:

- Apply DS followed by DS starting with every state in order to recognize all the states,
- Apply DS after every transition of the FSM.

Only a subset of FSMs possesses a DS, thus the applicability of this method is restricted to those machines.

It is shown in [Chow 78 and Kohavi 78] that if W- and D-sequences are applicable, they can detect all the faults in the FSM assuming that the IUT behaves like a FSM with a number of states smaller or equal to the specification. A transition tour does not have this property.

Upper bound length formulas for the above three methods and their complexities are given in [SaBo 84]. Actual lengths of test sequences applied to various

protocols are also given. A comparison with the upper bounds shows that actual lengths vary between 20 and 68 percent of the upper bounds depending on the complexity of the machine.

2.4. Synchronization Problem

So far we have assumed that any transition of a FSM can follow any other if they are in correct state order (next state of the previous transition matches with the present state of the current transition). This assumption is valid only if the test sides T and R of the test architecture can be synchronized directly. However, this is not desirable even if T and R may be in the same computer since it would require that the time taken by a sequence of interactions with the implementation involving only T (or R) can be known by R (or T). Thus it is desirable to synchronize T and R through interactions with the Implementation (IUT) by selecting only test sequences that are synchronizable (see below for a definition).

A transition in a test sequence can be seen as composed of an input message to be received by the implementation and zero, one, or two messages sent by IUT to be received by T or R or both. Also the test architecture can be modelled as three FSMs that communicate with FIFO queues. Using these ideas, [SaBo 84] gives the definition of a synchronization problem in a test sequence as follows:

Considering two consecutive basic transitions of the IUT, one of the test modules, say T (or R) faces a synchronization problem if T (or R) did not take part in the first transition and if the second transition requires that it sends a message to IUT.

According to the theorem given in [SaBo 84] any pairs of consecutive transitions in a test sequence should be a synchronizable pair of transitions in order that the test sequence contains no synchronization problems. This property will be used in the algorithms that generate synchronizable transition tours and W-sequences in Chapter 3.

Another basic problem is whether or not a FSM defining the protocol contains intrinsic synchronization problems. This is the case if there exists a transition that cannot be included in any synchronizable test sequence. Such a transition is called not synchronizable. If all transitions from a state are not synchronizable, this state is a nonsynchronizable state. An algorithm for finding intrinsic synchronization problems is outlined in [SaBo 84] and described in detail in Algorithm 2.1. It has a complexity of $O(n^2k)$ where n is the number of states and k is the number of possible inputs.

The language used in Algorithm 2.1. and all other algorithms in this thesis is the Pascal language extended with the construct:

(/ ... /);

where ... can be an English statement or predicate. This construct (taken from Subgroup B FDT) is used to express informally certain aspects of the algorithm.

The algorithm in Algorithm 2.1 assumes that each input to the FSM is specified with an indication of a side that initiates the input (such as Tester (T) or Responder (R) of the test configuration in Chapter 1). Also, each output generated by the FSM is assigned to a side to which the output is sent. Details of internal representation of the FSMs will be discussed in Chapter 3.

Using the side information, Algorithm 2.1 finds the set of all the sides from which a given state i can be reached through the transitions leading to state i and then checks if there is any transition from state i which should be initiated by a side that is not in the set found. All nonsynchronizable transitions (if any) from all states are found accordingly.

```
var
    nextstate:array[1..maxproduction] of integer;
    input_side,out_side1,out_side2:array[1..
            maxproduction] of integer;
    (*side information for input and outputs, and
      maxproduction is a parameter representing
      the maximum number of transitions accepted*)

procedure intrinsic(nonsyn,nonsynstate:integer);
    (*nonsyn:no. of nonsynchronizable transition,
    nonsynstate:no. of nonsynchronizable states *)
    arriving : array[1..maxnbside] of sendreceive;
(*maxnbside is a parameter representing the number of
distict sides allowed*)
    adr,i,j,k,term:integer;
begin

nonsyn:=0; nonsynstate:=0; (*initialize*)
for i:=1 to maxnbstate do (*for all the states*)
begin
    for k:=1 to maxnbside do (*for all the sides*)
        arriving[k]:=0; (*initialize arriving*)
    if i=1 then j:=2 else j:=1; (*initialize j*)
    repeat (*for all transitions*)
        (*find nextstate of the transition from state i
          under the next input*);
        for term:=1 to maxnbterm do
        begin
         adr:=i*maxnbterm + term;
         if nextstate[adr] = i then
         begin
            arriving[input_side[adr]]:=input_side[adr];
            if out_side1[adr]<>0 then
             arriving[out_side1[adr]]:=out_side1[adr];
            if out_side2[adr]<>0 then
             arriving[out_side2]adr]]:=out_side2[adr];
             if (/state i can be arrived from all the sides/)
                then goto 1;(*no more checks for state i*)
         end; (* of nextstate[adr] = i*)
        end; (* of for term*)
        j:=j+1;
        if j=i then j:=j+1;
    until j > maxnbstate;
(*now check the transitions from the same state,
  i.e., state i *)
    for term:=1 to maxnbterm do(*for all terminals*)
        adr:=i*maxnbterm+term;
        if (arriving[input_side[adr]] = 0) then
        begin
            print('transition from ',i,' on input ',term,
                'is not synchronizable');
            nonsyn:=nonsyn+1;
```

```
            end;
        if (/all transitions are not synchronizable/)
            then begin
                print('state ',i,' is nonsynchronizable');
                nonsynstate:=nonsynstate+1;
            end;

    1:end; (* of all states *)
    end; (*of intrinsic *)
```

Algorithm 2.1.Algorithm to Detect Intrinsic Synchronization Problems

2.5.Specification Enhancements for Testing

In order to make the W- and D-methods applicable when the original protocol specification does not possess a DS and/or a W-set, the following two approaches can be taken:

2.5.1.Special Test Transitions

The protocol specification may be enhanced by defining special test interactions and transitions called "read state" and "set state" [Piatkowski 80]. The "read state" input becomes a W-set and DS by definition thus these two methods can be applied and the resulting sequences have minimal lengths since the W-set/DS is of minimal length and all the transfer sequences necessary (including resets) can be implemented as "set states" which are of length one. A generalization of the special test transitions will be given in Chapter 6.

2.5.2.Completing Specifications

The specification may be completed until a W-set or DS is obtained by introducing an error state. This approach guarantees a W-set but not a DS. Synchronization considerations concerning the outputs for the added transitions are discussed in [SaBo 84].

2.6.Complexity of Test Sequence Generation

The D-method has the highest complexity mostly due to the high complexity for obtaining a DS [SaBo 84]. When generating synchronizable test sequences it is assumed that the complexity of checking for intrinsic synchronization problems $O(n^2k)$ becomes the most important factor, thus by adding the n^2k to the complexities of algorithms it is possible to obtain the complexities for finding synchronizable test sequences.

The complexity of the test sequences obtained from the FSM models (ignoring parameters, additional state variables, etc.) of the real protocols is not too high (see Table 2 in [SaBo 84]). Taking FSM models as approximations to the real protocols, FSM test techniques are useful in deriving test sequences. In Chapter 3 we explore the algorithms to implement some of them.

Complete testing of a protocol requires consideration

of parameter variations, additional state variables, etc. The resulting model is too complex because of the very high number of different inputs introduced by parameter variations of interactions and the very high number of states introduced by protocol variables used in the enabling condition of the transition types of a Subgroup B FDT specification. In order to reduce the complexity of complete testing, it becomes necessary to decompose the protocol into functions and design tests for these functions which are less complex than the complete protocol. This topic will be discussed in Part 2 of the thesis.

3.Test Sequence Generation Software

Programs written to generate test sequences according to the transition tour and W-methods of Chapter 2 will be described in this chapter. These programs read the input automaton which may be incomplete, check whether it is connected and minimal and whether it has any intrinsic synchronization problems. Then they generate a test sequence. Two programs have been developed for the transition tour method: one using a depth-first-search algorithm (DFS) [Tarjan 72], the other using a random input selection algorithm [NaTs 81]. One program has been developed for the W-method. These programs have been written in Pascal and are running under the operating system VMS of VAX 11/780 computer. A user manual is provided in [Sarikaya 84b].

In the following parts of this chapter we discuss the input and output formats and the internal representation of the test sequences generated by the programs, as well as modifications made to the basic algorithms [Tarjan 72, NaTs 81, Chow 78] in order to generate synchronizable test sequences.

3.1.Input and Output Formats and Internal Representations

The three programs share routines for error processing, automaton input, test sequence output and checks for intrinsic synchronization problems.

3.1.1.Automaton and Test Sequence I/O

The I/O formats depend on whether the sequence generated will be synchronizable or not. For the depth-first-search (DFS) algorithm which generates a transition tour without checking for synchronization, the input format is as follows. Each line of input represents one transition and has the form:

present state ß input ß output ß next state

The present state in the first line of the input file becomes the initial state.

An example input machine is shown in Figure 3.1 which models the Class 0 Transport protocol. An output of two symbols is represented by separating the two symbols by a comma, as shown in Figure 3.1.

If the input machine is incomplete, it is completed by the programs automatically. An error state is added and all incomplete transitions are filled as transitions leading to this state by giving the output ERROR.

The generated test sequence is stored in a double-line

format (states and inputs respectively) in an output file to be printed after the program terminates. A test sequence can be stored using many of these double lines depending on its length. In the following sections we include an example printout of the output file created by each program.

The input format changes slightly for the other programs, which also consider synchronization. The I/O symbol is replaced by an indication of the initiator, and the I/O symbol. The initiator indication has the value '^' when the output is empty, i.e., when no output is generated for a given input. No output is indicated by "VIDE".

The input machine of Figure 3.1 is shown in Figure 3.2. with the initiator indicated, where T stands for the Tester and R for the Responder in the test architecture of Chapter 1.

Synchronizable test sequences are stored in the same format as described above where the initiator indication is given as prefixes to the input and output symbols.

3.1.2. Internal Representation

Arrays are used to represent the machine internally. All four items in a line are converted into integers. Next states and outputs are stored in two arrays which are accessed by the index:

state * maxnbterm + term

where state is the integer corresponding to the input state, maxnbterm is the maximum number of terminals, i.e., inputs, allowed and term is the integer corresponding to the terminal. State number one is the initial state, zero is the error state (if any). Specified terminals and outputs are numbered as they appear in the input file. Output number zero is the output ERROR (if any).

A test sequence is represented as two arrays, one for terminals (inputs) and the other for states. When outputting the test sequence, the integers in the internal representation are converted into the symbols given by the user with the help of the conversion tables created during the input phase.

3.1.3. Synchronization Checks

Programs that generate synchronizable test sequences check the machine for any intrinsic synchronization problems. These checks are done by the algorithm given in Algorithm 2.1.

1	CC	ignore	1
1	DT	ignore	1
1	DR	ignore	1
1	CR	T_Cind	2
1	T_Creq	CR	3
2	T_Cresp	CC	4
2	DR	Err	1
2	DT	Err	1
2	CC	Err	1
2	CR	Err	1
2	T_Dreq	DR	1
3	CC	T_Cconf	4
3	DT	ignore	1
3	DR	T_Dind, N_Dreq	1
4	DT	T_DTind	4
4	T_DTreq	DT	4
4	T_Dreq	N_Dreq	1
4	CR	Err	1
4	DR	N_Dreq	1
4	N_Dind	T_Dind	1
4	N_Rind	T_Dind	1

Figure 3.1.An Input Machine Modelling Class 0 TP

1	T	CC	^ VIDE	^ VIDE	1
1	T	DT	^ VIDE	^ VIDE	1
1	T	DR	^ VIDE	^ VIDE	1
1	T	CR	R T_Cind	^ VIDE	2
1	R	T_Creq	T CR	^ VIDE	3
2	R	T_Cresp	T CC	^ VIDE	4
2	T	DR	T Err	^ VIDE	1
2	T	DT	T Err	^ VIDE	1
2	T	CC	T Err	^ VIDE	1
2	T	CR	T Err	^ VIDE	1
2	R	T_Dreq	T DR	^ VIDE	1
3	T	CC	R T_Cconf	^ VIDE	4
3	T	DT	^ VIDE	^ VIDE	1
3	T	DR	R T_Dind	T N_Dreq	1
4	T	DT	R T_DTind	^ VIDE	4
4	R	T_DTreq	T DT	^ VIDE	4
4	R	T_Dreq	T N_Dreq	^ VIDE	1
4	T	CR	T Err	^ VIDE	1
4	T	DR	T N_Dreq	^ VIDE	1
4	T	N_Dind	R T_Dind	^ VIDE	1
4	T	N_Rind	R T_Dind	^ VIDE	1

Figure 3.2.The Machine in Fig.3.1 With Initiator Indication

3.2.DFS Tour Program

This program (DFSTOUR) takes the input automaton as a connected graph and carries out a DFS of this graph. The algorithm is a slightly modified (to generate transfer sequences) version of [Tarjan 72]. This algorithm invokes a routine called transfer to generate a transfer sequence [SaBo 84]. The transfer routine finds a transfer sequence from the state in its first parameter to the state in its second parameter. The sequence found (not necessarily of minimal length) is added to the tour. The routine is outlined in Algorithm 3.1.

The tour generated by this program and all the other programs described in this chapter contain only specified transitions. Unspecified transitions are excluded from the tour. A DFS tour generated for the machine of Figure 3.1 is listed in Figure 3.3. This tour has a length of 37, thus it contains three more transitions than the minimal tour in [SaBo 84]. The DFS tour program does not necessarily generate tours of minimal length.

```
var nextstate : array[1..maxproduction] of integer;

procedure transfer(st1,st2:integer);
(*generates a transfer sequence from st1 to st2*)
(*and adds the sequence to the transition tour*)
var levels,terms,saveterms:array[0..maxnbstate] of integer;

    procedure assignlevel(state,level:integer);
    (*This recursive procedure assigns a "level", i.e.,
    distance from the start state st1 to the current
          "state"*)
    var
        term,adr,w:integer;

    begin (*find levels,terms, i.e. inputs from state*)
    levels[state]:=level;
    for term:=1 to maxnbterm do
    begin
        adr:=state*maxnbterm+term;
        w:=nextstate[adr];
        if levels[w] > level+1 then
        begin
         terms[state]:=term;(*store input of the transfer
                sequence in the array terms*)
         if w=st2 then saveterms := terms;(*transfer sequence
                found save it*)
         assignlevel(w,level+1); (*find levels of
             other states*)
        end;
    end; (*of all terminals*)
end; (*of assignlevel*)

begin (*transfer*)

    assignlevel(st1,0); (*assign level 0 to initial state*)
    st:=st1; (*initialize*)
    (*add the transfer sequence in saveterms to the tour*)
    repeat (*until st=st2*)
        (/add (st,saveterms[st]) to the tour/);
        st:=nextstate[st*maxnbterm+saveterms[st]];
    until st=st2;

end; (*of transfer*)
```

Algorithm 3.1.Transfer Sequence Finding Algorithm

```
1 CC       1 DT       1 DR       1 CR       2 CC    1 CR       2 DT    1 CR
  ignore     ignore     ignore     T_Cind     Err     T_Cind     Err     T_Cind

2 DR    1 CR       2 CR    1 CR       2 T_Cresp  4 DT
  Err     T_Cind     Err     T_Cind     CC         T_DTind

4 DR       1 CR       2 T_Cresp  4 CR    1 CR       2 T_Cresp  4 T_Dreq
  N_Dreq     T_Cind     CC         Err     T_Cind     CC         N_Dreq

1 CR       2 T_Cresp  4 T_DTreq  4 N_Dind  1 CR       2 T_Cresp
  T_Cind     CC         DT         T_Dind    T_Cind     CC

4 N_Rind   1 CR       2 T_Dreq   1 T_Creq  3 CC       4 DR       1 T_Creq
  T_Dind     T_Cind     DR         CR        T_Cconf    N_Dreq     CR

3 DT       1 T_Creq   3 DR
  ignore     CR         T_Dind,N_Dreq
                                          1
```

Figure 3.3.A DFS Tour Generated by DFSTOUR Program

3.3. Random Tour Program

It is possible to generate minimal transition tours by including the randomly selected transitions into the tour and reducing the unneccessary part from the last new transition once an unvisited transition (called **zero-input**) is added to the tour. The algorithm described in [NaTs 81] was programmed.

There are three procedures of main interest in the random tour program (RANTOUR) which implements the above algorithm modified to generate synchronizable tours. First we describe a function called "synchronizable" which returns

"true" if the pair (state,input) can be added to the tour without any synchronization problems in Algorithm 3.2.

```
type sendreceive=(null, send, receive);
var
    lastaction:array[1..maxnbside] of sendreceive;
    input_side,out_sidel,out_side2:array[1..maxproduction]
                of integer;
    firstransition:boolean;
(*lastaction is initialized to "null"s,
  firstransition is initialized to "true"*)

function synchronizable(state,input:integer):boolean;
var
    i, adr : integer;
begin
    (*lastaction contains the sides information
        from the last transition added*)
    adr:=state*maxnbterm+input;
    i:=input_side[adr];
    if lastaction[i] = null and not firstransition
    then
        synchronizable:=false
    else begin
    firstransition:=false;
    synchronizable:=true;
(*update lastaction array*)
    for j:=1 to maxnbside do
        if j=i then lastaction[i]:=send
        else lastaction[i]:=null;
    if out_sidel[adr] <> null then
    lastaction[out_sidel[adr]]:=receive;
    if out_side2[adr] <> null then
    lastaction[out_side2[adr]]:=receive;

end;

end; (*of synchronizable*)
```

Algorithm 3.2. Synchronization Check Algorithm

Secondly, there is the reduction procedure (see Algorithm 3.3) which eliminates redundant pairs included in the tour between the last new transitions.

```
var
    state, test:array[1..maxnbinput] of integer;
    (*arrays to store the tour, state for states
      and test for inputs, maxnbinput is a
      parameter of the program *)
procedure reduce(i,j:integer);
begin (*i is a pointer to the beginning and j to the
        end of the subsequence to be reduced*)
r:=i; (*r and s are internal pointers*)
while (r < j) do
begin
    s:=j;
    while (s > r) do
    begin
        if state[r]=state[s] then
        begin (*check for sychronization
            by calling Algorithm 3.2 before reducing*)
         if (/(r-1)th transition can be followed by
           s th transition without synchronization
           problems/) then
         (/delete the transitions from r th to (s-1)th/)
         else begin
          (/find the first transition that precedes s th
          synchronizably/);
          (/delete transitions up to (s-1)th/);
         end
        end
        else s:=s-1;

    end; (*of while s > r*)
    if s <= r then r:=r+1;

end; (*of while r < j*)

end; (*of reduce*)
```

Algorithm 3.3. Reduction Algorithm

The main procedure (RANTOUR) (Algorithm 3.4) uses a random number generator given in [ChDa 83] to select inputs randomly. The pointer T keeps the count of newly added transitions, once T reaches the number of specified transitions (nbtrans), the tour is completed.

A tour generated for the machine of Figure 3.2. is

listed in Figure 3.4. This sequence has a minimal number of transitions. Since the routine which finds transfer sequences does not necessarily find minimal transfer sequences, RANTOUR program may not always give sequences of minimal length.

```
var
    nextstate:array[1..maxproduction] of integer;
    nbtrans:integer;
procedure RANTOUR;
var adr, T, u, q0, x, q : integer;
    begin
    T:=1;
    (/generate a random input x which is specified from
        initial state q0/);
    if synchronizable(q0,x) then(*update lastaction/)
            (/add (q0,x) to the tour/);
        adr:=q0*maxnbterm+x;
        q:=nextstate(adr);
    while T <= nbtrans do
    begin
        u:=T-1;
        repeat (*until x is zero-input*)
            q0:=q;
            u:=u+1;
            repeat (*until synchronizable*)
             (/generate a random x specified from q0/);
             adr:=q0*maxnbterm+x;
             q:=nextstate(adr);
            until synchronizable(q0,x);
        (/add (q0,x) to the tour/);

        until (q0,x) is zero-input;

        reduce(T,u); (*Algorithm 3.3*)
        T:=T+1;

    end; (*of while T<=nbtrans*)

end; (*of RANTOUR*)
```

Algorithm 3.4.RANTOUR Algorithm

```
1 T DR   1 T CR       2 R T_Cresp  4 T DT       4 T N_Dind  1 T CR
  -        R T_Cind     T CC         R T_DTind    R T_Dind    R T_Cind

2 R T_Dreq  1 R T_Creq  3 T DR
  T DR      T CR        R T_Dind, T N_Dreq

1 R T_Creq  3 T CC      4 T DR      1 T DT  1 T CC  1 T CR       2 T DR
  T CR      R T_Cconf   T N_Dreq    -       -       R T_Cind     T Err

1 T CR      2 T CR   1 T CR      2 R T_Cresp
  R T_Cind    T Err    R T_Cind    T CC

4 T N_Rind  1 T CR      2 T CC   1 T CR      2 T DT   1 T CR
  R T_Dind    R T_Cind    T Err    R T_Cind    T Err    R T_Cind

2 R T_Cresp  4 T CR   1 T CR      2 R T_Cresp
  T CC         T Err    R T_Cind    T CC

4 R T_DTreq  4 R T_Dreq  1 R T_Creq  3 T DT  1
  T DT         T N_Dreq    T CR       -
```

Figure 3.4.A Synchronizable Tour Generated by RANTOUR

3.4.W-Method Program(SWMAIN)

The program first finds a W-set (guaranteed to exist since the machine is completed if necessary), and then lists all the subsequences in the set P.W, where P represents the testing tree [Chow 78]. SWMAIN is based on the program called CHOW which is described in [ChLeLeRi 81]. CHOW implements the method of [Chow 78] using the algorithm of [Gill 62] in finding a minimal W-set. When a machine

possesses a W-set of length one, the original CHOW program was finding a nonminimal W-set for the machine. Thus we added a routine (Algorithm 3.5) to check if the machine possesses a W-set of length one. Also, CHOW was not generating reset sequences, hence in SWMAIN the transfer sequence algorithm (Algorithm 3.2) is used. Since a reset sequence takes the machine to the initial state, the second parameter of the call to the transfer routine is specified as 1.

```
type
    pt = ^symbol;
    symbol=record val:integer;
           other,next:pt
    end;
var
    action:array[1..maxproduction] of integer;
           (*outputs of the transitions*)
    W:array[1..maxnbstate] of pt;
function checkforone:boolean;
(*checks if the machine posseses a W-set of length
  one and if it does returns the W-set in "W"*)
var
    alldifferent:boolean;
    adr1,adr2,term,count,i,k:integer;
    currentpt:pt;
begin
    checkforone:=false; (*initialize*)
    for term:=1 to maxnbterm do
    begin
        count:=1;
        for i:=1 to maxnbstate-1 do
        begin
            alldifferent:=true;
            for k:=i+1 to maxnbstate do
             adr1:=i*maxnbterm+term;
             adr2:=k*maxnbterm+term;
             alldifferent:=action(adr1)=action(adr2);
            if alldifferent then count:=count+1;
        end; (*of i:=1 to maxnbstate-1*)
(*check if count equals number of states*)
    if count=maxnbstate then
    begin
        new(currentpt);
        with currentpt^ do
        begin
         val:=term;
         other:=nil
         next:=nil;
        end;
        W[1]:=currentpt;
        checkforone:=true;
    end;

    end; (*of term:=1 to maxnbterm*)
end checkforone;
```

Algorithm 3.5.Checkforone Algorithm

SWMAIN uses a slightly modified version of the transfer routine to generate synchronizable transfer sequences. The modified routine is used in two places when generating synchronizable W-sequences:

i)In generating synchronizable reset sequences;

ii)In synchronizing the subsequences in P.W by generating a synchronizable reset sequence from the final state of the preceding subsequence to the initial state of the succeeding subsequence when necessary. A procedure to check a subsequence in P.W and add the above sequences if necessary is outlined in Algorithm 3.6.

```
type
    transition:record=state,input,action,next_state:integer;
        first_transition,last_transition:boolean
    end;
var
    state,input:array[1..maxnbinput] of integer;
    PW:array[1..maxinpw] of transition;
    (*maxnbinput and maxinpw are the parameters of the
      program*)
    i:integer;
(*initialize*)
i:=1;
(*state,input and PW are filled in SWMAIN*)

repeat (*for all the sequences in P.W*)
    with PW[i] do
    begin
    if not synchronizable(state,input) then
     if first_transition and (i > 1) then
        transfer(1,1); (*generate synchronizing sequence
            to synchronize two consecutive subsequences*)
    else
        print('P.W contains nonsynchronizable
               subsequences');

    if last_transition do
    if next_state <> 1 then
        transfer(next_state,1); (*generate a
            synchronizable transfer sequence
            to the initial state *)
    (/print the sequence/);
    i:=i+1;
    end; (* of with PW[i]*)

until i < maxinpw;
```

Algorithm 3.6.Synchronization Checks of P.W

A W-sequence found by SWMAIN for the machine of Figure 3.2 is listed in Figure 3.5. The length of this sequence is 69 and it contains only the specified transitions. The W-set is DR, as reported in [SaBo 84]. The sequence in Figure 3.5 contains no separate reset sequences since the W-set always

brings the machine to the initial state.

3.5.D-Method

We have not implemented the third test sequence generation method, namely checking sequences. A discussion on the implementation of the algorithms of [Gonenc 70] (ignoring synchronization problem) can be found in [Guitton 84].

```
1 T DR  1

1 T CC  1 T DR  1

1 T DT  1 T DR  1

1 T DR  1 T DR  1

1 T CR      2 T DR    1
  R T_Cind    T Err

1 T CR      2 T CC    1 T DR  1
  R T_Cind    T Err       -

1 T CR      2 T DT    1 T DR  1
  R T_Cind    T Err       -

1 T CR      2 T DR    1 T DR  1
  R T_Cind    T Err       -

1 T CR      2 T CR    1 T DR  1
  R T_Cind    T Err       -

1 T CR      2 R T_Cresp  4 T DR       1
  R T_Cind    T CC         T N_Dreq

1 T CR      2 R T_Cresp  4 T DT       4 T DR       1
  R T_Cind    T CC         R T_DTind    T N_Dreq

1 T CR      2 R T_Cresp  4 T DR       1 T DR  1
  R T_Cind    T CC         T N_Dreq       -

1 T CR      2 R T_Cresp  4 T CR    1 T DR  1
  R T_Cind    T CC         T Err       -

1 T CR      2 R T_Cresp  4 R T_Dreq  1 T DR  1
  R T_Cind    T CC         T N_Dreq       -

1 T CR      2 R T_Cresp  4 R T_DTreq  4 T DR       1
  R T_Cind    T CC         T DT            T N_Dreq

1 T CR      2 R T_Cresp  4 T N_Dind  1 T DR  1
  R T_Cind    T CC         R T_Dind       -

1 T CR      2 R T_Cresp  4 T N_Rind  1 T DR  1
  R T_Cind    T CC         R T_Dind       -

1 T CR      2 R T_Dreq  1 T DR  1
  R T_Cind    T DR          -

1 T CR      2 R T_Cresp  4 T N_Rind  1
  R T_Cind    T CC         R T_Dind

1 R T_Creq  3 T DR                1
  T CR        R T_Dind, T N_Dreq

1 R T_Creq  3 T CC        4 T DR       1
  T CR        R T_Cconf     T N_Dreq

1 T CR      2 R T_Cresp  4 T N_Rind  1
  R T_Cind    T CC         R T_Dind

1 R T_Creq  3 T DT  1 T DR  1
  T CR          -       -

1 T CR      2 R T_Cresp  4 T N_Rind  1
  R T_Cind    T CC         R T_Dind

1 R T_Creq  3 T DR                1 T DR  1
  T CR        R T_Dind, T N_Dreq      -
```

Figure 3.5.A Synchronizable W-sequence Generated by SWMAIN

Part 2

In Part 2 we develop some theory and a methodology for test design based on a specification of the protocol given in the Subgroup B FDT language. A protocol is assumed to be tested for conformance to its formal specification. As introduced in Chapter 1, the Subgroup B FDT supports modular protocol specification using an extended state transition model. Interaction primitives can have parameters, each module of the specification has its own state variables and a set of transitions corresponding to external and internal events. State variables include a major state variable and additional variables used in the specification.

We classify the protocol functions as follows:

Control Related Functions: These functions are related to the changes of the major state introduced by the transitions, and

Data Flow Related Functions: These functions arise from the flow of data from input interaction primitive parameters to the output interaction primitive parameters. We show how both of the above types of functions can be derived from the specification.

In Chapter 4, syntactic transformations are applied to the specification, and submodules are combined. The resulting specification has transitions with a single control path in the action. These transitions are called

normal form transitions.

Chapter 5 uses normal form transitions to derive the control and data flow functions. The control functions are derived from the finite state machine model of the normal form transitions. The data flow in the normal form transitions is modeled by a data flow graph. A partitioning of this graph gives rise to the data flow functions in terms of the **blocks** of the partition. Various dependencies among these blocks are discussed.

Finally, Chapter 6 outlines a **test design methodology** for protocols. Parameter variations for interaction primitives are considered as the main tool for generating test sequences. This variation is guided by a dependency classification and the structure of the blocks. The objectives of the different test categories, and multiple connection tests are discussed.

4.Transformations on Protocol Specification

In this chapter, we propose various transformations on the transition types of a protocol specification in order to obtain the transitions in a form which has input primitives (optional) from external interaction points, an action with a single path possibly containing one or more output statements to the external interaction points and a predicate modified to handle path conditions. Each transition in this form is called a **normal form transition.**

The transformations leading to normal form transitions can be done in two phases: In Phase 1, syntactic transformations are applied to FDT constructs [FDT 84]. **Symbolic execution** is used to enumerate paths and find path conditions in the BEGIN block. In Phase 2, modules are combined by combining the transitions with interactions. Finally, we discuss spontaneous transitions and nondeterminism in protocol specifications.

4.1.Sample Transition Types

The transformations of this chapter will be explained using an example protocol specification. This protocol has two modules called module1 and module2 and the substructure of the protocol is shown in Figure 4.1.

Two example transition types of this specification are

given in Figure 4.2. CONNECT_req over the interaction point called chanl in Figure 4.1. is an input primitive, A, B and C are expressions involving module variables and the input primitive parameters, S1 and S2 are blocks of assignment statements, and module2.event1 is an output statement to module2. **FROM** and **TO** clauses of Subgroup B FDT are used to specify present and next major state values, respectively. The second transition type given in Figure 4.2. has event1 which can be received from the first module as an input primitive with parameters called data_unit and length. S3 represents a block of statements and CONNECT_ind is an output to an external entity over chan2.

In the first transition type of Figure 4.2, the **BEGIN** block has two paths (because of the "if" statement) and it has an inter-module output statement which cannot be directly observed from external interaction points.

4.1.1.Normal Form Transitions

The resulting normal form transitions obtained by the transformations of this chapter to transition types like the ones in Figure 4.2 have the following form:

```
Normal Form Transition Block:

    ANY list_1
    [WHEN inp_1]
    PROVIDED pred_1
    BEGIN

        action_1

    END

    .
    .
    .

    ANY list_n
    [WHEN inp_n]
    PROVIDED pred_n
    BEGIN

        action_n

    END

end Normal Form Transition Block;
```

where [WHEN inp_i]° shows that some normal form transitions (those corresponding to spontaneous transition types) do not have **WHEN** clauses.

In general, there are more normal form transitions than transition types in the original specification since the paths in the transition types are enumerated. In order to obtain a finite number of normal form transitions the number of paths in any BEGIN block should be finite. Thus, we assume that no BEGIN block in the protocol specification contains loops with variable bounds.

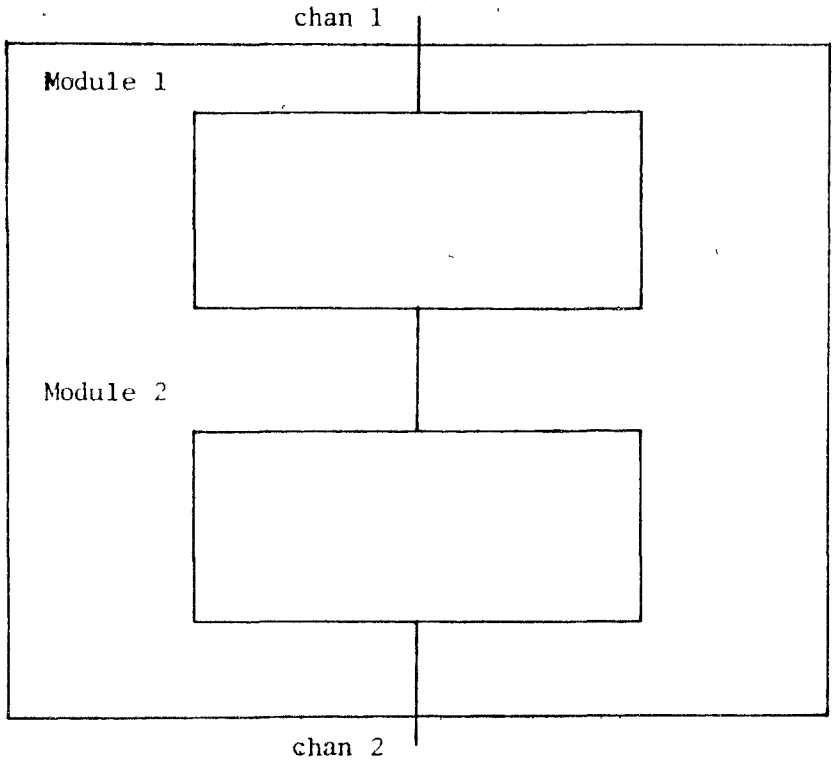


Figure 4.1.A Protocol Entity with Two Modules

```
(in module1)

    WHEN chan1.CONNECT_req
    FROM idle
    PROVIDED A and (B or C)
    TO connecting
    BEGIN

        if D then S1
            else S2;
        module2.event1(CONNECT_req,X+2)

    END;

(in module2)

    WHEN module1.event1(data_unit,length)
    PROVIDED data_unit = CONNECT_req and length <= 10
    BEGIN

        S3;
        chan2.CONNECT_ind

    END;
```

Figure 4.2.Two Sample Transition Types in FDT

4.2.Transformations on FDT Constructs (Phase I)

In Phase 1, we apply certain transformations on various constructs of the FDT. First FROM/TO clauses are eliminated, then conditional statements in the BEGIN block are removed. These transformations are done separately on all transition types of all the modules.

4.2.1.FROM/TO Clauses

The FROM clause is removed and an equality relation on the major state variable is added to the PROVIDED clause of the transition type as a conjuction. As an example, the FROM clause in Figure 4.2 transforms to:

```
WHEN chan1.CONNECT_req
PROVIDED state=idle and A and (B or C)
TO connecting
BEGIN
  if D then S1
...
```

The TO clause is removed and replaced by an assignment to the major state variable added to the BEGIN block. As an example, the TO clause of Figure 4.2. is removed as follows:

```
WHEN chan1.CONNECT_req
PROVIDED state=idle and A and (B or C)
BEGIN
  state:=connecting;
...
```

State expressions in the FROM clause such as

FROM state <> closed generates one or more normal form transitions corresponding to one transition type for every possible present state value (state values other than "closed" in the above example).

4.2.2. BEGIN block

To remove conditional statements and local procedure calls we adapt the techniques from symbolic execution of sequential programs [ClRi 81]. The idea is to create a new transition for every distinct path in the BEGIN block and to modify the PROVIDED clauses to reflect the conditions imposed from taking these paths.

Local procedure calls in the BEGIN block are translated by symbolically executing the local procedure body if the local procedure body is completely specified. No transformation is done on incompletely specified local procedures. We assume that the specified procedures are not recursive and do not contain any loops with variable bounds.

As an example, we continue with transforming Figure 4.2 to remove the **IF** statement in the first transition type:

```
WHEN chan1.CONNECT_req
PROVIDED state=idle and A and (B or C) and D
BEGIN

    state:=connecting;
    S1;
...

WHEN chan1.CONNECT_req
PROVIDED state=idle and A and (B or C) and ¬D
BEGIN

    state:=connecting;
    S2;
...
```

The **WITH** clause in the BEGIN block (as in Pascal) is eliminated by concatenating the record name with the proper variable names. The **CASE** statement is removed in a manner similar to the IF statement above, by generating a normal form transition for each case.

The **FOR** and **WHILE** loop statements are eliminated by repeating the body of the loop for every index variable value. If the loop index variable refers to the connection arrays (see Section 4.2.3 below) the loop is eliminated by including the index variable in the list of variables of the ANY clause. In this case no modification is done to the loop body except possibly for the modifications introduced to remove the WITH statement.

4.2.3.ANY Clause and WITH Statement

As discussed in Chapter 1, in a transition type, an ANY clause of Subgroup B FDT either introduces local variables or index variables to the connection arrays (the array name may appear in the WITH statement which follows the ANY clause).

Local variables introduced by the ANY clause are made global in Phase 1 and removed from the ANY clause (if the resulting ANY clause is empty, the ANY clause can therefore be eliminated). In some cases, the local variable may be of enumeration type and removing it requires that the transition type generates one normal form transition for each element in the enumeration type. The WITH statement associated with the ANY clause is removed in a manner similar to the WITH statement of the BEGIN block.

As far as the index variables to the connection arrays are concerned, if only a single connection is considered, the ANY clause can be completely removed. But this is not desirable since removing ANYs also removes the transition types in which two index variables of the same type are used (if any). Thus we consider the number of connections as a test system parameter (see Section 6.2.1) and do not remove the ANYs which contain index variables for connection arrays.

4.3.Combining Modules (Phase II)

The modules within a specification of a protocol may communicate with other modules in the specification generating an **internal** interaction in an output statement of their BEGIN block. It is assumed that the referred modules have transition types corresponding to these interactions. Intermodule communication may involve parameter passing.

Since we are interested in test case generation and since only the events at the external interaction points can be observed, the interactions referring to intermodule communication should be removed. Also, we assume "rendezvous" type communication between the modules of the protocol entity.

If the transition type corresponding to an internal input does not contain any internal output, the transition type can simply be removed by substituting its BEGIN block in all the statements that "call" the interaction using symbolic replacements for parameter values (if any). Also, the PROVIDED clause of the transition type is added to the PROVIDED clauses of the normal form transitions that contain intermodule "calls" as a conjunction, possibly with suitable symbolic replacements obtained through symbolic execution.

In general, a normal form transition corresponding to an internal input can also contain internal output statements to other modules. Thus, a given intermodule

communication can be modelled as a tree (called **intermodule communication tree** or ICT in short) with nodes being normal form transitions involved and arcs being the events referred in each transition and the root being the normal form transition initiating the communication. We assume that in none of the paths from the root to the leaves of the ICT an event is referred to more than once, thus a recursive algorithm can be found to remove the intermodule communication. A two step procedure will be given to combine modules:

Step 1 forms list structures of all the normal form transitions of a given event, i.e., it forms for every module the lists of normal form transitions corresponding to each input event.

Step 2 processes BEGIN blocks of the normal form transitions. It removes the intermodule output statements with textual substitutions as described above. Any intermodule output statement encountered during the processing of the BEGIN blocks is removed in the same manner, thus performing an inorder traversal of the ICT. If there exists more than one normal form transition corresponding to a given event, the list formed in Step 1 is used to generate different normal form transitions, one for each normal form transition in the list, until all the lists of communicating modules are exhausted.

The algorithm of Step 2 is outlined in Algorithm 4.1,

listed in Appendix D. The algorithm uses the following marking scheme to handle the case where more than one normal form transition exists for a given event. A node is marked when it is completely processed. In particular, a node is marked when its normal form transition(s) is processed and either it contains no output event, or all nodes corresponding to the output event of the normal form transition are processed. The marking is achieved by using the stack called event_headers.

In Step 2 we assume that in a given transition at most one intermodule output statement can occur. The algorithm works also for more than one intermodule output statement if there is only one normal form transition for each event. This restriction can be removed by using more complicated data structures.

As an example, Algorithm 4.1. is applied to combine the two modules of Figure 4.1. The Process_event routine is called for the two normal form transitions of module1 since they all contain intermodule output statements. The first elementary expression in the PROVIDED clause of the normal form transition of module2 is removed by the routine Process_event since it can be satisfied by a parameter replacement. The second relation is added to the PROVIDED clauses as a conjunction after parameter replacements. A complete listing of the normal form transitions corresponding to Figure 4.2. is given in Figure 4.3.

```
Normal Form Transitions Block

    WHEN chan1.CONNECT_req
    PROVIDED state=idle and A and (B or C)
                and D and (X<=8)
    BEGIN

        state := connecting
        S1;
        S3;
        chan2.CONNECT_ind

    END

    WHEN chan1.CONNECT_req
    PROVIDED state=idle and A and (B or C)
                and ¬D and (X<=8)
    BEGIN

        state := connecting;
        S2;
        S3;
        chan2.CONNECT_ind

    END

end Normal Form Transition Block;
```

Figure 4.3.Normal form Transitions of Figure 4.2.

4.4.Spontaneous Transitions

For testing purposes, spontaneous transitions have similar problems as intermodule output statements since they cannot be controlled by external input primitives, but they can be observed if they contain an output statement.

4.4.1.Nondeterminism in Protocol Specifications

Nondeterminism is expressed using WHEN and PROVIDED clauses of a protocol specification in Subgroup B FDT. In a given major state of the machine, for a given interaction, there may be more than one possible WHEN transition, i.e., PROVIDED clauses of the same interaction may not be mutually exclusive. In addition, spontaneous transitions are an important tool to specify nondeterminism since a spontaneous transition may be executed any time when its PROVIDED clause is satisfied. Thus the PROVIDED clause of a spontaneous transition may not be mutually exclusive with the PROVIDED clauses of other transitions (WHEN or spontaneous). [JaBo 83] contains a detailed discussion on nondeterminism in protocol specifications.

4.4.2.Removing Spontaneous Transitions

We have considered the option of removing spontaneous transitions by combining them with WHEN transitions.This option has been rejected for the following reasons:

1) Removing spontaneous transitions removes most of the nondeterminism in a protocol specification by making spontaneous transitions eligible only after external interactions are received.

2) Combining spontaneous transitions with WHEN transitions makes it impossible to express the repeated execution of a

spontaneous transition since loops are not allowed in normal form transitions.

4.4.3.Nondeterminism and Protocol Testing

In a given test in order to evaluate the responses of an implementation correctly, it is necessary to apply deterministic inputs. Nondeterminism in WHEN transitions can be easily removed by applying to the implementation unique inputs in a major state. But this solution is no longer valid for spontaneous transitions. Thus tests which involve spontaneous transitions should be designed adaptively to receive any output from spontaneous transitions and respond accordingly.

4.5.Conservation of the Semantics

It should be clear that the transformations of this section preserve the semantics of the original transition types.

Phase 1 transformations remove multiple paths in a given transition type by generating a normal form transition for each path. We have found the assumption of no loops in a BEGIN block with variable indexes to be satisfied for the protocol specifications we considered. Thus Phase 1 transformations preserve the semantics in the specification.

Incompleteness in the specification is preserved in terms of local procedure names corresponding to unspecified local procedures.

Phase 2 transformations combine modules by symbolic replacements of output statements with their transition types. Since loops are removed in Phase 1, Phase 2 assumes single path normal form transitions. Similar to Phase 1 above we have found the assumption of no (indirect) recursive intermodule "calls" to be satisfied for the protocol specifications we considered. Thus, Phase 2 transformations conserve the semantics of the specification by obtaining a product of the extended finite state machines of protocol modules.

Normal form transitions obtained from a protocol specification can be seen to completely represent the protocol specification hence in the following chapters of Part 2 we use only normal form transitions for further discussion.

4.6.A Real Example

Two transition types of a Subgroup B FDT specification of the Class 2 TP [ISO 82b] are given in Appendix A. The first transition type is a spontaneous transition for the decoding function of the transport protocol. Only the part related with a particular input primitive (CR) is

considered, the rest of the transition type is ignored. The second transition type has an input primitive from the interaction point called TS. Both transitions contain various intermodule output statements such as:
ATP.error_indication, ATP.forward and Map.forward.
The transition types corresponding to the ATP.forward and Map.forward are also given in Appendix A. Also, the first transition type refers to various local procedures such as: determine_PDU_length, implied_PDU_length, etc. The completely specified local procedures are listed in Appendix A.

The normal form transitions obtained from applying the two phases of transformations are given in Appendix B. They are numbered for identification purposes. More discussion on the Class 2 TP normal form transitions follows in Chapter 8.

5.GRAPH REPRESENTATIONS OF NORMAL FORM TRANSITIONS

In this chapter we identify two types of flow in a protocol specification:

- flow of control as the value(s) of the major state variable(s) (one for each module of the protocol entity) changes, and
- flow of data as the input primitive parameters change the values of the protocol variables and they in turn determine the output primitive parameters.

We model these flows as graphs and call them Control Graph (CG) and Data Flow Graph (DFG), respectively. After introducing these graphs, we will discuss decomposition of each graph, i.e., transition tours for CG and blocks for DFG. For the partitioning of the DFG into blocks, parameters of interaction primitives (input and output) and protocol variables are considered as distinct nodes, and the flow over each variable determines the most refined partition. Such a partition is found algorithmically for any given DFG. A heuristic procedure which requires user interaction is given to obtain less refined partitions of the DFG by combining the blocks.

The CG and DFG of a protocol specification are helpful in validating the design of the protocol as will be discussed last in this chapter.

Due to its relative simplicity, the Class 0 Transport

Protocol (TP) is used as an example in discussing these two graphs. A specification of Class 0 TP can be found in [ISO 82]. Normal form transitions obtained by applying the procedure in Chapter 4 to Class 0 TP specification are given in Appendix C. The normal form transitions in Appendix C are identified as P1,P2,...,P19 to be referred later in the graphs.

In the discussion on protocol design validation, Class 2 TP specification [ISO 82b] is used as an example. The normal form transitions of the Class 2 TP are documented in [Sarikaya 84].

5.1. Control Graph

The CG (also known as Finite State Machine model of the protocol) can be constructed from the normal form transitions as follows:

The nodes represent the values (tuples) the major state variable(s) can take. Each normal form transition is represented as an arc in this graph. The arc is drawn from the state in the provided clause to the next state specified in the begin block, and labelled with the identifier of the normal form transition. A CG for the Class 0 TP is given in Figure 5.1.

The labels in the CG are short-hand notations for

input / output

as used in the FSMs of [SaBo 84]. In here, "input" stands for the WHEN clause (if any) and the PROVIDED clause (with the expression on major state variable(s) removed) and "output" stands for any output statements to the external entities in the BEGIN block of the normal form transition.

For some protocols, a normal form transition is represented by more than one arc in the CG. This happens when the present and next state values of the normal form transition are specified only for one of the module state variable, making the normal form transition eligible for all values of other module's state variables.

The control graph models the part of the normal form transitions that are related to the state changes of a given connection for each module of the protocol entity. Normal form transitions referring to more than one connection cannot be represented in the CG. This happens when normal form transitions contain ANY clauses with more than one index variable of the same type, as discussed in Chapter 4. Therefore the CG ignores ANY clauses in the normal form transitions.

5.1.1. Subtours of the CG

Taking the identifiers of the transitions in the CG as inputs to the FSM, it is possible to generate transition

tours for any control graph since the transition tour method does not require any special property of the machine as discussed in Chapter 2. In what follows we assume that the initial state of the CG is also the final state. This assumption is realistic since, for a protocol, the initial state represents the idle state of a connection. Thus, a transition tour of the CG can be divided into smaller tours, each starting and ending in the initial state. These subdivisions of a transition tour will be called **subtours.**

It is possible to use a notation which has been used for regular expressions [Kohavi 78] to represent subtours. In this notation, a transition which is labelled by a list P1,P2,...Pn of identifiers is represented as

P1 + P2 + ... + Pn

instead of repeating the subtour for each of the choices above. Also, a transition labelled as P1,P2,...Pn which is a self-loop [Kohavi 78] is represented as:

$$(P1 + P2 + \ldots Pn)^{*}$$

Note that in a CG, there can be loops around more than one state not involving the initial state, these loops are represented similarly as the self-loops.

Subtours of Figure 5.1 are listed in Table 5.1. In Table 5.1, there are self-loops around one state but no loops.

5.1.2. Control Functions

In communication protocols, a sequence of normal form transitions occurring in the subtours represent distinct control phases, i.e., major state variable changes, such as connection establishment, data transfer, connection freeing, etc. Each subtour contains a number of control phases. More than one subtour may have the same pattern of control phases.

We define **a control function** of a protocol as a control phase of the protocol. A subtour may contain a sequence of control functions and the same sequence of control phases may occur in two different subtours. If all the normal form transitions are covered, this guarantees complete coverage of the control functions but not necessarily all the subtours. More discussion on test coverage follows in Section 6.1.3.

From Table 5.1, we identify the following control functions for the Class 0 TP, for each of the five subtours in top-to-bottom order:

- User-initiated connection establishment, data transfer, freeing
- Peer-initiated connection establishment, data transfer, freeing
- Call refusal by peer
- Call refusal by user
- Call refusal by protocol

5.1.3.Order of Transitions in a Self-loop

Normal form transitions which do not modify major state variable(s) are represented as self-loops in the CG. If there exists more than one normal form transition as a self-loop in a given state of the CG, the order of execution of these normal form transitions may become important. The "*" notation used to represent self-loops above suggests that the normal form transitions in the list may be executed any number of times in any order. This is not necessarily so since the order of execution of normal form transitions depends on:

a) the fact that the input primitive mentioned in the WHEN clause of a transition is received, and

b) the predicate, i.e., PROVIDED clause (which may contain expressions on protocol variables other than major state variable(s)) is satisfied for any spontaneous or WHEN transition.

Since a CG does not show the effects of protocol variables other than major state variable(s), the order of normal form transitions in a self-loop can not be determined from the CG. The data flow graph which will be discussed next should be consulted for this purpose.

P1(P6+P7) (P13+P14+P15+P16)* (P17+P18+P19)

(P3+P4)P10 (P13+P14+P15+P16)* (P17+P18+P19)

P1 (P8+P9)

(P3+P4) (P11+P12)

P2+P5

Table 5.1. Subtours of the Control Graph for Class 0 TP

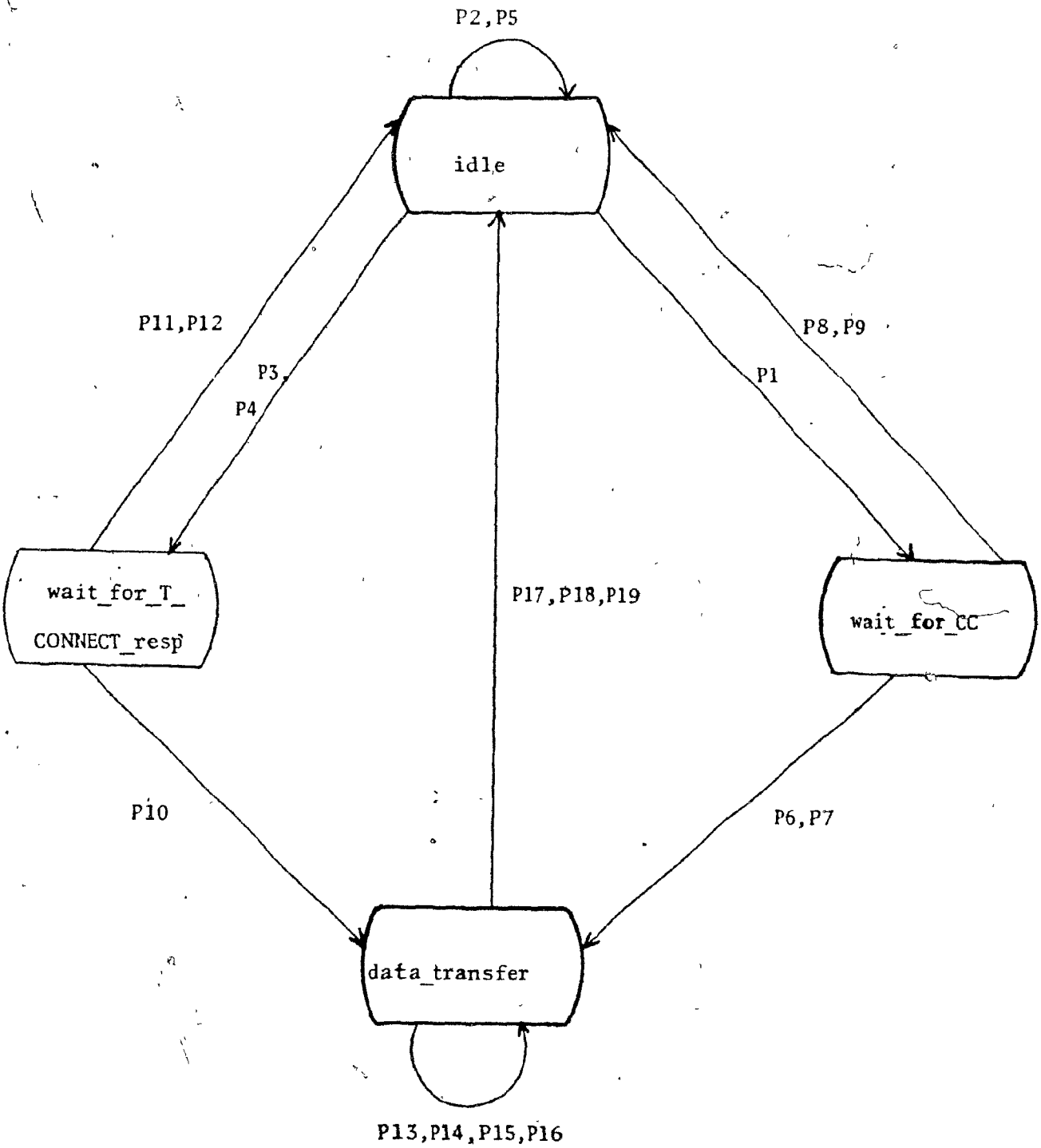
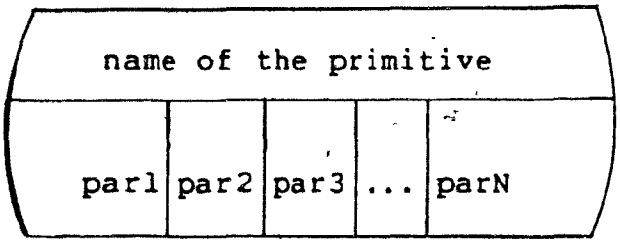


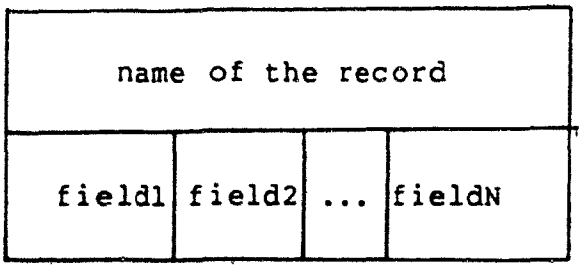
Figure 5.1. Control Graph of Class 0 TP

5.2. Data Flow Graph

A data flow graph models the flow of information in a protocol specification, excluding major state changes. A DFG contains four types of nodes: I-nodes to represent input primitives, D-nodes to represent protocol variables and constants, O-nodes to represent output primitives and F-nodes to represent certain functions on data. I-and O-nodes are shown in oval, D-nodes in rectangular and F-nodes in crossed-circular shapes to identify each of them. In each node, the name referred in the specification is written. The I- or O-nodes take the following form:



where par1,par2,...parN are the parameters of the primitive. To represent fields of records (i.e., Pascal records) in the I- or O-nodes or in D-nodes, the notation used is:



where fieldi (i=1,...,n) are the fields of the record.

In protocol specifications that contain connection

arrays, the D-nodes are prefixed with the array name which is indexed by the index variable in the ANY clause. If the ANY clause contains more than one index variable of the same type, data flow in these normal form transitions can easily be represented by replicating the D-nodes with different index variable names.

5.2.1. Formation of the Arcs

Arcs in the DFG are used to represent the flow as derived from the action (BEGIN block) of the normal form transitions. The effect of the predicates, i.e., the PROVIDED clause will be considered when we discuss dependencies in Chapter 6. Simple assignment statements in the BEGIN block are shown by arcs directed from the source nodes to the destination nodes. The output primitive parameter values directly carried out from input primitive parameter values are modelled as simple assignment statements.

Each arc in the DFG is labelled with the identifier of the normal form transition. These identifiers represent the **pre-condition** (WHEN and PROVIDED clauses) of the normal form transitions and the actions are modeled by the DFG (the state after execution of the action is called **post-condition**). Since a given BEGIN block can contain many statements there can exist more than one arc labelled with the same

identifier. Also, the same assignment used in more than one normal form transition can be represented as a single arc carrying more than one label.

If the BEGIN block of a normal form transition contains no assignment statements, an arc is created from the input primitive to the output primitive.

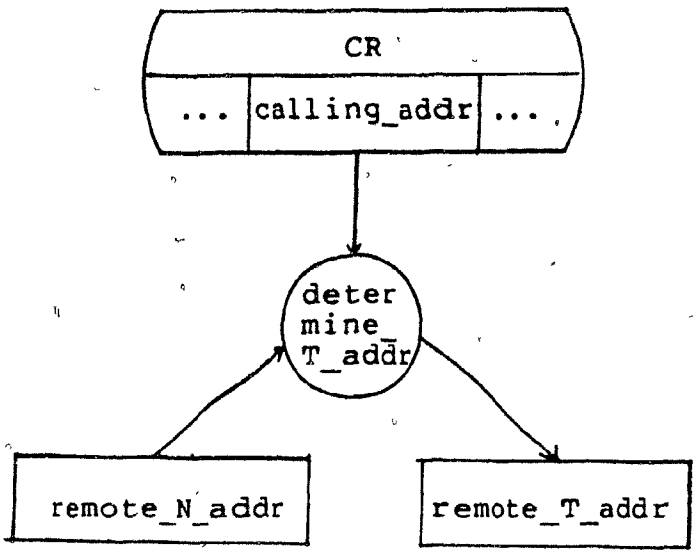
Three types of F-nodes are created:

Type 1) These nodes represent function calls which return a value and whose body is not specified, i.e., the value returned to be determined by the implementation. In the transformations phase, body replacement can not be done for these procedures, thus they are treated like assignment statements. In the F-node, parameters of the procedure or function call become the incoming arcs and an outgoing arc to the variable assigned is created. The F-node carries the name of the procedure. A DFG contains a F-node with the same name for each procedure call with different parameter list.

For example, an assignment statement like:

```
remote_T_addr:=determine_T_addr(remote_N_addr,
                                CR.calling_addr)
```

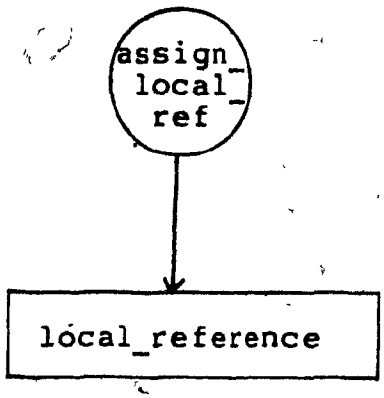
is represented as:



Type 2) These nodes represent assignment statements whose right hand side is not specified. A F-node which becomes a value source is introduced and an outgoing arc from this node to the destination variable mentioned in the statement is created. The F-node is labelled with "assign_name" where "name" is the destination variable name. For example, the assignment statement

```
local_reference := ...;
```

is represented in the DFG as :



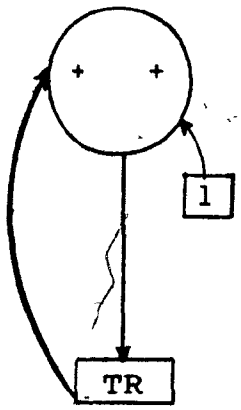
A F-node of the same type can be created for some local procedures whose input parameters are implicit in its name, e.g., a variable initialization procedure. As for F-nodes of Type 1, these nodes can be replicated for each variable initialized by the same procedure.

The F-nodes of Type 1 and 2 are abstracted representations of the incompleteness in the specification. It should be noted that the test designer should know the range of values these nodes can assign to the D-nodes, possibly by consulting the informal specification of the protocol.

Type 3) These nodes represent assignment statements containing an arithmetic or Boolean expression. A F-node is represented with incoming arcs from the operands of the expression and with an outgoing arc to the assigned variable. The F-node contains labels associated with the incoming arcs indicating the operators applied to the variables. For instance, the assignment statement:

TR := TR + 1

is represented as:



Similar to F-nodes of types 1 and 2, constant D-nodes (created for constants occuring in the actions of the normal form transitions) can be replicated when the same constant is assigned to more than one node.

When constructing a DFG, the assignment statements to the major state variable(s) are ignored. The labels on the arcs of the DFG can be used to refer to the subtours that include the normal form transitions. Also, the labels represent the pre-conditions, i.e., the PROVIDED clauses of the normal form transitions. A DFG for the Class 0 TP is given in Figure 5.2.

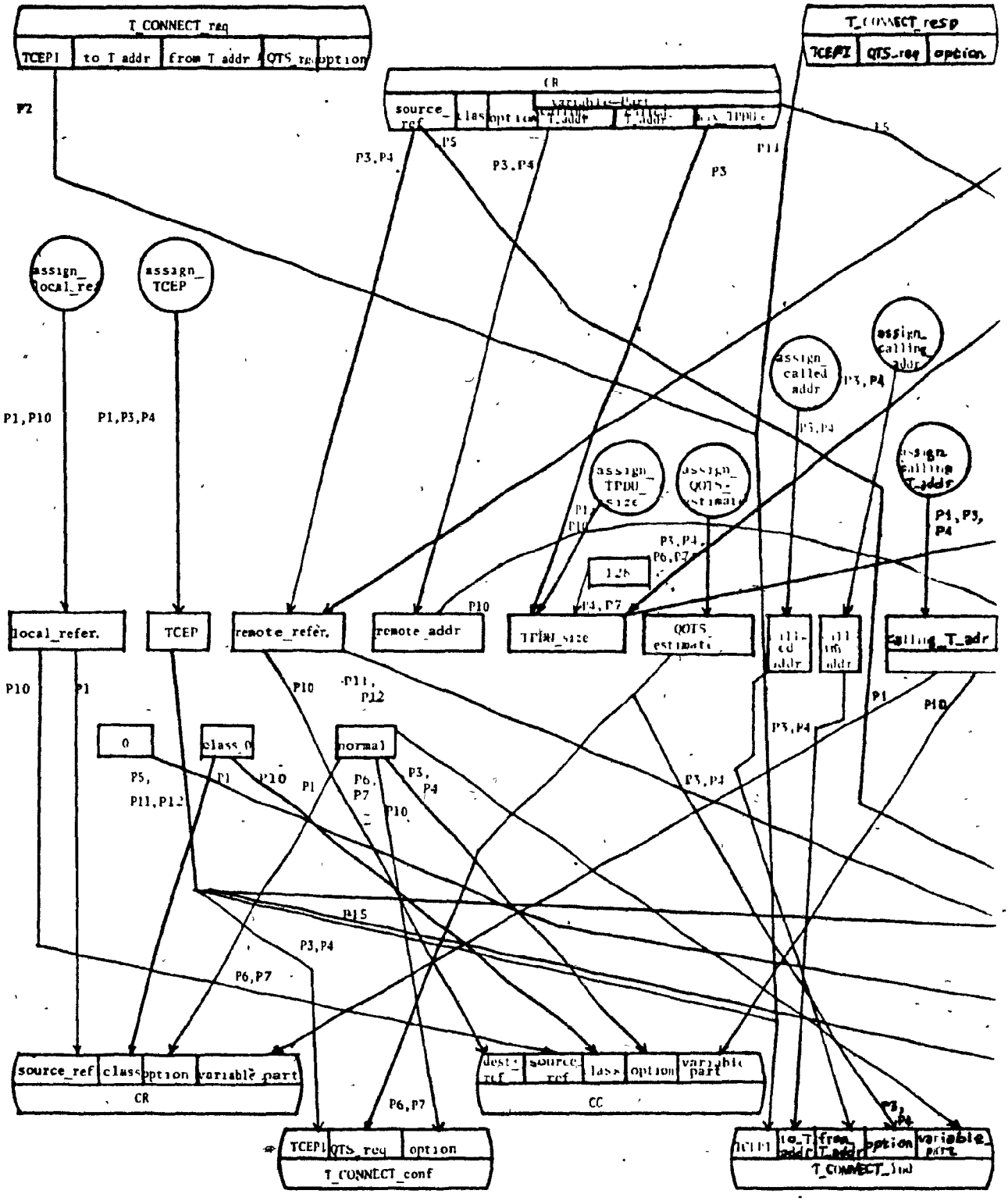


Figure 5.2.A DFG of the Class 0 TP

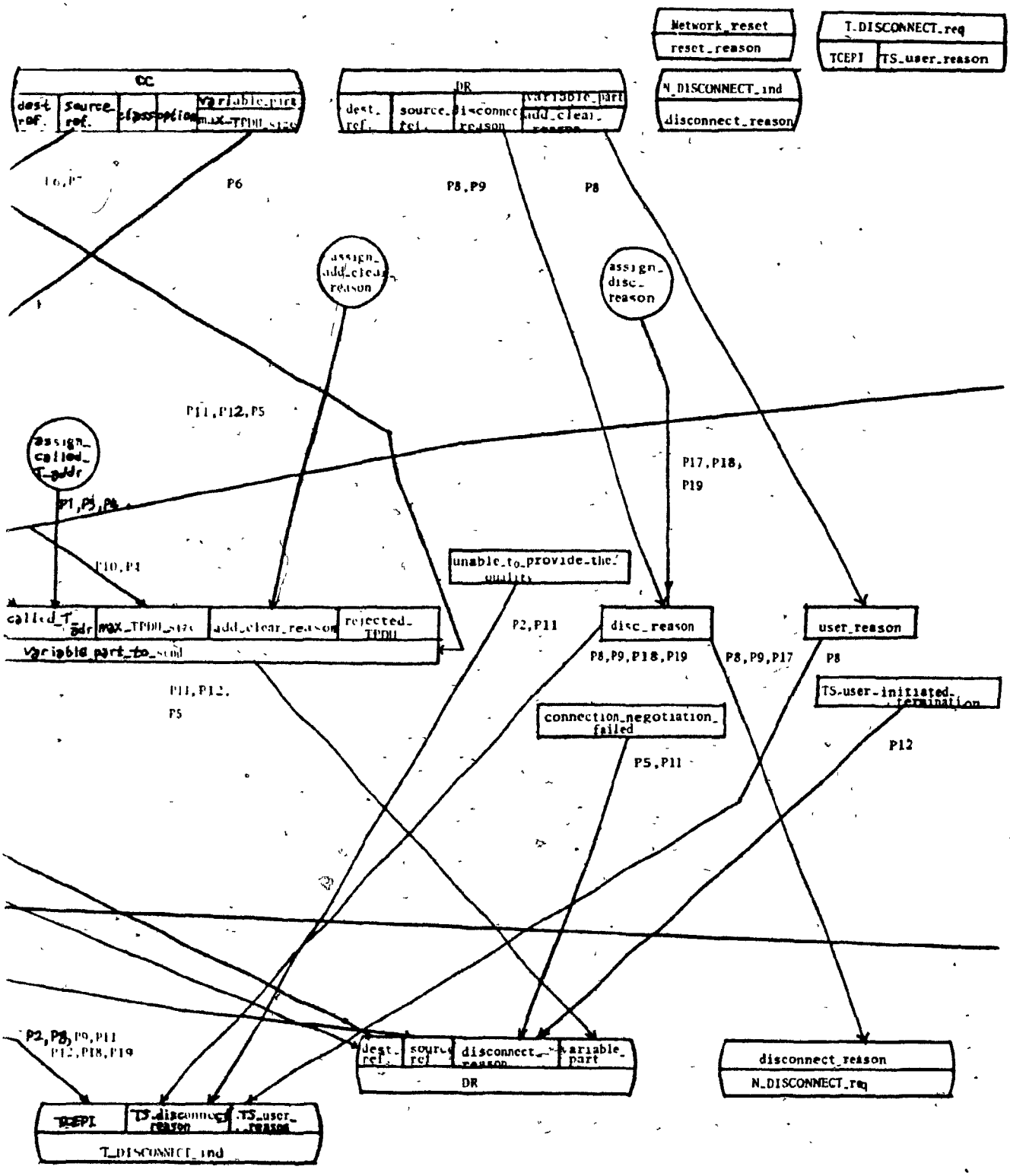


Figure 5.2. ... (continued) ...

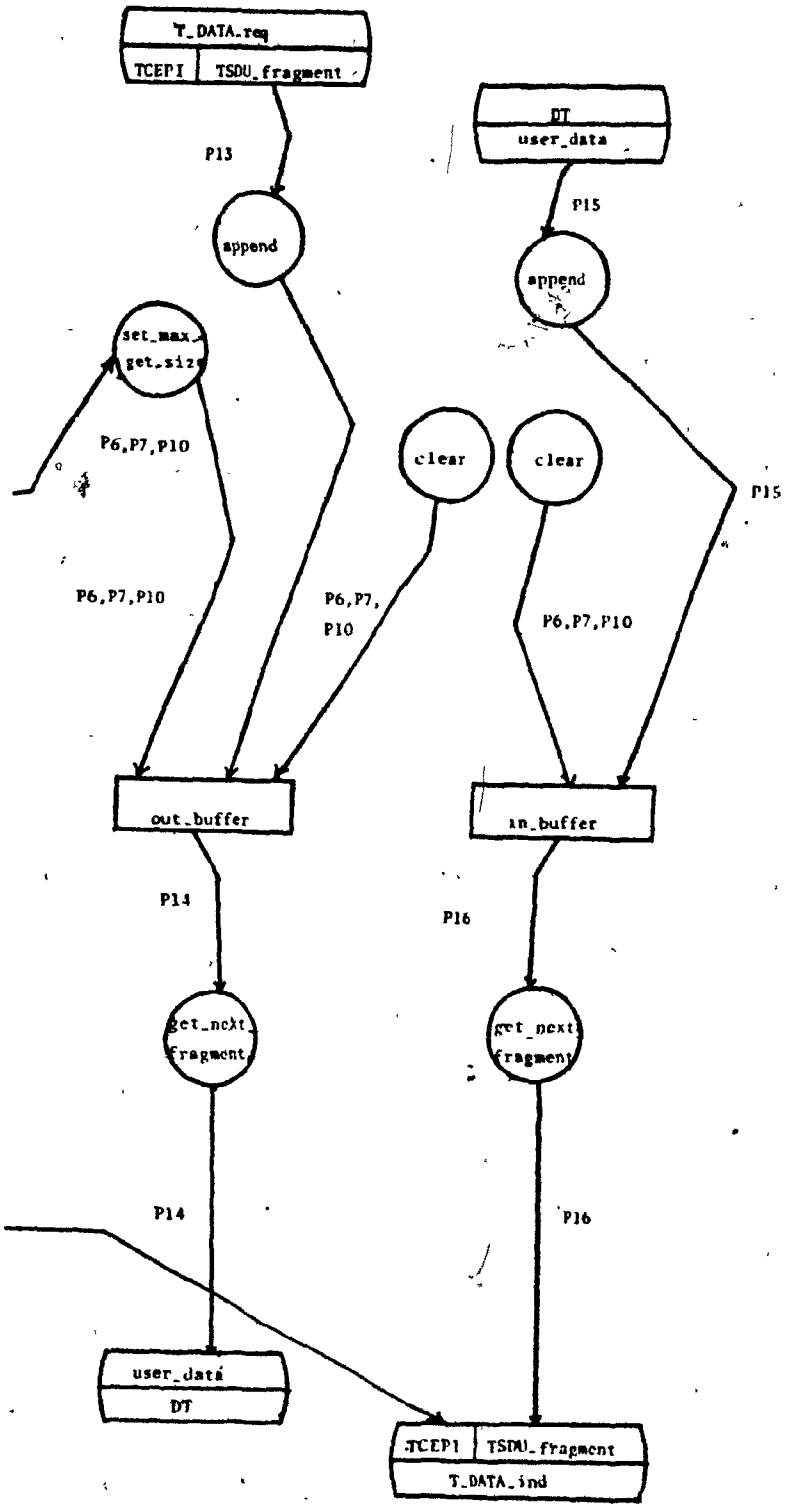


Figure 5.2. ... (continued) ...

5.3.Partitioning the Data Flow Graph

In this section, we consider each parameter (or field of the parameter) of the input and output primitives as separate I- and O-nodes respectively. The name of the node is obtained by prefixing the parameter name with the primitive name. Similarly, fields of protocol variables of type record are considered as separate D-nodes.

Splitting of I-, O-, and D-nodes into their component fields facilitates partitioning of the complex data flow in the DFG. In the remaining part of this section we introduce the concept of a **block** of the DFG (the blocks of a DFG represent different functions), and then give an algorithm to find disjoint blocks for a given DFG.

5.3.1.Blocks of the DFG

A **block** is a collection of nodes of types I-, F-, D- and O. Thus a block Bi of a DFG can be defined by the following four sets: the set of I-nodes, **SIN**, the set of F-nodes, **SFN**, the set of D-nodes, **SDN**, and the set of O-nodes, **SON** that it contains. We partition the DFG into disjoint blocks B1, B2, ..., Bn. Note that SFN and SDN of some blocks may contain F- and (constant) D-nodes carrying the same names because of the replication of some F- and D-nodes during the construction of the DFG, as explained in Section 5.2.1. We assume that these F- and D- nodes are

distinguished as different nodes.

All the incoming arcs to D-, F- and O-nodes and all the outgoing arcs from I-nodes belong to the block which contains the node. Some of the outgoing arcs from D- or F-nodes may be shared among blocks. The nodes associated with the incoming arcs to a D-node belong to the block containing the D-node. If the value of the D-node flows to the O-nodes, they are included in the block. If the value is needed by an F-node of Type 1 or 3 and the F-node has incoming arcs from other D-nodes, a separate block is created for the F-node. Different blocks are created for each O-node assigned directly by I-nodes or by F-nodes which in turn have incoming arcs from the I-nodes. Using these ideas, an algorithm to partition a DFG into (relatively small) disjoint blocks is given in Algorithm 5.1 listed in Appendix E. The algorithm uses the following sets in finding the blocks:

Definition 1. For each node in the DFG we define:
Set of Input I-Nodes (**SIIN** ~~in~~ short) is the set of I-nodes having outgoing arcs to the node under consideration.
Set of Input F_Nodes (**SIFN**) is the set of F-nodes having outgoing arcs to the node.
Set of Input D-Nodes (**SIDN**) is the set of D-nodes having outgoing arcs to the node.

Note that for I-nodes and constant D-nodes these sets are empty. Similarly, **SODN, SOFN** and **SOON** are defined to be the sets of output D-,F-, and O-nodes that have incoming arcs originating from the node under consideration. For O-nodes these sets are empty. The above sets of nodes define the

structure of the DFG, but not the labels of the arcs.

Definition 2. Set of Input Labels (**SIL**) is the set of labels i.e., identifiers of normal form transitions of the incoming arcs to the node in consideration,
Set of Output Labels (**SOL**) is the set of labels of the outgoing arcs from the node.

The above definitions can be extended to apply to more than one node:

SIIN(D1,D2,...,Dn) = SIIN(D1) u SIIN(D2) u...u SIIN(Dn)

Similarly for SIFN(D1,D2,...,Dn), SIDN(D1,D2,...,Dn), etc..

Algorithm 5.1 processes all variable D-nodes in the DFG by first creating a block for each unprocessed D-node. For any D-node in the SDN of the block, the algorithm includes all the nodes in SIIN, SIFN, SIDN and SOON into its sets SIN, SFN, SDN, and SON, respectively. For any O-node in the SON of the block, the algorithm includes all the nodes in SIIN, SIFN and SIDN into its sets SIN, SFN and SDN, respectively. For any F-node in the SFN of the block, the algorithm includes the constant D-nodes in SIDN and all nodes in SIIN into its sets SDN and SIN, respectively. The above procedure is repeated for the newly added D-nodes to the block.

Since there can be many levels of F-nodes before a D-node is assigned to an O-node, and since F-nodes can have more than one D-node in their SIDN, the nodes in SOFN of each D-node in the block receive special attention in Algorithm 5.1. If SIDN of each F-node in SOFN includes only variable D-nodes of the block, the F-node is added to SFN,

and SOON of the F-node is added to SON of the block. A similar process is applied to the F-nodes in the SFN of the block whose SIDN is empty and the SIFN contains the nodes that are already included in the SFN of the block.

Finally Algorithm 5.1 continues to form blocks containing no D-nodes. These blocks may contain only I- and O- nodes when input primitive parameters are directly assigned to output primitive parameters. The blocks containing only F- and O- nodes and (possibly) constant D-nodes are created by constant assignments to the O-nodes or through F-nodes which are not included in any of the blocks created above.

As an example, we apply this algorithm to the data flow graph in Figure 5.3 which represents a part of the DFG for Class 2 TP [ISO 82b] which will be discussed in Chapter 8. Algorithm 5.1 generates three blocks (B1, B2, B3) for the DFG in Figure 5.3, two for the two D-nodes and one for the F-node of "determine_add_addr". These blocks are described by the following sets:

```
SIN(B1)={N_CONNECT_ind.from_N_addr}
SDN(B1)={remote_N_addr}
SFN(B1)={assign_remote_N_addr}
SON(B1)={N_CONNECT_req.to_N_addr}
```

SIN(B2)={CR.calling_addr,T_CONNECT_req.to_T_addr}
SDN(B2)={remote_T_addr}
SFN(B2)={determine_T_addr}
SON(B2)=Ø

SIN(B3)=Ø
SDN(B3)=Ø
SFN(B3)={determine_add_addr}
SON(B3)={CR.called_addr, CC.called_addr}

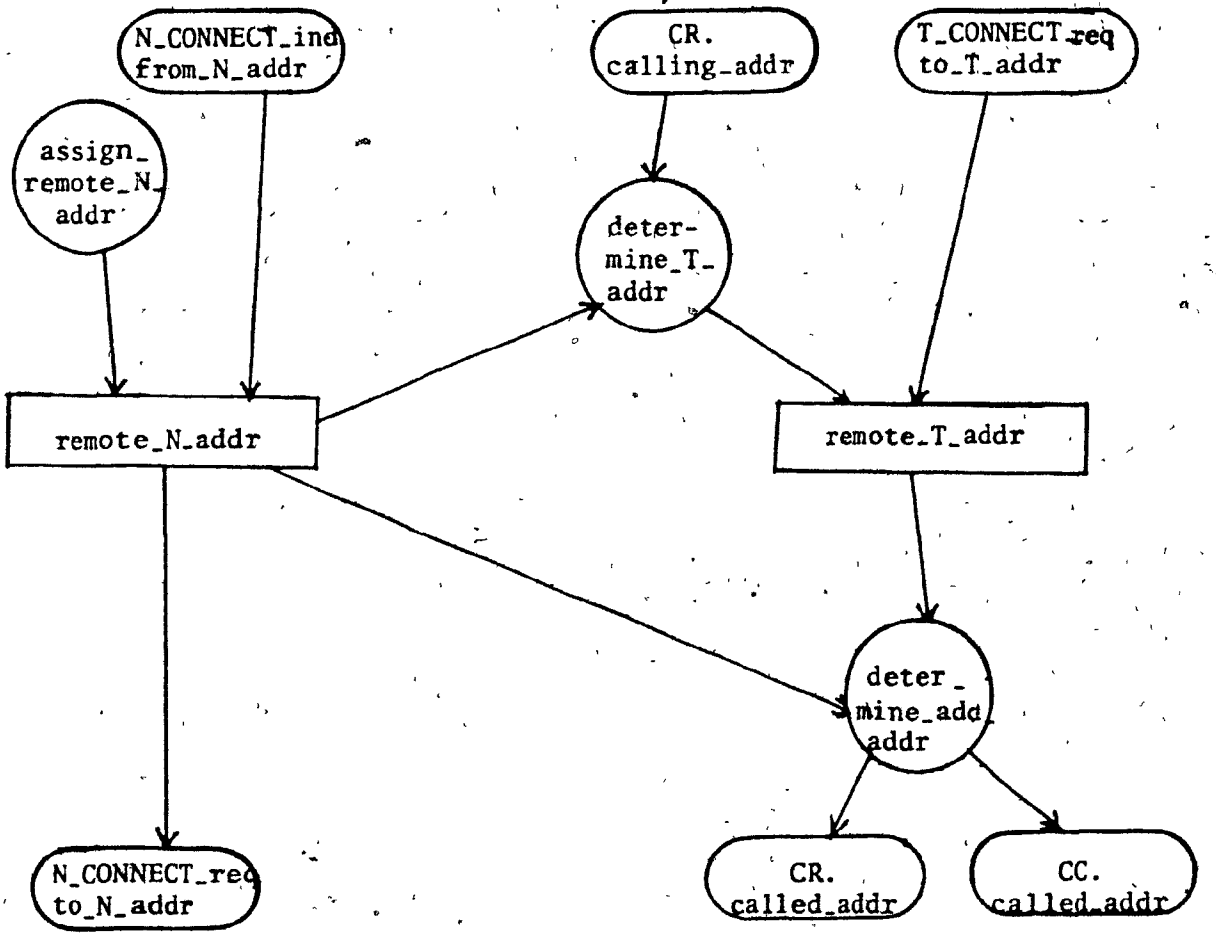


Figure 5.3.A Part of the DFG for Class 2 TP

Applying the algorithm to the DFG of the entire Class 0 TP given in Figure 5.2 produces the blocks shown in dotted lines in Figure 5.4.

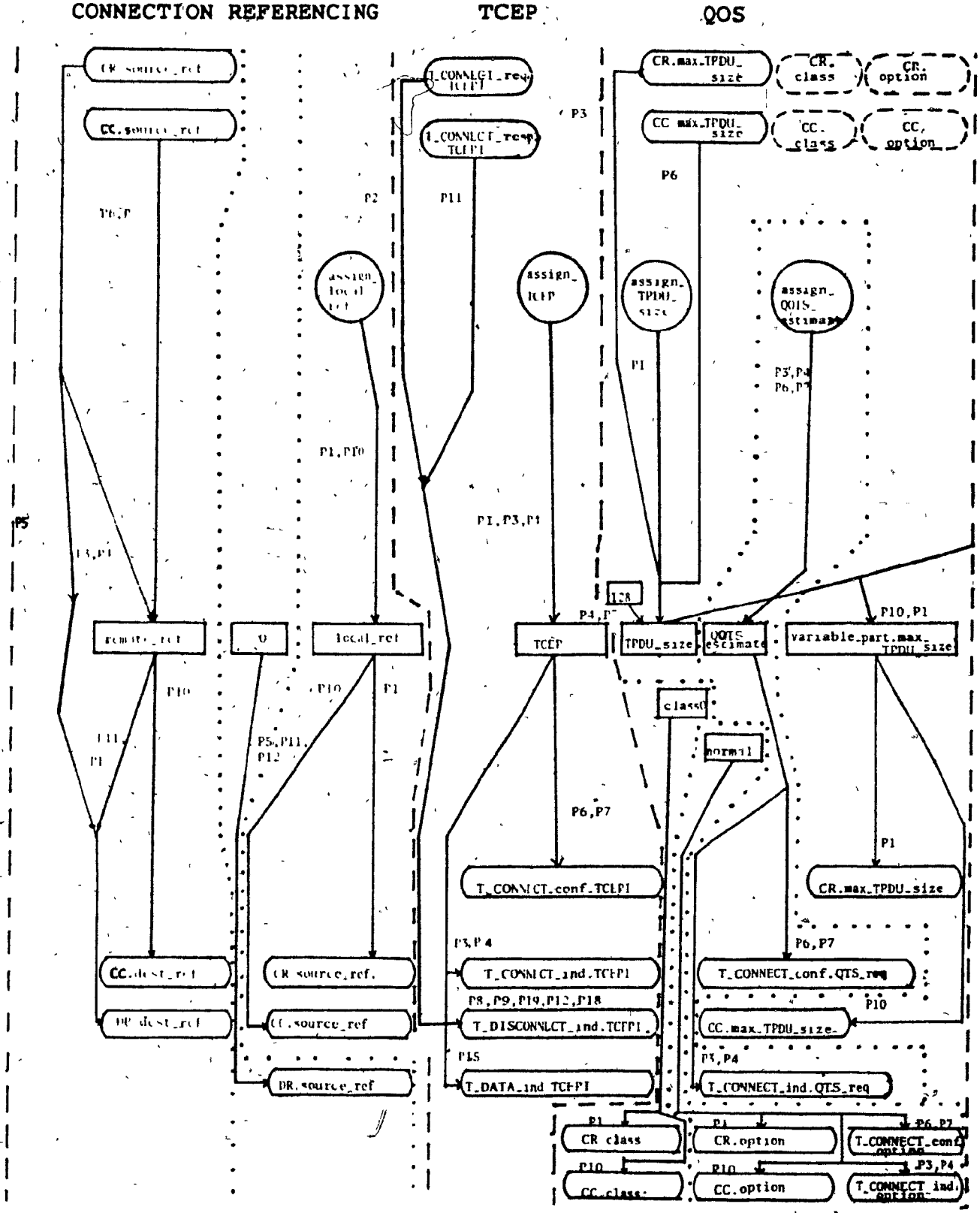


Figure 5.4.Partitions of the DFG for Class 0 TP

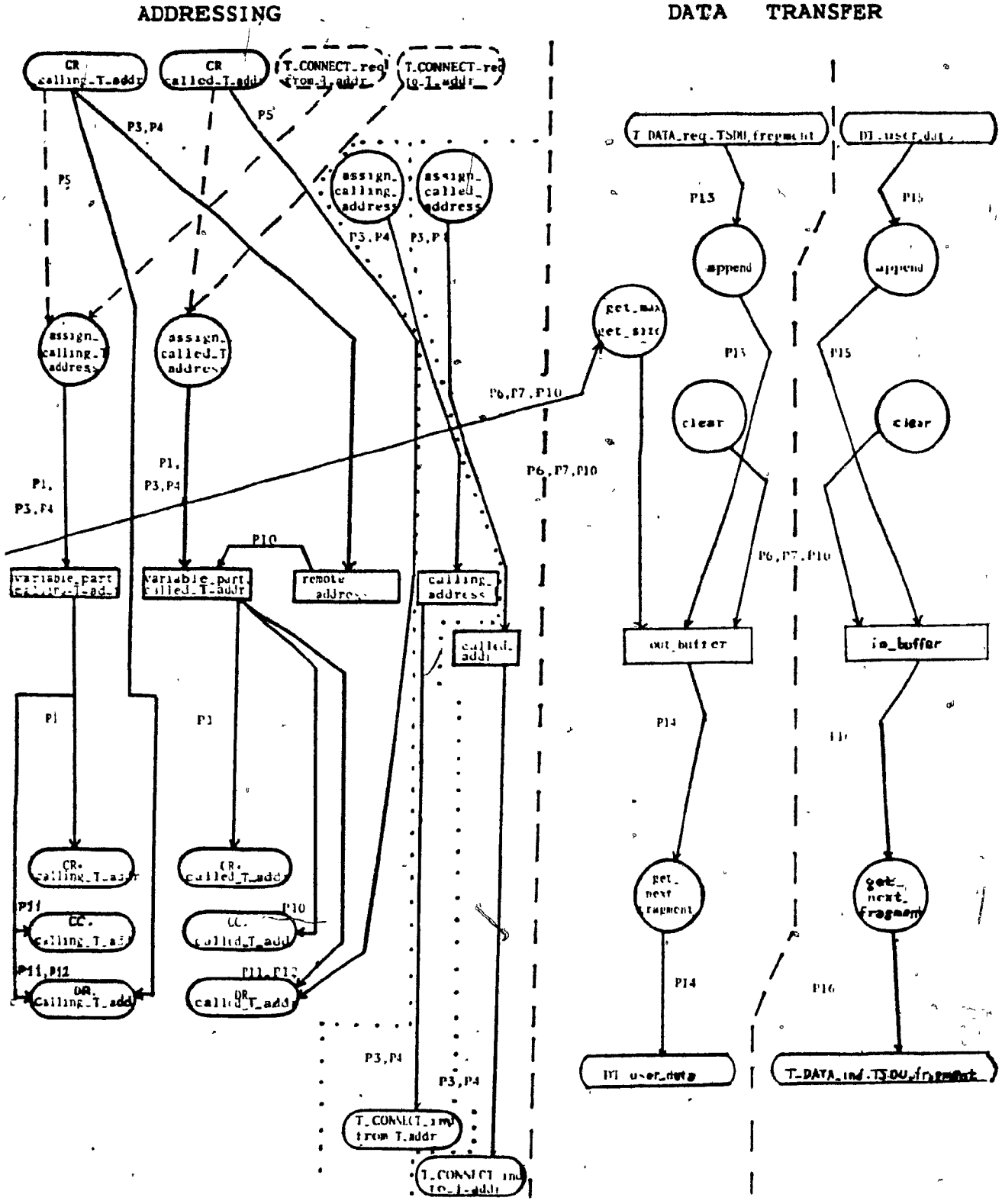


Figure 5.4. ...(continued)...

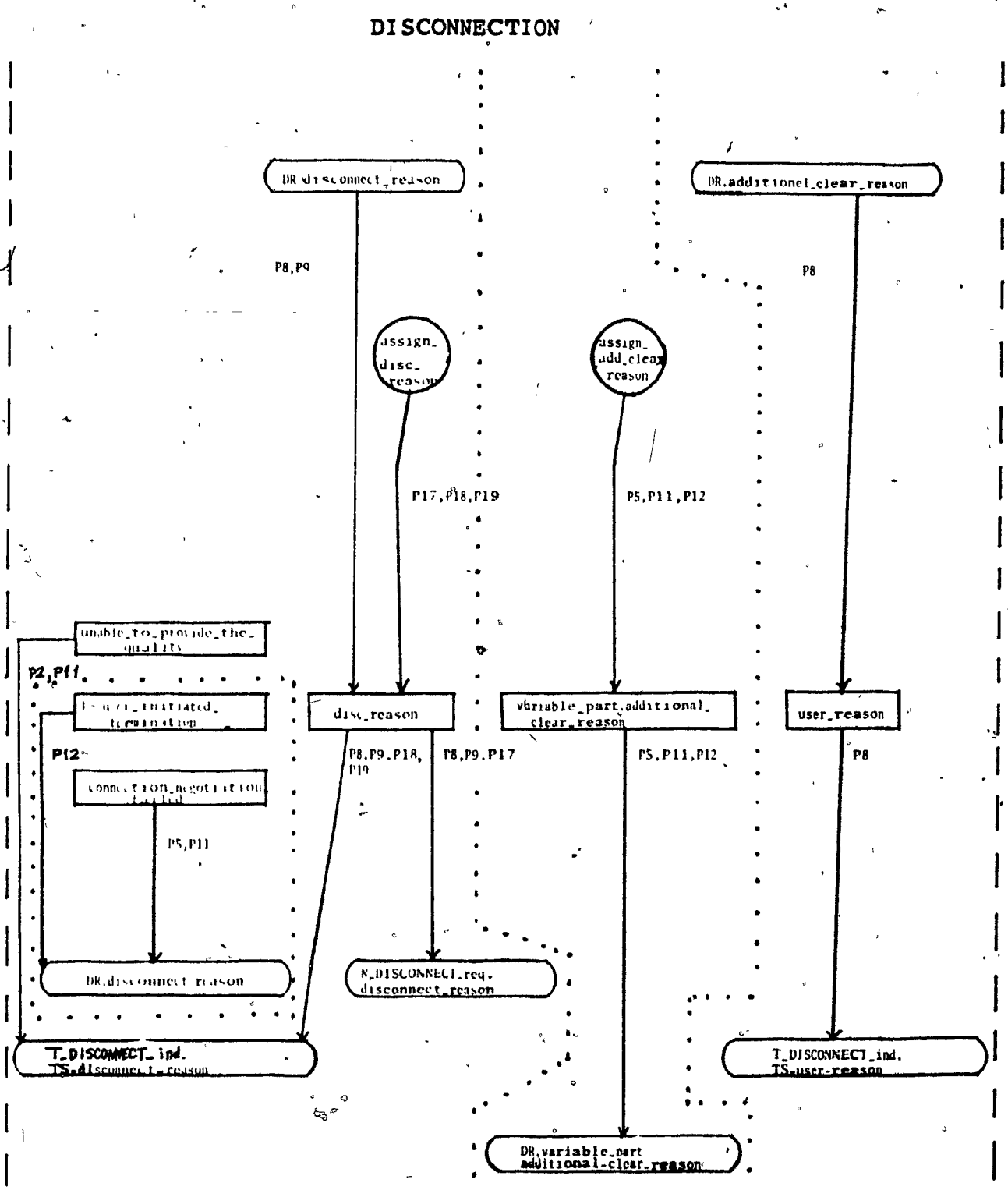


Figure 5.4. ...(continued)...

5.4. Functional Partitioning of the DFG

The partition obtained by Algorithm 5.1 is purely based on the structure of the DFG, i.e., interconnection of the various nodes determines the resulting blocks. The level of refinement of the blocks obtained from Algorithm 5.1 is not appropriate for testing purposes since a very high number of blocks can be obtained. A less refined partition of the DFG can be obtained by combining some of the blocks. We give in the following a heuristic procedure for such a combination. This procedure considers the semantics of I-D- and O- nodes and SIL (Definition 2) of the blocks. In some cases, the user can specify the blocks to be combined.

It should be noted that the following procedure is based on our experience with OSI protocols of layers up to 4; for higher layers or other types of protocols the procedure may need some modifications.

Block Merging Procedure

The procedure contains 6 steps. We start with the blocks obtained from Algorithm 5.1. Each round of application of the 6 steps results in a less refined partition of the DFG. The process is stopped when a partition is obtained such that no blocks can be combined in any of the steps of the procedure. Each step below will be followed by an explaination of its application to the DFG for the Class 0 TP (if any).

Step 1. Two blocks Bi and Bj are combined if SON(Bi) and SON(Bj) contain the parameter(s) of the same type.

This step combines the blocks in which corresponding parameters of different output primitives are assigned.

As an example, in Figure 5.4 the block containing "source_ref" of DR is combined with the block containing "local_ref" since the latter block contains "source_ref" parameters of CR and CC and all three parameters are of the same type. Step 1 also combines the block containing "disc_reason" with the block containing "disconnect_reason" of DR in order to combine all the blocks containing O-nodes of disconnect_reason type. Also the block of "called_address" is combined with the block of "calling_addr" since their O-nodes are of the same type.

Step 2. Independent blocks (blocks with no incoming arcs from other blocks) Bi and Bj are combined if the types of all the nodes in SIN(Bi) are the same as some of the nodes in SON(Bj).

This step combines two independent blocks if one block contains all I-nodes of the other block in its O-nodes provided that they are of the same type. The net effect of Step 2 together with Step 1 is to combine the blocks having a given type of parameter in various primitives as their I- and O-nodes.

When applied to Figure 5.4, Step 2 combines the block

of "local_ref" (Bi) with the block of "remote_ref" (Bj) since the O-nodes of Bi are of the same type as the I-nodes of Bj. Similarly, the block of "additional_clear_reason" is combined with the block of "user_reason". The blocks of "out_buffer" and "in_buffer" are not combined since the former is not an independent block.

Step 3. Let Bi and Bj be independent blocks. Bi and Bj are combined if SON(Bi) and SON(Bj) contain different but **related** parameters of the same primitive and

. SIL(Bi) $\supseteq$ SIL(Bj) holds.

Which parameters of a primitive are related is determined by the test designer.

Step 3 may be used to combine the blocks that assign similar parameters (or parameters of the same nature) of a given primitive in the same normal form transitions.

It is usually straightforward to identify related parameters of a primitive. In particular in Class 0 TP, we identify, "additional_clear_reason" and "disconnect_reason" parameters of DR. The blocks containing the above O-nodes are combined since SIL of the block containing "additional_clear_reason" is a subset of the SIL of the block containing "disconnect_reason".

Step 4. The blocks that contain only O- and F- nodes with F-nodes having incoming arcs from D-nodes of different blocks are combined with one of the blocks that contain the

D-nodes. The choice of the block depends on the relationship of the O- node and the D-node which is determined by the test designer.

Step 4 can be used to combine the blocks of some of the data transfer primitive parameters (O-nodes) with the blocks of the input and output buffers (D-nodes), since the buffers contain the data that flows to the O-nodes.

Step 4 does not apply to Figure 5.4 since the F-node "get_next_fragment" which assigns the O-node "DT.user_data" has only one incoming arc from the D-node "out_buffer", hence the F- and O-nodes are already included in the block of "out_buffer" by Algorithm 5.1. Similarly for "get_next_fragment", "T_DATA_ind.TSDU_fragment" and "in_buffer".

For the Class 2 TP, Step 4 can be used to combine blocks of "DT.user_data" and "DT.end_of_TSDU" with the block of "send_buffer". Similarly, the blocks of "T_DATA_ind.TS_user_data" and "T_DATA_ind.is_last_fragment" are combined with the block of "receive_buffer" (see Chapter 8).

Step 5. The blocks Bi and Bj are combined if SDN(Bi) and SDN(Bj) contain related D-nodes (variable or constant) that are used to specify different features of a relatively complex concept (such as quality of service, addressing, etc.), and

$SIL(Bi) \supseteq SIL(Bj)$ holds.

Which D-nodes are related is determined by the test designer.

Step 5 is a generalization of Step 3 to D-nodes. It may be used to combine the blocks that contain different parameters (O-nodes) that are assigned in the same normal form transitions by related D-nodes (as identified by the test designer). It can be seen that in the Class 0 TP, the concept of quality of service (QOS in short) is specified using the variables "TPDU_size", "QOTS_estimate", "max_TPDU_size", "class_0" (constant) and "normal" (constant). Similarly addressing is specified using the variables "calling_T_addr", "called_T_addr", "remote_address", "calling_address" and "called_address". Thus, Step 5 combines the blocks containing the above nodes creating only two blocks, one for qos and one for addressing.

Step 6. Let Di be a D-node with SOL(Di) being empty, i.e., Di does not assign any other node (used in the predicates), these D-nodes are called **internal D-nodes.** The independent block Bi containing Di can be combined with some Bj if

$$SIL(Di) \subseteq SIL(Bj).$$

If there exists more than one such Bj, Bi will be combined with the Bj chosen by the test designer. It should be noted that only the labels on the arcs that assign a value other than the initial value of the Di is considered in SIL(Di).

Step 6 combines internal D-nodes with one of the existing blocks. The block is chosen such that the D-nodes of the block and the internal D-node are assigned in the same normal form transitions except possibly for the initialization of the internal D-node. The Class 0 TP specification contains no internal D-nodes. The application of Step 6 to the Class 2 TP is discussed in Chapter 8.

5.4.1.Partitioning a DFG of the Class 0 TP

The partition resulting from the application of the above procedure to Figure 5.4 is also shown in Figure 5.4 with dashed lines among blocks. The application of the procedure to the blocks obtained from Algorithm 5.1 is stopped after combining the final blocks in Step 5 (the blocks of QOS and addressing as explained previously) since no further merging is possible. The resulting partition in Figure 5.4 has 7 blocks.

5.4.2.Data Flow Functions

We define each block of the partition obtained by the merging procedure as a **data flow function.** For communication protocols, data flow functions can be considered as specific refinements of different control phases (connection establishment, data transfer, connection

freeing, etc.). In particular, in the Class 0 TP specification, we identify the following four data flow functions for the connection establishment phase (in left-to-rigth order in Figure 5.4):

- connection referencing (remote and local references),
- transport user end point identification,
- quality of service,
- addressing.

The data transfer phase contains two data flow functions for the Class 0 TP specification:

- user to peer data transfer,
- peer to user data transfer.

Finally, the connection freeing phase has only one data flow function:

- disconnection.

The data flow functions for the data transfer also reveal the order of execution of the normal form transitions that are self-loops in the CG of the Class 0 TP. In Figure 5.4, from the block of user-to-peer data transfer we see that the spontaneous normal form transition labelled P14 must follow the normal form transition (a WHEN transition) labelled P13. Depending on the length of the data placed in "out_buffer", P14 may be executed one or more times. If the buffer is empty, P14 can no longer be executed. A similar execution order applies to P15 and P16.

5.4.3. Control Functions and Data Flow Functions

The normal form transitions in a data flow function (DFF in short) as determined from the labels of the arcs can be used to relate the data flow functions with the control functions.

A DFF may contain normal form transitions that occur in more than one subtour, i.e., one or more control functions. This is expected since a protocol variable in a data flow function is used for a given purpose which may be needed in more than one subtour. In particular, "remote_ref" of the Class 0 TP is assigned by "source_ref" of CR in one subtour and by "source_ref" of CC in another subtour (see Figure 5.4 and Table 5.1).

Since a subtour may contain more than one control phase normal form transitions of a subtour may occur in more than one data flow function. For example, the first subtour in Table 5.1 occurs in all of the 7 data flow functions in Figure 5.4.

Implications on protocol testing of the above two properties of the control and data flow functions will be investigated in Chapter 6.

5.4.4.Data Flow Dependencies

Data flow dependencies between the blocks arise from the arcs shared. More formally, let Bi and Bj be two blocks in a partition which is obtained by the procedure above. There is a **data flow dependency** when Bi and Bj share one or more arcs. A node Ni of Bi is called a **dependent** node if one of the incoming arcs of Ni is originated in Bj. Thus a block can have dependent F- and D-nodes. If a F-node is dependent all D nodes assigned by this node also become dependent.

The data flow dependencies in a partition of the DFG can be found by an algorithm which finds **cuts** in a graph [Even 79].

In Figure 5.4 we identify the following dependent nodes:
The D-node "out_buffer", F-node "set_max_get_size".

The impact of data flow dependencies on protocol testing will be discussed in Chapter 6.

5.5.Protocol Design Validation Using Flow Graphs

The control and data flow graphs discussed in this chapter can be used for protocol design validation. We assume in the following that the protocol is specified using a formalism such as Subgroup B FDT. Thus, transformations of Chapter 4 can be applied, and from the normal form

transitions obtained, a CG and a DFG can be constructed. These graphs are useful for protocol design validation as explained below. The DFG also visualises the flow of data over each protocol variable, as defined in the specification.

In this section we will classify different validation aspects that involve the use of the flow graphs and point out the errors found in the protocol specifications of the Class 0 and Class 2 TP.

5.5.1. Use of CG

The CG can be used for detecting missing assignments to the major state variables in the normal form transitions. In complex protocol specifications where local procedures and functions are used intensively and many modules are employed to specify the protocol entity, assignments to some of the major state variables may be missed. Errors of this type are detected when constructing the CG where assignments to the major state variable(s) are modeled.

In the Class 2 TP specification, one of the major state variables, i.e., NC_state of the Mapping module is assigned to the value "closed" in none of the normal form transitions although some normal form transitions do close the network connection. Consequently, a statement assigning "closed" to NC_state should be added to the BEGIN blocks of

all normal form transitions where the network connection is closed.

5.5.2. Use of DFG

The DFG can be used for detecting missing assignments to the protocol variables other than major state variable(s). In protocol specifications, errors of this type occur usually by having one or more unassigned variables in the parameter list of a statement generating an output primitive.

After a DFG is constructed, some of the variable D-nodes may have no incoming arcs, i.e., the SIL of the D-node may be empty. Some of these nodes have to be initialized during the initialization of the system, and afterwards they can be considered as constant D-nodes. In all cases, variable D-nodes with empty SILs represent missing assignments.

In the Class 0 TP specification, the D-node "TCEP" had no incoming arcs, thus an F-node of type 2 called "assign_TCEP" was added to assign the D-node "TCEP".

Since variables used in a normal form transition need not be assigned in the same normal form transition, detecting missing assignments may not always be straightforward. In some cases, both the CG and DFG should

be consulted as discussed below (Section 5.5.3).

5.5.3.Use of CG and DFG

Subtours of a CG indicate the state order of the normal form transitions. Thus a variable assigned in a normal form transition of a subtour may be used in a normal form transition which follows the assigning normal form transition in the same subtour. The flow over each variable of a DFG can be checked using the subtours to detect possible missing assignments. If such an error is detected, assigning statements should be added to the proper normal form transitions that precede the use of the variable.

Using this approach on the Class 2 TP (see Chapter 8) we have detected that the variable called "local_T_addr" was assigned for peer initiated connections but left unassigned for user initiated connections. Thus an F-node of type 2 was added to assign "local_T_addr" in the normal form transitions labelled PE1 and PE2.

5.5.4.Semantic Errors

Errors in the specification such as missing assignments, as discussed above, require no knowledge of the particular protocol; therefore, they may be classified as syntactic errors.

Errors that require a knowledge of the protocol (semantic errors) are more difficult to detect using the flow graphs. We discuss briefly a semantic error found in the Class 2 TP specification:

In one of the normal form transitions the "credit_value" parameter of the CR PDU is assigned to the protocol variable "R_credit". Unfortunately, however, "R_credit" is used for credits (number of data packets that can be sent before acknowledged) given to the peer entity, thus its value should not be determined by the peer entity. This error in the Class 2 TP was corrected by changing the assignment statement to assign "credit_value" of CR to the variable called "S_credit" instead of "R_credit".

5.5.5. Self-loop Spontaneous Transitions

As discussed in Section 5.1.3, the order of normal form transitions in self-loops may be determined with the help of the DFG. In some cases, though, an examination of the DFG and the preconditions may reveal that a self-loop normal form transition can be executed as many times as desired provided that the protocol stays in the same major state. For example, the PROVIDED clause of a self-loop normal form transition labelled Pi may have expressions on protocol variables whose values are not modified in the BEGIN blocks of none of the self-loop normal form

transitions in the same major state. Thus, once the protocol enters into the major state and the precondition of Pi is satisfied, Pi can be executed nondeterministically until the protocol changes its major state (see chapter 4 for a discussion on nondeterminism in protocol specifications). We call a normal form transition like Pi an **indefinitely executable** normal form transitions (IENT in short). IENTs require special attention for both implementation and validation purposes. In the remaining part of this section we classify IENTs into two classes and discuss their consequences.

IENTs can be divided into two classes: those that produce an output and those that do not produce any output. The former class allows the protocol to produce more than one consecutive PDUs carrying exactly the same information, i.e., parameter values. Similarly, the latter case allows the protocol to execute the BEGIN block of the spontaneous transition more than once assigning the same values to the same variables. If it can be assumed that the spontaneous transitions of the above types will be implemented so that they will be executed only when they produce different outputs (the former case above) or when they assign different values to the variables in the BEGIN block (the latter case above) then there is no design error; otherwise the existence of such spontaneous transitions may represent a design error.

If the protocol is allowed to send a PDU to the peer entity **any time** in a given major state, there should exist a WHEN transition to receive and process the same PDU any time in the same major state. Arguing as above, many consecutive PDUs received may contain the same values in the parameter(s) of the PDU and this results in the execution of the BEGIN block several times unnecessarily since each execution generates the same result, which again may be considered a protocol design error.

We have found two spontaneous normal form transitions in the Class 2 TP that are IENTs, one that produces an output (AK PDU) and the other that does not produce any output. The latter normal form transition assigns a value (representing available space in the "receive_buffer") to the variable called "R_credit" and the former that outputs an AK primitive with "R_credit" and "TR" (both "R_credit" and "TR" are modified in a WHEN transition receiving a DT from the peer entity). These normal form transitions can repeatedly assign R_credit with the same value and send AK PDUs carrying the same parameter values. Channel congestion due to consecutive AK PDUs is avoided by flow control as discussed in Chapter 8.

The Class 2 TP contains a WHEN normal form transition which processes the AK primitives received from the peer. Thus, the protocol allows consecutive AK PDUs to be sent with possibly the same parameters as discussed above.

Only spontaneous IENTs that produce output are important as far as testing is concerned. Spontaneous IENTs that produce no output have effects that can not be observed directly. Thus the tests that involve spontaneous IENTs producing output should be able to handle this nondeterministic behaviour of the protocol entity by being ready to receive the output and respond accordingly.

5.5.6. Normal Form Transitions That are not Firable

Some of the normal form transitions obtained from the procedure of Chapter 4 may never be eligible for execution. A normal form transition cannot be executed if its precondition (WHEN and PROVIDED clauses) is never satisfied. Normal form transitions with unsatisfyable preconditions (especially PROVIDED clause) may be obtained when modules are combined since only symbolic replacements are done (see Chapter 4).

To detect normal form transitions which are not firable, a reachability tree can be constructed considering all possible pre- and post-conditions (BEGIN blocks) of the normal form transitions. This method is expensive since the reachability tree may blow up. Instead, we propose a simpler method which, however, does not guarantee detection of all unfirable normal form transitions. The CG, DFG and preconditions are considered together to detect any

unsatisfyable expressions in the PROVIDED clauses. Using this method, it can be determined that in some major states of the protocol some protocol variables cannot have certain values. For example, in the Class 2 TP specification, it is easy to see that the internal D-node called "TC[TC_id].in_use" has the value true in all transport states other than "closed". Thus any precondition requiring that "TC[TC_id].in_use" be false in transport states other than "closed" cannot be satisfied. We have detected that 12 of the normal form transitions of the Class 2 TP specification cannot be fired because of conflicts of this type.

6. TEST DESIGN METHODOLOGY

This chapter presents a test design methodology, which is a generalization of the test sequence selection discussed in Chapter 2. This methodology should be applicable to the **real protocols** such as standard protocols in the context of the OSI Reference Model. An implementation of the protocol is considered as a black box and availability of a formal specification of the protocol in FDT is assumed. The formal specification is usually written based on a general description of the protocol in natural language called protocol standard. From the specification, its normal form transitions are obtained and CG and DFG of the protocol are constructed using these normal form transitions.

In the first part of this chapter, a number of categories of tests are listed, and methods for selecting interaction sequences are discussed. The tests in each category have certain objectives to meet, these objectives are discussed next.

Specification based tests, i.e. tests for the various blocks of the DFG are done by varying input primitive parameters. Parameter variations are based on a classification of the input primitive parameters. A summary of how each block is tested is given, and the data flow dependent test characteristics and inter-block dependencies are examined.

The sequencing of tests is discussed next, including optimizations regarding simultaneous execution of block tests. This is followed by a description of tests for multiple connections. The chapter concludes with an examination of the relationship between the FSM test techniques and parameter variations.

The methodology is illustrated by examples drawn from the Class 0 TP. The application of this methodology to the Class 2 TP is described in Chapter 8.

6.1. Overview of the Methodology

An implementation of a protocol, i.e., the implementation under test, or IUT in short, is tested for conformance to the protocol specification, i.e., the protocol standard. The IUT is assumed to be a single "black box" entity stimulated and observed from two service access boundaries, one which should provide the (N)-service and the other which uses the (N-1)-service [Rayner 82]. Stimulation/observation of the (N-1)-service is indirect as discussed in Chapter 1.

In Chapter 2, the test sequence selection was based on an FSM model which ignored many aspects of the protocol, such as primitive parameters, and pre- and post-conditions of the transitions. If additional protocol variables are included as state variables, the resulting number of states

may increase without bound. Parameter values of the input primitives increase the number of transitions of the protocol machine in a pure FSM model. If parameter values were varied exhaustively, the resulting machine would have a very large number of transitions (especially due to data variations). Thus it is impossible to drive the IUT into all of the states and apply all possible inputs. This result is to be expected since even a small program with two integer inputs can not be tested exhaustively. In order to cope with this complexity, we will base the test design methodology on the formal specification of the protocol in the extended FSM model of the FDT and its decomposition as discussed in Chapters 4 and 5.

We assume that, in addition to the informal specification of the protocol, a formal specification of the protocol defines the issues of:

a) **protocol conformance** considering only the interactions of the IUT through (N-1)-service (PDUs and control information), and

b) **service conformance** considering in addition the interactions of the IUT with its user through (N)-service [BoCeMaSa 83].

Usually the formal specification is incomplete, that is, some of the protocol functions may not be formally specified. These functions are called **"informal functions"**. Note that some of the informal functions may be related to

the protocol/service conformance such as encoding/ decoding, flow control, etc., or to the additional characteristics of the IUT such as options supported, error processing performed, etc. We assume that the informal specification contains all the possible options and a list of possible implementation behavior in error cases.

6.1.1. Categories of Tests

In view of the above discussion, we divide the tests of the IUT for conformance into the following categories:

a) Protocol/ service Conformance Tests which test the formally specified functions of the protocol. The test design is based on the graph models of the specification. The set of tests in this group will be called **block tests.** Block tests include the tests for determining the parameters of the IUT (see Section 6.2.1. for a definition), and the tests for determining the options supported.

b) **Informal Function Tests** include the tests for the protocol/ service functions that are not formally specified.

c) **Robustness Tests** include the tests about the handling of protocol errors (i.e., PDUs arriving in a major state for which it is not specified what to do with them, or PDUs carrying invalid parameter values), and service errors (i.e., unexpected user behavior).

Protocol/ service conformance tests can be obtained

from the graph models, i.e., control and data flow graphs of the formal specification. The partitioning into blocks, i.e., the data flow functions of the DFG, decrease the complexity of protocol testing since each block can be tested separately. We assume that the IUT respects the independence among the blocks in order that they may be tested independently, in other words, errors in one block are assumed not to influence the behaviour of other blocks. There are two exceptions to this independence: data dependencies introduced in Section 5.4.4 and predicate dependencies introduced in Section 6.2.3. We also assume that if the IUT were going to malfunction it would do so during the tests [Rayner 82].

6.1.2. Test Sequence Selection Considerations

Test sequences for all the tests except the informal function tests can be obtained from the graph models of the specification. Graph models are not completely helpful for informal function tests since no structural information on the function is available in the informal specification, except for a verbal description of the function.

For each of the above three categories of the tests, different test sequence selection methods can be used:

i) **Parameter Variations.** When a model of a data flow function is obtained from the specification, such as a block

of the DFG, a test sequence can be selected from the control graph. A subtour which includes one or more of the labels of the block is used, I-nodes of the block are varied and expected responses of the IUT are found from the O- nodes of the block. Either the subtour is repeatedly applied in consecutive connections one for each parameter variation, or the parameter variation is done in an inner loop (self loop) of the subtour requiring only a single connection. We assume that the subtour selected has no observability problems, i.e., all the D-nodes set by the normal form transitions of the subtour can be observed by the O-nodes in the same subtour. Such block tests will be elaborated in Sections 6.3 through 6.5.

ii) **Fault Models.** This method is used for informal function tests. Assumed fault classes in an implementation of an informal function are used to derive a test sequence for them.

Functional fault models were used in microprocessor testing to derive instruction sequences for testing a functional block of a microprocessor [ThAb 79b]. For protocols, it is difficult to find fault models to derive complete test sequences since the faults are not due to aging of components, but rather design errors. In general, we try to enumerate the possible observable effects of design errors and then, using the control graph, we construct a test sequence which attempts to verify if the

effects of the assumed errors can be observed. An example of test selection based on a fault model for the informal "encoding/ decoding" function of the Class 0 TP will be given in Chapter 7.

iii) **Unexpected Inputs.** Robustness tests can be obtained from the CG and DFG. For this purpose the CG is completed (see Chapter 2), (the responses will be determined by the tests) to include every primitive received in every state. It is then possible to select test sequences for unexpected PDUs and unexpected user interactions. The method consists of driving the IUT to a given state, applying the unexpected primitive, observing the response of the IUT and then disconnecting. An example of test sequence selection using this method will be given in Chapter 7.

The data flow functions are also helpful in robustness tests. PDUs carrying invalid parameter values can be selected by inspection from the DFG, i.e., the I-nodes in each block are searched for invalid values and the determined values are tried in the tests.

6.1.3. Objectives of the Tests

It is desirable to design tests that will guarantee the detection of all the errors in an implementation. Using the formal models of the hardware faults, the microprocessor tests of [ThAb 79b] are designed to guarantee detection of

the assumed faults. However, it is difficult to give fault models for software errors, in particular design errors. In addition, it is impossible to test for all possible input parameter values and all possible interaction sequences. Therefore, test sequences for software, and protocol implementations in particular, cannot guarantee detection of all errors, i.e., one can not expect to obtain 100% test effectiveness. We suggest that the effectiveness of the proposed test sequences be verified by experience.

The tests developed using our methodology will attempt to satisfy the following objectives:

a)All the control paths in the specification must be experienced. This corresponds to the "branch coverage" criterion in software testing [Prather 83]. This criterion can be met by passing through all the normal form transitions at least once, following at the same time the control graph. Note that the "branch coverage" criterion is weaker than the "path coverage" criterion which would require experiencing all the subtours of the CG. For simple cases such as the Class 0 TP, the two criteria are equivalent, but for more complex protocols, such as the Class 2 TP (see Chapter 8), a subset of the subtours may be sufficient for "branch coverage".

b) The data flow graph, i.e., the specified data flow functions, should be verified. Each function should be verified independently of the other functions (see the

assumption in Section 6.1.1) by setting all of its D-nodes and observing the D-nodes through all the arcs. Setting D-nodes is achieved by parameter variations, i.e., value instantiations to the I-nodes.

c) Every informal function should be verified using a fault model.

d) Robustness of the implemention should be verified. Robustness tests should verify the responses of the IUT to all the unexpected PDUs/ user interactions for all the major states. Some assumptions may be made concerning invalid parameters: only a single invalid parameter is selected for each primitive. PDUs with invalid PDU code should also be applied in all the major states.

6.2.Preliminary Test Design Considerations

6.2.1.Definitions

We define here the "parameters of the IUT", the "subtours of a block", the concept of a "flow" and a "transition".

The parameters of the IUT include the following information:

- the (N-1) service access point address(es),

- the level of quality of service (QOS) provided by the IUT,
- the number of multiple connections supported.

Some of these parameters are assumed to be known without testing (e.g., addresses), while others are determined by the block tests.

The subtours of a block are those subtours of the control graph which include normal form transitions that assign the D-node(s) (or O-nodes if the block contains no D-nodes) of the block. These subtours are used in the block tests assuming that they have no **observability problems,** i.e., the assigned values can be verified by observing the value from the O-node(s) in the same subtour.

The flow covered by a subtour is the set of arcs in the block containing the labels that occur in the subtour.

A normal form transition becomes a **transition** if specific values are assigned to all of its input primitive parameters, thus more than one transition can correspond to a given normal form transition.

6.2.2. PROVIDED clauses

The logical expression in the PROVIDED clause of a normal form transition obtained from the transformations described in Chapter 4 can be arbitrarily complex, making it difficult to select the test data satisfying the PROVIDED

clause. Therefore we propose here a method to simplify the PROVIDED clauses of normal form transitions, in order to facilitate the parameter variation process.

The Boolean expression in the PROVIDED clause is composed of one or more **elementary expressions**, where an elementary expression is either a Boolean variable or input primitive parameter of type Boolean or a relational expression on protocol variables or input primitive parameters. The PROVIDED clause is converted into a disjunctive normal form (DNF) such that every product contains all the elementary expressions exactly once. We call this form of the DNF a disjunctive canonical form (DCF). Since the products in the DCF are **mutually exclusive** by definition, the sums can be removed by generating a normal form transition for each such product. The resulting PROVIDED clauses contain only "and" and "not" logical connectives. No modification is necessary within the BEGIN blocks.

As an example we apply this transformation to the normal form transition given in Figure 4.3 (the first normal form transition in the figure). The corresponding DCF expression of the original PROVIDED clause and the resulting normal form transitions are listed in Figure 6.1.

Note that if the two operands of an "or" are mutually exclusive (i.e., "B and C" in Figure 6.1 can never be true) only two normal form transitions instead of three are

generated.

```
DCF expression:
(state = idle and A and B and C and D and X <= 8) or
(state = idle and A and B and ¬C and D and X <= 8) or
(state = idle and A and ¬B and C and D and X <= 8)

Normal Form Transitions:

WHEN chan1.CONNECT_req
PROVIDED state=idle and A and B and C
      and D and X <= 8
BEGIN
    state:=connecting;
    S1;
    S3;
    chan2.CONNECT_ind;
END;

WHEN chan1.CONNECT_req
PROVIDED state = idle and A and B and ¬C
      and D and X <= 8
BEGIN

      ...(*same as above*)

END;

WHEN chan1.CONNECT_req
PROVIDED state = idle and A and ¬B and C
      and D and X <= 8
BEGIN

      ...(*same as above*)

END;
```

Figure 6.1.Removing "or"s in the PROVIDED clauses

The transformation on the PROVIDED clause does not modify the arcs in the control and data flow graphs. However, the label of each normal form transition whose PROVIDED clause is transformed should be replaced by a list of labels in both graphs and also in the subtours.

6.2.3. Predicate Dependencies

In this section we explore the relationship between the predicates and the blocks of the DFG, while ignoring any expressions on the major state variables.

We consider all normal form transitions of a block. The predicates of these normal form transitions may contain expressions on variables, parameters of the input primitive and/or local function calls and/or natural language expressions. The parameters of local functions may include input primitive parameters, variables and/or constants.

Predicate dependencies are introduced when a predicate of a normal form transition contains expressions on protocol variables (including the variables used as parameters of local function calls) that belong to other blocks.

Since the predicates of the normal form transitions for the Class 0 TP (see Appendix C) do not contain any expressions on protocol variables, the blocks of Figure 5.4 are not involved in any predicate dependency. The impact of predicate dependencies will be discussed in Section 6.3.

6.2.4. Satisfying the Predicates

Assigning values to the input primitives, i.e., I-nodes of the DFG is done to satisfy the predicates of the normal form transitions.

The transformation of Section 6.2.2, i.e., the removal of "or"s, facilitates parameter variations by simplifying the predicates. The predicates show the elementary expressions to be satisfied simultaneously. The following considerations apply to satisfying individual elementary expressions of protocol predicates:

Elementary expressions including only input primitive parameters and constants are easily satisfied by parameter variations, while those which refer to variables can be satisfied by an earlier transition in the subtour setting proper values to these variables.

Sometimes, it is possible to obtain a pure FSM model of a block. This happens when the predicates of all the normal form transitions of the block can be satisfied by considering the D-nodes of the block as major state variables and obtaining a product state space. This method can be used when the predicates include protocol variables that are internal D-nodes of type Boolean and/ or D-nodes of enumeration type, in addition to input primitive parameters and constants. An example of this technique will be given in Chapter 8. More discussion on internal D-nodes follows in Section 6.3.2.

Elementary expressions including local function calls can be grouped based on the types of the local functions:

- The values returned by a local function can be controlled by an input primitive, for example, if it returns the length

of a primitive parameter or if it decides whether the primitive is "valid".

- The value returned by a local function can be determined only from the history of the transitions in the subtour. The value may depend on previous value assignments to certain D-nodes as discussed above.

The Subgoup B FDT allows the use of natural language expressions (NLE in short) in the predicates. NLEs are used in cases where the specification remains informal such as for the description of quality of service, addressing, etc. Typical expressions that may occur in the PROVIDED clauses are:

(/able to provide/) referring to qos, and

(/check addressing/) referring to addressing, also negations of the above. The NLEs of the above type can usually be related to certain parameters of the input primitive.

Other NLEs depend usually on informal protocol/ service functions such as:

(/flow control from the user is ready/) related to flow control. For this case in particular, a fault model will be used to generate its tests, as will be seen in Chapter 7.

In Appendix C (normal form transitions of the Class 0 TP), for example, the quality of service predicates can be related to the class, options, TPDU_size, and QTS_req parameters of the T_CONNECT_req, CR, T_CONNECT_resp and CC

interactions. The transitions P13 through P16 have predicates related to flow control.

6.2.5. Types of I-nodes

Different types of I-nodes are considered in the block tests for parameter variations. We classify the I-nodes of a DFG and explain how the values may be varied for each type.

An I-node may be of enumeration type (Pascal enumeration type) or have a continuous domain (integer, array of octets, etc.). Some continuous domain I-nodes may be considered of enumeration type if their set of possible values is sufficiently small.

I-nodes of enumeration type can be enumerated **exhaustively**, i.e., transitions are generated from the normal form transitions for each possible value while for other nodes only a certain number of different values can be considered, in order to limit the length of the resulting test sequence. If the block contains more than one I-node of enumeration type, the test designer may try to limit the number of different values to be considered by not including all possible combinations of the values.

The I-nodes of enumeration type in the DFG of Figure 5.4 are:

options of T_CONNECT_req and T_CONNECT_resp,

class,options,variable_part.max_TPDU_size of CR and CC,
disconnect_reason of DR.

The I-nodes of continuous domain can be divided into the following five groups (this list may change depending on the protocol considered):

Parametric I-nodes: The values of these I-nodes are determined by the implementation and they are considered as parameters of the IUT, their values are fixed. The I-nodes corresponding to the addresses of the user and peer entities belong to this group.

The parametric I-nodes of Figure 5.4 are:
calling_T_addr of CR and CC,
called_T_addr of CR and CC,
to_T_addr and from_T_addr of T_CONNECT_req.

Reference Value I-nodes: The I-nodes used as source and destination reference values for the connections can be arbitrarily selected, but must be nonzero. The methods of test data selection for software testing [Howden 80] can be applied for these I-nodes. Usually, three specific values are selected, namely, the two end points and some interior point of the domain.

The following I-nodes of Figure 5.4 are in this group:
source_ref,dest_ref of CR , CC and DR,

Large Integers: The I-nodes that are integers of one or more octets are in this group. The methods of test data selection for software testing [Howden 80] may be adopted for these

nodes, as in the case of reference value I-nodes.

Figure 5.4 contains the following I-nodes in this group:

QTS_req of T_CONNECT_req and T_CONNECT_resp.

We assume here that the QTS_req parameter represents the maximum TSDU fragment size which is an implementation dependent integer.

User data: I-nodes in this group include the length and content fields of the exchanged data. Although test data selection of [Howden 80] can be applied, due to the importance of the user data for protocols we suggest that all values for the length of the data be enumerated while the content of the data is varied systematically to verify the correct delivery of every octet in the data.

The following I-nodes of Figure 5.4 are in this group:

TSDU_fragment.length, and TSDU_fragment.data of T_DATA_req,

user_data.length, and user_data.data of DT.

End point identifiers (EPI):The interaction with the user takes place over an (N)-service access point. These interactions contain a parameter to identify the connection end point to which the interaction refers. This parameter is called EPI and its value is locally decided. EPIs are important in multiple connection tests since different values are used for different parallel connections. The multiple connection tests therefore achieve the necessary

parameter variations for these nodes.

The following I-nodes of Figure 5.4 are in this group: TCEPI of T_CONNECT_req and T_CONNECT_resp.

6.2.5.1.Optional Parameters

The informal specification of a protocol defines the mandatory and optional parameters for each PDU. If a PDU does not contain any value for an optional parameter its value is considered **undefined** and/ or a default value is assumed as given in the informal specification.

For example in Figure 5.4, the max_PDU_size parameter of CR and CC is an optional parameter. If it is missing, the default value of 128 is assumed.

The reaction of the IUT to missing optional parameters should be verified in the tests.

6.3.Block Tests

Designing block tests can be considered to be the process of finding subtours (sequences of transitions) that effectively test each block of the DFG. Therefore for the design of block tests one considers the DFG, CG, the predicates and parameter variations at the same time. We first outline this process in Section 6.3.1 and then give the details in the subsequent sections.

6.3.1. Overview of the Block Tests

In this section we give a summary of how each block is tested.

A subtour of the CG is selected (see the definition in Section 6.2.1). From the I-nodes of the subtour, i.e., the parameters of the input primitives, the ones that belong to the block (I-nodes in the partitioned DFG with SOLs nonempty) are determined. The values of the I-nodes of the block are enumerated as discussed in Section 6.2.5 and all other I-nodes are fixed to certain values (default or parametric). Sometimes, the F-nodes of the block may increase the number of I-nodes to be considered for the block, as discussed in Section 6.3.2. Natural language expressions (NLEs) in the PROVIDED clauses may also introduce I-nodes to the block. For example, the expression

(/able to provide/)

refers to all QOS parameters of the input primitives, some of which may not be used in the block, thus creating I-nodes with SOLs empty. Therefore the I-nodes introduced by NLEs must be added to the I-nodes of the block (see for example the dashed lines in the DFG of Figure 5.4), and their enumeration is done depending on their type.

The next step is the determination of the output primitives and parameters (O-nodes) corresponding to the subtour selected. The O-nodes of the block associated with these primitives are the only way of observing the effects

of I-node parameter variations. The expected values of the remaining O-nodes (belonging to other blocks) are determined from the fixed values assigned to the I-nodes of the other blocks. The test (application of the subtour to the IUT) has to validate the values observed at the O-nodes of the block in relation with the values assigned to the enumerated I-nodes and the flow covered by the subtour (see the definition in Section 6.2.1).

The predicates of the normal form transitions in the subtour selected must be satisfied in order to be able to execute the subtour. Predicate dependencies discussed in Section 6.2.3 indicate the influence of other blocks (i.e., their internal nodes). The flow covered by the subtour in these other blocks (possibly not yet tested) has to be considered to satisfy the predicates.

The steps above are repeated for a number of subtours of the block, since each subtour may test the block only partially. In this way new subtours are added until all the assigning arcs of the D-nodes (O-nodes for the blocks with no D-nodes) have been covered. At the end, all the outgoing arcs from the D-nodes will have been tested, since the tests will eventually consist of a complete tour of the normal form transitions.

6.3.2.Data Flow Dependent Considerations

Some details of the flow related to the I-, D-, F- and O-nodes of the block are considered in the test design. In what follows we discuss considerations applying to each particular node type.

Since the flow over all variable D-nodes of the block, (i.e., the assigning arcs) should be tested by the subtours, variable D-nodes assigned in different subtours increase the number of tests for a block. Each test may be different in nature because the flow covered by each subtour may be different. For example, a D-node may be assigned by both I- and F-nodes, and one of the tests involves parameter variations for the I-nodes and, the other tests the variation of the values assigned by the F-node.

Internal D-nodes (not directly observable through O-nodes) are tested by identifying those transitions that assign the nodes, in such a way as to satisfy the relations on them in the predicates of the subtour. The value assigned to the internal D-node by the IUT can be verified from the O-node values that indicate the outcome of the evaluation of the predicates. The I-node values must be selected carefully so as to be able to differentiate the various outcomes through the observation of O-nodes only. The internal D-nodes can also be helpful in finding the order of execution of the transitions if the subtour contains a self-loop.

F-nodes of a block are treated depending on their types:

F-nodes of Type 1: The test designer should be able to determine the values returned depending on the inputs given. These values are usually found by consulting the informal specification.

In Figure 5.4, the F-nodes "append" and "get_next_fragment" require special attention because they represent the operations on the variables of an abstract data type, namely "in_buffer" and "out_buffer". These operations can be observed from the O-nodes that they assign directly (for "get_next_fragment") or indirectly (for "append").

F-nodes of Type 2: They can be classified into three cases:
- F-nodes which are implicitely associated with I-nodes increase the number of I-nodes in a given block, they must be considered when enumerating the I-nodes of the block. As an example, the F-nodes "assign_calling_T_addr" and "assign_called_T_addr" are respectively associated with the "to_T_addr" and "from_T_addr" parameters of T_CONNECT_req. These new nodes and arcs are represented by dashed lines in Figure 5.4.
- F-nodes which assign implementation dependent values to D- or O-nodes are observed through the O-nodes. Sometimes, the observed values may be needed in assigning the I-nodes later in the same subtour. In Figure 5.4, the F-node

"assign_local_ref" assigns implementation dependent values to the D-node "local_ref".

- F-nodes which represent local procedures that **initialize** a variable are tested indirectly, i.e., by testing the block with normal form transitions that occur after the initializing normal form transition. The assumption here is that if the implementation does a wrong initialization of the D-node, the error can be observed later in the subtour by correctly setting up the I-node values. The same considerations apply to the initializations done by constant D-nodes. In Figure 5.4, the F-node "clear" initializes the D-nodes "in_buffer" and "out_buffer".

F-nodes of Type 3: The values returned by type 3 F-nodes related with Boolean expressions are easily determined from the incoming arcs to the F-nodes, since these usually assign Boolean type O-nodes.

Type 3 F-nodes related to arithmetic expressions are more difficult to deal with, because all possible values of all the D-nodes of a block assigned by type 3 F-nodes should be considered as different state values. Therefore, driving the implementation into all of these states may not be practical. Test design for blocks containing type 3 F-nodes should be based on the types of the D-nodes assigned and on the set of arcs relating these D-nodes. The detailed test design for the Class 2 TP (see Chapter 8) contains such a case.

Blocks with no variable D-nodes usually contain I-nodes flowing directly to the O-nodes, or O-nodes assigned by a collection of I-, F- and (constant) D-nodes. Each such flow should be tested (possibly in different subtours).

6.3.3. Dependent/Independent Blocks

The blocks that have incoming arc(s) from other blocks are called dependent blocks, because they are involved in data flow dependencies (see Section 5.4.4). All other blocks are independent. Independent blocks as well as dependent blocks may be involved in predicate dependencies (see Section 6.2.3).

The flow in independent blocks can be tested by parameter variations of its I-nodes independently of other blocks (except for predicate dependencies), assuming that the same independence is observed by the implementation. Once the independent block has been tested, fixed values for its I-nodes can be used in testing other blocks having I-nodes of the same primitive(s). Sometimes such blocks can be tested simultaneously.

In Figure 5.4, all blocks except the block of "out_buffer" are independent, without any predicate dependencies.

Data dependent blocks are usually created when D-nodes assigned in one control phase (dependency causing D-nodes) are used in the BEGIN blocks of the normal form transitions in other control phases. The tests for dependent blocks first set the dependency causing D-nodes to a specific value (possibly in a preceeding control phase) and then do parameter variations of the I-nodes of the dependent block. Dependent D-nodes as well as dependency causing D-nodes may be assigned in the same control phase.

Predicate dependencies are similar to the data flow dependencies, the only difference is that the D-nodes involved in predicate dependencies do not take part in the flow of the block.

6.4. Test Sequencing and Test Optimizations

In this section we discuss the considerations that apply to the sequencing (i.e., order) of the tests for the blocks, combining parameter variations of different blocks in one test (optimizing the number of tests), the number of connections required to test a block and the structure of the subtours (the "+" operator in particular).

6.4.1.Test Ordering

The following rules should be followed in sequencing the tests for the blocks:

Tests for the blocks whose D-nodes (O-nodes for the blocks with no D-nodes) are assigned in earlier control phases are done before the tests for the blocks whose D-nodes are assigned in later control phases. If there are more than one block to test in a given control phase, independent blocks are tested first (considerations related to testing them simultaneously are given in Section 6.4.2). Independent blocks that are involved in a minimum number of predicate dependencies (measured as the number of variables borrowed from other blocks) can be given priority in the test ordering, since they can be considered more independent than the other blocks.

The ordering of the tests for the blocks of Figure 5.4 will be given in Chapter 7.

Tests that determine the parameters of the IUT are done before the tests that use these parameters. As an example, the tests for the blocks whose normal form transitions include predicates such as

(/able to provide/)

determine the qos parameters of the implementation. The level of QOS provided can be determined by varying the I-nodes related with the QOS (see Section 6.2.4) adaptively

until the unacceptable level is found. Later, these parameter values are used for testing the call refusal blocks.

6.4.2. Optimizations

The following rules apply to combining parameter variations of different blocks in one test:

The blocks whose D-nodes are assigned in the same control phase can be tested using the same subtour. The blocks whose D-nodes or O-nodes are assigned in different control phases can be considered as containing more than one **subblock** (not necessarily disjoint), one for each control phase. Tests for each subblock are done in a subtour, possibly combining parameter variations with the blocks whose D-nodes are assigned in other control phases of the same subtour. For instance, the "disconnection" data flow function of Figure 5.4 can be divided into 4 subblocks involving the normal form transitions (P17,P18,P19), (P8,P9), (P11,P12) and (P2,P5) respectively. The flows corresponding to each of these subblocks can be determined from Figure 5.4. The subblock containing (P17,P18,P19) can be tested with one of the blocks of the "connection establishment" phase and all the other subblocks are tested separately, because Figure 5.4 contains no other blocks with D-nodes assigned by the normal form transitions of these

subblocks.

The blocks containing only continuous domain parametric I-nodes (see Section 6.2.5) need not be tested separately, these blocks may be tested with any other block whose D-nodes are assigned in the same subtour(s). As an example, in Figure 5.4, the addressing block may be tested with the connection referencing, or quality of service blocks.

Independent blocks may be tested with any other block whose D-nodes are assigned in the same subtour(s). In the example of Figure 5.4, the addressing, QOS, TCEP and connection referencing blocks are such blocks, thereby they may be tested simultaneously.

6.4.3.Number of Connections in a Test

The number of consecutive connections required to test a block depends on the normal form transitions of the block. If the normal form transitions assigning the D-nodes occur in the self-loop, a single connection is sufficient to test the block by parameter variations done in the self-loop. Otherwise, a new connection is established for each variation of input parameters. Nevertheless, even blocks tested in a self-loop may require consecutive connections if they are dependent.

Usually, there exists more than one block that can be tested in a self-loop. If the parameter variations are combined, it is important to add another test phase in which the I-nodes of all the blocks are varied simultaneously (see Chapters 7 and 8 for data transfer tests). In Figure 5.4, the blocks of "in_buffer" and "out_buffer" can be tested in a self-loop using one of the subtours containing the self-loop. Parameter variations of the two blocks can be combined, giving two test phases for data transfers. Another phase should be added to vary their I-nodes simultaneously. Since the block of "out_buffer" is data dependent, the tests are repeated for different TPDU sizes in consecutive connections.

6.4.5.Structure of the Subtours

A list of normal form transitions containing the "+" operators (see Section 5.1.1) such as P1+P2+...+Pn represented as a single transition in a control graph may be created by

a) Normal form transitions with the same primitive. They result from various paths in the transition type of the Subgroup B FDT specification, and/or from the transformations on the PROVIDED clauses to remove the "or"s (see Section 6.2.3).

b) Normal form transitions with different input primitives. These normal form transitions are created by different

transition types having the same present and next states.
c)Spontaneous transitions.

Note that a given list of normal form transitions may have normal form transitions from a combination of the above cases. In Table 5.1, the lists (P3+P4), (P6+P7) and (P8+P9) belong to the first case, the lists (P11+P12) and (P17+P18+P19) to the second case, and (P14+P15+P16) to the first and last cases above.

If the subtour used to test a block includes a list belonging to the case a) above, the I-nodes to be varied and the I-nodes to be fixed determine the choice of the normal form transitions from the list. Either the complete list is selected, since the I-nodes to be varied require inclusion of all the normal form transitions, or only some of the normal form transitions in the list can be selected since the I-nodes to be fixed do not allow the rest of the normal form transitions to be included. For example, in Figure 5.4, when testing the block of connection references using the second subtour in Table 5.1, only P3 is selected since both P3 and P4 have predicates on an I-node (max_TPDU_size) which is fixed because it belongs to the QOS block.

If the subtour used to test a block includes a list belonging to the case b) above, the test designer has to select one of the primitives to be used in the test. The choice usually depends on the synchronization of the test sequence in the subtour (see Chapter 2). The tests with the

first/ second subtour (user/ peer initiated connection establishment) in Table 5.1 selects P17/ P18 from the list (P17+P18+P19) because P17/ P18 frees the connection from the user/ peer side.

Sometimes, one of the inputs in the list is **uncontrollable**, i.e., it cannot be applied deterministically by one of the testing sides. These inputs are generated by the (N-1)-layer and applied to the (N-1)-service boundary of the IUT. The specification usually includes transition types related to the inputs of this type, because any implementation of the protocol should be ready to handle such cases. In the case of subtours including uncontrollable inputs, it is assumed that the same input is applied to all communicating entities, thus the Tester side and also the Responder side have to be ready to handle any such input.

In Table 5.1, P19 has an input called "network_reset" which is not controllable unless a special test device is used (see Chapter 1). This input is initiated by layer 3 in the case of a transmission error (nondeterministically).

The subtours containing "+" operators in places other than self-loops require consecutive connections to repeat the subtour for every choice introduced by the "+" operator.

6.5.Multiple Connection Tests

Handling multiple parallel connections is an important aspect of a protocol implementation. The protocol specification specifies (sometimes implicitly) the mapping of the multiple connections onto lower level connections. For example, in the Class 0 TP, the mapping of the transport connections to the network connections is done in a one-to-one fashion. This fact is implicit in the Class 0 TP specification [ISO 82] which does not include any arrays and ANY statements. All the variables of the Class 0 TP except TCEP can be thought of as belonging to an array indexed by TCEP.

In the Class 2 TP, a given network connection may be used for more than one transport connection. The Class 2 TP specification defines two arrays of variables indexed by "NC_id" for network connections and "TC_id" for transport connections (see Chapter 8).

D-nodes representing the connection array sizes may be treated as being of enumeration types. The enumeration of these D-nodes requires establishment of parallel connections. The number of parallel connections supported by the implementation is one of its parameters. The value of this parameter can be determined by trying to establish as many parallel connections as possible in a test. The subtour(s) to be applied is selected from the block containing the array size. Once the number of parallel

connections supported is determined, the other blocks can then be tested using parallel connections. Testing all the blocks using multiple parallel connections may not be practical due to the increased number of tests. Instead, only the blocks and informal functions which may be shared by multiple connections are tested. The block containing the data buffers and some informal functions, such as flow control, may be considered here.

Multiple connection test design for protocols having two array variables will be detailed in Chapter 8.

6.6.Parameter Variations and FSM Test Techniques

In this section we discuss the relationship between test selection methods for FSMs discussed in Chapters 2 and 3 and parameter variations, and generalize the definitions of special test inputs (see Section 2.5.1).

The subtours of the CG are parts of a transition tour of the FSM representing the protocol. Performing parameter variations implies application of the sequence in the subtour as many times as necessary. Combining parameter variations means that the number of I-nodes to be varied is increased by the I-nodes of the blocks considered together, but these I-nodes are not varied exhaustively.

As discussed in Chapter 2, a transition tour does not

necessarily have full fault detection, but some measures can be taken to increase the fault detection capability of a tour: A subsequence can be added to the tour in order to verify the state of the machine. For example, a subsequence accomplishing two-way data transfer can be added to the tests of one of the connection establishment blocks to verify that the IUT really enters the data transfer state. An example of this process will be given in Chapter 7.

As discussed in Section 6.1, parameter variations, pre- and post-conditions, nondeterminism and spontaneous transitions make it impossible to use FSM test techniques to generate test sequences for real protocols automatically, using programs such as those described in Chapter 3. This fact remains true even when a single subtour of the CG is considered in isolation. In some cases, considering only the normal form transitions of a single block, it may be possible to obtain a FSM modelling the sequence of normal form transitions. This is done by considering the variables of the block that occur in the predicates as state variables. Based on the resulting FSM, test sequences for the block may be obtained. An example of these techniques will be given in Chapter 8.

However, the programs that generate transitions tours (see Chapter 3) may be used to obtain subtours for any given control graph.

6.6.1.Use of W- and D-Methods

The FSM for a test of a block usually does not possess a W-set or a DS. A W-set or a DS can be obtained by (partially) completing the machine. The completion process has the problem of assuming responses for unspecified inputs that may not match with the responses of a given implementation. Also, the resulting W- or D-sequences can have synchronization problems. Thus a practical use of the W-, and/or D-methods for protocols is only possible if primitives for state recognition/ setting such as "read state" and "set state" [SaBo 84] are included in the specification.

When the PROVIDED clauses are considered, the state space of a given test is effectively determined by the variables that occur in all the predicates. Therefore the definitions of the read state and set state primitives may be generalized as follows: Instead of "reading" only the major state values, "read state" returns the values of all state variables, similarly for "set state". This set may then be used in the tests for each block of the DFG as a W-set (or a DS).

7.Class 0 Tests

This chapter describes the tests designed for testing implementations of Class 0 TP. These tests were applied to the TP implementations described in [Leveille 84] and [Serre 84] using the test system described in [Maksud 83] which implements the test architecture described in Chapter 1.

These Class 0 tests were designed before the theory introduced in Chapters 4, 5 and 6 was developed. An adhoc functional division and a dependency structure among the functions were the major considerations for structuring these tests as described in [BoCeMaSa 83]. In this chapter we give a detailed description of each of the tests and discuss the following points:

(a) Test sequence selection using a fault model,

(b) Solutions to certain synchronization problems. These problems arise in tests which establish consecutive connections (called single-connection synchronization problem) or simultaneous multiple connections (called multi-connection synchronization problem).

(c) Time-outs used in some tests, i.e., unexpected stimulation tests.

(d) Relation of these tests with the test design methodology of Chapter 6.

First these tests were formally described in the Subgroup B FDT. Each test description consists of two parts: one that describes the actions of the Tester (called

Tester(T) part), and one that describes the actions of the Responder (called Responder(R) part). The test descriptions were then translated manually into Pascal programs with proper interfaces to the Tester and Responder system routines.

In discussing test sequences for each side we use the notation:

$$S_{kind} \quad \text{or} \quad R_{kind}$$

where kind indicates the interaction primitive involved, and S and R stand for send and receive, respectively.

7.1. Classification of the Tests

A complete testing sequence consists of 15 tests. They may be classified into four groups:

i) Single Connection Tests: This group contains the basic tests (TTBAS0, TCFR), quality of service tests (TQTC, TQCI), call refusal tests (TCRT, TCRR) and data transfer test (TDTSC).

ii) Multiple Connection Tests: This group consists of the basic tests (TMC1, TMC2, TMC3), and the data transfer test (TDTMC).

iii) Unexpected Stimulation Tests: This group may further be subdivided into peer unexpected stimulation tests (TUS1, TUS2), and user unexpected stimulation test (TUS3).

iv) Call and Disconnect Collision Tests: There is a single

test in this group called TCDC.

7.2. Single Connection Tests

7.2.1. Basic Tests

The tests (TTBAS0 and TCFR) were designed to check if the implementation is able to encode/ decode the PDU's, to address its user and peer entity, and to establish and free a connection. The test sequence selection is based on a fault model for encoding/ decoding and a FSM model (i.e., similar to the control graph of Chapter 4) which gives the sequencing of the interactions.

7.2.1.1. Fault Model

We assume independence of encoding from decoding and also a correct network layer in the following discussion. Encoding/ decoding of the primitives can be modeled by defining the functions Fe and Fd respectively. These functions are mappings from I (the set of primitives) to I u {Ø} where Ø denotes null or invalid primitive [ThAb 79b].

Decoding: An input primitive Ii introduced by the network layer is decoded with Fd into zero, one or more primitives.

Fd(Ii) = {Ii} if there is no fault in decoding.

Encoding: An output primitive Oi to be introduced to the network layer is encoded with Fe,

Fe(Oi) = {Oi} if there is no fault in encoding.

To introduce faults, we consider each primitive with its parameters:

Ii(P0,P1,P2,...,Pn) for an Ii with P0 being the field which identifies Ii and P1,...,Pn being the fields corresponding to its n parameter values. Similarly,

Oi(P0,P1,P2,...,Pn) for an Oi.

We allow the following faults:

Decoding: Fd(Ii) = {Ø} where the implementation either cannot decode Ii or ignores it.

Fd(Ii) ={Ij} where Ii is decoded as Ij with Ii<>Ij, or

Fd(Ii(P0,P1,P2,...,Pn)) = {Ii} but one or more Pis get wrong values.

Encoding. Fe(Oi) = {Ø} where the implementation cannot introduce Oi to the network due to the network interface problems or any other fault that is not observable.

Fe(Oi) = {Oj} where Oi is introduced as Oj with Oi<>Oj, or

Fe(Oi(P0,P1,P2,...Pn)) = {Oi} but one or more Pis are introduced to the network layer with wrong values.

Note that the above fault model assumes only "first order" faults, i.e., higher order faults where Ii/ Oi would be decoded/encoded into more than one primitive are not allowed. If higher order faults occur they would appear as first order faults where one or more of the parameters are

decoded/ encoded with wrong values, thus we assume that a test sequence which detects all first order faults would also detect higher order faults. It should be noted that in the Class 0 TP "concatenation" is not allowed, i.e., a given NSDU can contain only one TPDU. This fact makes the first order fault assumption above more realistic.

In order to introduce/ receive a maximum number of primitives to/ from the implementation, the sequences which involve connection establishment, data transfer, and connection freeing from two sides (one from T (TTBASE), and one from R (TCFR)) are selected as test sequences using the FSM model of Class 0 TP. Parameters of all the primitives are fixed to their default (or arbitrary, for exchanged data) values and addressing parameters are used to test for a basic addressing capability.

The test that starts the connection from the Tester side can detect all the decoding faults for CR and DT, and all the encoding faults for CC and DT primitives. The other test can detect all the decoding faults for CC and DT, and all the encoding faults for CR and DT primitives. To prove this we consider for example the decoding faults in the first test:

a) if the implementation ignored the received CR (as may be detected by timer interrupt), the Responder part would not have received a T_CONNECT_ind (similarly for DT); then it would not have received a T_DATA_ind.

b) The CR and DT were not decoded into other primitives, since if they were they would become unexpected stimulations and the Tester would have received an "Err" PDU and terminated the test. It is assumed that there is no additional error in the implementation for handling unexpected situations. The other possibility is that the IUT may have ignored the unexpected stimulation which would be detected by timer interrupt and the Tester would terminate the test in this case.

c) The parameters of the CR were decoded correctly since the parameters of the CC received in response to the CR were verified for correctness, and since the user (Responder part) received a T_CONNECT_ind and verified its parameters. The parameters of the DT can be verified by the Responder to be equal to the data sent by generating the data with the same procedure as the Tester.

Similar proofs can be made for the encoding faults and also for both faults of the second basic test.

It should be noted that the original test programs (TTBAS0 and TCFR) contained no time-out mechanism, thus according to the fault model timer should have been used in the places mentioned above.

7.2.2.Quality of Service(QOS) Tests

There are two tests corresponding to the QOS negotiation for connections initiated from the Tester and the Responder sides, respectively. These tests enumerate the possible values of the max_TPDU_size parameter of the CR and vary the QTS_req parameter of the T_CONNECT_req, respectively. QOS tests are done with consecutive connections, one per QOS value applied. Although there is in general no direct relation between the max_TPDU_size parameter of the CR and the QTS_req parameter of the T_CONNECT_req, in TQCI for simplicity, we have enumerated QTS_req in the same manner as max_TPDU_size.

The test initiated by T (TQTC) can be described as:

```
T:  S                                R    S
      CR                               CC   N_DISCONNECT_req

R:    R                  S                    R
        T_CONNECT_ind  T_CONNECT_conf         T_DISC_ind
```

The above sequence is repeated each time increasing the TPDU_size parameter of the CR starting with the default value, i.e., 128, up to the maximum of 1024. The enumeration is stopped when a TPDU size not supported by the implementation is reached, i.e., when the implementation either:

a) refuses the CR by sending a DR, or

b) accepts the CR and decreases the QOS value in the responding primitive CC.

If case (a) happens, the Responder part receives no response and will still be expecting a T_CONNECT_ind when the Tester part terminates the test. Thus the following subsequence has been added to synchronize both parts and to terminate the test in a synchronized manner:

A connection/ disconnection with default TPDU_size initiated by the Tester.

Note that this termination works also for case (b) although it introduces an extra connection to the test sequence.

The test sequence for the test initiated by R (TQCI) is:

```
R: S                    R                   S
    T_CONNECT_req        T_CONNECT_conf     T_DISCONNECT_req
T:    R       S                               R
       CR      CC                              N_DISC_ind
```

The above sequence is repeated each time increasing the QTS_req parameter of the T_CONNECT_req which stands for the TSDU_fragment_size supported, in the same manner as for CR above. The implementation's response to the unsupported fragment sizes depends on the user interface. In the tested implementation [Leveille 84], such a request was rejected, thus causing a synchronization problem similar to the case above. A solution can be given in a way similar to above, i.e., the Responder initiates one extra connection with a default QOS parameter value.

Each connection establishment with a new QOS value in both of the above tests is followed by a two-way data transfer with the maximum negotiated PDU size. This data transfer has been added to increase fault detection since it verifies that the implementation really goes in to the data transfer state and supports the negotiated PDU size.

7.2.3.Call Refusal Tests

There are two tests in this group, one initiates calls from the Tester (TCRT), the other from the Responder (TCRR). They test the cases where the implementation or the called user refuses the call.

The test sequence is as follows: For TCRT,

```
T: S  R  S          S                R      S
    CR DR N_DISC_req CR                DR     N_DISC_req
R:                      R          S
                        T_CONN_ind T_DISC_req
```

The first CR carries a QOS parameter value (class = class_2) which is expected to be rejected by the implementation. The second CR carries only default parameters.

The test sequence for TCRR is:

```
R: S          R          S          R
    T_CONN_req T_NOT_ind T_CONN_req T_DISC_ind
T:                        R   S      R
                          CR  DR     N_DISC_ind
```

The first T_CONN_req containes a QOS parameter value (an unsupported TSDU_fragment_size) which should be rejected by the implementation, and the second contains default parameters.

7.2.4.Data Transfer Tests

This group consists of a single test (TDTSC) initiated from the Tester for correct data delivery, simultaneous data transfer and flow control.

TDTSC contains three phases and these phases are repeated for each TPDU_size supported (128 and 256 for the implementation tested) in consecutive connections.
Phase 1 tests the data transfer from the peer to the user. The user_data parameter of the DT primitive is varied for length and content. Over all, each byte position in the data parameter of the DT PDU gets all possible values from 0 to 255. Correct delivery is checked by acknowledgements sent by the Responder in a 8-byte DT PDU which carries an error report of the data received after receiving a complete TSDU. Phase 2 tests data transfer from the user to the peer. Parameter variations for the T_DATA_req primitive are done

for length and content. Correct delivery of a unit of data (last PDU is indicated by setting the end_of_TSDU bit) is checked by the Tester.
Phase 3 includes the actions of Phases 1 and 2 simultaneously.

An adhoc fault model is used in TDTSC to test flow control. In Phase 1, where flow control from the peer to the user is tested, the Tester sends DT PDUs to the implementation independently of the responses received (error reports). The Responder can check the data delivered for correctness since it knows the contents of the data to be received and sends the error report at the end of a TSDU. Thus if there is an error in flow control, it is assumed that the implementation will either be unable to stop the Tester and crash or it will deliver the data to the Responder in wrong order or with losses. Similar remarks apply to Phase 2 where data flows from the Responder to the Tester. Simultaneous flow control from both user and peer to peer and user, respectively, and buffer management are tested in Phase 3.

7.3.Multiple Connection Tests

Parallel multiple connection tests are done to determine the maximum number of connections supported by an implementation and to observe the effects of multiple connections on data delivery. There are two groups of multiple connection tests: basic and data transfer tests.

7.3.1.Basic Tests

This group includes three tests: one to create simultaneous connections by multiple CRs (TMC1), one to create simultaneous connections by multiple T_CONNECT_reqs (TMC2) and one to create simultaneous connections in both directions.

The TMC1 test sequence is obtained from the single connection basic test TTBAS0 by simultaneously applying it for more than one connection. Since an implementation may not be able to support all of the connections initiated by the Tester part, the Responder part may be waiting for the T_CONNECT_inds for some of the connections. Thus when the test terminates for the connections supported, the Tester part will terminate, but the Responder part will not. This situation is similar to the synchronization problem for a single connection. We call it the **multi-connection synchronization problem.** We present two possible solutions: (a) if the test sequence for each connection includes data

transfer from the initiating side then all the established connections can be synchronized delaying the data transfer until all of the possible connections are established. When the Responder receives the first data, it knows that no further connection is to be expected.

(b) One of the connections (with lowest source_ref/ TCEP value) takes the responsability of synchronizing the connections. This connection after all others are disconnected, sends data, receives its echo and then disconnects. At the moment the Responder receives the data it knows that no further connection is to be expected.

Both of the above solutions were used in TMC1, connections are synchronized at the beginning of the data transfer and an additional data transfer by one of the connections was included in the test sequence.

TMC1 determines the number of connections to be used in all the other multiple connection tests, thus the other multiple connection tests do not have the multi-connection synchronization problem. In here, we assume that the parameter (number of connections supported) remains constant regardless of the initiating side of the calls.

The TMC2 test sequence is generated from the single connection test TCFR by simultaneously applying it for all connections.

TMC3 tries both of the sequences in the basic tests,

half of the number of connections supported is initiated by the Tester, the other half by the Responder.

7.3.2.Data Transfer Tests

This group includes a single test (TDTMC) initiated from the Tester. The sequence is the same as the Phase 3 of the TDTSC test applied simultaneously on all connections but only for maximum TPDU_size supported. TDTMC tests buffer management and flow control in the presence of multiple parallel connections.

7.4.Unexpected Stimulation Tests

These tests are designed to determine the behavior of the implementation in situations which the protocol specification does not define. They are divided into two groups:

(a) peer unexpected stimulation tests (TUS1 and TUS2) and
(b) user unexpected stimulations tests (TUS3).

The sequence of TUS1 may be represented as follows:

```
repeat
    establish network connection;
    send a primitive other than CR;
    receive the response (if any);
    clear the network connection;
until all_primitives_tried;

repeat
    establish network connection;
    send the CR;
    send any primitive;
    receive the response (if any);
    clear the network connection;
until all_primitives_tried;

repeat
    establish network connection;
    send the CR
    receive the CC;
    send a primitive other than DT (or an erroneous DT)
    receive the response (if any);
    clear the network connection;
until all_primitives_tried;
```

Since each of the three phases (repeat-until loops above) require a different response from the Responder and the number of stimulations in each phase are not necessarily known by the Responder, the end of each phase is indicated by the Tester to the Responder by establishing a connection, sending data and then disconnecting. Once this is done after stimulations in the data transfer state, the test terminates. This mechanism partly synchronizes the sequence of TUS1, however, a time-out mechanism is necessary in cases where a primitive sent generates no response from the implementation, as discussed below.

TUS2 tests the peer unexpected stimulations in the case where all of the consecutive connections are initiated

from the Responder.

TUS3 tests the user unexpected stimulations. All unexpected service primitives are tried in different states and the responses of the implementation are sent to the Tester at the end of each phase.

A time-out mechanism is necessary in the unexpected stimulation tests because an implementation may ignore some of the peer or user interactions. When waiting for the response from the implementation, a timer is started. If no response is received before the expiry of the timer, it is assumed that the implementation has ignored the interaction. The connection is cleared to allow for the next stimulation to proceed.

7.5.Call and Disconnect Collision Tests

This group includes only one test which tries to create a call and a disconnect collision.

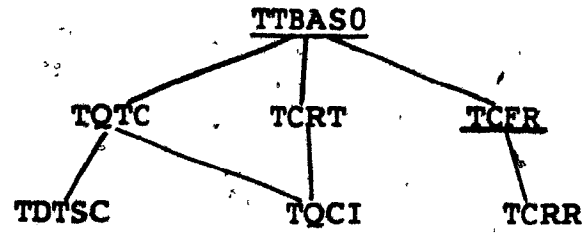
Both Responder and Tester initiate a connection as soon as the test starts for the purpose of creating a call collision. Since there is a one-to-one correspondence between a transport and a network connection in the Class 0 TP, the two calls initiated should create two simultaneous connections. After the connections are established, both sides send data and after receiving the data both sides

initiate a disconnection on both connections for the purpose of creating a disconnect collision. Whether the collision will occur or not depends on the delay in the network. Thus this sequence does not guarantee a disconnect collision.

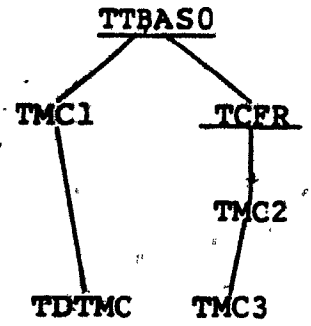
7.6. Sequencing of the Tests

Test sequencing for the four groups of the Class 0 tests is shown in Figure 7.2. The sequencing in each group is represented as a tree by taking the basic tests as the roots. A given test should only be applied after all the tests in the paths from the root to the node corresponding to this test are applied.

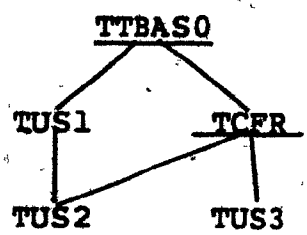
The global ordering of the test phases is also indicated in Figure 7.2. First basic tests are applied, followed by the tests for specified functions. The testing of functions not formally specified is either performed by "specified function tests" (for the case of flow control) or covered separately after all other tests are done.



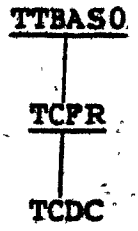
a)Single Connection Tests



b)Multiple Connection Tests



c)Unexpected Stimulation Tests



d)Call and Disconnect Collision Test

Figure 7.2. Test Sequencing for Class 0 Tests

7.7. Relation to the Test Design Methodology

The relation of the Class 0 tests described above to the test design methodology developed in Chapter 6 is considered in this section. Possible modifications to the test sequences are discussed in view of making them conform to the methodology described in Chapter 6.

The single connection basic tests (TTBAS0 and TCFR) correspond to the first part of the test design process, thus no modification is necessary for these tests (except for the addition of a time-out mechanism , see Section 7.2.1).

The QOS tests (TQTC and TQCI) are associated with the "quality of service" block (see section 5.4.2) of the DFG in Fig 5.4. The I-nodes max_PDU_size of CR (of enumeration type) and QTS_req of T_CONNECT_req (of large integer type) are enumerated in the two tests for the two subtours of the block. Some modifications to these tests are proposed below in relation with other blocks.

The call refusal tests are associated with the "disconnection" block of the DFG. Enumeration of the I-node "disconnect_reason" of DR should have been done, and the F-node "assign_add_clear_reason" should have been observed by possibly varying the "user_reason" parameter of the T_DISCONNECT_req. Thus the two tests in this group should be modified as follows:

(a) TCRR using the subtour P1(P8+P9) by enumerating disconnect_reason of DR, (b) TCRT using the subtour (P3+P4)(P11+P12) by varying user_reason of T_DISCONNECT_req. These tests may include the subtour (P2+P5) in one test since the subtour contains single interactions, i.e., it is not worth designing a separate test for P2+P5. In the block for disconnection, there are three predicates (P17, P18 and P19) which are the labels of the F-node called assign_disc_reason. These labels occur in the subtours of the QOS tests, thus the F-node can be observed for different values in the different consecutive connections of the QOS tests. These cases should be included in the QOS tests.

Data transfer tests are associated with the user to peer and peer-to-user data transfer blocks of the DFG. Since "out_buffer" is a dependent D-node, the QOS tests (which test the dependency causing D-node "TPDU_size") are done before data transfer tests. The initializing F-node "clear" for both of the above blocks is tested by observation of correct delivery (see Chapter 6 for a discussion on initializing nodes in the DFG).

Since the SILs of these two blocks occur in the same self-loop, these two blocks can be tested with one test. Thus Phase 1 of TDTSC corresponds to peer-to-user data transfer, and Phase 2 corresponds to user-to-peer data transfer. Phase 3 tests the two blocks simultaneously. Flow control is tested using a fault model as discussed in

Section 7.2.1. Therefore the data transfer test requires no modification.

The multiple connection tests have the effect of enumerating the D-node called TCEP of the DFG. The basic tests called TMC1, TMC2, TMC3 require no modification since the SIL of "TCEP" occurs in two subtours. The multiple connection data transfer test (TDTMC) is necessary for testing the flow control over simultaneous connections.

The unexpected stimulation tests correspond to part 4 of the test design process, no modification is necessary.

7.7.1. Uncovered Blocks

The Class 0 tests described above contained no tests for the blocks of connection referencing and addressing (see Figure 5.4). For the latter block, any variation of its I-nodes becomes an unexpected stimulation. This block is tested with correct address values in the tests for other blocks and may be tested with random address values in the unexpected stimulation tests.

Separate test(s) may be designed for the block of connection referencing, or its parameter variations can be combined with the QOS tests since their labels occur in the same subtours. Tests with invalid (more) and incorrect reference parameters are included in the unexpected

stimulation tests called TUS1 and TUS2.

7.8.Error Detection by Class 0 Tests

Errors detected in two implementations of the Class 0 TP with the above tests are reported in [CeBoMaLeSeSa 84]. In general, unexpected stimulation tests detected most of the errors. Also, Phase 3 of the data transfer test TDTSC proved to be very effective in detecting errors related with flow control, buffer management, etc..

8.Test Design for the Class 2 TP

In this chapter we apply the test design methodolgy to a considerably complex protocol, i.e., the Class 2 TP [ISO 82b]. The main features of the Class 2 TP are the support of multiple transport connections over a single network connection, and normal and expedited data transfer. Flow control for the normal data transfer is achieved using an acknowledgement scheme with receive/send credits (windows) for the two directions of transfer. Flow control for the expedited data is simply done by receiving/sending the next data only after the acknowledgement for the previous data.

We discuss the test design for the Class 2 TP in the following order: First the normal form transitions are explained. Then the control graph, its subtours, and the data flow graph are discussed. Next a partition of the DFG is obtained, and the dependencies between the blocks of this partition are discussed. Finally the tests for the blocks of the partition are described.

8.1.Normal Form Transitions

By applying the transformations of Chapter 4, 134 normal form transitions are obtained for the Class 2 TP, they are documented in [Sarikaya 84]. Some of the normal form transitions are given in Appendix B, they correspond to the transition types given in Appendix A, as discussed in

Chapter 4.

For obtaining the normal form transitions we made the following assumptions:

i) The transport service interface function called TS.user_ready which returns the length of the data that can be received by the user is modelled by introducing a variable called "length" which is assigned by a F-node of type 2 in all the normal form transitions corresponding to the transition type where TS.user_ready is used.

ii) To handle the case where an AK primitive is received and the negotiated value of the class is class 0, a major state value called T_Err_sent is introduced. The protocol enters into this state when AK is received and an Err PDU is sent to the peer in order to wait for a N_DISCONNECT_ind. When the N_DISCONNECT_ind is received the network connection is cleared. The same modification is made to all specified error cases.

8.2.Control Graph and Subtours

There are two major state variables, "state" and "NC_state" corresponding to the two modules ATP_type and mapping, respectively, of the Class 2 TP specification. After these modules are combined, the BEGIN block of the normal form transitions contains an assignment statement for each major state variable. The corresponding control graph

is shown in Figure 8.1. From this CG we observe that the state space of the protocol is not equal to the cross product of the state spaces of its modules. This is expected since most of the "state" values can exist only when the "NC_state" is "open".

In order to keep Figure 8.1 readable, we represent the transitions by single letters A thru Z and a thru f, each letter representing a list of labels of normal form transitions. Table 8.1 contains the list of labels corresponding to each letter used in Figure 8.1. In Table 8.1 and the data flow graphs of the following sections, a list of labels corresponding to consecutively numbered normal form transitions is written in an abbreviated form. For example, the consecutive normal form transitions numbered 81 thru 84 are written as P81 ... 84. A normal form transition labelled Pij is derived from the i-th transition type of the original specification [ISO 82b]. The associated input primitives for the labels of Table 8.1 are listed in Table 8.2. For spontaneous transitions either the output generated is listed, or the functionality (the variables set, primitives encoded/ decoded, etc.) is given.

Unlike the Class 0 TP specification, primitives from the peer entity (TPDUs) do not appear in the WHEN clauses nor directly in the output statements. This is so because the TPDUs are manipulated in the buffers which are moved or filled to/ from NSDUs in spontaneous transitions which

handle the encoding/ decoding and concatenation functions of the Class 2 TP.

8.2.1.Subtours

The initial state of the CG in Figure 8.1. is the state which contains the initial major state values of the two modules i.e.,

{NC_state=closed, state=closed}.

There is another state for which loops are obtained:

{NC_state=open, state=closed}.

The loops correspond to successive transport connections using the same network connection. Six states, including the data transfer state {NC_state=open, state=open} have self-loops.

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...604,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...610,PF04
Q	PN7
R	P91
S	P103,104,110,112,113,115,116,118,P2,P3,P81...84, P92,93,Pa1...A3,PB1,PB2,PF04,PG04,PG08,PG14, PN6,PQ1,PP1,PP2,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,PF03,PG03,PG07,PG13,PR3
b	P414
c	PG17,PK1
d	PL1
e	P106,P109,P116,P2,P3,P407,P413,P611
f	P2,P3,P108

Table 8.1.Labels of the Transitions in Figure 8.1.

P101...118	Spontaneous transitions which encode TDPUs to be sent, in particular
P101,102,117:	CR
P103,104,118:	CC
P105..108:	DR
P109:	DC
P110,111:	DT
P112:	AK
P113,114:	EDT
P115:	EAK
P116:	ERR
P2	Spontaneous transition which outputs N_DATA_req
P3	N_DATA_ind
P401...415	Spontaneous transitions which decode N_DATA_ind into CR,
P501...513	idem, into CC,
P601...612	idem, into DR,
P71...72	idem, into DC,
P81...84	idem, into DT,
P91...93	idem, into AK,
PA1...PA3	idem, into EDT,
PB1...PB2	idem, into EAK,
PC1	idem, into ERR.
PD1...PD3	Spontaneous Transition that outputs T_DISCONNECT_ind
PE1,PE2	Spontaneous Transition that sets class and max_PDU_size
PF01...10	N_RESET_ind
PG01...18	N_DISCONNECT_ind
PH1	Spontaneous Transition that outputs N_CONNECT_req
PI1	N_CONNECT_conf
PJ1	N_CONNECT_ind
PK1	Spontaneous Transition that outputs N_DISCONNECT_req
PL1	T_CONNECT_req
PM1,PM2	T_CONNECT_resp
PN1...N7	T_DISCONNECT_req
PO1	T_DATA_req
PP1,PP2	Spontaneous Transition that outputs DT
PQ1	Spontaneous Transition that outputs T_DATA_ind
PR1...R4	Spontaneous Transitions that update R_credit
PS1	Spontaneous Transition that outputs AK
PT1	T_EX_DATA_req
PU1	T_EX_D_READY_req

Table 8.2.Primitives Corresponding to the Labels in Table 8.1

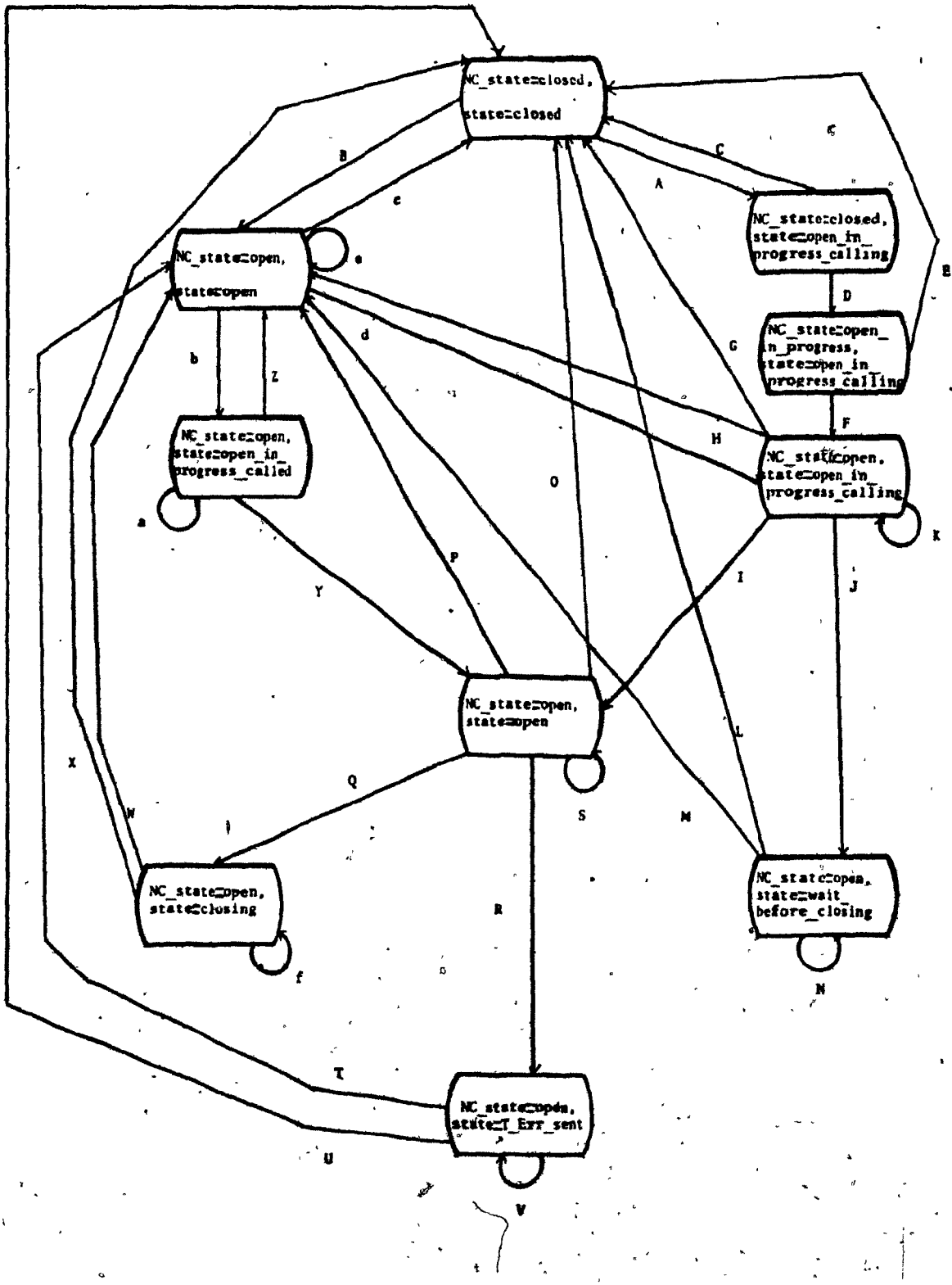


Figure 8.1. A Control Graph of Class 2.TP

Subtours of the CG are listed in Table 8.3. This table contains a relatively high number of subtours (115). This is due to the fact that the CG of this specification includes network and transport connection management for Class 0 and Class 2, thereby increasing the number of major states. One of the objectives set by the test design methodology (see Chapter 6) requires that all the normal form transitions are covered in the subtours to be used in the tests. This criterion decreases the number of subtours for the Class 2 TP to 25, i.e., the subtours numbered 1 thru 25 are the subtours to be used in the Class 2 tests. It is easy to see from Figure 8.1 that the discarded subtours are derived from the state {NC_state=open, state=open} and involve more than one control function such as "call refusal" followed by "connection establishment, data transfer, disconnection" over one network connection and these control functions are already covered by the above 25 subtours.

The control functions corresponding to some of the subtours of Table 8.3 are listed in the Table (for the transport connections initiated by the peer/ user entity).

All the self-loops in the CG contain the normal form transitions labelled P2 and P3 which receive/ send NSDUs. Since these two normal form transitions are needed in all control functions, the self-loops are repeated in the subtours (see for example the self-loop labelled "e" above).

```
1         A C
2         A D E
                 *
3         A D F K G
                   *   *
4         A D F K J N L
                   *   *
5         A D F K I S O
                   *   *   *
6         A D F K I S R V U
                   *   *   *
7         A D F K I S Q f X

8-25        B LOOP
                   *
26-43       A D F K H LOOP
                   *   *
44-61       A D F K J N M LOOP
                   *   *
62-79       A D F K I S P LOOP
                   *   *   *
80-97       A D F K I S R V T LOOP
                   *   *   *
98-115      A D F K I S Q f W LOOP
```

with LOOP containing:

```
 *
e c           Protocol errors in "closed" state
  *  *  *
(e ba Z) c    Call refusal by the user
 *  *  *
e ba YS O     Connection establishment data transfer
                freeing by peer/user (Class 0)
  *  *  *  *
(e ba YS P) c idem, by peer (Class 2)
 *  *  *  *
e ba YS Qf X  idem, by user (Class 0)
  *  *  *  *  *
(e ba YS Qf W) c idem, by user (Class 2)
 *  *  *  *
e ba YS RV U    Connection establishment, disconnection
                  due to protocol error in data transfer
                  (Class 0)
  *  *  *  *  *
(e ba YS RV T) c idem, (Class 2)
  *
dK G          Call refusal by the peer/user
   *  *
(dK H) c      Peer initiated connection establishment
              disconnection due to protocol errors
   *  *
dK JN L       Call refusal by the user
```

```
      *  *  *
(dK JN M) c  idem, by the peer
      *  *
dK IS O      Peer initiated connection establishment
             freeing by the peer/ user (Class 0)
      *  *  *
(dK IS P) c idem, by peer (Class 2)
      *  *  *
dK IS RV U   Peer initiated connection establishment
             disconnection due to protocol errors
             in data transfer (Class 0)
      *  *  *  *
(dK IS RV T) c Peer initiated connection establishment
              data transfer freeing by the user
              (Class 2)
      *  *  *
dK IS Qf X     idem, (Class 0)
      *  *  *  *
(dK IS Qf W) c idem, by the peer (Class 2)
```

Table 8.3.Subtours of Class 2 TP

8.3.Data Flow Graph

A DFG of the Class 2 TP is shown in Figure 8.2 (each page in the figure is numbered from 1 thru 10). Page 1 belongs to the encoding/ decoding, pages 2 thru 4 belong to the connection establishment, pages 5 thru 7 to the data transfer (including error cases) followed by pages 8 and 9 for the disconnection phase, and page 10 to the error cases related to duplicate connections. D-nodes that are used in more than one of these parts are replicated in the figure and idicated by an "off-page" connector (see the D-nodes NC_id, TC_id, max_PDU_size, etc.). In the specification, the TPDUs received/ sent are stored in an array called PDU_buffer. In Figure 8.2, the TPDUs decoded from a

N_DATA_ind are represented anywhere except in Page 1 as I-nodes, and the TPDUs encoded into N_DATA_req as O-nodes. More discussion on this point follows in Section 8.4.

One of the aspects of the DFG is the presence of many internal D-nodes, namely:

(a) in_use of boolean type, indicates if the transport connection is being used.

(b) this_side of enumeration type, indicates the initiating side of the network connection.

(c) supports_class_0 of boolean type, indicates if the class negotiated is Class 0,

(d) S-credit of enumeration type, is used as credit value for the outgoing data,

(f) EX_D_sent of boolean type, indicates whether expedited data from the user has been sent to the peer,

(g) EX_D_received of boolean type, indicates whether expedited data from the peer has been received.

The D-node called local_N_addr is assigned by none of the normal form transitions, thus we assume that it is correctly initialized.

The F-node of type 1 called determine_TC is assumed to have the effect of checking the validity of the "dest_ref" parameter of the primitive received, thus in what follows we ignore this node and all of its replications and add a relation about the dest_ref to all of the involved predicates.

The D-node called "assigned_NC" is set by the "NC_id" which is a record containing "id" and "local_N_addr" as its fields. We assume that a transport entity has only a single local network address. Then the NCEP value assigned to the transport connection alone is sufficient to associate the transport connection with a network connection, therefore in what follows "assigned_NC" is assumed to be assigned by the value of "NC_id.id". The D_node called "corresponding_TC_id" is ignored since it is not used anywhere else in the specification.

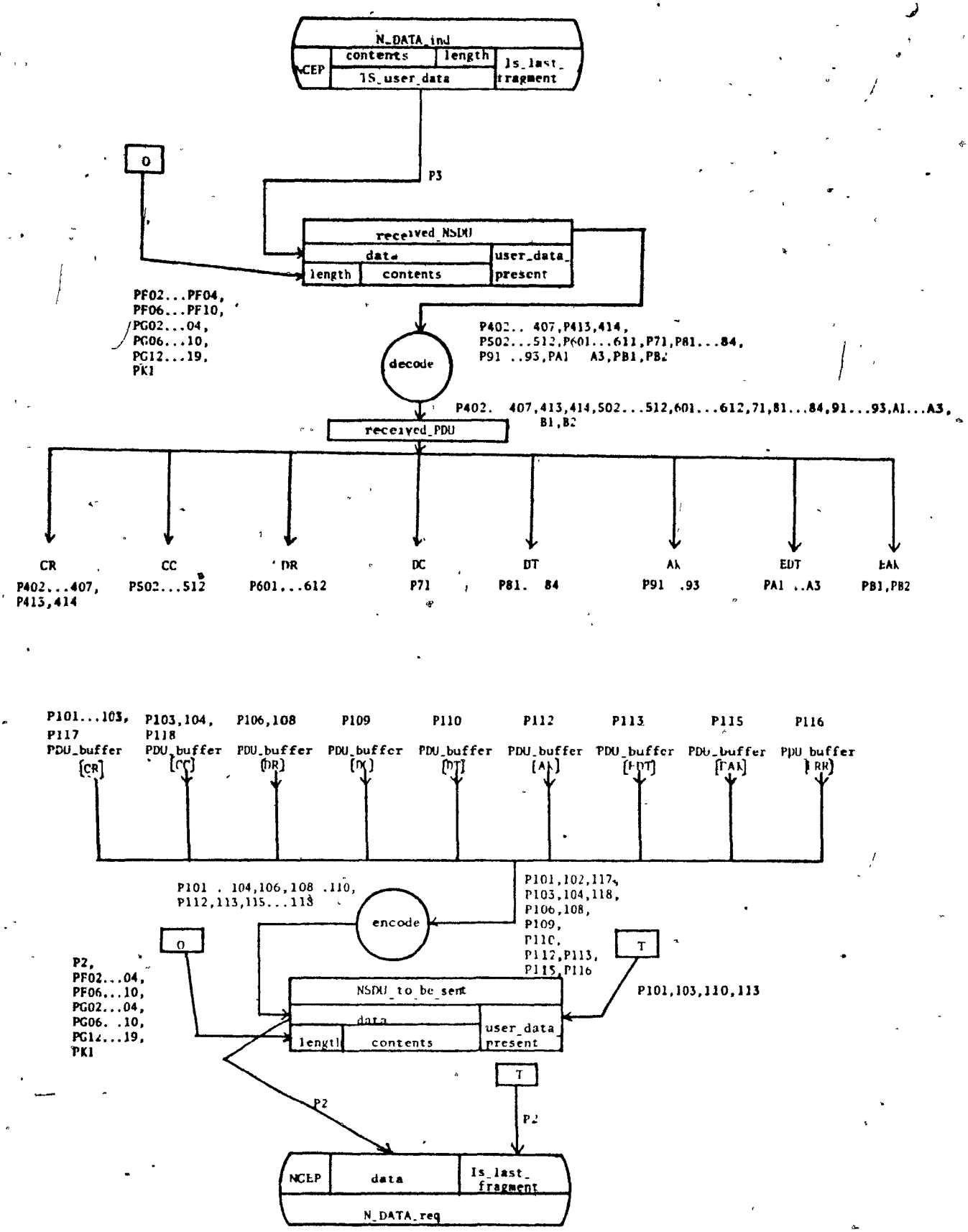


Figure 8.2 A DFG of the Class 2 TP

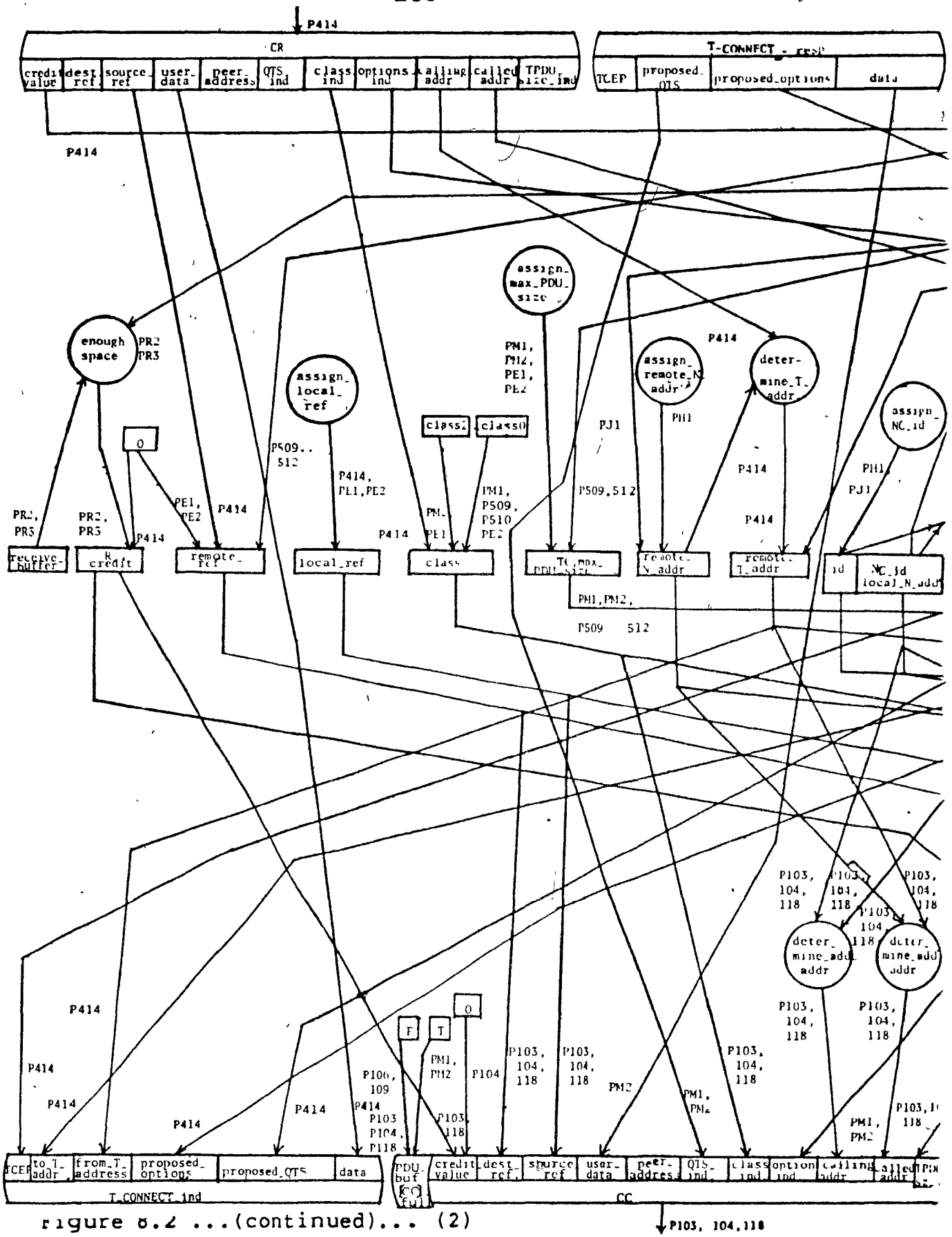


Figure 8.2 ...(continued)... (2)

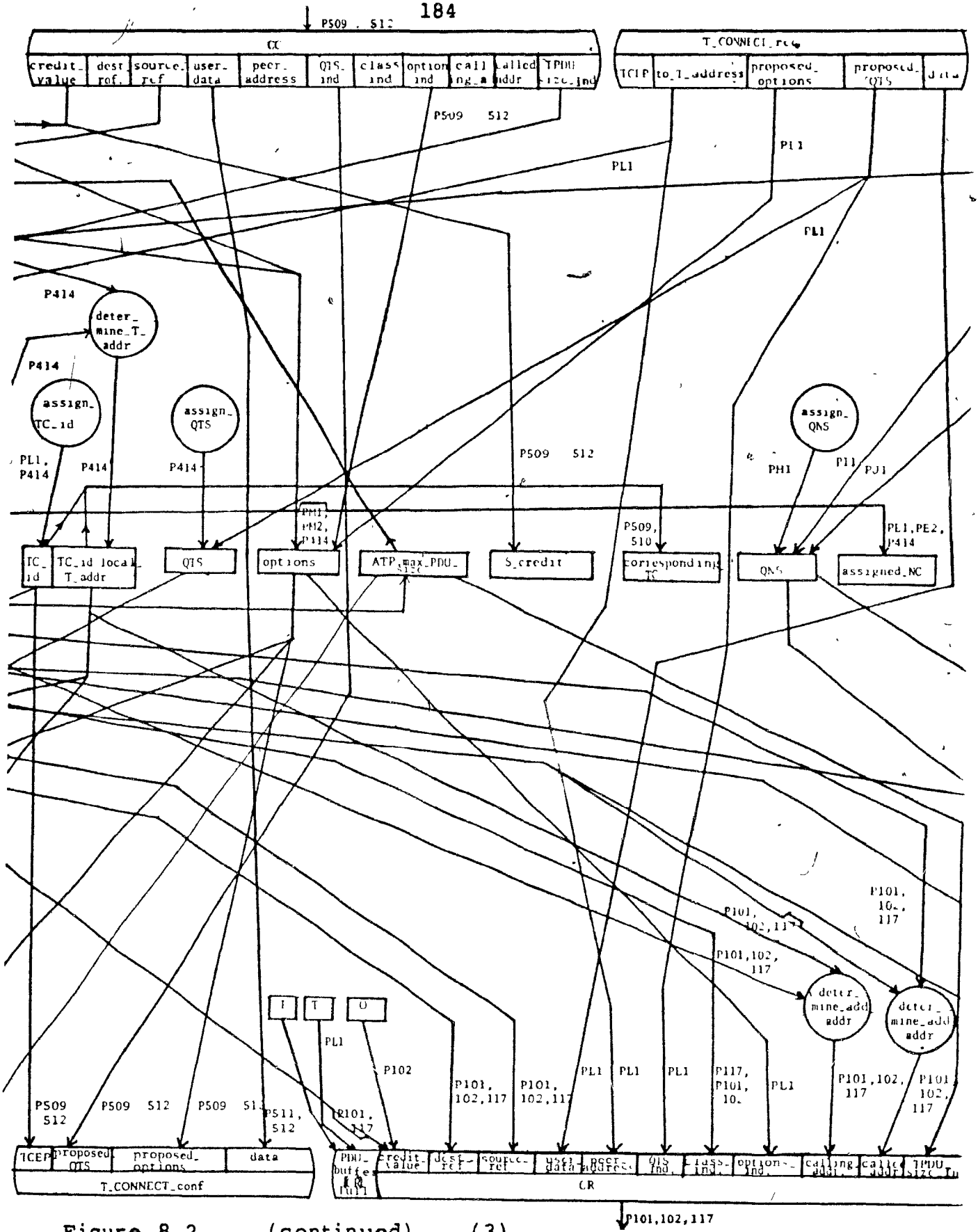


Figure 8.2. ...(continued)... (3)

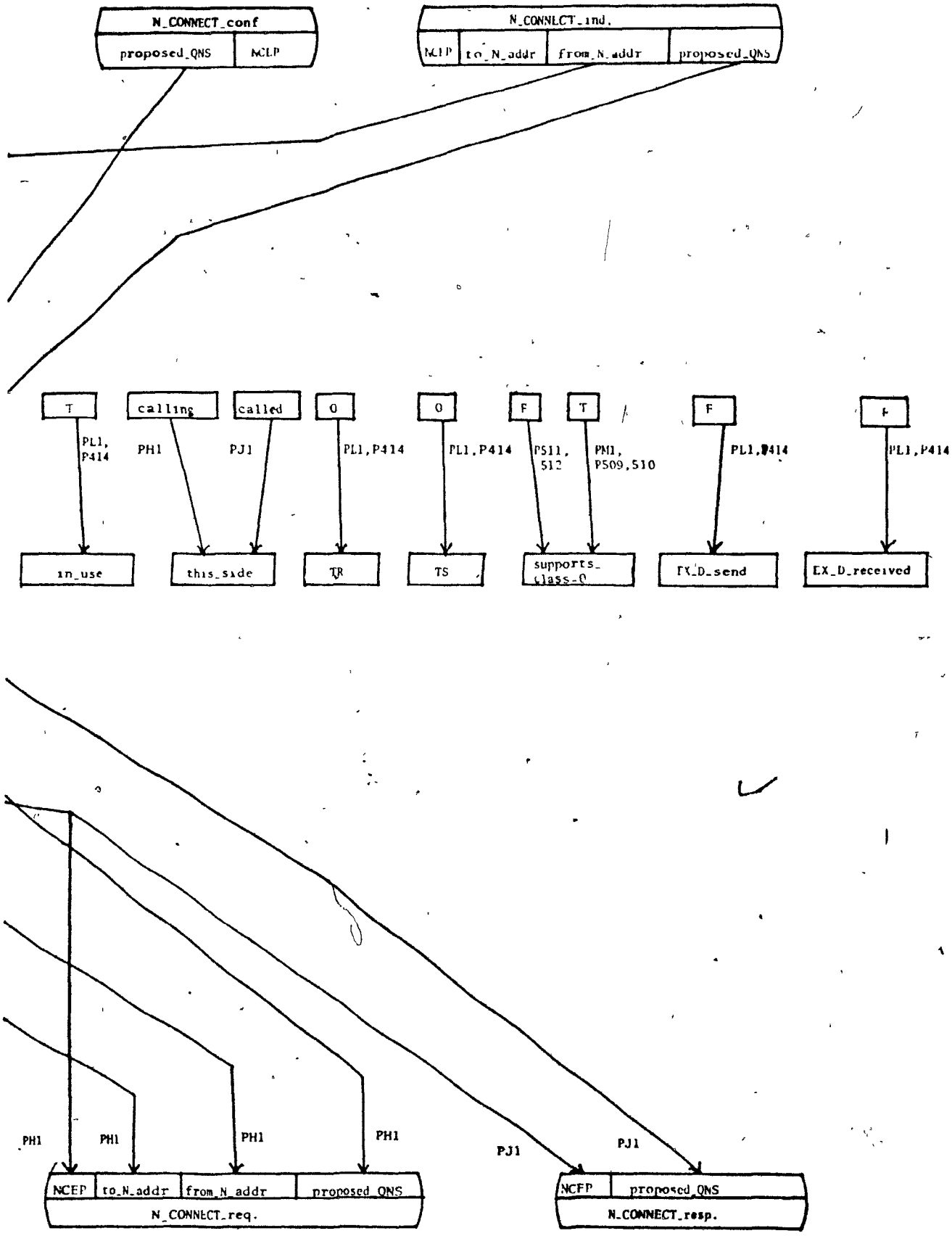


Figure 8.2. ...(continued)... (4)

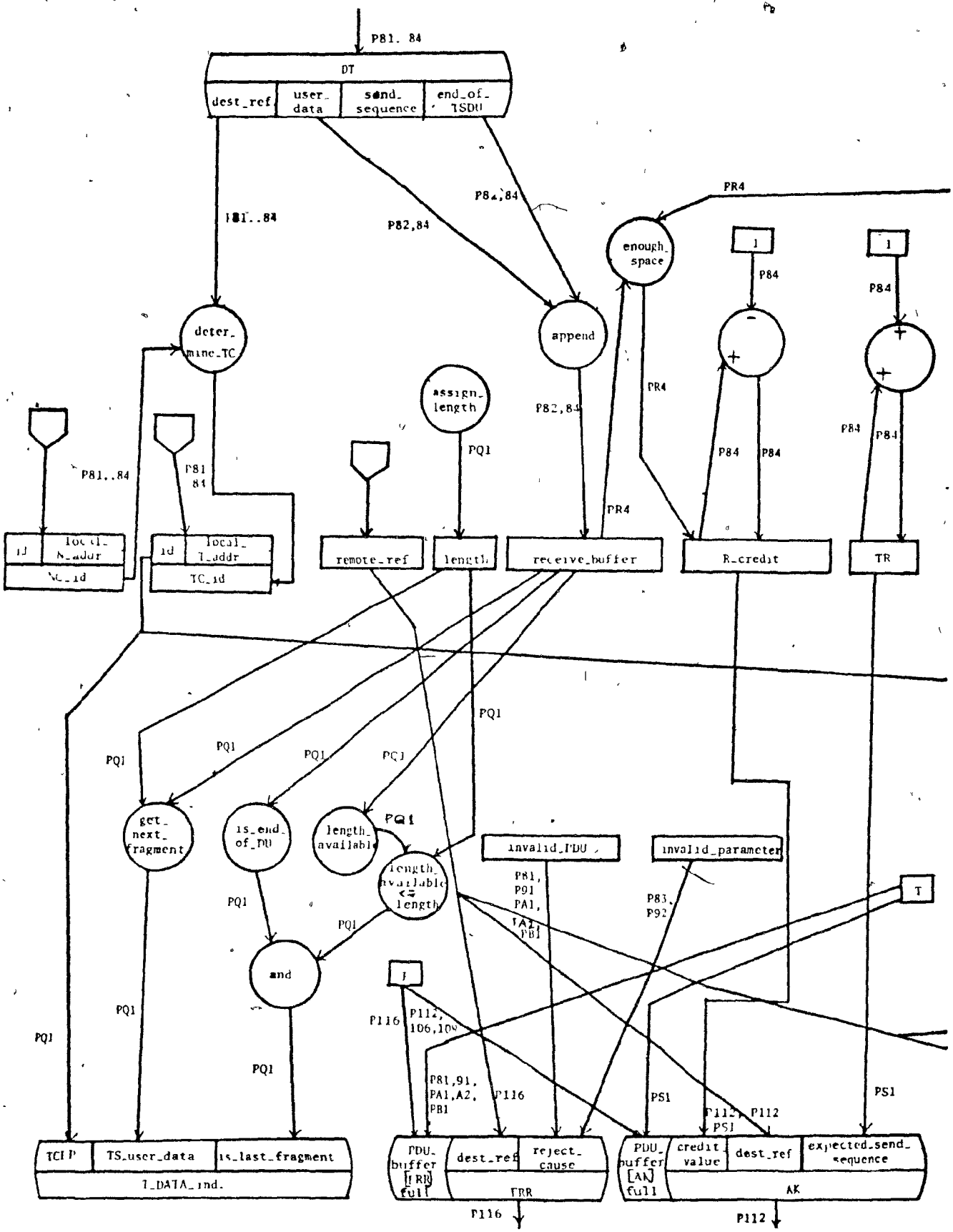


Figure 8.2. ...(continued)... (5)

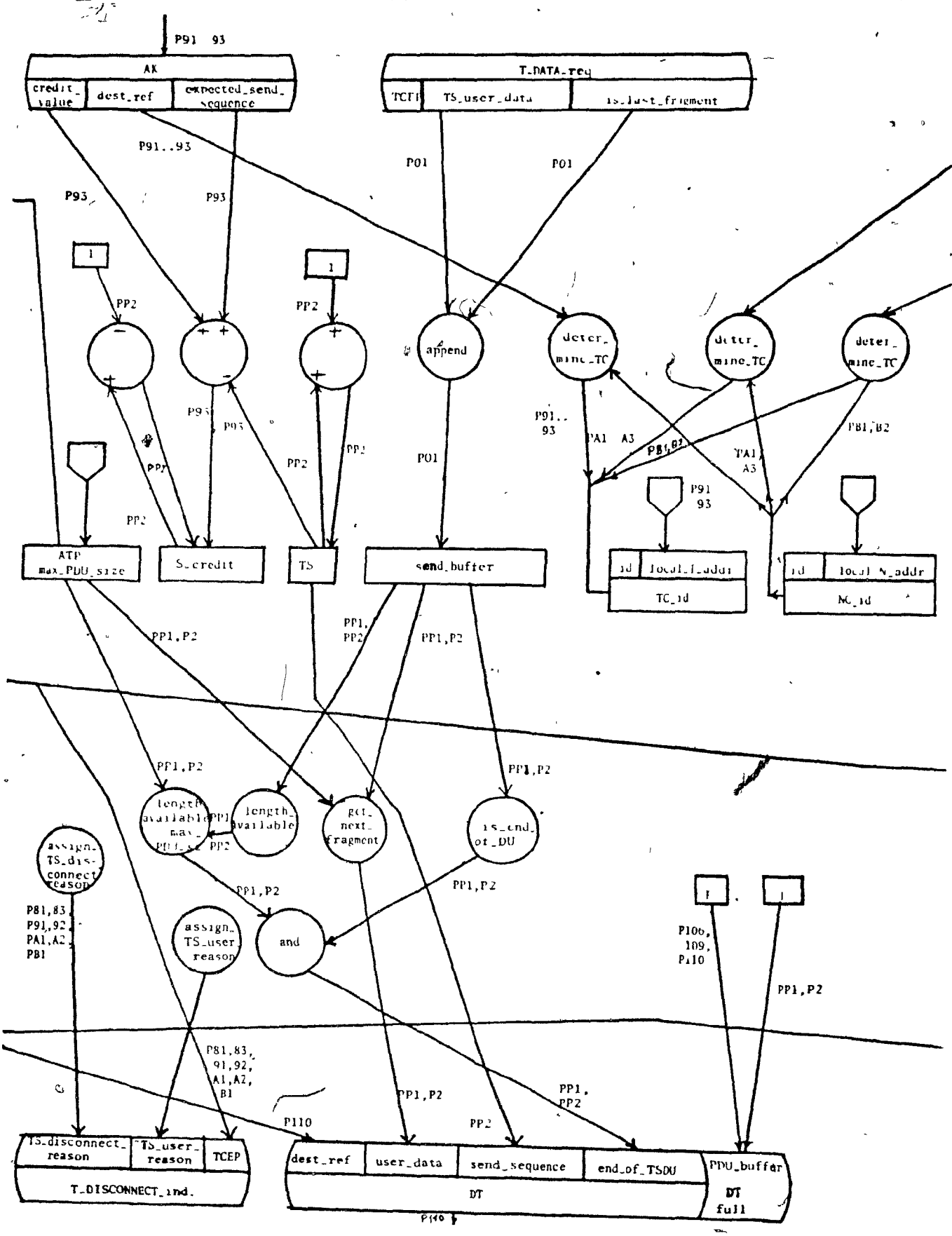


Figure 8.2. ...(continued)... (6)

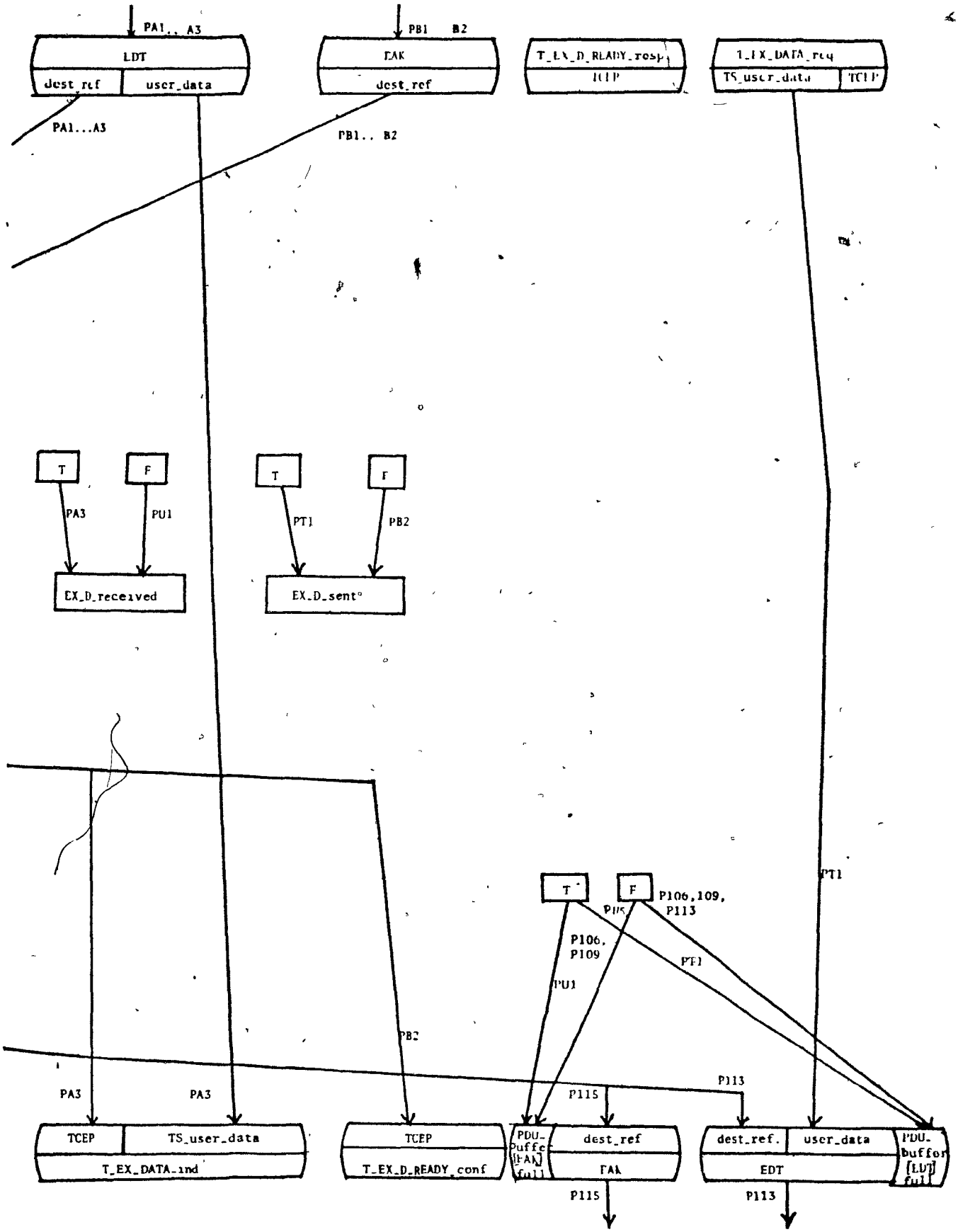


Figure 8.2. ...(continued)... (7)

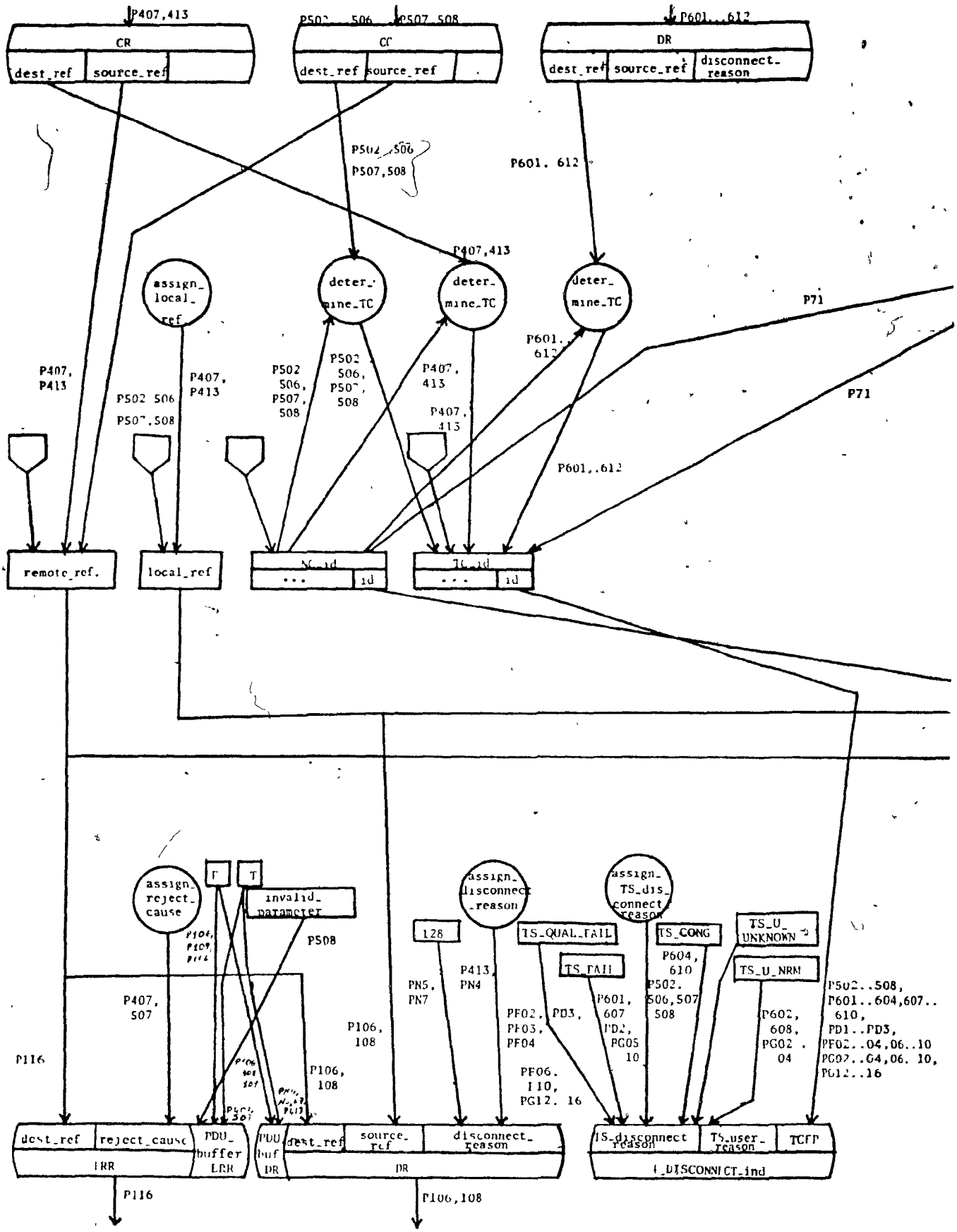


Figure 8.2. ...(continued)... (8)

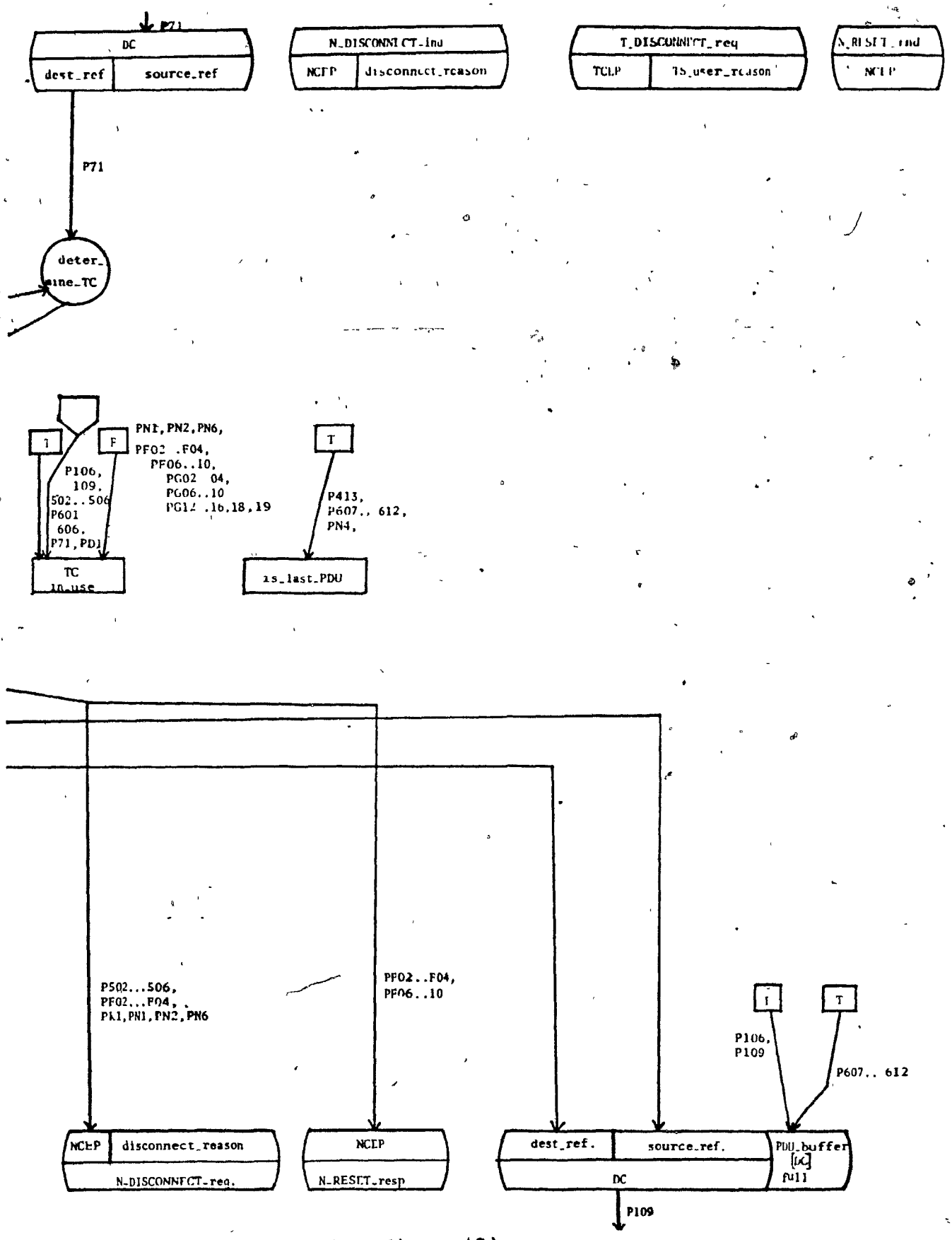


Figure 8.2. ...(continued)... (9)

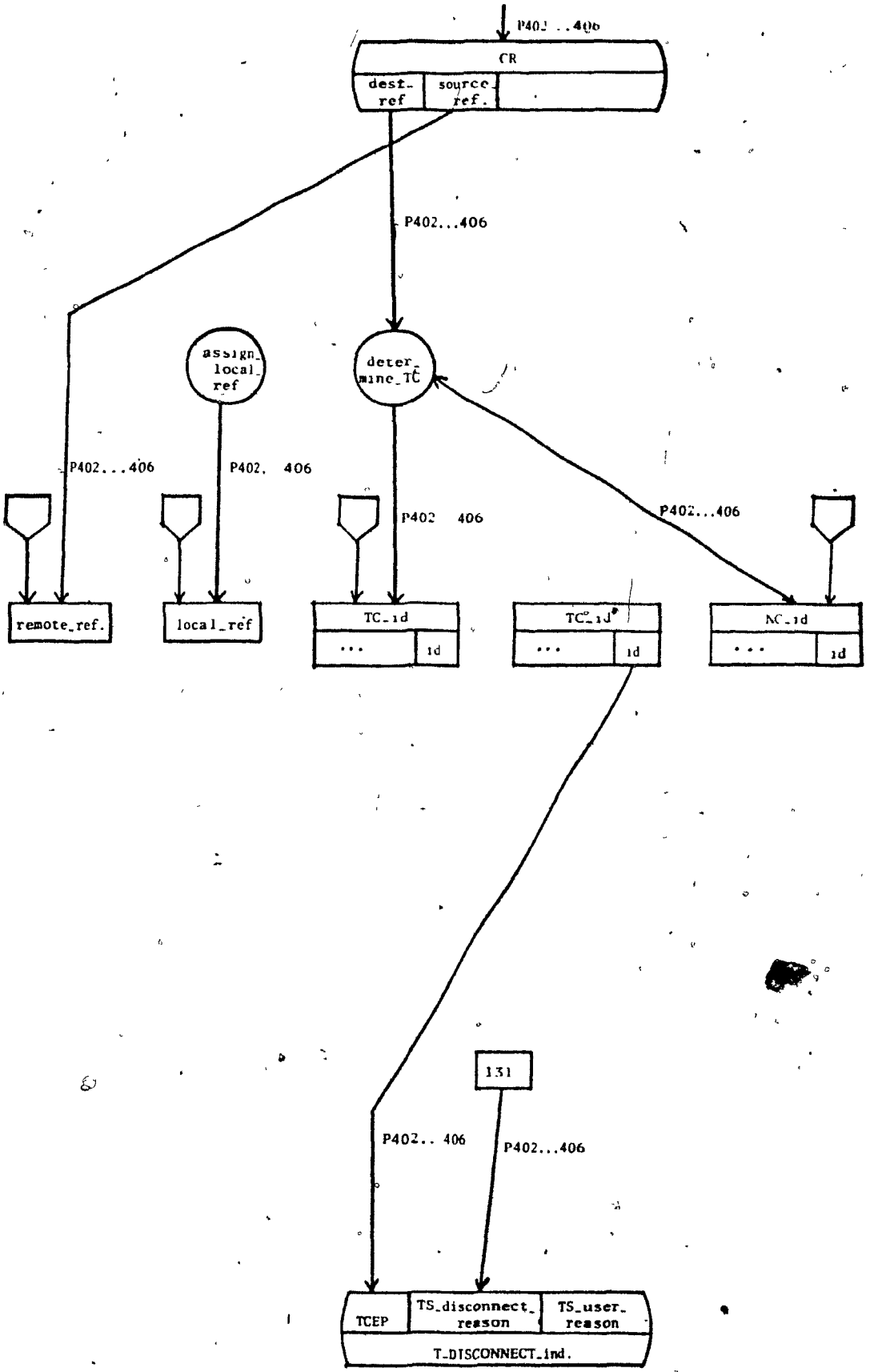


Figure 8.2. ...(continued)... (10)

8.4.A Partition of the Class 2 TP

In order to obtain the initial blocks of Figure 8.2 we apply Algorithm 5.1 in two phases:

Phase 1 treats the variable "received_PDU" and the array "PDU_buffer" (page 1 in Figure 8.2) as D-nodes and 4 blocks are obtained from page 1 of Figure 8.2. These blocks contain the F-nodes "decode" and "encode", the D-node "NSDU_to_be_sent.user_data_present" and the O-node "is_last_fragment" of N_DATA_req, respectively.

Phase 2 considers the rest of the DFG in Figure 8.2, i.e., pages 2 thru 10. Decoded TPDUs become I-nodes and each entry in the array "PDU_buffer" containing TPDUs becomes an O-node. The resulting DFG is similar to the DFG of the Class 0 TP (see Figure 5.2), but more complicated.

In order to obtain the data flow functions of the Class 2 TP we apply the block merging procedure of Section 5.4 to the blocks obtained by Algorithm 5.1. Details of how these blocks (data flow functions) are obtained are given below, considering the blocks in a left-to-right order (the resulting block boundaries are shown with dashed lines in Figure 8.3):

Encoding/ Decoding Block (page 1 in Figure 8.3): Step 3 combines the block containing "is_last_fragment" of N_DATA_req with the block containing the F-node "encode". The block of the internal D-nodes "NSDU_to_be_sent.user_data_present", "is_last_PDU",

"PDU_buffer[CR].full" thru "PDU_buffer[ERR].full" are combined with the resulting block in Step 6. The resulting block is combined with the block containing the F-node "decode" in Step 2 obtaining a single block for the encoding/ decoding data flow function.

Connection Referencing Block (page 2 in Figure 8.3): The block containing "remote_ref" is combined with the block containing "local_ref" in Step 2.

TCEP Block (page 2 in Figure 8.3): To the block containing "TC_id.id" the block of the internal D-node "in_use" is added in Step 6.

User Data Block (page 2 in Figure 8.3): Step 1 combines the blocks containing "user_data" parameters of CR and CC and "data" parameters of T_CONNECT_ind and T_CONNECT_conf, respectively. The resulting two blocks are combined in Step 2, obtaining a single block representing the data exchange during connection establishment.

Addressing Block (Page 3 in Figure 8.3): The block containing "calling_addr" of CR (and some other O-nodes) is combined with the block containing the D-node "local_T_addr" in Step 2. Step 2 also combines the block containing "called_addr" of CR (and some other O-nodes) with the block containing "remote_T_addr". The resulting blocks are combined in Step 2 to obtain a single block for transport addressing.

network Connection Establishment Block (page 3 in Figure 8.3): In Step 5, the blocks representing network addresses

(containing the D-nodes "local_N_addr" and "remote_N_addr") and NCEP (containing "NC_id.id") and network Quality of Service (containing "QNS") are combined obtaining a single block representing network connection establishment. The block of the internal D-node "this_side" is added to the resulting block in Step 6.

QOS block (page 4 in Figure 8.3): The Steps 1 and 2 combine the blocks containing "QTS_ind" parameter of CR and CC and proposed_QTS parameter of T_CONNECT_ind and T_CONNECT_conf. The resulting block is combined with the blocks containing the D-nodes "class", "options" and "max_PDU_size in Step 5. The block of the internal D-node "supports_class_0" is added to the resulting block in Step 6.

Error Block (page 4 in Figure 8.3): The block containing "reject_cause" parameter of ERR does not combine with any other block.

Disconnection Block (page 5 in Figure 8.3): The block containing "disconnect_reason" parameter of DR is combined with "TS_DISCONNECT_reason" of T_DISCONNECT_ind in Step 1.

Peer-to-user Data Transfer Block (page 6 in Figure 8.3): The block containing "receive_buffer" is combined with the block containing "TS_user_data" and "is_last_fragment" parameters of T_DATA_ind in Step 4 obtaining a single block for data transfer from the peer to the user.

User-to-peer Data Transfer Block (page 6 in Figure 8.3): The block containing "send_buffer" is combined with the block containing "user_data" and "end_of_TSDU" parameters of DT in

Step 4.

Flow Control Block (page 6 in Figure 8.3): The blocks containing the D-nodes "R_credit", "TR", "S_credit", and "TS" are combined in Step 5 since they represent credit and sequencing aspects of the flow control.

Expedited Data Transfer Block (page 7 in Figure 8.3): Step 2 combines the blocks containing "user_data" parameter of EDT and "TS_user_data" of T_EX_DATA_ind. The blocks of the internal D-nodes "EX_D_received" and "EX_D_sent", respectively are combined to the resulting block in Step 6.

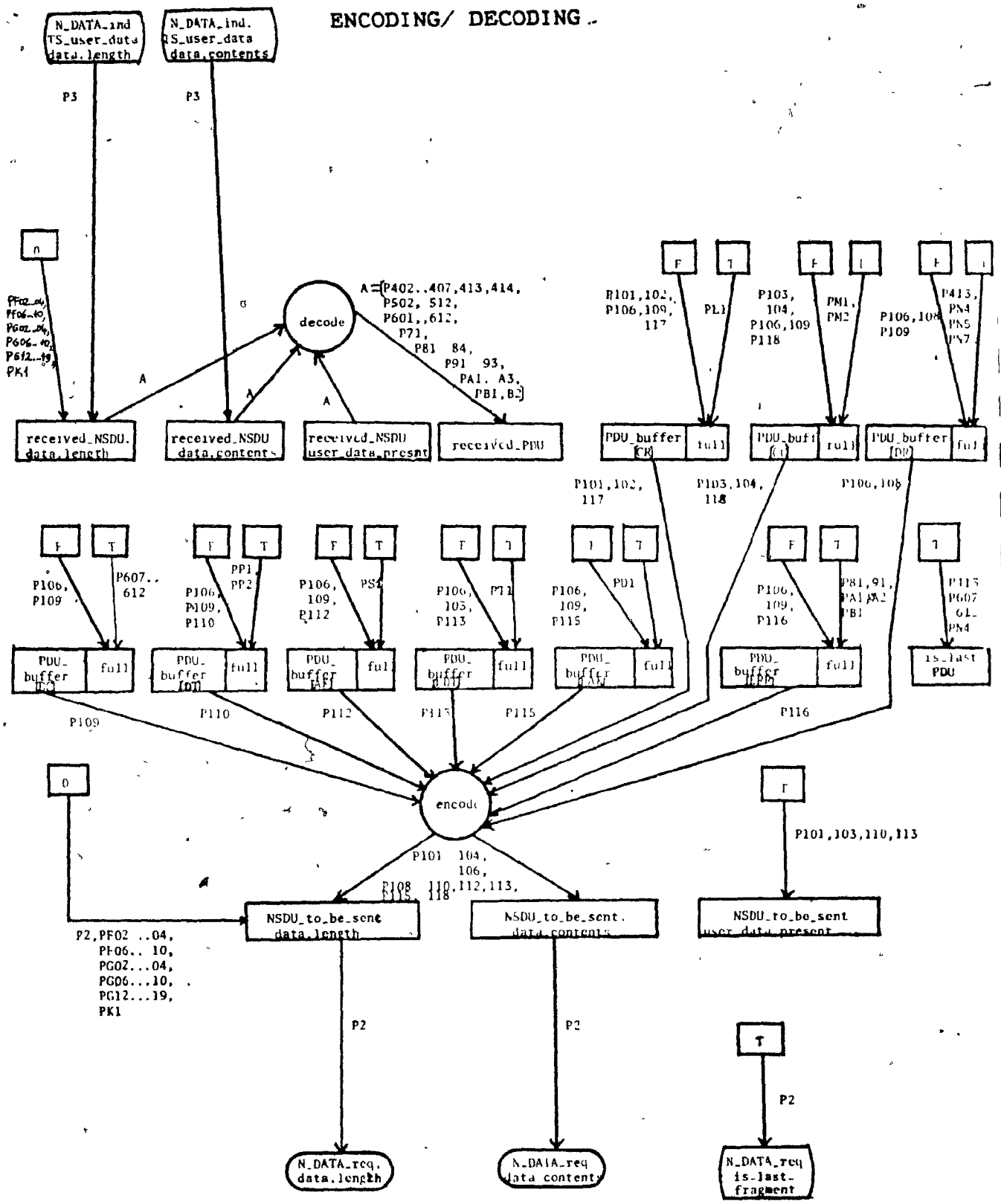


Figure 8.3. A Partition of the Class 2 TP DFG

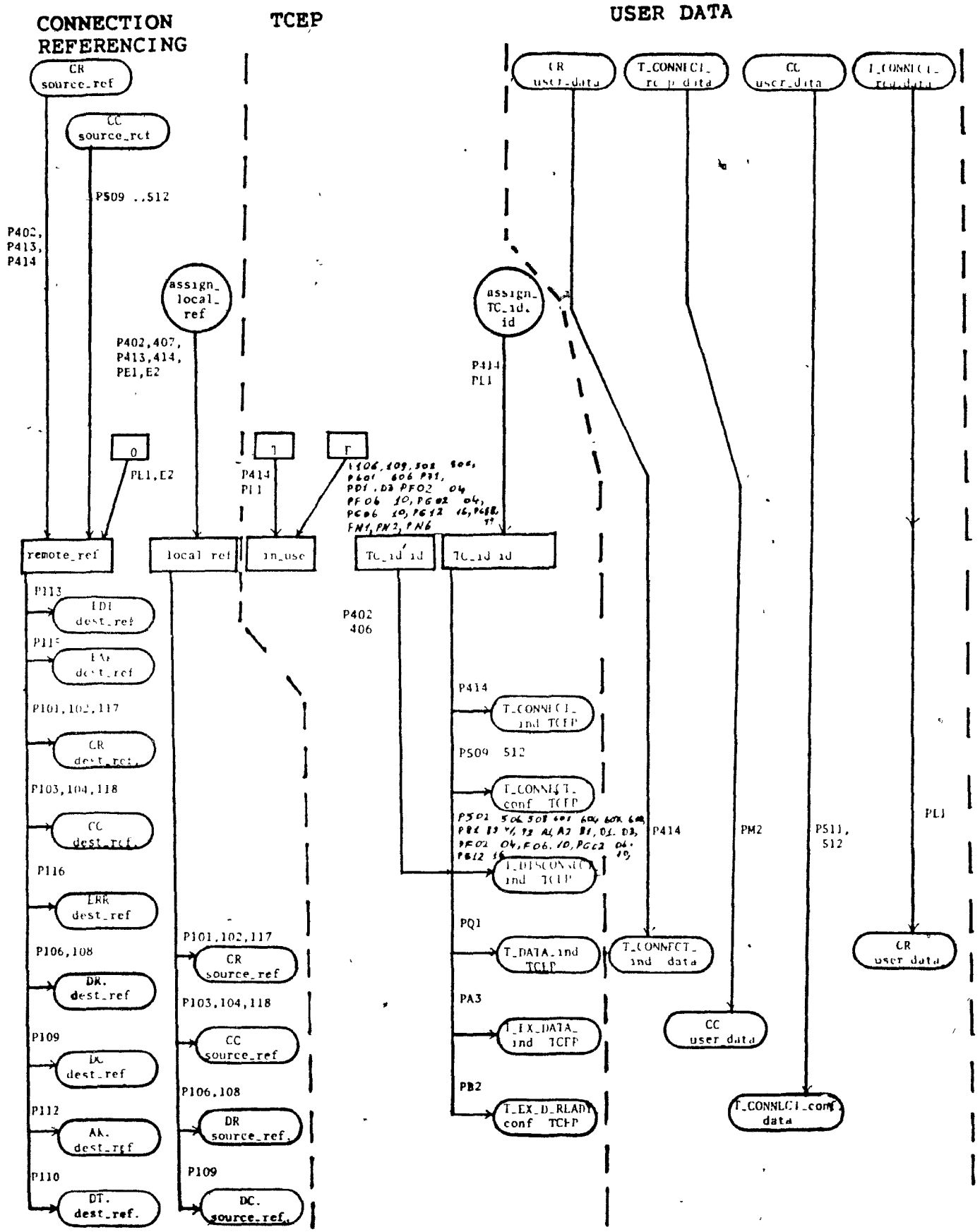


Figure 8.3. ...(continued)... (2)

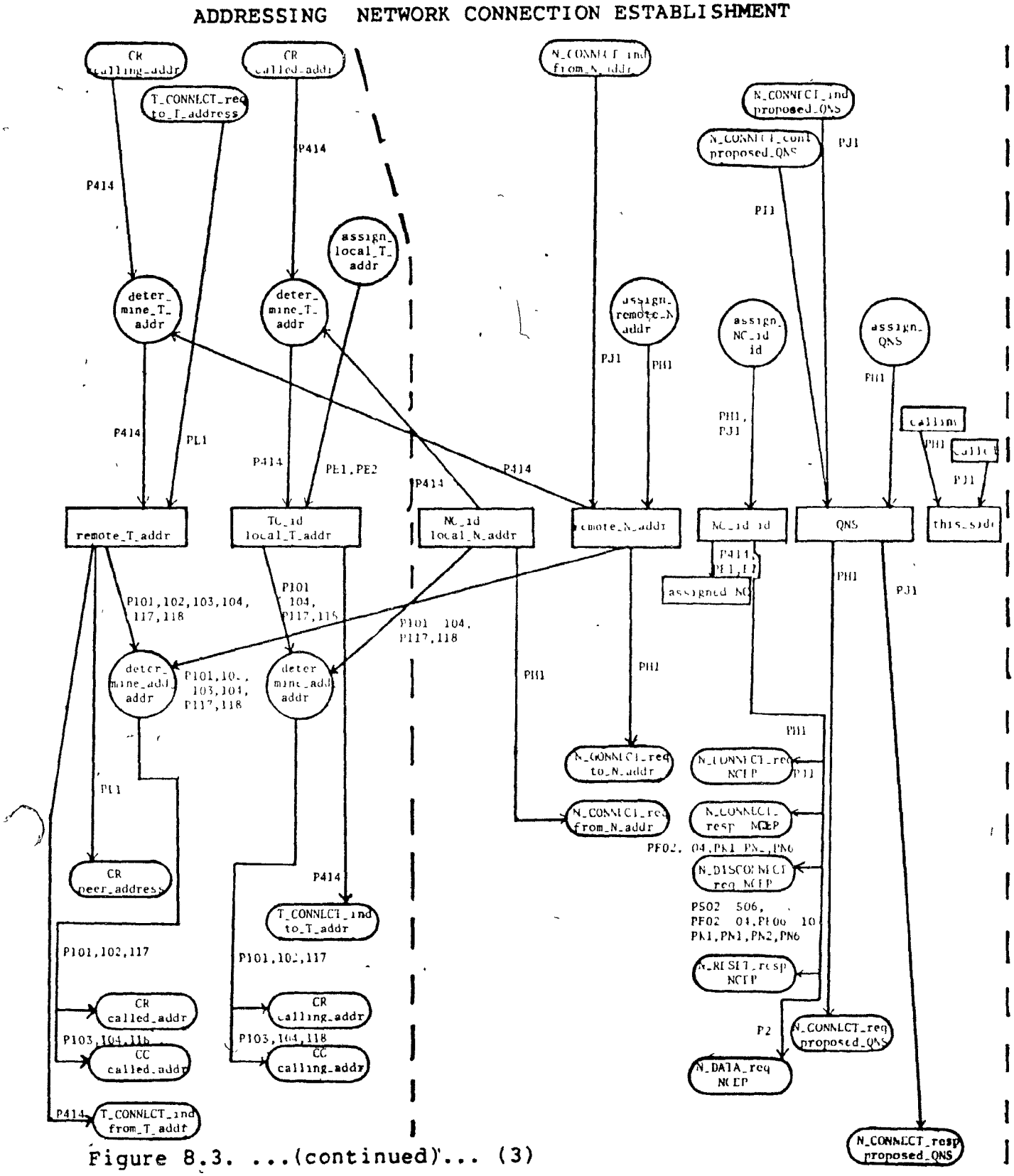


Figure 8.3. ...(continued)... (3)

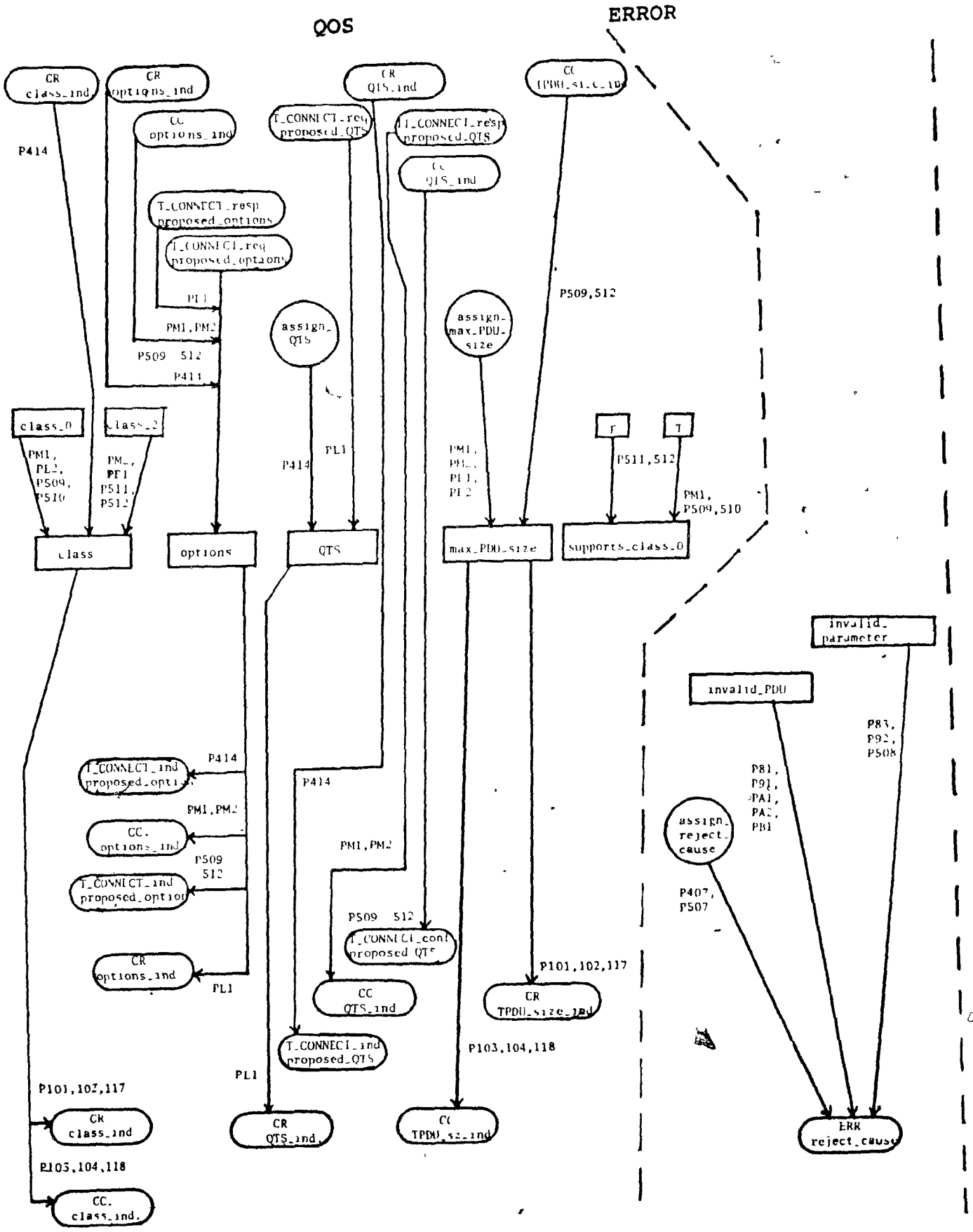


Figure 8.3. ...(continued)... (4)

DISCONNECTION

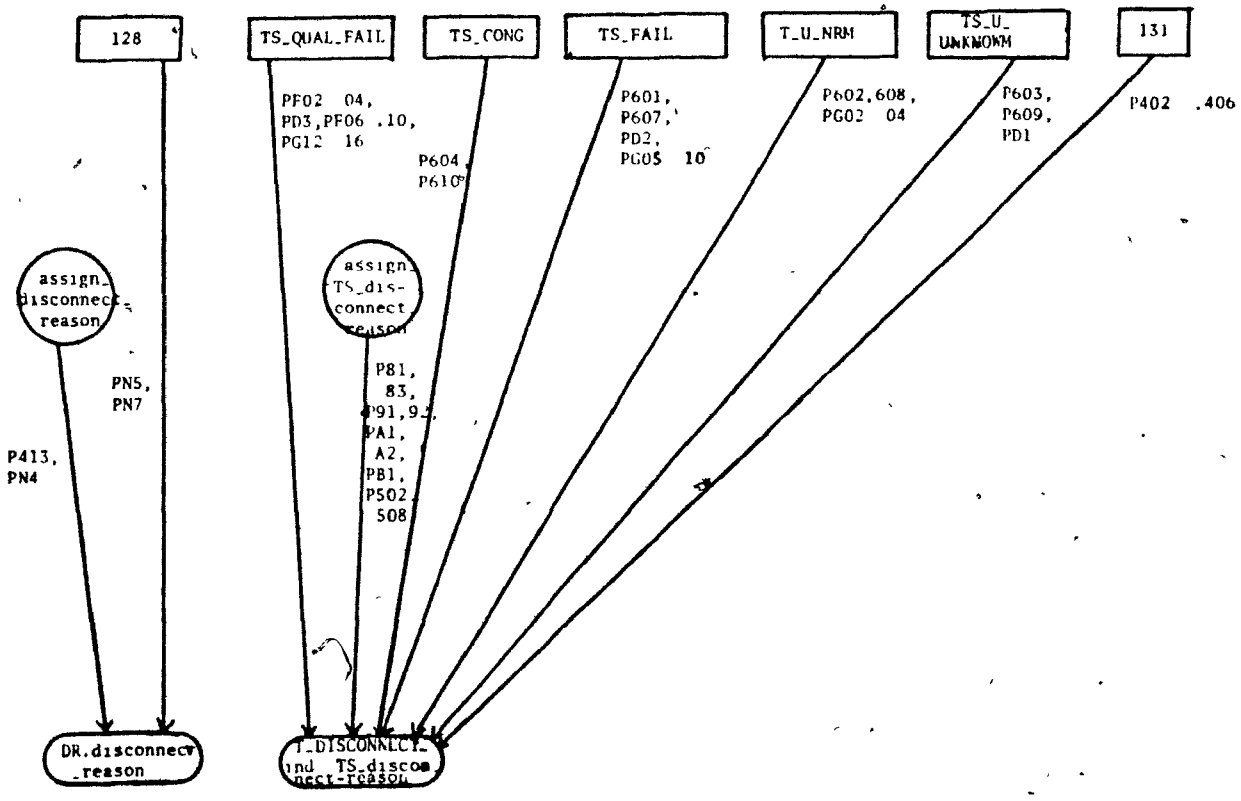


Figure 8.3. ...(continued)... (5)

PEER-TO-USER USER-TO-PEER DATA TRANSFER FLOW CONTROL

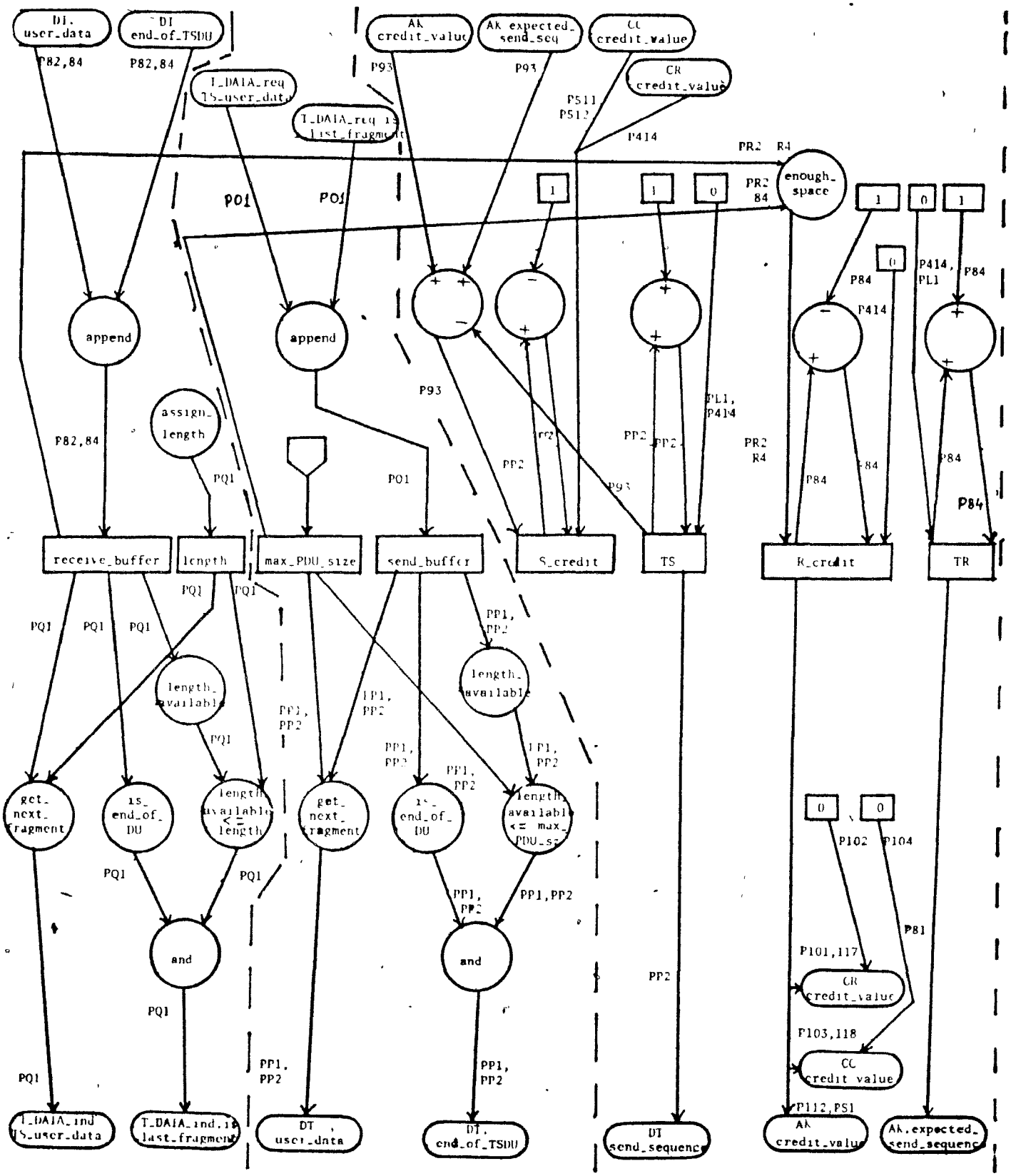


Figure 8.3. ...(continued)... (6)

EXPEDITED DATA TRANSFER

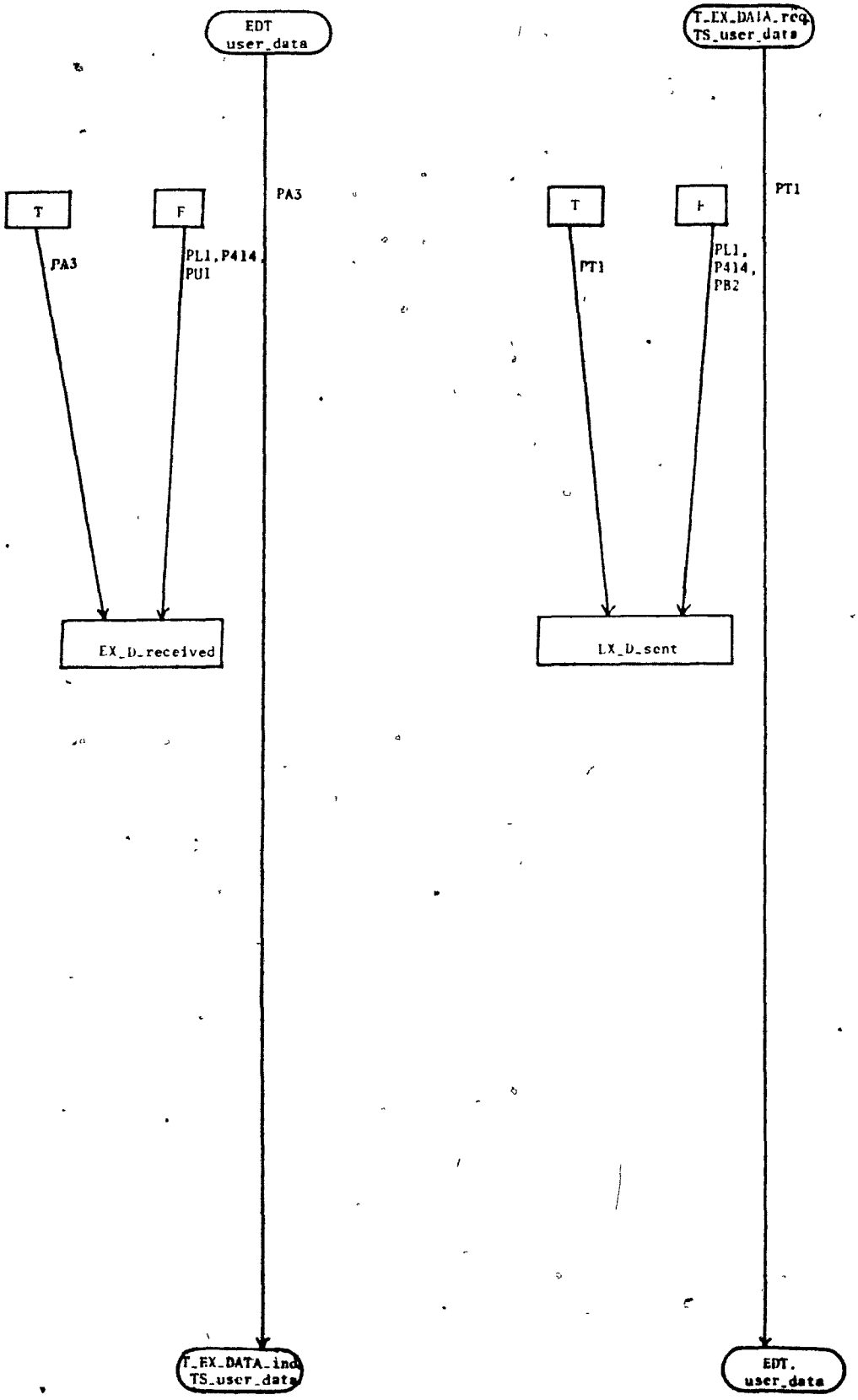


Figure 8.3. ...(continued)... (7)

8.5.Dependencies in the Class 2 TP

The partition in Figure 8.3 gives rise to many data and predicate dependencies. There are 3 dependent blocks in Figure 8.3, namely, addressing, user-to-peer data transfer and flow control blocks. In the addressing block, the D-nodes "remote_T_addr" and "local_T_addr" and the F-nodes "determine_T_addr" and "determine_add_addr" are dependent D- and F-nodes, respectively. These dependencies are caused by the D-nodes "local_N_addr" and "remote_N_addr". The user-to-peer data transfer block has the F-nodes "get_next_fragment" and "length_available" as dependent nodes, all caused by the D-node "max_PDU_size". Finally, the flow control block has the dependent D-node "R_credit" and F-node "enough_space" caused by the D-nodes "max_PDU_size" and "receive_buffer".

The predicate dependencies for the different blocks are listed below. For each given block the D-nodes of other blocks which are used in a predicate of at least one of the normal form transitions of the block are listed.

Encoding/ Decoding: assigned_NC, in_use, class, this_side, local_T_addr, local_N_addr, remote_T_addr, remote_N_addr, max_PDU_size, supports_class_0, receive_buffer, R-credit, TR, TS, S_credit, options, EX_D_recieved.

Connection Referencing: this_side, options, QTS, local_T_addr, local_N_addr, remote_T_addr, remote_N_addr, in_use, max_PDU_size, assigned_NC,

received_NSDU.data.length.

TCEP: received_NSDU.data.length, this_side

User data: received_NSDU.data.length, this_side, in_use, class, local_T_addr, local_N_addr, remote_T_addr, remote_N_addr, max_PDU_size, options.

Addressing: received_NSDU.data.length, in_use, this_side, QTS, options.

Network Connection Establishmeht: PDU_buffer[CR].full.

QOS: in_use, received_NSDU.data.length, in_use, this_side, remote_T_addr, local_T_addr, local_N_addr, remote_N_addr.

Error: received_NSDU.data.legth, in_use, this_side, class, supports_class_0, receive_buffer, max_PDU_size, R_credit, TR, S_credit, options, EX_D_received.

Disconnection: received_NSDU.data.length, in_use, this_side, class, assigned_NC, supports_class_0, receive_buffer, max_PDU_size, R_credit, TR, S_credit, EX_D_received, PDU_buffer[CR].full.

Peer-to-user data transfer: received_NSDU.data.length, in_use, supports_class_0, max_PDU_size, class, R_credit, TR.

User-to-peer data transfer: max_PDU_size, PDU_buffer[DT].full, class, S_credit.

Flow control: receive_buffer, max_PDU_size, received_NSDU.data.length, in_use, supports_class_0, send_buffer, PDU_buffer[DT].full.

Expedited data transfer: in_use, received_NSDU.data.length, options.

The above predicate dependencies show that every data

flow function has a predicate dependency caused by a variable in the encoding/ decoding block. This fact supports the idea of testing encoding/ decoding data flow function before all the other blocks.

The initializations in Figure 8.3 can be listed as follows:

I-node "credit_value" of CR and CC for "S_credit",

Constant D-node 0 for "TR", "TS", and "R_credit",

Constant D-node F(false) for "EX_D_received" and "EX_D_sent".

8.6.Overview of Test Design for the Class 2 TP

In this section we give an overview of the application of the test design methodology of Chapter 6 to the Class 2 TP. The detailed design of the block tests is discussed in the next section.

The first block to consider is the block of (encoding/ decoding) in Figure 8.3 (see Section 8.5). The tests are similar to the Class 0 TP basic tests with the additional tests for concatenation. The concatenation feature can be tested in the data transfer phase by sending NSDUs containing more than one TPDUs (DT with AK and EAK or EDT with EAK and AK, etc.). An inherent restriction in these tests is that the Responder has no way of forcing the IUT to send NSDUs containing concatenated PDUs.

8.6.1.Types of I-nodes

In the following we list the types of the I-nodes of the Class 2 TP, as shown in Figure 8.3. This list is used for determining the input parameter variations as explained in Section 6.2.5.

I-nodes of enumeration type:

credit_value,class,options_ind of CR and CC,

proposed_options of T_CONNECT_req and T_CONNECT_resp,

disconnect_reason of DR,

TS_disconnect_reason of T_DISCONNECT_req,

reject_reason of ERR,

credit_value,expected_send_sequence of AK,

send_sequence of DT.

D-nodes of enumeration type set by F-nodes of Type 3:

R_credit, TR,

S_credit, TS.

Also the array sizes of

TC_id.id and NC_id.id are enumeration types.

Parametric I-nodes are:

from_N_addr of N_CONNECT_ind,

called_addr, calling_addr, peer_addr of CR and CC,

to_T_addr of T_CONNECT_req and T_CONNECT_resp.

Reference value I-nodes:

Source_ref of CR.

Large integer I-nodes:

TPDU_size_ind of CR and CC,

proposed_QTS of T_CONNECT_req and T_CONNECT_resp,

proposed_QNS of N_CONNECT_ind and N_CONNECT_conf.

I-nodes that are related with exchanged data:

user_data of CR and CC,

data of T_CONNECT_req and T_CONNECT_resp,

user_data and end_of_TSDU of DT,

TS_user_data and is_last_fragment of T_DATA_req and N_DATA_ind,

user_data of EDT,

TS_user_data of T_EX_DATA_req.

End Point Identifiers:

NCEP of N_CONNECT_ind,

TCEP of T_CONNECT_req.

8.7.Block Tests

In the tests for each block, parameter variations for each I-node of the block are done and the data flow in each block is considered as discussed in Chapter 6. We briefly discuss the tests for each block of Figure 8.3.

8.7.1.Connection Establishment Tests

Connection establishment tests can be divided into two sets of tests. The first set combines parameter variations for the blocks of "connection referencing", "TCEP", "user_data", "addressing" and "network connection

establishment". Parameter variations of these blocks are done similarly as in the Class 0 tests (see Chapter 7). The "user data" block which does not exist in the Class 0 TP can be tested by varying the I-nodes and verifying the correct data delivery from the O-nodes using a subsequent transport connection over the same network connection.

The second set of connection establishment tests does parameter variations for the "QOS" block. The "class" and "options" parameters of CR, CC, T_CONNECT_req and T_CONNECT_resp are enumerated, and the "QTS" and "PDU_size" parameters of CR, CC, T_CONNECT_req and T_CONNECT_resp are varied (possibly trying boundary and middle values of their domains). The tests in this set are adaptive since the QOS parameters of the implementation must be determined.

Connection establishment tests are done using the subtours numbered 11, 13, 21, and 25 for peer initiated connection establishment and 7 for user initiated connection establishment.

8.7.2.Call Refusal Tests

The disconnection block in Figure 8.3 can be divided into subblocks as discussed in Section 6.4. The subblocks that represent call refusal by the user, the peer and the protocol are tested similarly as in the Class 0 TP. The protocol call refusal tests require that the QOS parameters

of the implementation are determined by the QOS tests.

The tests for call refusal by the user and the peer entity involve enumerations of the "TS_disconnect_reason" of T_DISCONNECT_req and the "disconnect_reason" of DR, respectively.

The subblock of the disconnection block which represents connection freeing can be tested with connection establishment tests. This may be done by simply observing the values assigned to the O-nodes "disconnect_reason" and "TS_disconnect_reason" by the type 2 F-nodes "assign_disconnect_reason" and "assign_TS_disconnect_reason", respectively.

Call refusal tests can be done using the subtours 16, 18, 19, 1 thru 4, and 9.

8.7.3. Expedited Data Transfer Tests

The expedited data transfer block in Figure 8.3 is an independent block and its normal form transitions occur in a self-loop (except for the initializations) of the major state "open". This block is involved in the predicate dependencies from the "TCEP", "QOS" and "encoding/ decoding" blocks. Ignoring the predicate dependency on the encoding/ decoding function, we describe in the following a method for satisfying the expressions on the internal D-nodes. This

method models the normal form transitions of the block as a pure FSM from which test sequences may be obtained, as described in Chapter 2.

The predicates of the expedited data transfer block contain three internal D-nodes "in_use", "EX_D_received", "EX_D_sent". By inspection of Figure 8.3, we see that "in_use" is always "true" in the major state "open", thereby satisfying the expressions containing "in_use" in the normal form transitions of the block. The other internal D-nodes belong to the block, thus they are considered as state variables. Predicates of the block contain expressions on the D-node "options" which is set to a certain value before the major state becomes "open". Once "options" is set to "expedited_data", all the normal form transtions of the block can be modelled as a FSM which defines the order of the normal form transitions. This FSM is shown in Figure 8.4. The transitions in Figure 8.4 are labelled with the labels of the corresponding normal form transitions. The initial state of the machine in Figure 8.4 is

EX_D_received=F, EX_D_sent=F, NC_state=open, state=open

since the initializing normal form transitions P414 and PL1 of the block initialize the internal D-nodes to these values.

Subtours of Figure 8.4, as listed in Table 8.4, give the order of the normal form transitions. Parameter variations of the I-nodes and encoding/ decoding should be

considered for obtaining complete tests for the expedited data transfer block.

Expedited data tests can be done using any of the subtours numbered 11, 13, 21, or 25.

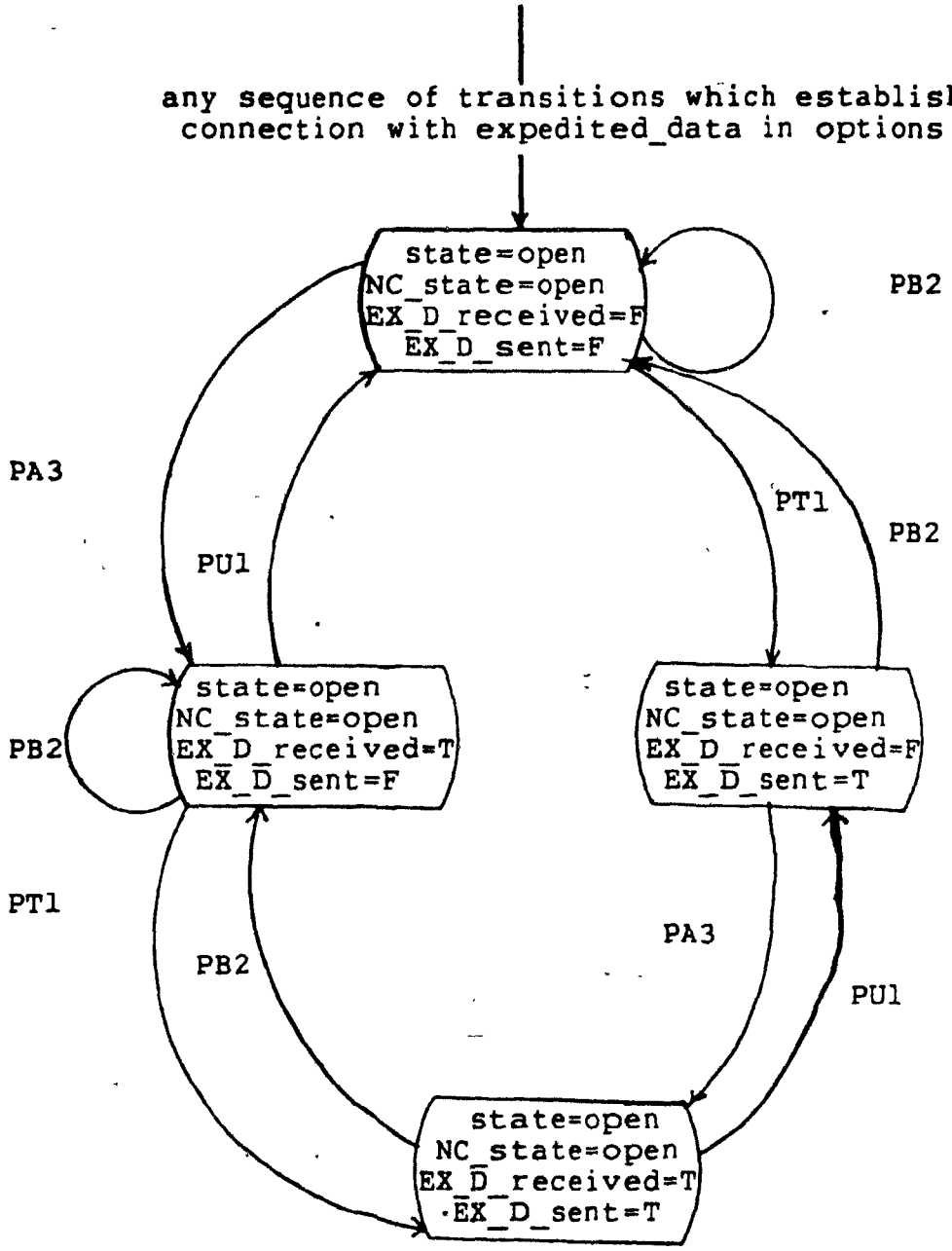


Figure 8.4.A State Diagram for Expedited Data Transfer

```
          *
PA3 PB2 PU1
          *       *
PA3 PB2 PT1 PB2 PU1

PA3 PT1 PU1 PB2
                  *
PA3 PT1 PU1 PA3 PB2 PU1

PT1 PB2

PT1 PA3 PU1 PB2
              *
PT1 PA3 PB2 PU1
              *           *
PT1 PA3 PB2 PT1 PU1 PB2
```

Table 8.4. Test Sequences for Expedited Data Transfer

8.7.4. Data Transfer Tests

In this section we discuss the data transfer tests for the Class 2 TP. Data transfer contains three data flow functions, the user-to-peer and peer-to-user data transfer, and the flow control blocks of Figure 8.3. These blocks contain various F-nodes of all types. Test design considerations related with F-nodes were discussed in Section 6.4. Some considerations on arithmetic type 3 F-nodes are given below.

The parameter variations of the above three blocks of normal data transfer can be combined since their normal form transitions occur in the same self-loop.

Since "R-credit" in the flow control block is internally assigned, there is no direct way of enumerating

its values, but the values decided by the implementation can be observed in AK PDUs. Therefore a test sequence can be selected so that the number of unacknowledged data packets sent by the Tester is equal to the received credit value. At the same time, variations of the I-nodes of the peer-to-user data transfer block can be done, also testing the type 3 F-node which decrements R_credit. R_credit is a dependent D-node (see Section 8.5), i.e., its value depends on the negotiated max_PDU_size and the receive_buffer. Thus it is important to repeat the tests for peer-to-user data transfer for a number of different max_PDU_size values which are tested in the QOS tests.

"TR" in the flow control block is also internally assigned and the value received in AK PDU is used as the sequence number of the next DT to be sent by the Tester.

Verification of correct data delivery from the peer to the user (and also from user to peer) can be done with a mechanism similar to the one used in the data transfer tests for the Class 0 TP. This will also be the test for the F-nodes called "append" and "get_next_fragment". The latter takes the length parameter from the user interface function TS.user_ready, as discussed in Section 8.1.

The D-node "S-credit" of the flow control block can be enumerated (probably not exhaustively) since it can be initialized by either "credit_value" of CR or CC (depending on the initiating side of the transport connection) and

modified later in the data transfer phase by the "credit_value" of AK. After it is set to a certain value, the Responder part can create data flow in order to test the correct handling of S_credit, i.e., the F-node of type 3 assigning S-credit.

"TS" of the flow control block can easily be observed from the data, sent by the Responder, which is supposed to be delivered to the Tester in DT PDUs.

The F-nodes "get_next_fragment" and "length_available" in the block of user-to-peer data transfer are dependent F-nodes. The value of the negotiated max_PDU_size determines the maximum length of the DT PDUs to be sent, thus data transfer tests should be repeated in consecutive connections with different max_PDU_size values.

The F-node "is_end_of_TSDU" of the two data transfer blocks is tested by varying the parameters of DT.end_of_TSDU and T_DATA_req.is_last_fragment, respectively.

Since the normal form transitions of the data transfer blocks occur in the same subtour as the transitions of the expedited data transfer block, they should also be tested simultaneously.

Data transfer blocks can be tested with one of the subtours 11, 13, 21, or 25.

8.7.5.Tests for Error Cases

The last uncovered block, called error block, can easily be tested by the normal form transitions which assign the O-node "reject_cause" of ERR. The error cases are divided into two parts, the errors during connection establishment and the errors during data transfer. These two parts can be tested in two separate tests.

The error block is tested using the subtours 6, 8, 14, 15, 17, 22, and 23.

8.8.Multiple Connection Tests

Tests for enumerating array sizes for the Class 2 TP should be different from the multiple connection tests of the Class 0 TP because of the existence of two D-nodes for array sizes.

Exhaustive enumerations of these two variables should be done in both of the following ways:

i) By fixing "NC_id.id", tests are designed to enumerate "TC_id.id". The number of tests are determined from the number of subtours of the block containing "TC_id.id".

These tests are performed to determine the number of transport connections that can be multiplexed over a single network connection. Also, data transfer blocks should be tested, in order to observe possible reductions in the

number of credits given to the parallel transport connections.

The tests in this group should test duplicate connection cases as specified in the specification. The corresponding data flow is found from one of the subblocks of the disconnection block (see Figure 8.3).

ii) "NC_id.id" is enumerated simultaneously as "TC_id.id" is enumerated. These tests are done to determine the number of network connections that the implementation supports under multiple transport connections. The data transfer block, should also be tested under multiple network and transport connections.

Note that the third enumeration possibility which establishes multiple network connections with a single transport connection each, corresponds to the multiple connection tests for the Class 0 TP, described in Chapter 7.

Multiple connection tests can be done using the subtours of the connection establishment tests of Section 8.7.1.

8.9. Some Observations on the Class 2 TP Test Design

The test design discussed above gives rise to the following observations:

(a) From the Control Graph of a complex protocol, a considerably large number of subtours may be obtained. Considering only the subtours that represent single control functions, i.e., discarding those that are concatenations of more than one control function, it is possible to decrease the number of tests and yet cover all the normal form transitions.

(b) If we assume that any Class 2 TP implementation would also support the Class 0 TP, it is possible to apply the Class 0 tests (see Chapter 7) to the Class 2 TP implementations (slight modifications in the user interface may be necessary). This further decreases the number of additional subtours necessary. For example, the subtours 5, 10, 12, 20, and 24 can be covered by the Class 0 tests.

(c) The form in which the encoding/ decoding of PDUs is described in the specification makes it difficult to apply Algorithm 5.1 to the DFG since only a single large block would be generated. The problem can be solved by considering some of the D-nodes as I- or O-nodes, as discussed in Section 8.3. The test designer can easily identify these nodes by determining the D-nodes that represent the PDUs in the DFG.

9.Conclusions

We have developed a methodology for designing tests of communication protocols. The methodology is based on decomposing the protocol into its control and data flow functions. These functions can be derived from a formal specification of the protocol which is assumed to be available. In the following paragraphs we summarize the results of each chapter.

In **Chapter 1,** the problem of protocol testing is introduced. The work in several related areas is surveyed including an introduction to the test architecture proposed for the testing of protocol implementations for conformance to the standard natural language specifications. We have adapted ideas from various fields: finite state machine and microprocessor test techniques, control system verification, symbolic execution and specification based software validation.

Chapters 2 and 3 apply finite state machine test techniques to the protocols to obtain test sequences. The three major approaches (transition tours, characterization and checking sequences) are used to generate test sequences for protocols modelled as incompletely specified finite state machines. A synchronization problem which may arise in the application of some of these test sequences with the architecture of Chapter 1 is defined. Avoiding the synchronization problem may be impossible for certain

protocols which are defined in such a way that they have intrinsic synchronization problems. In **Chapter 3**, we modify the basic algorithms that generate tests sequences for the transition tour and characterization sequences to obtain only synchronizable sequences.

In order to account for parameters of the protocol primitives and the effects of the protocol variables other than the major state variable, a different test design approach is proposed. **Chapter 4** introduces the first step of the test design: normal form transitions. From the formal specification of the protocol, a simpler specification is obtained by applying syntactic transformations based on symbolic execution. Modules of the protocol are combined by removing any inter module communication. The resulting transitions are called normal form transitions.

Chapter 5 models various aspects of normal form transitions by graphs: the changes in the major state variable are modelled by a control graph, and the other aspects by a data flow graph. The transition tours of the control graph are divided into several subtours, each representing a sequence of control functions of the protocol. An algorithm to partition the data flow graph into disjoint blocks is developed. The resulting blocks are partly combined by a merging procedure which requires interaction with the test designer. The blocks obtained by

the merging procedure are defined to be the data flow functions of the protocol. The flow graphs (control and data flow) are used to validate the protocol design. Syntactic and semantic errors in a specification can also be detected using these graphs.

Using the control and data flow functions, **Chapter 6** develops a test design methodology. The various steps of the test design are identified, and test sequence selection guidelines are detailed for each step. Fault models for functions of the protocol that are not formally specified can be used for test sequence selection. Tests for data flow functions are based on parameter variations of the input primitives of the block. The structure of the block, its data and predicate dependencies, the types of the input parameters and the subtours have to be considered in the tests. The test design for multiple connection and unexpected stimulations are also based on the flow graphs.

Chapters 7 and 8 apply the test design methodology to the Transport protocols Class 0 and Class 2, respectively.

9.1.Future Work

None of the steps of the test design methodology presented in Chapters 4, 5, and 6 was implemented. Applications to the two protocols were developed manually. However, it is desirable to generate tests for protocols

automatically.

We divide the future work into three categories: implementations, theoretical investigations, and applications. Considering first implementation, different steps of the test design methodology developed in this thesis could be automated:

- To obtain normal form transitions from any Subgroup B FDT specification.
- To obtain the flow graphs automatically from the normal form transitions. Sometimes, user interaction may be necessary, as in the case of merging the blocks.
- A graphics package could be developed for displaying and manipulating the data flow graph.
- After obtaining the data and predicate dependencies, test sequences could be automatically derived. Test sequences could be expressed using a test specification language (Subgroup B FDT, for example).
- Obtaining test programs (one for the Tester and one for the Responder) from the test specification will be the last step of automatic test generation for protocols.

Theoretical investigations would be useful in the following areas, which are not covered in this thesis:

- Complexity analysis of the method, in particular the complexities of the algorithms given in Chapter 4 and 5, i.e., the algorithm that partitions a DFG, and the algorithm that combines the modules.

- Further work on the use of the flow graphs in protocol design validation.
- An expert system which is capable of deducing the considerations introduced in Chapter 6 from the control and data flow graphs would be a breakthrough for the automation of the protocol test design process.

Concerning applications of the test design methodology developed in this thesis, we may distinguish the following areas:

- Straightforward application of the methodology, for instance to other classes of the ISO/ CCITT transport protocol, or protocols of link, network and session layers.
- Application in areas where the methodology may have to be adjusted in order to take into account certain characteristics of the system specification which were not encountered in the areas of application considered in this thesis. For instance, the testing of application layer protocols may require a methodology with certain modification. It could also be investigated whether the developed methodology is applicable in other areas of software development, including the testing of executable specifications.

REFERENCES

[Ansart 81] J.P. Ansart,"Test and Certification of Standardized Protocols", in Protocol Testing- Towards Proof? D.Rayner and R.W.S. Hale (eds.), Vol.2, NPL, 1981, pp.119-126.

[Ansart 81b] J.P. Ansart,"Tools for the Certification of Standardized Protocols: Cerbere and Genepi", in Protocol Testing- Towards Proof? D.Rayner and R.W.S. Hale (eds.), Vol.2, NPL, 1981, pp.127-130.

[BoCeMaSa 83] G.v. Bochmann, E. Cerny, M. Maksud, B. Sarikaya, "Testing Transport Protocol Implementations", Proc. of CIPS'83, May 1983.

[BoCeLa 81] G.v. Bochmann, E. Cerny, C. Lacaille, "Formal Specification of a Transport Service", document WASH 9 of ISO/TC97/SC16/WG1 ad hoc group on FDT, September 1981.

[Bochmann 83] G.v. Bochmann,"Concepts for Distributed System Design", Springer-Verlag, 1983.

[CeBo 83] E. Cerny, G.v. Bochmann,"An Experimental Protocol Implementation Testing System", Report prepared for DOC of Canada, 1983.

[CeBoMaLeSeSa 84] E. Cerny, G.v. Bochmann, M. Maksud, A. Leveille, J.M. Serre, B. Sarikaya, "Experiments in Testing Communication Protocol Implementations", Proc. of 14.th Symp. on Fault Tolerant Comp., Orlando, June 1984.

[ChDa 83] A. Champeville, K. Daher, "Systeme de Generation Automatique de Sequences de Test", MSc. Thesis, Univ. of Bordeaux I, June 1983.

[ChLeLeRi 81] H. Chaigne, M. Leport, M. Lety, O. Ridoux, "Un Generateur de Test pour Systemes Modelises par Automates d'Etats Finis", BIGRE of IRISA, pp.19-24, December 1981.

[Chow 78] T.S. Chow, "Testing Designs Modelled by Finite-State Machines", IEEE Trans. on SE, 4-3, 1978.

[ClRi 81] L.A. Clarke, D.J. Richardson, "Symbolic Evaluation Methods for Program Analysis", in Program Flow Analysis by S.S. Muchnick, N.D. Jones(Eds.), Prentice Hall, 1981.

[Even 79] S. Even, "Graph Algorithms", Computer Science Press, 1979.

[FDT 84] "A FDT based on an Extended State Transition Model", Subgroup B of ISO TC 97/SC16/WG1 ad hoc group on FDT, March 1984.

[Gerber 83] G. Gerber, "Une Methode d'Implantation Automatisee de Systemes Specifies Formellement", M.Sc. Thesis, Univ. de Montreal, 1983.

[Gill 62] A. Gill, "Introduction to the Theory of Finite State Machines", McGraw Hill, 1962.

[Gonenc 70] G. Gonenc, "A Method for the Design of Fault

Detection Experiments", IEEE Trans. on Computers, 19-6, 1970.

[Guitton 84] P. Guitton, "Description, Validation et Test de Conformite de Protocole de Communication", PhD Thesis (3eme Cycle), Univ. of Bordeaux I, January 1984.

[Henley 81] RFL Henley(Ed.), "Implementation Assessment of Transport and Network Services: The Test Responder Specification", NPL Report DNACS 46/81, July 1981.

[HeRa 81] RFL Henley, D. Rayner, "Implementation Assessment of Transport and Network Services: An Informal Description of Tests for Public Comment", NPL Report DNACS TM 5/81, July 1981.

[Howden 80] W.E. Howden, "Functional Program Testing", IEEE Trans. on SE, Vol. SE-6, No.2, March 1980.

[INWG 83] Several papers in the proceedings of 3.Int. Workshop on Protocol Spec., Testing and Verification, North-Holland, 1983.

[ISO 82] G.v. Bochmann, "Examples of Transport Protocol Specifications", ISO TC97/SC16/WG1 Subgroup B, April 1982.

[ISO 82b] G.v. Bochmann, "Example of a Transport Protocol Specification", ISO TC97/SC16/WG1 Subgroup B, Nov. 1982 (Doc. de Travail No.146, Univ. of Montreal).

[JaBo 83] C. Jard, G.v. Bochmann, "An Approach to Testing Specifications", Proc. of the ACM SIGSOFT/SIGPLAN Software Eng. Symposium on High-Level Debugging, March 1983, pp.53-59.

[King 76] J.C. King, "Symbolic Execution and Program Testing", CACM Vol.10, No.4, 1976.

[Kohavi 78] Z. Kohavi, "Switching and Finite Automata Theory", McGraw Hill, 1978.

[Kowalski 79] R. Kowalski, "Logic for Problem Solving", Elsevier North Holland, New York, 1979.

[Leveille 84] A. Leveille, "Implantation et Certification d'un Protocole de Transport", M.Sc. Thesis, Univ. of Montreal, 1984. (to appear)

[LiMc 83] R.J. Linn, W.H. McCoy, "Producing Tests for Implementations of OSI Protocols", in INWG 83.

[LiNa 83] R.J. Linn, S.J. Nightingale, "Some Experience with Testing Tools for OSI Protocol Implementations", in INWG 83.

[Maksud 83] M. Maksud, "Un Systeme de Test de Protocoles de Communications", M.Sc. Thesis, Univ. de Montreal, 1983.

[NaTs 81] S. Naito, M. Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours", Proc. of FTCS, pp.238-243, 1981.

[Piatkowski 80] T.F. Piatkowski, "On the Feasibility of Validating and Testing ADCCP Implementations", NBS Trends and Applications, May 1980.

[Prather 83] R.E. Prather, "Theory of Program Testing - An Overview", The Bell System Technical Journal, Vol. 62, No. 10, pp. 3073-3105, December 1983.

[PrSkUr 83] R.L. Probert, D.R. Skuce, H. Ural, "Specification of Representative Test Cases Using Logic Programming", Proc. of 16th Hawaii Int. Conf. on System Sciences, 190-196, 1983.

[PrUr 83] R.L. Probert, H. Ural, "High Level Testing and Example-Directed Development of Software", SRRG Report, Dept. of CS, Univ. of Ottawa, 1983.

[Rayner 81] D. Rayner(Ed.), "Protocol Implementation Assessment: Philosophy and Architecture", NPL Report DNACS 44/81, April 1981.

[Rayner 82] D. Rayner, "A System for Testing Protocol Implementations", Computer Networks 6,6, Dec. 1982.

[SaBo 82] B. Sarikaya, G.v. Bochmann, "Some Experience with Test Sequence Generation for Protocols", Proc. 2nd Int. Workshop on Protocol Specification, Testing and Verification, North-Holland, 1982, pp. 555-567.

[SaBo 83] B. Sarikaya, G.v. Bochmann, "Synchronization Issues in Protocol Testing", Proc. of SIGCOMM' 83, March 1983, pp. 121-128.

[SaBo 84] B. Sarikaya, G.v. Bochmann, "Synchronization and Specification Issues in Protocol Testing", To appear in IEEE Trans. on Communications.

[Sarikaya 84] B. Sarikaya, "Normal Form Transitions of Class 2 Transport Protocol", Doc. de Travail, Computer Science Dept., Univ. of Montreal, 1984.

[Sarikaya 84b] B. Sarikaya, "Manuel d'Utilisation du Logiciel Servant a Generer des Sequences de Tests", Doc. de Travail, Computer Science Dept., Univ. of Montreal, 1984.

[Serre 84] J.M. Serre, "Une Implantation de Protocole de Transport Classes 0 et 2", M.Sc. Thesis, Univ. of Montreal, 1984. (to appear)

[Tarjan 72] R. Tarjan, "Depth-First Search and Linear Graph Algorithms", SIAM J. Computing, Vol.1, No.2, 1972.

[ThAb 78] S.M. Thatte, J.A. Abraham, "A Methodology for Functional Level Testing of Microprocessors", Proc. 8th FTCS pp.90-95, 1978.

[ThAb 79] S.M. Thatte, J.A. Abraham, "Test Generation for General Microprocessor Architectures", Proc. 9th FTCS, pp.203-210, 1979.

[ThAb 79b] S.M. Thatte, J.A. Abraham, "User Testing of Microprocessors", Proc. of COMPCON Spring 1979, pp.108-114.

[VaDi 78] R. Valette, M. Diaz, "Top-Down Formal Specification and Verification of Parallel Control Systems", Digital Processes, 4, 181-199, 1978.

[UrPr 83] H. Ural, R.L. Probert, "User-guided Test Sequence Generation", in INWG 83.

[Zimmermann 80] H. Zimmermann, "OSI Reference Model- The ISO Model of Architecture for Open Systems Interconnection", IEEE Trans. on Communication, Vol. Com-28, No.4, April 1980.

APPENDIX A

EXTRACT FROM CLASS 2 TP SPECIFICATION

```
const
            ... (*see [ISO 82b]*)

type
            ... (*see [ISO 82b]*)
    TPDU_and_control_information = record
    ...
    CR(credit_value:credit_type;
        dest_ref,source_ref:reference_type;
        user_data:string_of_octets;
        peer_address:T_address_type;
        QTS_ind:quality_of_TS_type;
        class_ind:class_type;
        options_ind:option_type;
        calling_addr,called_addr:
              additional_address_information;
        TPDU_size_ind:PDU_size_type) ;
    ...
    end;

channel PDU_and_control(protocol,mapping);
    by protocol,mapping:
        forward(PDU:TPDU_and_control_information);
        ...

end PDU_and_control;

module Mapping(ATP:array[TC_id_type] of
              PDU_and_control(mapping);
    NS:array[NC_id_type] of NCEP_primitives(user));

        ...

function implied_PDU_length(size: optional PDU_size_type):
    PDU_size_type;
begin if size = undefined
    then implied_PDU_length := 128
    else implied_PDU_length := size end;

procedure close_and_clear_buffers (TC_id : TC_id_type);
begin with TC[TC_id] do begin
    in_use := false;
    ATP [TC_id].idle;
    for kind := CR to ERR do PDU_buffer [kind]. full := false;
    end end;
```

```
when ATP[TCEP_id].forward
begin with TC[TCEP_id] do begin
PDU_buffer [PDU.kind].full := true;
PDU_buffer [PDU.kind].PDU := PDU;
with PDU_buffer [PDU.kind].PDU do begin
    case kind of
    ...
    DR, DC, DT, EDT, EAK, ERR:;
    end;
end end;

when ATP[TCEP_id].implicit_termination
begin with TC[TCEP_id] do begin
    if assigned_NC = undefined
    then (*wait_for_NC state; no action*)
    else NS [assigned_NC].N_DISCONNECT_req;
    close_and_clear_buffers(TCEP_id);
    end; end;

any NC_id : NC_id_type do with NC [NC_id] do
provided received_NSDU.data.length <> 0
    and not ( (/PDU_kind(received_NSDU.data) /) = DT
    and supports_class_0 and ATP[corresponding_TC_id].
    ready_for_receiving)
var received_PDU : TPDU_type;
    TC_id : TC_id_type;
function determine_TC(NC_id : NC_id_type; ref : reference_type):
            TC_id_type;
    begin (/ determine_TC(NC_id, ref) =
        if exists TC_id such that with TC[TC_id] holds
            in_use and assigned_NC = NC_id and local_ref = ref
        then TC_id
        else TC_id' such that not TC[TC_id].in_use;
     i.e., find the TC associated with the reference
     "ref" over the NC
        or assign some TC_id not inuse;
        if "ref" = 0 then such a new TC is assigned. /)
begin
    (/decode(received_NSDU, received_PDU/)
    with received_PDU do begin
        TC_id:=determine_TC(NC_id,dest_ref);
        with TC[TC_id] do
        case kind of
        CR:if not in_use then begin
            remote_ref:=source_ref;
            local_ref:=...;
            if dest_ref <> 0
            then ... (*error*)
            else if (/exists TC_id' <> TC-id
              such that with TC[TC_id'] holds
               in_use and assigned_NC = NC_id
               and remote_ref = source_ref;
               i.e., this is a duplicated CR/)
             then ATP[TC_id'].close_indication(131)
```

```
            else if determine_PDU_length(received_PDU) >
              implied_PDU_length(TPDU_size_ind)
              or class_ind = class0 and this_side=calling
              then ATP[TC_id].error_indication(...)
              else if (/not able to provide service
                or destination address unknown/)
             then begin
              PDU_buffer[DR].full := true;
              with PDU_buffer[DR].PDU do begin
               kind:=DR;
               disconnect_reason:=...;
               is_last_PDU:=true;
               end;
              end;
            else begin
            TC_id.T_addr:=determine_T_addr(
              NC_id.local_N_addr,called_addr);
            TC_id.id:=...;
            remote_T_addr:=determine_T_addr(
              NC[NC_id].remote_N_addr,calling_addr);
            received_PDU.peer_address:=remote_T_addr;
            QTS:=...;
            received_PDU.QTS_ind:=QTS;
            remote_ref:=source_ref;
            assigned_NC:=NC_id;
            ATP[TC_id].forward(received_PDU);
            end;

    ...

module ATP_type(TS:TCEP_primitives(provider);
        Map:PDU_and_control_primitives(protocol));
    ...
(*definition of interface predicates*)
Map.ready_for_receiving :=
enough_space(receive_buffer, max_PDU_size -
        (/ DT header length /) );

when Map.forward (PDU) provided PDU.kind = CR
from closed to open_in_progress_called
begin
    in_use := true;
    options := option_ind;
    TR := 0;
    TS := 0;
    S_credit := credit_value;
    EX_D_sent := false;
    EX_D_received := false;
    TS.T_CONNECT_ind (local_T_addr, PDU.peer_address,
            options, PDU.QTS_ind, PDU.user_data);
    end;

when TS.T_DISCONNECT_req
from open_in_progress_called
```

```
    to closed
begin
    with PDU do begin
        kind:=DR;
        is_last_PDU := true;
        if class = class0 then
          disconnect_reason := (/1 or 2/);
        else
          disconnect_reason:=128;
    end;
    Map.forward(PDU);
end;
```

APPENDIX B

NORMAL FORM TRANSITIONS FOR THE TRANSITIONS DEFINED IN APPENDIX A

```
P401:any NC_id : NC_id_type, TC_id : TC_id_type;
provided NC[NC_id].received_NSDU.data.length <> 0
  and NC[NC_id].received_PDU.kind = CR
  and not TC[TC_id].in_use and ATP[TC_id].state = closed
  and NC[NC_id].received_PDU.dest_ref <> 0
begin
  (/decode(NC[NC_id].received_NSDU,NC[NC_id].received_PDU)/);
  TC_id:=determine_TC(NC_id,NC[NC_id].received_PDU.dest_ref);
  TC[TC_id].remote_ref:=NC[NC_id].received_PDU.source_ref;
  TC[TC_id].local_ref:= ...;
  (/error/);
end;

P402:any NC_id:NC_id_type;TC_id,TC_id':TC_id_type;
provided NC[NC_id].received_NSDU.data.length <> 0
  and NC[NC_id].received_PDU.kind = CR
  and not TC[TC_id].in_use
  and NC[NC_id].received_PDU.dest_ref <> 0
  and (/exists TC_id' <> TC_id such that TC[TC_id'].in_use
        and TC[TC_id'].assigned_NC = NC_id and
        TC[TC_id'].remote_ref=NC[NC_id].received_PDU.source_ref/)
  and ATP[TC_id'].state = open_in_progress_calling
begin
  (/decode(NC[NC_id].received_NSDU,NC[NC_id].received_PDU)/);
  TC_id:=determine_TC(NC_id,NC[NC_id].received_PDU.dest_ref);
  TC[TC_id].remote_ref:=NC[NC_id].received_PDU.source_ref;
  TC[TC_id].local_ref:= ...;
  TS[TC_id'].T_DISCONNECT_ind(131,...);
  ATP[TC_id'].state:=closed;
end;

P403:any NC_id:NC_id_type;TC_id,TC_id':TC_id_type
provided NC[NC_id].received_NSDU.data.length <> 0
  and NC[NC_id].received_PDU.kind = CR and not TC[TC_id].in_use
  and NC[NC_id].received_PDU.dest_ref = 0
  and (/exists TC_id' <> TC_id such that TC[TC_id'].in_use
        and TC[TC_id'].assigned_NC = NC_id and
        TC[TC_id].remote_ref=NC[NC_id].received_PDU.source_ref/)
  and ATP[TC_id'].state = open_in_progress_called
begin

  ... same as above ...

end;
```

```
P404:any NC_id:NC_id_type,TC_id,TC_id':TC_id_type;
provided  ... same as above except
    ATP[TC_id'].state=open
begin

  ...same as above...

end;

P405:any NC_id:NC_id_type,TC_id,TC_id':TC_id_type;
provided  ... same as above except
    ATP[TC_id'].state=wait_before_closing
begin

  ...same as above...

end;

P406:any NC_id:NC_id_type,TC_id,TC_id':TC_id_type;
provided  ...  same as above except
  ATP[TC_id'].state=closing
begin

  ...same as above...

end;

P407:any NC_id:NC_id_type,TC_id_type;
provided NC[NC_id].received_NSDU.data.length <> 0
  and NC[NC_id].received_PDU.kind=CR
  and not TC[TC_id].in_use
  and NC[NC_id].received_PDU.dest_ref=0
  and not (/exists TC_id'<>TC_id such that TC[TC_id'].in_use
  and TC[TC_id'].assigned_NC=NC_id and TC[TC_id'].
    remote_ref=NC[NC_id].received_PDU.source_ref/)
  and (determine_PDU_length(NC[NC_id].received_PDU) >
  implied_PDU_length(NC[NC_id].received_PDU.TPDU_size_ind))
  or NC[NC_id].received_PDU.class_ind = class_0
  and NC[NC_id].this_side = calling)
  and ATP[TC_id].state=closed
begin
  (/decode(NC[NC_id].received_NSDU,NC[NC_id].received_PDU)/);
  TC_id:=determine_TC(NC_id,NC[NC_id].received_PDU.dest_ref);
  TC[TC_id].remote_ref:=NC[NC_id].received_PDU.source_ref;
  TC[TC_id].local_ref:=...;
  ATP[TC_id].state:=closed;
  ATP[TC_id].PDU.kind:=ERR;
  ATP[TC_id].PDU.reject_cause:=...;
  TC[TC_id].PDU_buffer[PDU.kind].full:=true;
  TC[TC_id].PDU_buffer[PDU.kind].PDU:=ATP[TC_id].PDU;
end;
```

```
P408:any NC_id:NC_id_type,TC_id:TC_id_type;
provided    ...same as above except
  ATP[TC_id].state=open_in_progress_called
begin

  ...same as above with the following added to the end...
  TS[TC_id].T_DISCONNECT_ind(...);

end;

P409:any NC_id:NC_id_type,TC:TC_id_type;
provided    ...same as above except
  ATP[TC_id].state=open_in_progress_calling
begin

  ...same as above...

end;

P410:any NC_id:NC_id_type,TC_id:TC_id_type;
provided ... same as above except
   and ATP[TC_id].state=open
begin

  ... same as above  ...

end;

P411:any NC_id:NC_id_type,TC_id:TC_id_type;
provided ... same as above except
   and ATP[TC_id].state=wait_before_closing
begin

  ... same as above ...

end;

P412:any NC_id:NC_id_type,TC_id:TC_id_type;
provided  ... same as above ...
    and ATP[TC_id].state=closing
begin

  ... same as above ...

end;

P413:any NC_id:NC_id_type,TC_id:TC_id_type;
provided NC[NC_id].received_NSDU.data.length <> 0
and NC[NC_id].received_PDU.kind=CR and not TC[TC_id].in_use
and NC[NC_id].received_PDU.dest_ref=0
and not (/exists TC_id' <> TC_id such that ...(*see above*)/)
and not (determine_PDU_length(NC[NC_id].received_PDU)>
implied_PDU_length(NC[NC_id].received_PDU.TPDU_size_ind)
or NC[NC_id].received_PDU.class_ind=class_0 and
```

```
NC[NC_id].this_side=calling) and
(/not able to provide service or destination address unknown/)
begin
  (/decode(NC[NC_id].received_NSDU,NC[NC_id].received_PDU)/);
  TC_id:=determine_TC(NC_id,NC[NC_id].received_PDU.dest_ref);
  TC[TC_id].remote_ref:=NC[NC_id].received_PDU.source_ref;
  TC[TC_id].local_ref:=...;
  TC[TC_id].PDU_buffer[DR].full:=true;
  TC[TC_id].PDU_buffer[DR].PDU.kind:=DR;
  TC[TC_id].PDU_buffer[DR].PDU.disconnect_reason:=...;
  TC[TC_id].PDU_buffer[DR].PDU.is_last_PDU:=true;
end;

P414:any NC_id:NC_id_type,TC_id:TC_id_type;
provided NC[NC_id].received_NSDU.data.length<>0
and NC[NC_id].received_PDU.kind=CR and not TC[TC_id].in_use
and NC[NC_id].received_PDU.dest_ref=0
and not (/exists TC_id'<>TC_id s.t. (* see above *)/)
and not (determine_PDU_length(NC[NC_id].received_PDU) >
implied_PDU.length(NC[NC_id].received_PDU.TPDU_size or
NC[NC_id].received_PDU.class_ind=class_0
and NC[NC_id].this_side=calling)
and (/able to provide and destination address known/)
and ATP[TC_id].state=closed
begin
  (/decode(NC[NC_id].received_NSDU,NC[NC_id].received_PDU)/);
  TC_id.T_addr:=determine_T_addr(NC_id.local_N_addr,
    received_PDU.called_addr);
  TC_id.id:=...;
  TC[TC_id].remote_T_addr:=determine_T_addr(NC[NC_id].
      remote_N_addr,
    received_PDU.calling_addr);
  received_PDU.peer_address:=TC[TC_id].remote_T_addr;
  TC[TC_id].QTS:=...;
  received_PDU.QTS_ind:=QTS;
  TC[TC_id].remote_ref:=source_ref;
  TC[TC_id].local_ref:=...;
  TC[TC_id].assigned_NC:=NC_id;
  ATP[TC_id].state:=open_in_progress_called;
  ATP[TC_id].in_use:=true;
  ATP[TC_id].options:=received_PDU.option_ind;
  ATP[TC_id].class:=received_PDU.class_ind;
  ATP[TC_id].TR:=0;
  ATP[TC_id].TS:=0;
  ATP[TC_id].R_credit:=0;
  ATP[TC_id].S_credit:=received_PDU.credit_value;
  ATP[TC_id].EX_D_sent:=false;
  ATP[TC_id].EX_D_received:=false;
  TS[TC_id].T_CONNECT_ind(TC.local_T_addr,received_PDU.
    peer_address,ATP[TC_id].options],received_PDU.QTS_ind,
    received_PDU.user_data);
end;
```

```
P415:any NC_id:NC_id_type,TC_id:TC_id_type;
provided received_NSDU.data.length <> 0
and received_PDU.kind=CR and TC[TC_id].in_use
begin
  TC_id:=determine_TC(NC_id,received_PDU.dest_ref);
  (/error/);
end;


PN4:any TC_id:TC_id_type
when TS[TC_id].T_DISCONNECT_req
provided ATP[TC_id].state=open_in_progress_called
and ATP[TC_id].class=class_0
begin
  ATP[TC_id].state:=closed;
  ATP[TC_id].PDU.kind:=DR;
  ATP[TC_id].PDU.is_last_PDU:=true;
  ATP[TC_id].PDU.disconnect_reason:=(/1 or 2/);
  TC[TC_id].PDU_buffer[PDU.DR].full:=true;
  TC[TC_id].PDU_buffer[PDU.DR].PDU:=ATP[TC_id].PDU;
end;

PN5:any TC_id:TC_id_type;
when TS[TC_id].T_DISCONNECT_req
provided ATP[TC_id].state=open_in_progress_called
and ATP[TC_id].class<>class_0
begin

  ...same as above except
  ATP[TC_id].PDU.disconnect_reason:=128;

end;
```

APPENDIC C

NORMAL FORM TRANSITIONS FOR THE CLASS 0 TP

(Based on [ISO 82])

```
when TSAP.T_CONNECT_req
P1:provided state = idle and
     (/Transport entity able to provide the quality
        of service asked for /)
begin
    state:=wait_for_CC;
    local_reference:=...;
    TPDU_size:=...;
    variable_part_to_send:=...;
    CR(0,local_reference,class_0,normal,
        variable_part_to_send);
end;

when TSAP.T_CONNECT_req
P2:provided state = idle and
     (/Transport entity not able to provide the
        quality of service asked for /)
begin
    state:=idle;
    T_DISCONNECT_ind(TCEPI,
        inability_to_provide_the_quality);
end;

when mapping.CR
P3:provided state = idle and variable_part.max_TPDU_size
        <> undefined and
         (/able to provide the quality of service/)
begin
    state:=wait_for_T_CONNECT_resp;
    remote_reference:=source_reference;
    TPDU_size:=variable_part.max_TPDU_size;
    remote_address:=variable_part.calling_T_address;
    TCEP:=...;
    called_address:=...;
    calling_address:=...;
    QOTS_estimate:=...;
    T_CONNECT_ind(TCEP,called_address,calling_address,
        QOTS_estimate,normal);
end;
```

```
when mapping.CR
P4:provided state = idle and variable_part.max_TPDU_size
        = undefined and
         (/ able to provide the quality of service/)
begin
    state:=wait_for_T_CONNECT_resp;
    remote_reference:=source_reference;
    TPDU_size:=128;
    remote_address:=variable_part.calling_T_address;
    TCEP:=...;
    called_address:=...;
    calling_address:=...;
    QOTS_estimate:=...;
    T_CONNECT_ind(TCEP,called_address,
        calling_address,QOTS_estimate,normal);
end;

when mapping.CR
P5:provided state = idle and
         (/ not able to provide the QOS/)
begin
    state:=idle;
    variable_part_to_send.
        additional_clear_reason:=...;
    DR(source_reference,0,connection_negotiation_
        failed,variable_part_to_send);
end;

when mapping.CC
P6:provided state = wait_for_CC and
    variable_part.max_TPDU_size<>undefined
begin
    state:=data_transfer;
    remote_reference:=source_reference;
    TPDU_size:=variable_part.max_TPDU_size;
    QOTS_estimate:=...;
    T_CONNECT_conf(TCEP,QOTS_estimate,normal);
    in_buffer.clear;
    out_buffer.clear;
    out_buffer.set_max_get_size(TPDU_size);
end;
```