CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

AN APPROACH TO LIFE-CYCLE TESTING OF COMMUNICATION PROTOCOLS

By

HASAN URAL

A Ph.D. Thesis

submitted to the School of Graduate Studies and Research

in partial fulfillment of the requirements

for the degree of Ph.D. in Electrical Engineering

University of Ottawa

Ottawa, Ontario, Canada

May 3, 1984

ABSTRACT

The effective use of data communications networks requires reliable protocols. However, protocols are often developed from inconsistent or incomplete specifications and with inadequate validation efforts. We propose a validation approach based on software testing principles which covers the entire protocol development life-cycle. The testing approach includes test sequence specification, generation, evaluation, and refinement; construction of executable specifications and designs; and consistency checking via observable behaviors of specifications, designs, and implementations.

Both the specification of test sequences and the construction of executable specifications and designs are based on attributed context-free grammars. A technique to automatically generate syntactically defined test sequences is presented. Shortcomings of systematic generation of test sequences are pointed out and a methodology aiming at integrating a tester's understanding of protocol semantics into the test construction process is proposed. This methodology is used to build generators for semantically meaningful test sequences.

A method of two-way complementary consistency checking which is facilitated by the construction of executable communication service and protocol specifications is presented. Algorithms for generating and validating externally observable behaviour of executable specifications and designs are described. As an integral part of consistency checking, test architectures which are applicable to pre-release and post-release protocol implementation testing are proposed.

The feasibility of this approach is demonstrated by a detailed application to ISO transport protocol class 0.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1. Context and Motivation for Life-Cycle Testing

Communication protocols form an essential part of any communication system where the interactions between communicating entities are based on the exchange of messages. In such a system, information about the state of a communicating entity is made available to the other entities only if it is explicitly released via a message (e.g., a finite sequence of bits) sent by the entity. Communicating entities in such a system utilize a set of rules for the orderly exchange of messages between them. This set of rules is called a communication protocol and is described in a protocol specification.

Recently, a great deal of attention has been given to the development of techniques for specifying communication protocols and for validating protocol specifications, designs, and implementations [IFIP 82, IFIP 83]. The term validation is used here to include any exercise aimed at increasing confidence in correct functioning of the entity being analyzed (e.g., specifications, implementations, etc.). Traditionally, natural language-based protocol specifications and informal validation techniques have been used in communication protocols development environments. This approach has been found to be inadequate and insufficient since protocol specifications in a natural language can be ambiguous, incomplete, inconsistent and

typically allow only manual and relatively informal validation techniques such as "walkthroughs". Such validation techniques are in general incapable of investigating and validating all possible behaviors of protocols and therefore are of limited help in increasing our confidence that the protocol specification, design, or implementation being validated has or does not have certain desirable properties. In fact, protocols such as CCITT X.21 and X.25, among others, which were developed using informal techniques, suffered "a disturbing number of errors or instances of unexpected and undesirable behavior" [Boch 77, BoSu80, BeLy 77, Lai 81].

Formal specification techniques provide a basis for adequate and relatively error-free validation activities in developing reliable communication protocols. These techniques vary from extended finite state machine based formalisms [BoGe 77] to temporal logic based techniques [Pnue 77]. Since a protocol specification is taken as a reference for subsequent development and validation of protocol designs and implementations, its validation takes on a critical importance. In fact, any development and validation effort applied to protocol designs and implementations may be wasted if the underlying protocol specification is invalid, incomplete, ambiguous, or inconsistent with the intentions of the specifier.

Generally speaking, validation activities in a protocol

development environment may be grouped under two headings; namely, verification and testing. Analysis of an entity for the purpose of validation is based on logical means in verification and on observed behavior of the entity in testing.

Verification techniques applicable to protocol specifications (and possibly to designs and implementations) are closely related to underlying protocol specification techniques and vary from reachability analysis [Suns 75] to program proving techniques [Sten 76]. Many of these techniques concentrate on checking certain general properties of protocols such as absence of deadlocks, unspecified receptions, etc. [BoSu 80]. In the case of complex protocols, automation of these techniques becomes a necessity due to the combinatorial explosion of the size of the associated state space.

On the other hand, testing aims at detecting the inconsistencies (if any) between the entity under test (EUT) and its specifications (implicit or explicit) through the observations of controlled interactions of the EUT and its environment. Often, testing is employed as a practical means of checking the conformance of an implementation to its specifications. Recently, specification testing (as opposed to implementation testing) has been considered as a mechanism to check the consistency of specifications given at different levels of abstraction [Gogu 80, JaBo 83]. Such

a mechanism is particularly desirable in a development process which progresses via step-wise refinement. One such development process is implied by the layered protocol architecture defined by the OSI Reference Model [Zimm 80].

Considering the protocol layers in the OSI Reference Model, a typical protocol development life-cycle involves a sequence of phases in which a protocol implementation is constructed through successive refinements from its specifications. This step-wise refinement process produces a series of system functionality representations (i.e., specifications, designs, and implementations) in decreasing levels of abstraction. Each such representation describes the system functionality in terms of externally observable behavior of the system (i.e., possible types and orderings of interactions that the system exchanges with its environment). Each step of refinement results in a more refined representation of the system functionality than the one developed in the preceeding step. A major concern in a protocol development life-cycle, is to ensure that these distinct but highly related representations are in fact mutually consistent.

1.2. Objectives and the Scope of the Thesis

Our primary objective is to define a life-cycle validation approach that can potentially be applicable throughout a typical protocol development life-cycle. Here,

we attempt to define such an approach that employs testing as a means of checking the consistency of system functionality representations. In particular, we concentrate on adapting testing to checking the consistency of those representations constructed at earlier phases of the life-cycle.

In the proposed approach, the consistency checking is based on dynamic analysis of externally observable behavior of representations. Specifically, this entails,

1) Constructing "executable" representations

2) Generating (i.e., revealing and observing) the system functionality captured by these representations in terms of their externally observable behavior (i.e., in terms of input and output sequences that they exchange with their environment)

3) Checking whether

   (a) each observed (actual) interaction sequence obtained during the execution of the representation under test is permitted by a more abstract and already validated representation.

   (b) each intended (expected) interaction sequence (i.e., test sequence) obtained during the execution of a more abstract representation is realizable via the representation under test.

An overview of the approach is given in chapter 3. This approach is intended for a protocol development environment which progresses via step-wise refinement and which involves formal specifications of services and protocols. Because of the wide interest in layered architectures and in open system interconnection (OSI), we restrict ourselves to the context of OSI Reference Model and OSI protocols.

1.3. Summary of Main Contributions

The main contributions of the thesis are the demonstration of a means of developing executable system functionality representations and methods of exercising these executable representations in a structured manner to carry out validation activities throughout the protocol development life-cycle. In particular, we demonstrate the usefulness of two highly related attributed grammar based formalisms, namely one for developing executable representations and one for constructing test (sequence) specifications. The first formalism is used to develop executable communication service and protocol specifications which facilitate the validation of their mutual consistency through testing. The second formalism is intended to integrate the tester's understanding of protocol semantics into the test construction process and thus is used to construct semantically meaningful test sequences. It is our belief that development of such test specifications early in

the protocol development life-cycle is a high level analogue of "redundant development" technique proposed by D. Panzl [Panz 81] and expanded upon in [PrUr 82a, PrUr 82b].

Other contributions include

a) a technique for automatic derivation of syntactically defined tests (sequences) based on the grammatical structures underlying protocol specifications

b) algorithms for

(1) generating intended or observed interaction sequences of executable service and protocol specifications

(2) checking whether a given (actual or expected) interaction sequence is allowed by an executable representation

c) architectures for pre-release and post-release protocol implementation testing.

1.4. Outline of the Thesis

A brief survey of techniques for communication protocols specification, verification, and testing is given in chapter 2 together with discussions of their applicability and limitations.

In chapter 3, an introduction to the proposed life-cycle testing approach is presented. Basic definitions are followed by an overview of the approach. Various consistency checking practices in protocol development environments are discussed.

Chapter 4 discusses the construction of executable representations: an interaction model underlying the representations, an extended finite state machine model and an attributed grammar based formalism to describe representations, a method of encoding representations in a logic programming language PROLOG, and techniques to carry out transformations between these formalisms. Examples include ISO Transport Service and Protocol specifications.

In chapter 5, generation of actual (observed) and expected (intended) behavior of executable representations is discussed. Three different modes of system behavior generation are identified; namely, systematic, user-guided, and input-directed generation. Construction of minimum covers and test sequence grammars are presented. An algorithm describing input-directed system behavior generation is given.

Chapter 6 describes checking the mutual consistency of distinct representations of the same system functionality. The chapter presents algorithms for invocation routines which automatically invoke a PROLOG procedure corresponding to an executable representation in order to check whether an

intended or observed system behavior is allowed by that representation. The chapter concludes with test architectures for protocol implementation testing.

Chapter 7 contains the summary, conclusions, and suggestions for future research.

2. PREVIOUS WORK

2.1. Protocol Specification and Verification

This section reviews previous work on communication protocols specification and validation. The review is based on a classification of the protocol specification techniques which categorizes them into three classes, namely: the "Transition-Based" techniques, the "Programming Language-Based" techniques, and the "Hybrid (Unified)" techniques. Protocol validation methods are reviewed during the discussion of associated protocol specification techniques. The discussion of each class of specification techniques includes an introduction describing the common characteristics of those techniques within the class and the validation methods applicable to them. This introduction is followed by the presentation of representative specification techniques in the class. The presentation of each representative specification technique consists of the description of the technique, the description of the associated validation methods, and a critique of both. Surveys on protocol specification and validation techniques can also be found in [Suns 76,78a,79,81], [Merl 79], [Thar 79], [BoSu 80], [Teng 80], and [Schw 81] among others.

2.1.1. Transition-Based Methods

The transition-based methods have been motivated by the observation that protocols involve communicating entities which respond to certain internal and external stimuli such that their behavior can be described by state machines. These methods may be classified into Finite State Machine Models, Formal Language Models, and Graph Models. In general, these models are used to describe the control aspects of protocols (i.e., rules governing the exchange of messages during initialization, connection establishment, connection termination, etc.). For the representation of the data transfer aspects of protocols (i.e., the parameters involved during transfer of data such as the sequence numbers of the messages, message texts, etc.) additional states are necessary. For example, different states are required to represent each possible sequence number that a message may take. Hence, for relatively complex protocols and/or realistic assumptions on the transmission medium (i.e., arbitrary transmission delay, message loss, message duplication, and message distortion) the number of states becomes intractably large (i.e., state explosion).

The validation approaches associated with transition-based formalisms are based on a basic state exploration technique (reachability analysis) [Suns 75, Boch 78]. This technique consists of exploring all possible interaction sequences of communicating processes by generating

exhaustively all global states <1> (composite states) reachable from a given initial global state <2>. Starting from the initial global state, the state exploration technique generates all possible global state transitions in the form of a transition diagram which corresponds to the reachability tree (graph) rooted in the initial global state. The traversal of the reachability tree is equivalent to the exploration of the complete interaction domain of the overall communication system (i.e., communicating processes and the transmission medium).

The state exploration technique is effective for checking implicit requirements and control aspects of protocols which are expressible in terms of the structure of the reachability tree and the contents of the global states. Examples of these implicit requirements (more commonly, the "general properties" [BoSu 80]) are excluding any possible occurrences of unspecified receptions <3>, state deadlocks <4>,

---

<1> A global state is generally defined as a combination of the states of the communicating processes and the transmission medium.

<2> The initial global state represents the initial states of all communicating processes and an empty transmission medium (i.e., no messages in transit).

<3> An unspecified reception occurs when a message reception that may take place at a state is not specified in the protocol specification.

<4> A state deadlock occurs when no message transmissions are possible from the current states of the processes and the transmission medium is empty (i.e., no messages are in

non-executable interactions <5>, state ambiquities <6>, channel overflows <7>, unbounded growth <8>, and infinite idle loopings (or tempo blockings)<9>; establishing proper synchronization; providing self-synchronization; and terminating at a desired final state [Suns 79]. Once error conditions corresponding to violations of required properties are defined based on the specification methods, these conditions are checked on the context of each global state reached during reachability analysis [Rudi 78]. A major limitation of the state exploration technique is state explosion, that is the size of the reachability tree grows very rapidly with the complexity of the protocol and makes it impractical to generate and check all reachable global states. Automated generation of reachability trees and error condition checks are reported in [DaBr 78, Rudi 78, West

---

transit).

<5> A non-executable interaction is present when the protocol specification has receptions and/or transmissions that can not occur under normal operating conditions (i.e., similar to deadcodes in programs).

<6> A state ambiguity exists when a state in one process can coexist with several different states in the other processes while the transmission medium is empty (i.e., terminal global states).

<7> A channel overflow occurs when the number of messages in the channel exceeds the predefined capacity of the channel.

<8> An unbounded growth occurs when one of the processes transmits messages at a rate which is much faster than the processing rate of the receiving process.

<9> A tempo-blocking is a loop in the transition diagram in which processes may execute without any effective progress.

78a, West 78c, Zafi 80].

Another form of state exploration is the symbolic execution technique which attempts to reduce state explosion [BrJo 79]. A reachability graph is constructed starting from an initial symbolic global state. In this graph, each node corresponds to a symbolic global state which represents a large number of explicit global states.

2.1.1.1. Finite State Machine Models

Finite State Machine (FSM) models include "Pure FSM" [Bart 69], "Duologue Matrix" [Zafi 78], "Phase Diagram" [West 78c], "Perturbation" [West 78b, Zafi 80], and "Colloquy" [LeMo 73, DaBr 75] approaches. In these approaches, each process is represented by a FSM where the states of the FSM correspond to the states of the process and the transitions between the states of the FSM correspond to the transitions in the process associated with receiving or transmitting stimuli and responses respectively. The coupling between FSMs is done by using queues (buffers) connecting the inputs of one FSM to the outputs of the other and vice versa, or by direct coupling (i.e., specifiying the transitions in each FSM that must occur together) when the message tranmission delay is not important. Although the coupling of FSMs corresponds to the underlying transmission medium, the specific behavior of the transmission medium may be explicitly modelled by a separate FSM.

As an example of FSM models, we briefly describe the "Perturbation Approach". This approach extends the applicability of duoloque matrix theory [Zafi 78] to protocols governing the interactions between more than two processes without the restriction that the processes need return to their initial states after a finite number of interactions [West 78b, Zafi 80].

Specification Technique

This approach is based on a communication system model which includes a FSM for each communicating process (there may be more than a pair of processes) and a pair of simplex channels, one for each direction of communication, between each pair of processes. An instance of the communication system is represented by a system state (i.e., global state) which is a unique set of current states of the processes and of all simplex channels <10>.

Validation Method and Remarks

The perturbation analysis is equivalent to the state exploration: it starts with the initial system state and generates the complete interaction domain of the communication system by investigating all possible ways in which the initial state and all subsequent states can be

---

<10> At any given time, the state of a simplex channel is the ordered set of messages in transit in the direction of communication that the simplex channel stands for.

perturbed. A perturbation of a system state results in a new system state by executing a single transition <11> in one of the processes in the system while other processes remain unchanged. Simultaneous transitions of more than one process are represented as a sequence of perturbations. During successive perturbations, a system state that has been previously generated by perturbations of an earlier state need not be further perturbed and is therefore omitted. This procedure continues until no more perturbations are possible. During this process, each generated system state is analyzed to determine whether it represents a non-occurable interaction, state deadlock, unspecified reception, state ambiguity, or channel overflow, etc.

The perturbation approach is applicable to a much wider range of protocols than duologue matrix and phase diagram approaches. It offers all of the capabilities of both of the previous approaches except the analysis of race conditions performed by the phase diagram approach in a restricted environment (i.e., perfect transmission medium). That is, this approach does not allow explicit timing constraints such as the time taken to transport messages between processes to be specified (like the duologue matrix approach). Other limitations of this approach are caused by

---

<11> This implies a change in the state of the process and also a change in the state (content) of the channel which is affected by that transition.

the assumptions on the transmission medium as well. The transmission medium is not allowed to reorder messages and there is a predefined upper bound on the number of messages in transit at any instant of time.

A decomposition technique proposed by Vuong [VuCo 82] has extended the applicability of the perturbation approach to more complex (but structured) protocols such as session negotiation procedures of the CYCLADES transport protocol and the packet level of X.75 protocol.

2.1.1.2. Petri Nets and Other Graph Models

Petri Nets (and the related "UCLA Graphs" [Post 74, RaEs 80]) are the main graph models which are theoretically applicable to a broader range of protocols than FSM models [Merl 79]. In particular, the specification of protocols having a possibly infinite number of states may be done by Petri nets which have the property to accumulate unbounded number of messages at their nodes. Such protocols, i.e., the ones allowing the transmission of any message an arbitrary number of times, can not be represented by a finite state machine even theoretically [Merl 79].

Petri nets are graphical representations of possible control flow in concurrent processes and were first introduced by Petri [Petr 62]. They provide a detailed description of the conditions related to the control flow in

processes and correspond, in a more abstract form, to flowcharts, or to natural language assertions of the control flow [Dant 80].

Specification Technique

In this formalism, protocols are represented by Petri nets which are digraphs with nodes called as conditions (places) and events (transitions) [Merl 74]. A directed arc connects either a condition to an event which specifies that condition to be an input condition to the event or an event to a condition which is called as an output condition for the event. An instance of a Petri net is represented by a token distribution among the conditions (places) in the net which is called a marking. From an initial marking of the Petri net, it is possible to derive the set of markings reachable from it. A token distribution at any instance is affected by the occurrence of events. For an event to occur (to be enabled), each of its input conditions should hold (i.e., have at least one token). The occurrence (firing) of an enabled event consists of removing one token from each of its input conditions and adding one token to all of its output conditions.

The basic Petri net model of a protocol is a single composite Petri net obtained by combining Petri nets representing the communicating processes. This combination is achieved by identifying the input and output places of each component's Petri net representation and by connecting

them according to the assumptions made on the transmission medium.

Validation Method and Remarks

The validation methods for the basic Petri net models are based on the reachability analysis: Starting from an initial marking, all possible markings of the composite Petri net are generated. The set of markings reachable from the initial marking and the possible transitions between them define a state machine called token machine where each state corresponds to a marking [Merl 76]. Thus, checking some general properties of protocols such as liveness, absence of deadlocks, etc. is based on the context of each state of the token machine (i.e., the corresponding token distribution).

The major shortcomings of Petri net approach are not different than those of FSM approaches. To overcome some of these shortcomings several extensions and generalizations of the basic Petri net model have been proposed. One of these extensions, Time Petri Net, is a direct derivative of the basic model and is related to incorporating timing constraints to the events (transitions) in the Petri net to allow the analysis of recoverability <12> (i.e., self-synchronization) [MeFa 76].

<12> The ability to return to its normal mode of operation within a finite time after a perturbation occurs.

2.1.1.3. Formal Language Models

The inclusion of the formal language models in the transition-based methods is based on the correspondence between state machines and formal grammars. The starting symbol of the formal grammar corresponds to the initial state of the state machine whereas the non-terminal symbols to the states, the terminal symbols to the input and output sets, and the production rules of the grammar to transitions in the state machines.

Specification Technique

In this formalism, the interaction sequences defined by a protocol between communicating processes are viewed as the sentences of a formal language. Thus a protocol is described by a formal grammar which defines a formal language and generates all valid sentences to represent the possible interaction sequences between the communicating processes. That is, the formal language is based on an alphabet which is composed of symbols representing the events in the communication system and the sentences generated by the formal grammar describe the sequences of events (say dialogues) defined by the protocol.

If the interaction sequences are viewed as sequences of terminal symbols (as described above), the formal grammar describing a protocol as a set of rules that define those interaction sequences is called an action grammar [TeLi 78].

The complex formats of the messages exchanged between the communicating processes may also be represented in this formalism by defining a message grammar for the formats of the fields in each message [TeLi 78]. It is possible to substitute the productions of the message grammar for the terminal symbols of the action grammar to have a detailed representation of the protocol which is being modelled.

Validation Method and Remarks

The validation methods for formal language models are based on formal properties of the grammars used. These methods (e.g., "grammar cleaning" techniques) may detect loops in the interaction sequences, improper termination or deadlocks, etc. For the formal language models based on regular grammars, the state exploration techniques can be applied on the corresponding state machines obtained by a proper transformation.

The formal language models and the associated validation methods have the same restrictions as FSM models and the state exploration techniques, particularly in representing and validating the data transfer aspects of the protocols. This approach has been applied to HDLC link protocol by Harangozo [Hara 77, 78]. This application involves a regular grammar model of the protocol and an indexing technique to accommodate sequence numbers of the messages exchanged. However, the indexing technique which is used to define groups of productions parameterized by a seguence number

leads to a rapid increase in the number of the productions (i.e., similar to a state explosion). For the solution to this dimensionality problem, Teng proposes the use of context-free or context-sensitive grammars [TeLi 78, Teng 80].

2.1.2. Programming Language-Based Methods

Protocols may be viewed as procedures containing concurrent processes which can be described as programs in high-level programming languages. Depending on the level of abstraction provided by the programming languages, one advantage of the programming language-based methods is that the protocol specification may be quite near to an implementation of the protocol [BoSu 80]. Another advantage of these methods is their capability to represent data structuring aspects of the protocols which is a major shortcoming of the transition-based methods for protocol specification.

Specification Technique

The basic programming language model consists of programs for each communicating process. The underlying transmission medium is specified by the assumptions formulated as assertions on its behavior. The communication between separate modules is achieved through the use of shared variables or message passing [Boch 75, Sten 76, Good 77,

GoCo78, Haje 78, Mier 79].

Validation Method

The basic validation method applied to programming language models is based on program proving approach [Eloy 67]. This approach involves first formulating appropriate assertions which reflect the desired correctness properties corresponding to explicitly stated requirements (e.g., intentions or service specifications) or to general (implicit) properties. The next step is to prove the correctness of protocols by verifying these assertions. In contrast to the state exploration technique, the program proving technique has the capability that allows the validation of data transfer aspects of protocols. However, both generating the assertions and carrying out the proof require considerable amount of expert knowledge and human ingenuity. This requirement prevents the technique to be fully automated. However, the use of some standard automatic theorem provers may be employed to have some degree of automation. But the lack of sufficiently rich assertion languages to express the assertions covering a wide variety of data types and functional primitives limits the applicability of these partially automated tools for the time being.

Symbolic execution is another validation activity applicable to programming language models [BrJo 78], [Mier 79]. Theoretically, symbolic execution may be used as a

program proving technique: by attaching the assertions at certain points in the programs together with some predicates representing the choice of paths, it is possible to check those assertions by symbolically executing the programs. However, practical limitations restrict the effective use of symbolic execution to the generation of paths to be proved correct and assertions and/or verification conditions. These outcomes of the symbolic execution may then be used by a verifier (a human or a tool) for correctness proofs. Note that although symbolic execution can be automated, in many cases it still requires considerable human interaction (e.g., during executable path generation) and usually covers only a special subset of the syntax of the programming language that the automated tool is designed for.

Remarks

It has been noted that the use of programming languages as specification tools may generate "not a satisfactory specification because it is impossible to separate the essential features of what the program is supposed to do from the particular way chosen to accomplish those functions" [Suns 81]. In fact, it is rather difficult to specify or model a protocol with programming language based techniques and not to include any hints about the process with which the protocol might be implemented. The less procedural and more abstract specification techniques for which the underlying model is not different than that of

programming language-based models may reveal the "required" distinction between a specification and an implementation of that specification. These techniques employ a more abstract notation than programming languages to specify protocols and may be classified as "Sequencing Expressions" [Schi 80], "State Deltas" [Over 81], "Abstract Data Types" [Thom 81], and "Temporal Logic" [Hail 81, Lamp 80, ScMe 81].

2.1.3. Hybrid (Unified) Methods

The formal methods for protocol specification and validation reviewed so far are powerful in representing either the control or the data transfer aspects and properties of protocols, but not both [Merl 79]. Several specification methods combining the distinct features of both classes of specification methods have been proposed [BoGe 77, DaBr 78, Suns 78b, Boch 80, Dant 80, Schu 80].

Specification Technique

Typically a hybrid model is a state machine augmented with context variables and procedures representing states and transitions of the state machine written in a high level language. The states correspond to the subfunctions of the functionally decomposable features of the protocol being represented, namely: initialization, connection establishment, data transfer, interrupts, error recovery, and such. Procedures for the states describe the action to

be taken depending on the input (stimulus) and the values of the context variables. For example, the sequence number of an arriving message can be compared with a variable storing the next expected sequence number to determine whether to accept the message, what the next state should be, and how to update the expected sequence number [Boch 80].

Validation Method and Remarks

The validation methods applicable to hybrid models generally employ both the reachability analysis (to check the general implicit properties) on the state machine and the program proving techniques (to verify the explicit requirements such as sequenced message delivery) on the context variables and procedures [BoGe 77]. Another validation method which is applicable to hybrid models is symbolic execution.

Hybrid models can be viewed as extensions of transition-based models (i.e., augmenting a transition-based model by context information and procedures). Several hybrid models are based on Keller's transition model for parallel programs [Kell 76]. This model is essentially a Petri net complemented with a set of variables X. The transitions in the net are associated with enabling predicates on the variables in X and with actions assigning new values for X. Hence, the state of the communication system at any given time (when no action is being executed) is determined by the token distribution in the net (i.e., marking) and the values

of the variables in X.

Some hybrid models are quite similar to Keller's transition model. Two of these models are given in [Symo 80] and [RaEs 80]. In the following section we will present an extension on the above model.

General Transition Model

The general transition model is an extension of Keller's transition model [BoGe 77, Boch 80] to include the possibility of having several disjoint subsystems and some means of communication between them. That is, a protocol is specified by separate descriptions for a pair of subsystems (e.g., entities in a protocol layer) executing the protocol. Each subsystem is modelled by the formalism of Keller's transition model and the predicates and actions (local actions) of a subsystem refer only to the local variables in that subsystem. In order to specify the variables, predicates, and actions in a subsystem a notation similar to that of Pascal has been used [BoGe 77].

The actions of the separate subsystems are interrelated by using distantly initiated actions (DIA) which allow one subsystem to change the local variables of the other subsystem. A DIA in a subsystem may assign new values to the local variables in that subsystem but unlike a local action it is not associated with a transition in the subsystem. DIAs are initiated by a distant subsystem with the execution

of an initiating statement in a local action in that distant subsystem. The initiating subsysytem may pass parameter values for the execution of the DIAs. The initiation of the DIAs of a distant subsystem and the execution of DIAs in that subsystem represent message transmission and receptions respectively.

The state of the overall system is determined by the token distribution, values of the variables, and set of DIAs which have not yet been executed. This set corresponds to the content (state) of the transmission medium (i.e., messages in transit).

The validation method involves stating assertions about the state of the overall system and using a form of induction to prove these assertions. The assertions are predicates which state the desired properties of the protocol in terms of states of the system. The form of induction used states that the predicates are in fact invariant assertions: for each transition t, if a predicate P holds before t has fired and if t is enabled then P holds after t has fired. That is, the predicate P is system invariant if it is preserved by the actions associated with t.

2.2. Protocol Testing

Another major activity in communication protocols development environment is to assess the conformance of a protocol implementation to the protocol specification. Although various validation methods such as simulation or program proving techniques can be used for protocol implementation assessment the related research is directed mainly towards conformance (assessment through) testing [Boch 83]. Investigations have focused particularly on developing test architectures and generating test sequences for conformance testing.

2.2.1. Test Architectures

The work on test architectures for protocol implementation assessment is very recent [Ansa 81, Ansa 82, AnDa 82, BoCe 82, Boch 83, Henl 81, Nigh 82, Rayn 82, UrPr 83a]. According to a typical proposed test architecture [Rayn 82] (cf. Figure 2.1), a protocol implementation is tested as a "black-box" over a communication medium between an assessment (certification) site and a client site. The assessment site configuration includes an "Active Tester" (AT). The client site configuration includes the protocol implementation under test (IUT) and a "Test Responder" (TR) acting as the user of the service to be provided by the IUT. During assessment, the conformance of the IUT to the specifications is assessed through the observations of

controlled sequences of interactions between AT and TR which are generated as a result of the application of a set of test sequences.

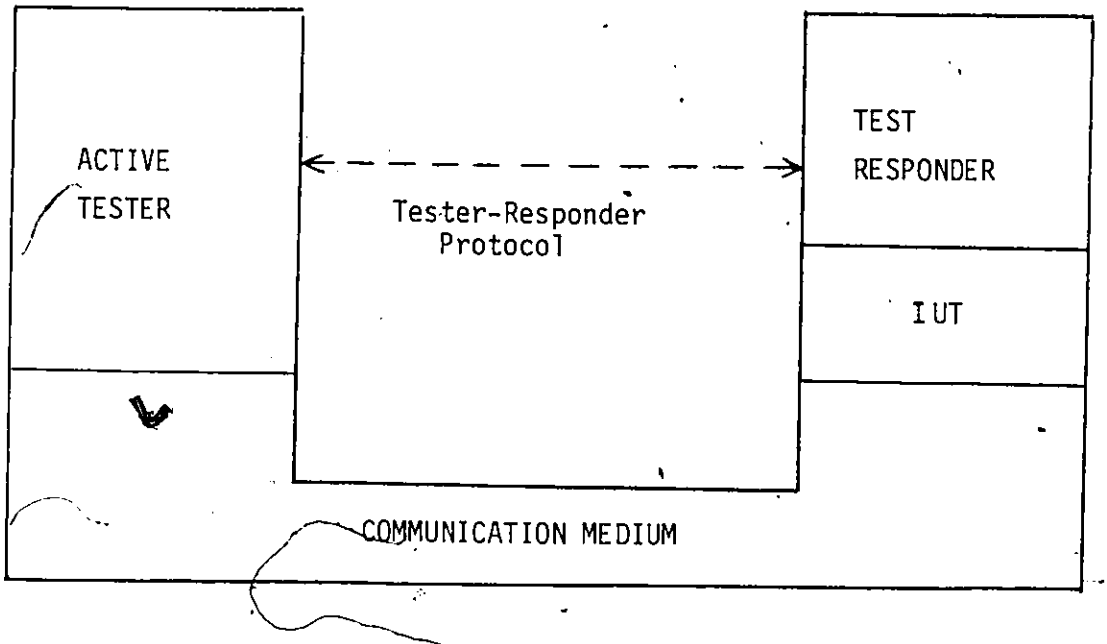


Figure 2.1 A Test Architecture for Protocol Implementation Testing

2.2.2. Test Sequence Generation

One of the major tasks in protocol implementation testing is the generation of effective test sequences (sequences of input stimuli and possibly corresponding expected responses). In software engineering, complementary testing techniques are used involving both black-box and white-box (structural) test generation strategies in order to improve the effectiveness of tests. However, a common assumption in protocol testing is that the internal structure (i.e., source code) of the IUT is not available [Boch 83, Rayn 82]. This assumption restricts the applicable test sequence generation strategies to black-box or specification-based testing techniques; i.e., the design and the internal structure of the IUT can not be taken into account to guide the construction of the test sequences. For example, a recent attempt to generate test sequences to be used for conformance testing is based on protocol specifications [SaBo 82, SaBo 83]. In this attempt, various systematic methods for testing finite state machines (FSM's) are applied to communication protocol specifications modelled as pure FSM's. Other attempts are reported in [LiMc 83, UrPr 83b]. Reports of some of the most recent work on protocol specification, verification, and testing can be found in [IFIP 82, IFIP 83].

3. LIFE-CYCLE TESTING APPROACH

This chapter presents the basic definitions; an overview of the approach which introduces a method of two-way complementary consistency checking; specific consistency checking exercises in a typical protocol development life-cycle; and the basis for the approach.

3.1. Basic Definitions

Consider the Reference Model of protocol hierarchies for "Open System Interconnection" (OSI) being developed within ISO [Zimm 80, 81]. The layered protocol architecture defined by the OSI Reference Model is given in Figure 3.1 and an overview of layers can be found in [Zimm 80]. In this layered protocol architecture , protocol layer N provides a particular communication service (i.e., the N-service) to layer N+1. The N-service is specified in the service specification for laver N (i.e., the N-service specification). The N-service specification describes the overall behavior of layer N in a highly abstract manner. That is, the N-service is defined in terms of possible types and possible execution sequences of interactions (i.e., N-service primitives) exchanged between layer N and layer N+1 at the N-service access points.

However, according to the OSI Reference Model, the N-service is provided by the collaboration of protocol

entities (denoted N-entities) in layer N where each N-entity uses the service provided by the (N-1)-service provider as shown in Figure 3.2. This corresponds to a distributed implementation of the N-service provider with N-entities (usually at physically remote locations) communicating with each other via the (N-1)-service and serving their local users in layer N+1 (i.e., (N+1)-entities). For example, the transport protocol layer is implemented with transport protocol entities communicating to each other via a network service in order to provide a transport service to session layer entities. In OSI, the network service is implemented in a similar manner, and so on.

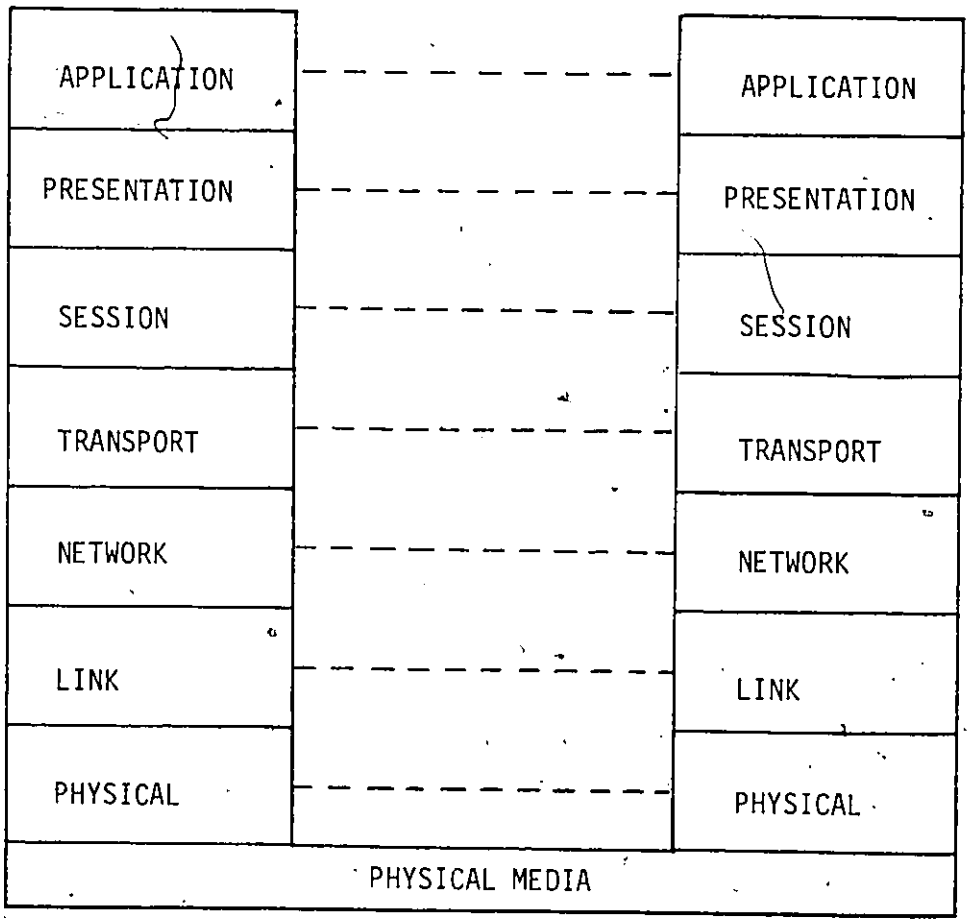


Figure 3.1 Protocol Hierarchy of the OSI Reference Model

The role of an N-entity in providing the N-service is defined in the protocol specification for layer N (i.e., the N-protocol specification). The N-protocol specification specifies the behavior of each N-entity in terms of its responses to requests (N-service primitives) from its users ((N+1)-entities), to messages (or Protocol Data Units (PDU's)) from other N-entities received via the (N-1)-service, to control messages ((N-1)-service primitives) received from the (N-1)-service, and to internal events such as timeouts. That is, the N-protocol specification is an abstract representation of the behavior of each implemented protocol entity in layer N. Thus, a complete implementation of an N-entity (i.e., N-protocol implementation) may be constructed by successively refining the N-protocol specification. For example, a transport protocol prototype implementation can be obtained directly by refining the transport protocol specification. The refinement steps may involve defining lower and upper layer interfaces through which the protocol implementation interacts with entities in lower and upper layers respectively, algorithms implementing protocol actions and rules, data structures, etc.

Hence, the protocol development process (or life-cycle) related to a protocol layer in the OSI Reference Model may be defined as a sequence of phases starting from formulating intentions of the desired service (or software functionality) and converging to implementations of protocol

entities. In fact, in the above context, the protocol development life-cycle for protocol layer N may be viewed as a step-wise refinement process involving the following phases:

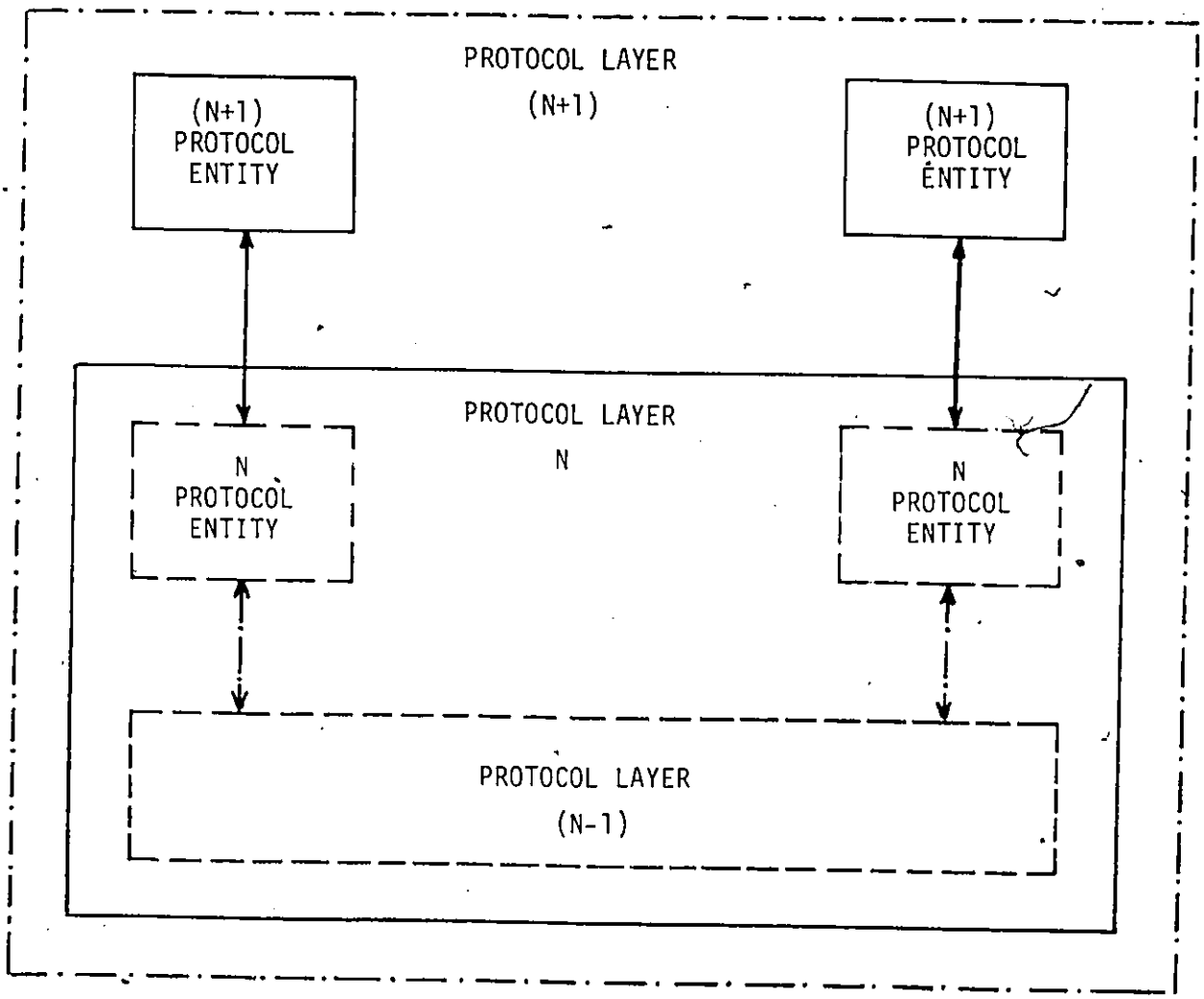


Figure 3.2 Structure of a Protocol Layer N

1) Formulating the intended functionality of the desired N-service. These intentions are generally implied by the purpose of layer N and the service requirements of layer N+1.

2) Constructing a natural language representation of the intentions (This representation is often the first explicit statement of intentions and usually is denoted the N-service specification), and then deriving a corresponding formal N-service specification.

3) Constructing a formal N-protocol specification portraying the role of each N-entity in the provision of the N-service.

4) Constructing a protocol implementation by refining the N-protocol specification in increasing levels of detail and developing necessary implementations of service interfaces.

In each of these phases, a representation of either the service of layer N (as in phases 1 and 2) or of the role of N-entities in providing this service according to the distributed implementation of the N-service provider, implied by the OSI Reference Model (as in phases 3 and 4), is developed. We call these representations of desired system behavior functionality representations. Each such representation describes the system functionality in terms

of externally observable behavior of the system (i.e., possible types and orderings of interactions that the system exchanges with its environment). These representations (namely, intentions, service specifications, protocol specifications, protocol implementations, and their modified and/or refined versions) are all highly related to each other and are formulated at different levels of abstraction. For example, consider the service specification and the protocol specification. The N-service specification views layer N as a "black box" whereas the N-protocol specification is based on a refinement of layer N where the N-service is provided by the collaboration of functionally identical N-entities over the (N-1)-service.

In this step-wise refinement process, each functionality representation (except intentions) is derived by successively refining the more abstract representation immediately preceeding it. A large number of errors may be introduced into the representations during these derivations (i.e., transformations from one representation to another) mainly due to misunderstandings among project personnel and lack of clarity of the more abstract representation, as well as omissions, ambiguities, and logical inconsistencies of the more abstract representation. Such errors can propagate throughout subsequent representations unless they are detected and corrected in the representation where they originate. However, note that even this correction activity is a potential source of new errors.

Thus the purpose of validation in a protocol development process is to ensure that these distinct representations are in fact consistent. That is, validation is an activity of checking the consistency of a more refined representation against a more abstract representation. Accordingly, considering protocol layer N:

1) The purpose of service validation is to show that the N-service specification is consistent with the intentions implied by the purpose of layer N and the service requirements of layer N+1.

2) The purpose of protocol validation is to demonstrate that the collaboration of N-entities, as described in the N-protocol specification via the (N-1)-service provides the N-service specified in the N-service specification. In other words, protocol validation is to check the consistency of a refined representation of layer N (i.e., as in Figure 3.2), which is composed of N-protocol specifications and the (N-1)-service specification, with respect to the N-service specification. It is assumed that the (N-1)-service specification has already been validated.

3) The purpose of protocol implementation validation is to show that an implementation of an N-entity is consistent with the N-protocol specification.

In general, checking the consistency of a refined system functionality representation (representation K) with an abstract representation of the same functionality (representation K-1) includes demonstrating that:

a) every externally observable system behavior described by representation K that is actually possible satisfies (the constraints imposed by) representation K-1 (e.g., intentions of the specifier, functional specifications, etc.), and

b) every externally observable system behavior described by representation K is actually possible [Boch 80].

Point a) is also called "safeness" or "partially correctness" which relates the system behavior described by representation K to that of representation K-1. Point b) is also called "liveness" which includes some "general properties" of any system functionality representation such as absence of deadlocks, infinite loopings, unspecified receptions, etc. [BoSu 80].

There are basically two approaches to validation, namely, verification and testing [JaBo 83]. Verification is based on formal and logical means of analyzing the system functionality representations. Verification methods include both reachability analysis [Zafi 80] and program proving techniques [Sten 76]. While reachability analysis attempts to check general properties of system functionality

representations, program proving techniques involve proving assertions reflecting desired correctness properties (i.e., partial correcness and termination) [BoSu 80].

Testing, on the other hand, attempts to reveal inconsistencies between representation K and representation K-1 by comparing the system behavior observed during the execution of representation K with the system behavior described by representation K-1. However, testing can not guarantee that representation K is "consistent" with representation K-1 unless all possible system behavior patterns described by these representations are explored and compared against each other. In general, this corresponds to "exhaustive" testing which is practically infeasible [Piat 80]. Thus, testing aims at detecting particular, highly likely (if not all) instances of inconsistencies between representation K and representation K-1.

3.2. Overview of the Approach

In this thesis, we present a life-cycle validation approach for developing reliable communication protocols. The life-cycle validation approach employs testing as a means of checking the consistency of system functionality representations. As a dynamic analysis technique, an inherent requirement of testing is that the entity under test is executable. Therefore, the proposed approach involves constructing system functionality representations

that can be executed.

Construction of executable representations of system functionality is based on the following assumptions:

1) each system (e.g., the N-service provider) is characterized by the interaction model given in [Boch 80]. Accordingly, each system is described as a collection of processes, each process interacting with its environment (i.e., other processes) through a number of interaction points.

2) each process is represented by an extended finite state machine where extensions are defined in terms of additional state variables and interaction parameters [ISO 83b].

Using the extended FSM descriptions of processes in a given system, an executable representation of the system functionality is constructed in an attributed grammar formalism [UrPr 83b]. This is in turn implemented as a PROLOG procedure. The details of the interaction model, the extended finite state machine and the attributed grammar based formalisms, logic programming implementations, and the techniques to carry out transformations between these formalisms are described in chapter 4. For now, assume that these transformation techniques have been carried out, resulting in a set of PROLOG procedures. These procedures

characterize in some sense, the permissible sequences of input and output interactions between the representation and its environment.

In the proposed approach, these PROLOG procedures are used as generators and validators of the system functionality described by the representations that they implement. That is, a PROLOG procedure implementing an abstract representation K (e.g., the design of a system) can be used

1) as a generator to obtain interaction sequences in which representation K participates during its execution. Specifically, it can be executed as a generator to produce

   i) all possible interaction sequences,

   ii) a set of all possible output and/or input interaction sequences observed at the interaction points of representation K for a given input (or output) interaction sequence

   iii) a set of all possible interaction sequences observable at the interaction points of representation K satisfying a certain functional property. This will be explained and illustrated in detail in chapter 5.

2) as a validator in recognizer or in acceptor mode to

check whether

a) observed interaction sequences obtained during the execution of a less abstract representation K+1 (e.g., an implementation of the system) are permissible

b) intended interaction sequences (e.g., test sequences) obtained by executing a more abstract representation K-1 (e.g., the specification of the system) are achievable.

Generation and validation of representative sets of observed and intended interaction sequences for system functionality representations are discussed in chapter 5 and 6, respectively. In fact, the proposed approach involves construction of invocation routines which can be used to automatically invoke PROLOG procedures corresponding to executable representations for the purposes of generating or validating observable interaction sequences. Algorithms for these invocation routines are also described in chapter 5 and 6.

In this approach, checking the consistency of representation K with respect to representation K-1 is based on comparison of their observable behaviors (i.e., the system functionality captured by representation K is checked against that of representation K-1). In general, assuming

that representation K-1 has already been validated, the following validation scenarios may be used for checking the consistency of representation K with representation K-1:

1) Obtain a selected set of execution histories of representation K (i.e., sequences of interactions in which representation K participates during its execution). Each execution history is a sequence of input stimuli and responses recorded during the execution of representation K. We call each such sequence a trace of representation K. Each trace of representation K corresponds to an instance of actual behavior of representation K which in turn corresponds to an instance of externally observable system behavior described by this representation. Since representation K-1 has already been validated, the system behavior described by representation K-1 can be considered as the expected behavior of representation K. That is, each trace of representation K can be checked for consistency with the expected behavior described by representation K-1. Therefore, apply each trace of representation K to representation K-1 to check whether it is permitted by representation K-1. This validation activity is called "trace checking".

2) Obtain a selected set of interaction sequences from representation K-1. This set contains intended (or

expected) sequences of interactions between representation K and its environment. We call each such interaction sequence a trajectory for representation K which corresponds to an instance of expected behavior (i.e., potential trace) of representation K. Then apply each trajectory to representation K to check whether it is actually allowed by representation K. We call this validation activity "trajectory checking".

In fact, this is a two-way complementary consistency checking method. Trajectory checking can be employed to check whether each representative expected behavior of representation K is actually realizable by representation K. The aim in this case is to reveal any expected behaviors that representation K fails to perform or performs incorrectly. On the other hand, trace checking can be employed to check whether each representative actual behavior of representation K is allowed by representation K-1. The aim in this case is to reveal any behavior of representation K that was not intended. As well, the actual limits and constraints that are implemented by representation K may be revealed during trace checking. In each of these cases, discrepancies between actual and expected behaviors of representations are reported for subsequent analysis and reconciliation. In the following section, we present specific consistency checking exercises

in a typical protocol development life-cycle [UrPr 84a].

3.3. Consistency Checking in Protocol Development Life-Cycle

Based on the definitions given in sections 3.1, validation activities in protocol development environments are aimed to check the consistency of:

(a) the service specification with respect to the intentions,

(b) the protocol specification with respect to the service specification,

(c) a protocol implementation with respect to the protocol specification.

3.3.1. Validating Service Specifications

Due to the absence of a higher level formal specification, checking the consistency of the N-service specification with respect to the intentions of the specifiers is an informal activity. Two directions can be taken:

a) The N-service specification may be executed to yield a representative set of execution histories (traces). This set can be inspected by the specifier manually; any discrepancies which are detected are analyzed and

reconciled. Where necessary the N-service specification can be corrected or augmented.

b) The specifier may formulate a small set of representative and discriminating trajectories which contain selected, intended sequences of interactions between the N-service provider and the N-service users. Each trajectory may then be (automatically) applied to the N-service specification to check that it is allowed by the N-service specification. Alternatively, input (or output) interactions alone may be applied to the N-service specification, and its reaction validated by the specifier.

3.3.2. Validating Protocol Specifications

Protocol Specification validation is defined as checking the consistency of a more refined specification (the N-protocol entities together with the (N-1)-service provider) against a more abstract specification of the N-service. This corresponds to validating the behaviour of the refined specification of the N-service provider which consists of two copies of the executable N-protocol specification interacting through an executable (N-1)-service specification. Thus, in each of the following scenarios, we are able to validate the participation of the N-protocol specification in the provision of the N-service.

a) The N-service specification is executed to generate a selective set of (possibly all representative) observable interactions as trajectories. Then, these trajectories are applied to the refined specification to check whether they are allowed by the cooperation of N-protocol specifications over the (N-1)-service specification.

b) The refined specification is executed to generate a representative set of traces. Then these traces are applied to the N-service specification to check whether they are permissible execution histories according to the N-service specification.

c) The same set of sequences of input stimuli are applied to the N-service specification and the refined specification, either one after the other ("off-line testing") or to both of them at the same time ("on-line" testing). In either case, each pair of corresponding observed interaction sequences are compared in order to detect any occurrence of discrepancies.

It should be noted that the N-protocol specification may not be in its final form. A final version of the N-protocol specification may be derived through a step-wise process involving a series of refinements from a high level abstraction to a detailed representation (c.f. section 4.3.2). The above scenarios may be employed at each step of

such a step-wise refinement to validate the design in its most recent form, and to compare the behavior of the most recent design to that of its predecessors.

3.3.3. Validating Protocol Implementations

Protocol implementation validation (or assessment) is to check the conformance of a given implementation of an N-protocol entity to the N-protocol specification. Protocol implementation assessment may be required basically in two distinct contexts:

a) in-house validation which is conducted by the developer who may have a direct access to both upper and lower interfaces of the N-protocol entity implementing the N-protocol specification.

b) certification which is conducted by an agency (possibly a recognized third party such as a certification centre) which may not have a direct access to the lower layer interface of the protocol implementation. We restrict ourselves to the first type of testing. Details and particulars of certification issues can be found in [Rayn 82, Boch 83], among others.

In the first context, the test sequences given in terms of upper and lower interface stimuli allowable by the protocol implementation and its responses are monitored at its

accessible interfaces. The conformance of the protocol implementation under test to the N-protocol specification is determined by checking that the observed interaction sequences are possible sequences according to the given specification. The incompatibility of the types and parameters of the interactions can be eliminated by using encoder-decoder modules as proposed in [UrPr 83a].

3.4. Basis for the Approach

In our approach, executable system functionality representations are described in attributed context free grammars. A context-free (CF) grammar G is a 4-tuple (VN,VT,P,S0) where

VN = set of non-terminals

VT = set of terminals

P = set of production rules

S0 = start symbol which is an element of VN

Note that each production rule is in the form of:

X0 --> X1 X2 .... Xn where

X0 is in VN and Xi is in VN U VT.

An attributed context-free grammar is a CF grammar G augmented by a set of semantic equations:

1) For each non-terminal S in VN, one associates a set A=Ai U As where

a) Ai is a set of data types associated with S which pass information to S. Each element of Ai is called an inherited attribute.

b) As is a set of data types associated with S which return information from S. Each element of As is called an synthesized attribute.

2) For each non-terminal S occurring in a production rule

X0 --> X1 X2 .... Xn where S is Xi for some i, one associates a set of semantic equations which define the attributes associated with S in the following manner:

a) For each inherited attribute i of S, there is a semantic equation i(Xi)=f(i(X0),s(X1),.....,s(Xn)) which defines the inherited attribute i of Xi in terms of the inherited attributes of X0 and synthesized attributes of some symbols occurring at the right hand side of the rule,

b) For each synthesized attribute s of S, there is a semantic equation s(X0)=f(s(X1),.....,s(Xn),i(X0)) which defines the synthesized attribute s of X0 in terms of its inherited attributes and synthesized attributes of some symbols occurring at the right hand side of the rule,

Attributed context-free grammars can be used to generate any Type 0 language [MiFi 79], and thus, theoretically, are applicable to any effectively-computable problem.

In general, including attributes in a grammar

a) makes explicit how information is passed and updated during the expansion of grammatical symbols. This property provides a means of interconnecting processes described as grammatical rules which exchange messages (cf. section 4.4),.

b) allows the possibility of the values of attributes to control the choice of grammatical rules and symbols via guards (i.e., enabling predicates) embedded into the grammar,

c) provides the capability of incorporating context-sensitive information (e.g., interaction parameters) and of encoding adaptations of various testing heuristics (e.g., boundary value analysis) into the grammar.

d) allows guiding the syntax-directed translation of the generated strings which is performed by computation rules (i.e., action routines). This translation process provides a means of generating input interaction sequences, output interaction sequences or sequences of both input and output interactions. Thus, both trace and trajectory generation as well as

test sequence generation can be supported by attributed grammars.

An attributed grammar, employed as a pure specification language, is independent of implementation constraints. It does not impose any direction as to how the system being described is implemented. Thus, a specification given in this formalism can be made free from implementation-specific decisions and details.

The attributed context-free grammar formalism provides an interface between our validation approach and a wide variety of specification methods used to describe system functionality representations. The power of attributed context-free grammars makes the translation from natural languages and extended FSM models rather easy [Prob 83]. Besides, this approach can be adapted to other specification techniques [Logr 84] such as the ones employing abstract data types [Logr 83] and Horn-clauses [Sidh 83].

In the next chapter, we demonstrate how our attributed grammar approach encapsulates specifications written in extended FSM's and how the approach can be embedded effectively in a straightforward subset of the logic programming language PROLOG.

4. "EXECUTABLE" SYSTEM FUNCTIONALITY REPRESENTATIONS

This chapter presents the construction of executable system functionality representations in an attributed context-free grammar formalism. In the first two sections, models underlying executable representations, namely, an interaction model and an extended FSM model are summarized. Following some examples of system functionality representations, an attributed context-free grammar formalism is presented and PROLOG implementations of grammar representations are discussed.

4.1. An Interaction Model

An interaction model characterizing distributed systems is given in [Boch 80, BoRa 82, JaBo 83]. In this model, each system is considered as a number of concurrent processes. Examples of such systems include the N-service provider (see Figure 4.1), the distributed implementation of the N-service provider (see Figure 4.2), etc. Each process in this model includes a number of interaction points through which it interacts with its environment (i.e., other processes). In addition, each interaction that a process participates in is associated with one of the interaction points of that process. In fact, an interaction between two processes takes place at a pair of corresponding interaction points (one in each interacting process) which are associated with that interaction. We call each pair of

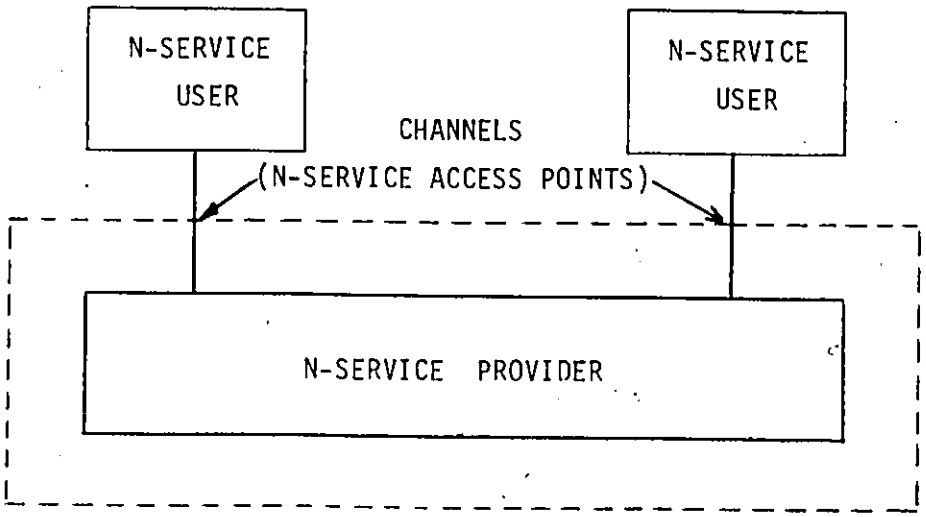


Figure 4.1 Interaction Model for an Abstract N-Service Provider

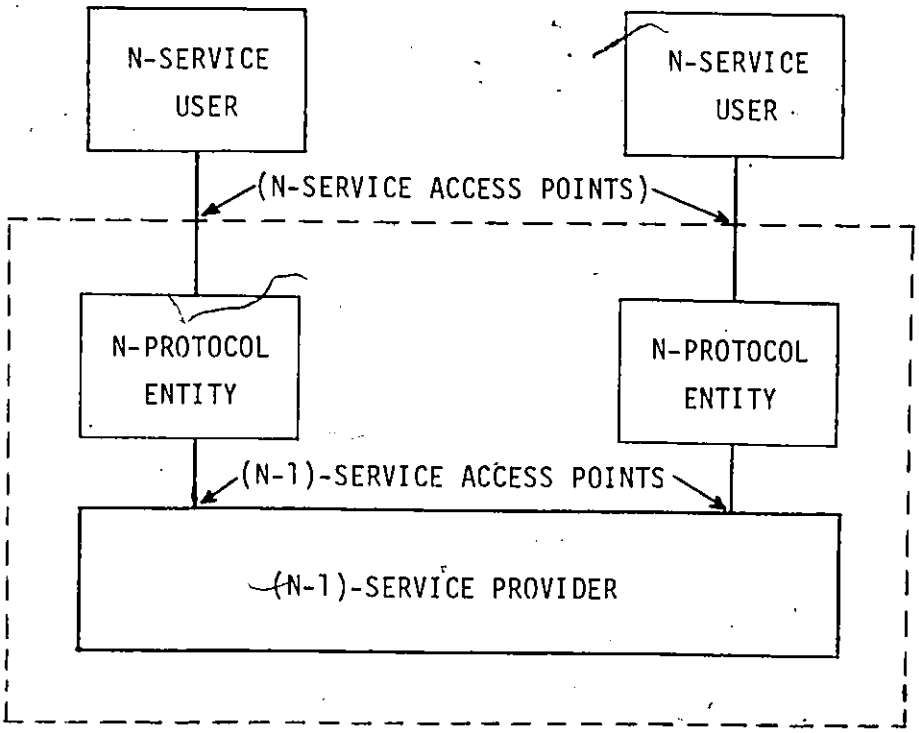


Figure 4.2 Interaction Model for a Refined N-Service Provider

corresponding interaction points between a pair of interacting processes a channel (e.g., an access point). An example of a channel is an (N-1)-service access point through which the (N-1)-service provider and an N-protocol entity interact (see Figure 4.2).

An interaction is characterized by a name, a number of parameters, and ranges of valid values for those parameters. An occurrence of an interaction between two processes involves a particular assignment of valid values to parameters and a transfer of this information from one process to the other over a particular channel between these processes. That is, an interaction transfers information only in one direction. The direction of information transfer taken by each interaction can be used to distinguish between interactions transferring information from and to a process: Interactions that transfer information from a process are considered as "output" interactions by that process whereas the ones transferring information to the process are considered as "input" interactions. As well, any interaction associated with a channel can be initiated (i.e., outputted) by only one of the two processes interacting over that channel. Hence, the interactions between two processes (e.g., process A and process B) over a channel can be classified into two groups; namely, those initiated by process A and those initiated by process B. Clearly, the interactions initiated by process A are considered as "input" interactions by process B and vice

versa.

The model allows for the queuing of outputs from one process before they are considered as inputs by the other. In general, infinite or finite (including zero) length queues are possible. A zero length queue implies that a "rendez-vous" type of interaction exists between processes: A process which initiates an (output) interaction at a particular interaction point waits for the other process to invoke the same interaction at the corresponding interaction point as an input interaction. The interaction between these two processes is realized when both processes are ready to invoke the interaction. Rendez-vous type interactions between processes may give rise to deadlocks which are difficult to resolve in a natural way [ISO 82].

Therefore, we assume that each input interaction is queued before it is considered by the process that inputs it. Hence, each process has an input queue at each of its interaction points: Each input interaction to a process is first put at the tail of the input queue associated with the interaction point at which the interaction reaches the process. An input interaction is considered by a process when it has moved to the head of the queue at which it was received.

Moreover, it is assumed that each interaction is atomic and parameter values included in an interaction are determined by the process that outputs the interaction.

We assume that only one interaction can take place at a given interaction point at any given time. In order to identify the interactions occurring at a given interaction point within a given sequence of observed interactions and to determine whether they are input or output interactions, some attributes such as interaction point number (e.g., service access point or connection end-point address), type (e.g., "i" for input, "o" for output interactions), etc. can be attached to interactions. Similarly, the relative order of interactions occurring at different interaction points of a process can be determined by the order of appearance of interactions in the given interaction sequence.

Furthermore, processes may have internal buffers to store some data that they receive through an input interaction before they send it to another process via an output interaction. These internal buffers may be represented by queues [Logr 83].

Based on this model, an extended FSM formalism is being developed in ISO and CCITT as a formal description technique for the specifications of communication services and protocols (e.g., FDT [ISO 82, BoRa 82]). In this formalism, a formal description of a process is given in terms of its externally observable behavior (i.e., possible types and execution orders of interactions of the process with its environment). In the following section, the main features of the extended FSM model are briefly summarized.

4.2. An Extended Finite State Machine Model

In this extended FSM formalism, a system functionality representation consists of specifications of one or more channel types followed by specifications of one or more process (called "module" in [ISO 83b]) types. In general, there may be more than one occurrence of a channel or a process type in a given system. For example, the system in Figure 4.1, consists of one occurrence of the process type "N-service provider" and two occurrences of the the channel type "N-service access point". On the other hand, the system in Figure 4.2, consists of two occurrences of the process type "N-protocol entity", one occurrence of the process type "(N-1)-service provider", and two occurrences of each of the channel types "N-service access point" and "(N-1)-service access point".

Each channel type corresponds to an abstract pair of related interaction points between two processes. Here, the concept of "role" is introduced in order to distinguish between two process types using the channel for their interactions. For example, at an N-service access point, the "user" role is played by an (N+1)-protocol entity and the "provider" role is played by the N-service provider. According to their roles, each process initiates or responds to certain interactions on the associated channel.

Briefly, the specification of a channel type includes a channel name, the roles of two process types, and an

enumeration of possible types of interactions (along with their parameters) initiated by each role player. For an example of a channel type specification, the reader may refer to section 4.3.1.

The specification of a process type consists of the specifications of the process "header" and the process "body". The specification of the process header includes the process name, the names of the channels corresponding to the interaction points of the process, and the roles that the process plays at each of those interaction points. For example, an N-protocol entity plays the role of a user at its interaction point corresponding to the (N-1)-service access point and plays the role of a provider at its other interaction point corresponding to the N-service access point.

The specification of process body is given in terms of an extended FSM. Accordingly, the possible orders of interactions of a process is described by the state-space of the process that defines all the states in which the process may be at any given time, and by the possible state transitions. The state-space of a process is specified by a set of variables. At any given time, the values of these variables uniquely determine the state of the process. One or more of these variables are called "major state variables" which correspond to basic functions or phases of the process such as data transfer, disconnect, etc. The

remaining variables are referred to as "additional state variables" which represent various attributes of the process such as, for example, "calling" or "called" user, etc. Thus, extensions to the FSM are defined in terms of additional state variables and interaction parameters whereas the finite state part of the machine is defined in terms of "major states" and transitions between those states. In brief, the specification of process body consists of declarations of major states and major state sets (i.e., the set of allowable values for major state variables), declarations of additional state variables, initializations of variables, declarations of functions and procedures to be used during execution of transitions, and an enumeration of state transitions corresponding to the possible orderings of interactions of the process.

A state transition in the extended FSM is defined by an enabling predicate which describes a condition in terms of variables and parameters and an action which may update some state variables and may generate some output interactions. Depending on whether a transition is initiated by the reception of an input interaction or not, transitions may be classified as follows:

1) input-initiated transitions which are initiated by some input interaction and may produce some output interactions,

2) internal transitions which are spontaneous and do not

produce output interactions,

3) output generating transitions which are spontaneous and produce some output interactions.

Accordingly, a general format for the representation of transitions in the extended FSM model is as follows;

```
FROM <present major state>
WHEN <input interaction>
ANY  <identifier>
PROVIDED <condition>
TO       <next major state>
BEGIN    <action> END;
```

The execution of a transition depends on the truth value of the associated enabling predicate which is any combination of the clauses FROM, WHEN or ANY, and PROVIDED, in conjunctive form. The transitions containing a WHEN clause are subject to execution when the specified input interaction arrives at the head of the queue at the specified interaction point. A transition containing an ANY clause is a spontaneous transition (i.e., either an internal transition or an output generating transition). Spontaneous transitions are usually included in process descriptions in order to represent some events such as failures that may occur any time. In fact, non-determinism in process descriptions is introduced by the existence of spontaneous transitions and/or the presence of more than one input-initiated transition for a specific input interaction at

some states. An ANY clause includes an identifier (variable) with an arbitrary value within the range that has already been defined. Such a clause represents the fact that the associated transition is defined for each legal value of that identifier.

When an enabling predicate is satisfied, the associated transition is executed; the major state of the machine becomes the major state specified in the TO clause and the action defined in the BEGIN clause is performed. The actions are usually defined in terms of some imperative statements which may involve updating some additional state variables, performing some predefined functions, and producing some output interactions.

More details and the syntax and semantics of the specification language used in this formal description technique can be found in [ISO 83b].

4.3. System Functionality Representations

4.3.1 An Example of System Functionality Representations

Consider the N-service provider shown in Figure 4.1. Here, each N-service user is associated with an N-service access point through which it interacts with the N-service provider. When an attempt is made to characterize the N-service provider in the interaction model given in section 4.1, each N-service access point is considered as a channel

between two processes; namely, the N-service provider and an N-service user. Accordingly, the N-service provider is represented as a process interacting with its environment (i.e., user processes) through a number of interaction points "connected" to the interaction points of user processes. In this model, the process representing the N-service provider (call it process P) can identify each user at any of its interaction points where it may initiate or respond to certain interactions.

As stated previously (see section 3.1), the N-service specification describes the N-service in terms of possible types and possible execution orders of interactions exchanged at N-service access points. More precisely, the N-service specification consists of:

1) possible types of interactions taking place at an N-service access point,

2) rules determining possible execution orders of interactions at N-service access points, including the desired end-to-end properties of the N-service.

Referring to the previous section, the description of process P is given in terms of its externally observable behavior. That is, the specification of process P consists of declarations of channels corresponding to interaction points of process P and an extended finite state machine (FSM) specification which defines the rules restricting the

possible orderings of interactions taking place at the interaction points of process P. Notice that the specification of process P refers to the specifications of channel types which describe the possible types of interactions that process P participates in. Accordingly, point 1) corresponds to the specification of the channel type "N-service access point", and point 2) corresponds to the specification of process P ("N-service provider").

Now let us give an example of how to construct an N-service specification by describing (the associated service provider) process P in the extended FSM model presented in section 4.2. The example is based on the draft transport service specification given in [ISO 81]. In order to simplify the example:

1) only a single transport connection between two users is considered,

2) quality of service parameters, various options, and expedited data transfer are not included,

3) declarations of constants, data types, and identifiers are omitted.

The transport service specification given in [ISO 81] consists of the specification of the channel type "TS_access_point" and the specification of the process type "TS" which corresponds to the transport service provider.

The specification of the channel type "TS_access_point" (TSAP) can be given as follows:

```
TS_access_point(TS_user,TS_provider);
   by TS_user:
      T_CONNECT_req(TCEPI           : TCEP_identifier_type;
                    to_T_address    : T_address_type;
                    from_T_address  : T_address_type;
                    QOTS_request    : quality_of_TS_type;
                    options         : option_type;
                    TS_connect_data: TS_connect_data_type);
      T_ACCEPT_req(*associated parameters*);
      etc.

   by TS_provider:
      T_CONNECT_ind(*same parameters as above*);
      T_DISCONNECT_ind(*associated parameters*);
      etc.

end TS_access_point;
where T     stands for TRANSPORT
      TS        "     "  TRANSPORT SERVICE
      TCEPI     "     "  TRANSPORT CONNECTION ENDPOINT ID
      QOTS      "     "  QUALITY OF TS PARAMETERS
```

Explanation of TSAP: The roles of the two players interacting at this channel are indicated in parentheses after the name of the channel as "TS_user" and "TS_provider". For each role, the interactions that may be initiated by the role player are listed along with their parameters in a BY clause. Accordingly, the first BY clause contains interactions initiated by the "TS_user" (which are inputs to the process playing the role of "TS_provider"). The interactions given in the second BY clause are initiated by the "TS_provider" (and thus are outputs of the process playing the role of "TS_provider").

The specification of process TS is given as follows:

```
process TS (AP1,AP2:  TS_access_point (TS_provider));
* Major states and major state sets declarations *
state1, state2:  (idle, wait_for_acc, data_transfer,disconnect);
 .
 .
 * Additional state variable declarations*
 .
 .
* initialization *
state1:=idle;
state2:=idle;
 .
 .
* procedure and function declarations *
 .
 .
* Transitions *
from (state1=idle and state2=idle)
     when AP1.T_CONNECT_req (... parameters ...)
          provided (no_congestion)
          to (state1=wait_for_acc and state2=idle)
          begin
            TCEP1 :=TCEPI;
            caller:=from_T_address;
            called:=to_T_address;
            options:=requested_options;
            connect_data:= TS_connect_data;
            buffer21.clear;
            buffer12.clear;
          end;
 .
 .
```

Explanation of TS: The specification of process TS starts with a declaration of the process header. In the process header, the names of the channels corresponding to the interaction points of the process TS and the role that the process TS plays at these channels are given as AP1, AP2, and TS_provider, respectively. Thus, process TS is the player of the role "provider" at AP1 and AP2 which are the names of two occurrences of the channel type TS_access_point. Referring back to the specification of

channel type TS_access_point, it is understood that process TS (denoted TS_provider) outputs the interactions enumerated in the second BY clause and inputs the interactions given in the first BY clause.

The process header is followed by the process body. Here, the major state of process TS is defined as a pair (statel,state2) where statel and state2 are two state variables which stand for the state of process TS as seen at AP1 and AP2, respectively. The major state set, following the declaration of the major state variables, defines the possible values of state variables. In addition, an example state transition between the major states(idle,idle) and (wait_for_acc,idle) is given above. This transition occurs when a T_CONNECT_req is received by process TS at the transport service access point AP1 and when process TS is able to provide the service (i.e., when there is no congestion). The actions associated with the transition are naming the calling and called users, setting the options as requested, and clearing the buffers associated with each transport service access point. The complete example is given in Appendix A. The interested reader may also refer to [ISO 81].

4.3.2. Refinement of System Functionality Representations

The step-wise refinement of distributed systems is supported by the interaction model described in section 4.1.

A system characterized by this model may be refined by defining its structure in terms of concurrent subsystems and their interconnections through channels. Refinements of each subsystem may be obtained in a similar way; defining its subcomponents and the interconnections between these subcomponents.

As an example, consider the N-service provider given in Figure 4.1. A refinement of this system is given in Figure 4.2. This refinement corresponds to a functional decomposition of the N-service provider into two N-protocol entities and an (N-1)-service provider (for details, see section 3.1). Each N-protocol entity may be further refined by defining its subcomponents and their interconnections [ISO 83b]. Similarly, a refinement of the (N-1)-service provider can be constructed so as to correspond to its distributed implementation with the (N-2)-service provider and two (N-1)-protocol entities, and so on. In fact, the above step-wise refinement is the specification philosophy underlying the ISO Reference Model [Zimm 80].

A parallel argument can be given for the step-wise refinement of system functionality representations (as opposed to systems themselves). In the preceding section, a system functionality representation is constructed by considering the system (e.g., N-service provider) as a single process (i.e., as a "black box") and by describing its behavior in terms of the possible types and execution

orders of interactions in which it participates. A refinement of the above system functionality representation may be based on a system refinement corresponding to a functional decomposition of the system which is given in terms of subsystems and their interconnections. A refined system functionality representation can then be constructed by describing each subsystem and their interconnections. Following the example given in the previous section, a refined N-service specification can be constructed by the N-protocol specification and the (N-1)-service specification. Assuming that the (N-1)-service specification already exists (which is often the case in practice), this refinement step reduces to the construction of an N-protocol specification.

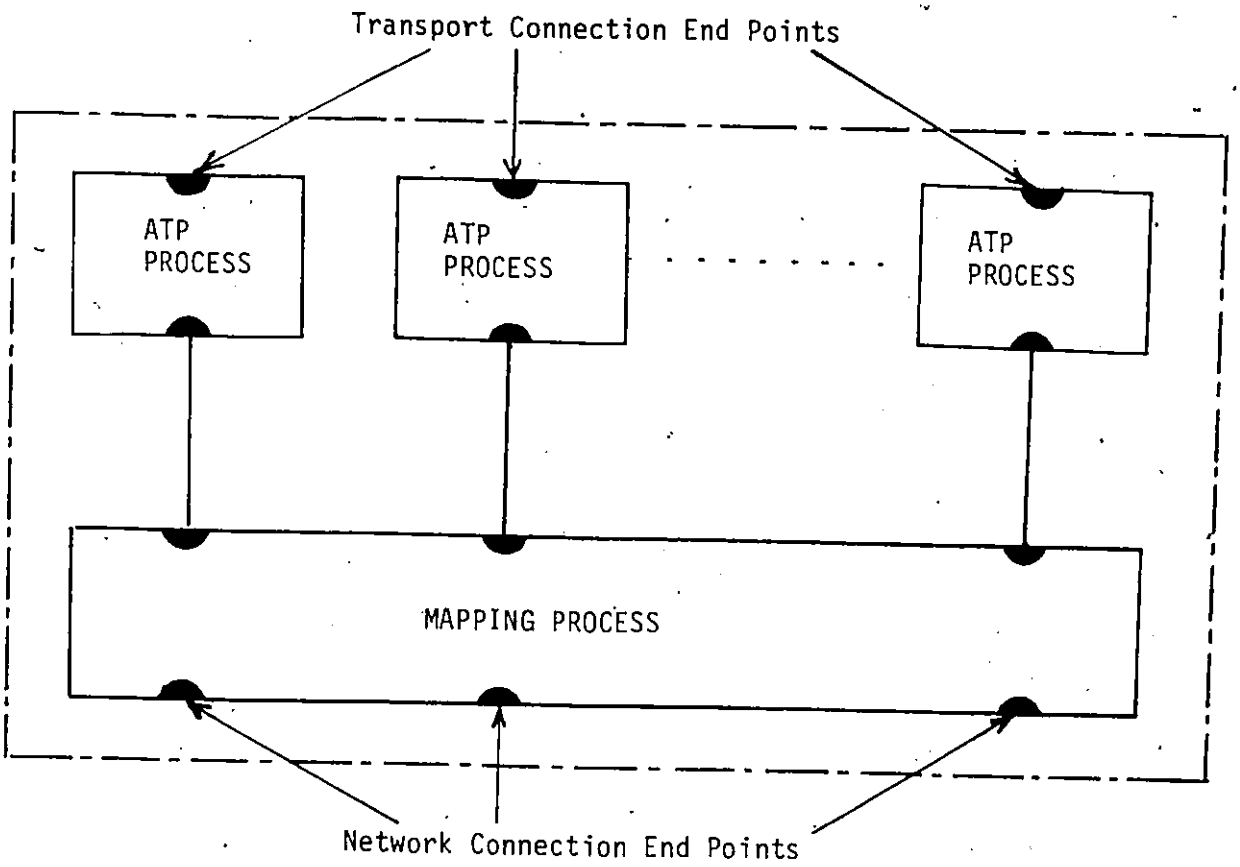


Figure 4.3 Structure of a Transport Protocol Entity

A transport protocol specification, based on a substructure of a transport protocol entity (shown in Figure 4.3) is given in [ISO 83a]. The transport protocol entity in Figure 4.3 supports an arbitrary number of transport connections over an arbitrary number of transport service access points (TSAP's). The protocol entity uses an arbitrary number of network connections at an arbitrary number of network service access points (NSAP's). The behavior of this transport protocol entity is specified in the transport protocol specification given in [ISO 83a]. In this specification, a transport entity is considered to be composed of one "mapping" process and an arbitrary number of "abstract protocol" processes (called ATP). The assumption is that there is an ATP process for each transport connection end point at each TSAP.

Thus, the given protocol specification consists of separate specifications of an ATP process, a "mapping" process, and the interconnection between these process types. The specification of the ATP process type defines the rules restricting the possible orderings of interactions (transport service primitives) with a transport service user and transport protocol data units (PDU's) exchanged with a peer ATP process at another site. These rules describe the behavior of a transport protocol entity in terms of transport service primitives exchanged at TSAP's and transport PDU's exchanged with a peer protocol entity through the network service provider. The specification of

the "mapping" process determines the mapping between transport PDU's and the network service data units. Further details of the specification can be found in [ISO 83a].

As pointed out previouslv (in section 3.1), step-wise refinement results in a new system functionality representation given in a lower level of abstraction. In general, these differences between the levels of abstraction concentrate on the following points:

a) Global rules (e.g., end-to-end properties of the desired service) are specified in detail in more refined representations.

b) The internal channels between subsystems or subcomponents of subsystems in a more refined system functionality representation have no counterpart in a more abstract representation. An example of internal channels is an (N-1)-service access point between the (N-1)-service provider and an N-protocol entity.

c) The channel types defined in a more abstract representation and the interactions taking place at those channels may also be refined in a more detailed representation. Such refinements include specifications of parameter types and ranges of possible values for those parameters, etc.

Hence, step-wise refinements tend to lead to more and more detailed representations of the desired system

functionality, converging finally to an implementation of the system.

4.4. An Attributed Grammar Based Formalism

It has been claimed that the extended FSM formalism briefly summarized in section 4.2 lends itself to the construction of executable specifications [JaBo 83]. However, at the time of writing this thesis, it is not clear how the system structure and subsystem structure (i.e., the interconnections of subsystem descriptions and the descriptions of occurrences (instances) of process types and channel types) are defined in this formalism [ISO 83b]. For example, it is not clear how an executable specification of a refined N-service provider is constructed within this formalism by interconnecting two occurrences of the N-protocol specification with an occurrence of the (N-1)-service provider specification as shown in Figure 4.2.

In this thesis, the executable system functionality representations are described in an attributed context-free grammar formalism. In this formalism, each executable representation is constructed by using the extended FSM based descriptions of process types and of associated channel types given in communication service and protocol specifications. Briefly, an executable representation is constructed as a grammar defining all possible orderings of input and output interactions of the system via the possible

tours of the state space of the global system status. The global system status is determined by the status values of all the processes in the system. An integral component of such a grammar is the definitions of all possible single global system status changes. Each possible global system status change corresponds to a state transition in one of the processes in the system.

The use of attributed context free grammars presents some advantages: In extended FSM models, the transition to be executed at a state where more than one possible state transition exists is not determined by the extended FSM formalism [ISO 83b]. Normally, the choice of which transition to execute is made randomly. This is the case when one considers only the syntax of an attributed grammar; the grammar merely shows what choices are possible for right hand sides of production rules, but it does not specify how this choice is made. However, in attributed grammar formalisms, the choice of which alternative term to use when expanding a non-terminal can be guided by the user via attributes [DuHu 81]. Hence, it is possible to guide the execution of system functionality representations described in attributed grammar based formalisms. In addition, attributes provide mechanisms for:

1) incorporating context sensitive information (e.g., state variables, interaction parameters in extended FSM representations) into the executable system

functionality representations

2) making explicit how information is passed during the execution of representations and therefore, observing the external behaviour of these representations. As well, the interconnections of process descriptions are based on this information passing capability provided by attributes. This point is discussed in more detail in section 4.4.2.

3) encoding adaptations of various testing heuristics such as boundary value analysis, equivalence partitioning [Myer 79], error-based testing [Weyu 81], and functional testing [Howd 80] into test sequence specifications.

The specification notation that we will use in this thesis is described by the extended BNF notation below:

```
grammar          ::= production_rule {production_rule}.
production_rule  ::= nonterminal ["("attributes")"]
                     ":="
                     expression
                     ".".
expression       ::= term {"|" term}.
term    ::= factor {factor}.
factor  ::= nonterminal ["("attributes")"]
          | terminal ["("attributes")"]
          | "["expression"]"
          | "("expression")"
          | "{"expression"}".
nonterminal ::= identifier.
terminal    ::= literal.
attributes  ::= attribute {attribute}.
attribute   ::= identifier | literal.
identifier  ::= letter {["_"] letter_or_digit}.
letter_or_digit ::= letter | digit.
literal         ::= "'" string_of_characters "'".
string_of_characters ::= character {character}.
```

```
character          ::= letter_or_digit | "_".
```

```
|   alternative
[ ] optional       i.e., [a] stands for empty | a
{ } repetition     i.e., {a} stands for empty | a | aa | ....
( ) factorization  i.e., (a(b|c)) stands for ab|ac
```

Figure 1. An Extended BNF Notation

The attributes pertaining to a grammatical symbol are enclosed in parantheses following the symbol. As in the usual attributed grammars [Lewi 76], attributes can be inherited (to pass information to a grammatical symbol) or synthesized (to return information from a symbol). Each synthesized attribute of the left hand side non-terminal of a production rule must be assigned a value in the right hand side of the production rule. It is assumed that when a synthesized attribute is assigned a value in a term in the right hand side of a production rule, all occurrences of that attribute are instantiated to the same value in that rule. Similarly, during the expansion of a production rule, all occurrences of an inherited attribute in the rule are instantiated to the same value.

We introduce guards in the attributed grammar formalism to allow values of attributes to dynamically control (to guide) the choice of rules during the expansion of non-terminals. The capability of controlling the choice of rules through values of attributes was introduced by Milton [MiFi 79] with attributed LL(k) grammars. A similar effect is accomplished by allowing each grammatical term to start

with an optional guard [DuHu 81] as follows:

```
term  ::=  [ guard ]  factor { factor }.
```

Each guard is a boolean expression (cf. section 4.4.1) which may involve inherited attributes of the left hand side non-terminal of a production. For example, the following guard corresponds to an enabling predicate which holds when the input interaction I received at "state1" is "T_CONNECT_req" and there is congestion in the system:
"STATE EQ "state1" AND I EQ "T_CONNECT_req" AND congestion"
It is understood that the rest of the term is evaluated when the guard associated with that term holds. Otherwise, the evaluation of the term terminates.

Moreover, we introduce computation rules to correspond to actions performed during the execution of a transition in the extended FSM formalism. Computation rules should be placed immediately after the guard (if any) in a term. Accordingly, the definition of a factor can be rewritten as follows:

```
factor  ::=  computation_rule
           |  nonterminal ["(" attributes ")"]
           |  terminal ["(" attributes ")"]
           |  "["expression"]"
           |  "("expression")"
           |  "{"expression"}".
where a computation_rule is defined as:
computation_rule ::= identifier ["(" attributes ")"]
                   |  assignment_statement.
```

Accordingly, a computation_rule can be an identifier (followed by an optional list of attributes) which may

correspond to a procedure name or can be an assignment statement which may assign a value to a synthesized attribute. For example, the quard given in the above example may precede the following computation rules:

output("T_DISCONNECT_ind") NEXTSTATE = "statel"

The first computation rule is a procedure call which invokes procedure "output" to generate an output interaction (e.g., T_DISCONNECT_ind) and the second rule is an assignment statement which defines the next state.

Procedures referred to in productions as computation rules are in fact action routines which provide mechanisms for defining specific actions taken during the execution of corresponding transitions. An important usage of procedures as computation rules may be handling of interaction parameters, namely defining ranges of valid values for parameter fields, or only those values which are on the boundaries of allowable range of values, specifying invalid values for parameters such as the values outside the allowable range, and assigning or checking the values of parameter fields. For example, consider the following definition of a procedure, denoted "parameter":

```
parameter(VALUE, 'valid')    ::= VALUE = 1
                               | VALUE = 10
                               | VALUE GE 1 AND VALUE LE 10.

parameter(VALUE, 'invalid') ::= VALUE = 0
                               | VALUE = 11
```

```
| VALUE LE 0 OR VALUE GE 11.
```

Procedure "parameter" defines both valid and invalid values of a parameter, say X. The range of valid values of X is defined as $1 \leq X \leq 10$ and thus invalid values of X are those which are outside of this range (i.e., $X>10$ or $X<1$). Accordingly, the first definition of procedure "parameter" specifies the valid values of X whereas the second definition specifies the invalid values of X. Notice that both definitions consist of two assignment statements and a boolean expression. The assignment statements in the first definition assigns two barely allowable values (i.e., 1 and 10) as the valid values of X. These values are on the boundaries of allowable range of valid values and are called "on-by-one" values [Fost 80]. On the other hand, the assignment statement in the second definition of procedure "parameter" assigns two invalid values (i.e., 0 and 11) which are just outside the allowable range of X. These values are called "off-by-one" values [Fost 80]. In fact, this is an implementation of "boundary value analysis" testing strategy [Myer 79]. That is, instead of generating all possible valid and invalid values, this testing strategy aims to construct test cases considering only those values which are on the boundaries and off the boundaries of valid range of values.

Thus, procedure "parameter" returns "1" or "10" and "0" or "11" as valid and invalid values of parameter X when it

is invoked by "parameter(X,'valid')" and "parameter(X,'invalid')", respectively. Moreover, procedure "parameter" can be used to check whether a given value of X is valid or invalid. For example, the invocation "parameter(5,Y)" returns Y as 'valid' while the invovation "parameter(0,Y)" returns Y as 'invalid'.

Note that it is also possible to selectively assign any on-by-one or off-by-one values to X via employing an additional attribute in conjunction with guards in definitions of procedure "parameter". For example, consider the following modified definitions of the procedure:

```
parameter(VALUE, 'valid', SPECIFIER)  ::=
          SPECIFIER EQ 'lower' OR 'both'  VALUE = 1
        | SPECIFIER EQ 'upper' OR 'both'  VALUE = 10
        | VALUE GE 1 AND VALUE LE 10.

parameter(VALUE, 'invalid', SPECIFIER) ::=
          SPECIFIER EQ 'lower' OR 'both'  VALUE = 0
        | SPECIFIER EQ 'upper' OR 'both'  VALUE = 11
        | VALUE LE 0 OR VALUE GE 11.
```

Via the attribute SPECIFIER, one can select either the lower boundary value, the upper boundary value, or both of these values as the returned values of parameter X.

Similar definitions of parameter fields of interactions can be employed in constructing executable system functionality representations. However, for simplicity of the presentation, we do not include the specifications of

interaction parameters in our examples, except the one given in section 5.2.2.

In the following sections, we present the syntax and semantics of executable system functionality representations described in the attributed context free grammar formalism. We call these system functionality representations executable representations. These representations are "testable" in the sense that they can be executed and their external (i.e., input/output) behaviour is observable. Being testable, these representations provide a basis for validation activities in a protocol development environment: In particular, their consistency with respect to their more abstract predecessor representations (that they are derived from) can be checked during their executions. Moreover, test sequence (or in the more general context, trace and trajectory) generators or trace and trajectory validators may be constructed by employing these executable representations (cf. sections 5.3 and 6.1).

4.4. The Syntax of Executable Representations

In the proposed attributed context free grammar formalism, an executable system functionality representation is described as a grammar defining all possible orderings of externally observable input and output interactions via the possible tours of the state space of the global system

status. Here, the global system status is defined as the combination of the status values of all the processes in the system. The status of each process is determined by the state of the process and the contents of its internal buffers and input (output) interaction queues.

An integral part of this grammar defines all possible single global system status changes. In fact, each global system status change corresponds to a change in the status of one of the processes. This requires definitions of all possible single status changes of each process which are constructed from extended FSM based process descriptions by identifying the state transition in the extended FSM corresponding to the change in the status of the process.

The general syntax of executable system functionality representations can be described by the following grammar using the notation given in Figure 4.4:

```
executable_system_functionality_representation "("attributes")"::=
   single_global_status_change "("attributes")"
   executable_system_functionality_representation "("attributes")"
 | empty.

single_global_status_change ::= process_description
                                {process_description}.
process_description ::= process_header ":=" process_body.

process_header      ::= process_name
                        "("
                          [input_interaction]
                          [output_interaction]
                           present_status
                           next_status
                        ")".
process_name        ::= identifier.
input_interaction   ::= interaction.
output_interaction  ::= interaction.
present_status      ::= status.
```

```
next_status          ::= status.
interaction          ::= interaction_id
                         interaction_parameter
                        {interaction_parameter}.
interaction_id           ::= identifier | literal
interaction_parameter    ::= identifier | literal.
status                   ::= interaction_queues
                             states
                             internal_buffers.
interaction_queues       ::= input_interaction_queue
                             output_interaction_queue
                             {interaction_queues}.
input_interaction_queue  ::= {input_interaction}.
output_interaction_queue ::= {output_interaction}.
states               ::= state {state}.
state                ::= identifier | literal.
internal_buffers     ::= {internal_buffer}.
internal_buffer      ::= {data_item}.
data_item            ::= identifier | literal.

process_body         ::= transition  {"|" transition}.
transition           ::= [enabling_predicate]  factor {factor}.
enabling_predicate   ::= boolean_expression.

boolean_expression ::= boolean_term  {"OR" boolean_term}
                     |  literal_expression
                        relational_operator
                        literal_expression.
boolean_term         ::= boolean_factor  {"AND" boolean_term}.
boolean_factor       ::= boolean_identifier
                     |  boolean_function
                     | "("boolean_expression")"
                     | "NOT" boolean expression.
boolean_function     ::= identifier ["("attributes")"].
boolean_identifier   ::= ".true." | ".false.".

factor               ::= identifier ["("attributes")"]
                     |  computation_rule
                     | "("process_body")"
                     | "["process_body"]"
                     | "{"process_body"}".
computation_rule     ::= identifier ["("attributes")"]
                     |  assignment_statement.
assignment_statement  ::=identifier "=" literal_expression.
literal_expression    ::= identifier | literal.
relational_operator   ::= "LT" | "LE" | "EQ"
                        |  "GT" | "GE" | "NE".
identifier  ::= letter  {[" "] letter_or_digit}.
letter_or_digit ::= letter | digit.
literal          ::= "'" string_of_characters "'".
string_of_characters ::= character  {character}.
character            ::= letter_or_digit | "_".
attributes  ::= attribute  {attribute}.
attribute   ::= identifier | literal.
```

The following translation rules are applied when constructing system functionality representations in the attributed grammar formalism from those given in the extended FSM formalism. To aid his understanding, the reader may wish to follow the translation from Appendix A (FSM) to Appendix B (Attributed Grammar).

1) Each description of an occurrence of a process type is given a unique name and is represented as a production rule which consists of alternative terms.

2) Each term corresponds to a single change in the status of the process which is caused by a state transition defined in the extended FSM based process description. Each term usually consists of a guard and computation rules.

3) A guard corresponds to the enabling predicate and a computation rule corresponds to the action of the associated transition, respectively.

4) A guard may define a predicate in terms of certain attributes corresponding to an input interaction, input/output queues, and various state variables including the major state variables. The input interactions appearing in the guards as terminal symbols are the ones given in channel type descriptions as the interactions responded to (inputted) by the process being described.

5) A computation rule can be used to assign values to some synthesized attributes including the next major state and various additional state variables, to update input/output queues , and to define the output interaction. The output interactions appearing in computation rules as terminal symbols are the interactions indicated as initiated (outputted) by the process being described in the associated channel type descriptions.

6) Each evaluation or expansion of a production rule defining all possible single changes of the status of the process corresponds to an execution of a transition in the corresponding process.

4.4.2. The Semantics of Executable Representations

The semantics of executable system functionality representations are described as follows:

1) Each representation consists of a number of process occurrence descriptions Pij where j(=1,2,...,m) stands for the occurrence number of process type i(=1,2,...,n). For example, Figure 4.5.a describes a system functionality representation which is composed of two occurrences of process description P1 (i.e., P11 and P12) and one occurrence of process description P2 (i.e., P21). Assuming that this is a

refined representation of the transport service provider, P11 and P12 correspond to occurrences of the transport protocol specification and P21 corresponds to an abstract representation of the network service provider.

2) Each Pij refers to a number of channel occurrences Cpq where q(=1,2,...,y) stand for the occurrence number of channel type p(=1,2,...,z). Continuing with the above example, C11 and C12 are two occurrences of channel type C1 which corresponds to a TSAP, and C21 and C22 are two occurrences of channel type C2 which corresponds to an NSAP.

3) Each channel Cpq is associated with at least one Pij. If a channel Cpq is referred to by only one Pij, it is called an external channel. For example, channels C11 and C12 (corresponding to TSAP's) in Figure 4.5.a are external channels. That is, each external channel connects two processes, one of which is not described in the system functionality representation (e.g., session protocol entities). On the other hand, if a channel is referred to by exactly two processes, it is called an internal channel. That is, it is a channel connecting two processes which are described in the system functionality representation. For example, channels C21 and C22 that correspond to NSAP's are internal channels in the refined

representation of the transport service provider (Figure 4.5.a).

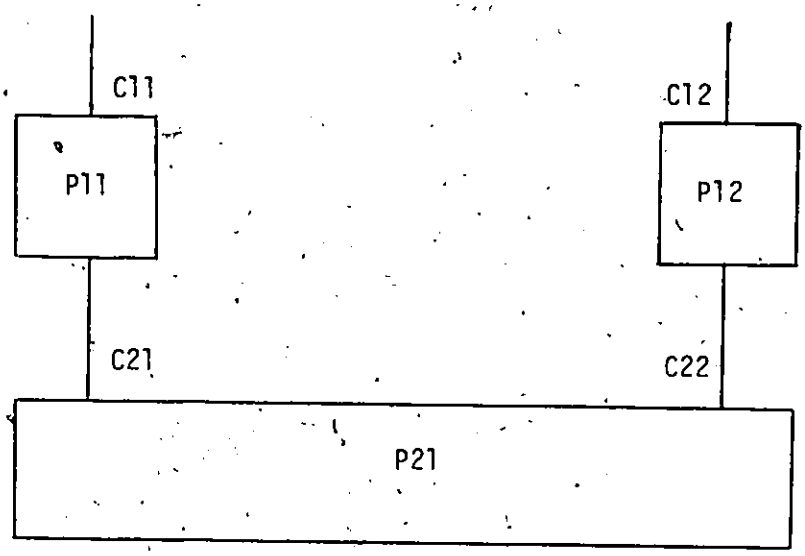


Figure 4.5.a Process Descriptions and Channels

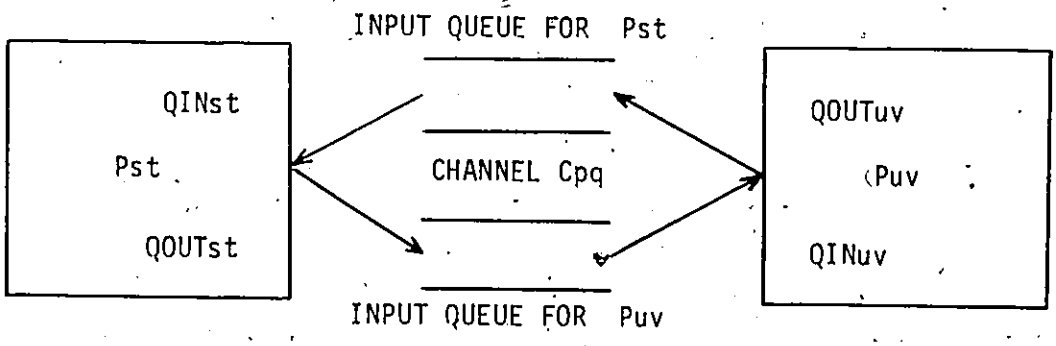


Figure 4.5.b Channels Between Processes

4) Each reference made in a Pij to a channel Cpq is associated with two queues; an input and an output interaction queue. For example, each reference to channel Cpq in process Pst is associated with the input interaction queue QINst and the output interaction queue QOUTst (see Figure 4.5.b). Recall that each interaction point at which a process interacts with another process is associated with an input queue (see section 4.2). Hence, each channel is represented in our model by a pair of interaction queues; one for each direction of information transfer as in Figure 4.5.b. The convention is that the input queue at the interaction point of process Puv is the output queue at the interaction point of process Pst, and vice versa. Thus, in order to observe the interactions between these two processes:

i) the input interaction which is considered by a process is represented by the inherited attribute "input-interaction". The value of this attribute is determined by the process when it is ready to execute a production corresponding to an input initiated transition. The process chooses an input queue and gets the interaction at the head of that queue as the input interaction.

ii) the output interaction which may result during the expansion of a production rule is placed at the

rear of the corresponding output queue. This output interaction becomes the value of the synthesized attribute "output-interaction".

5) Two process occurrences are interconnected by two queues as shown in Figure 4.5.b. Thus, the interconnections between descriptions of two process occurrences are provided by the attributes associated with input and output interaction queues.

6) At any instant of time, a Pij is considered to be either in a single state or executing a transition (i.e., between states). The status of a Pij is considered to be indeterminate if it is executing a transition. The status of a Pij is determined by the values of state variables and contents of queues associated with the Pij. The latter includes all input and output queues at the interaction points of the Pij and all those queues corresponding to internal buffers.

7) The global status of the system is determined by the status of all Pij when none is executing a transition.

8) The global status of the system changes as a result of a change in the status of one of the Pij's. Thus, the grammar is intended to specify all possible orderings of externally observable system

interactions through all possible global system status tours. The grammar includes the definitions of all possible single global system status changes as production rules. All possible single global system status changes are defined in terms of possible single changes in the status of processes.

4.4.3. Two Examples of Executable Representations

In general, there may be more than one interaction point through which an executable representation interacts with its environment. Accordingly, an observed or expected interaction sequence of an executable representation consists of input and output interactions initiated at those interaction points. The representation that we use for interaction sequences involves an augmentation of interactions with special attributes to explicitly indicate the interaction point at which the interaction is initiated and whether the interaction is an input or an output interaction. Specifically, we employ the following conventions:

i) an identifier immediately preceding an interaction denotes the interaction point associated with that interaction,

ii) "i" or "o" immediately preceding the interaction point identifier denotes that the interaction is an

input or ouput interaction, respectively. Whenever it is obvious from the context, for the sake of simplicity, "i" and "o" are omitted.

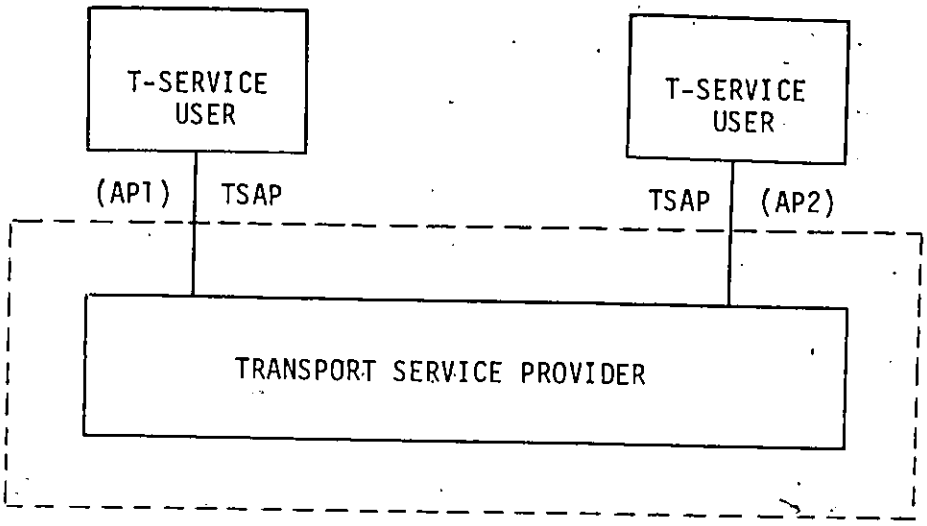


Figure 4.6 Interaction Model for Transport Service Provider

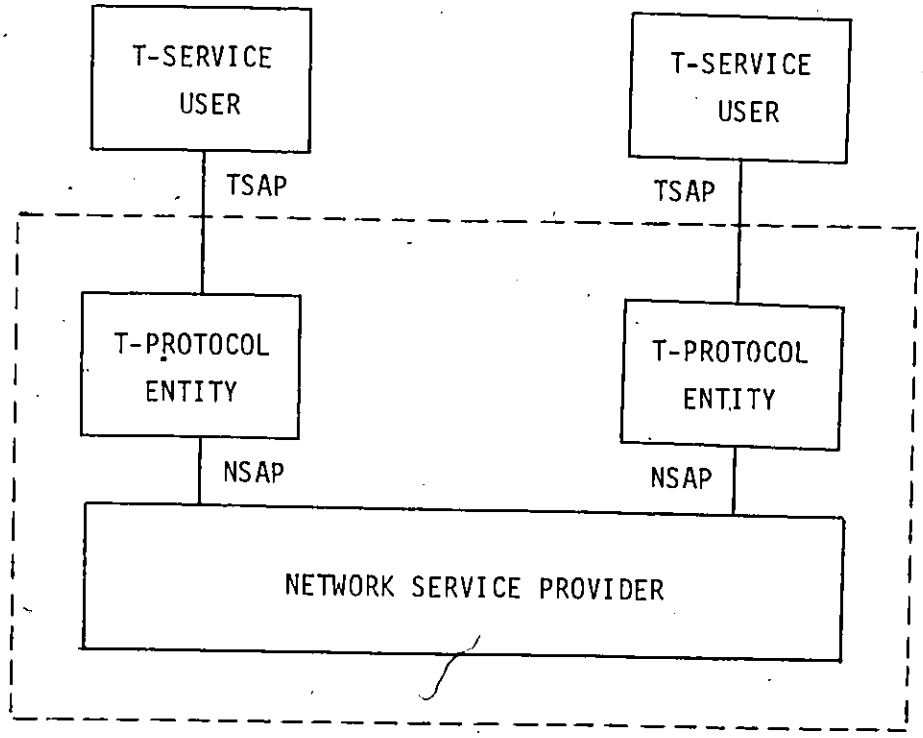


Figure 4.7 Interaction Model for Refined Transport Service Provider

As an example, the following interaction sequence of an executable transport service specification represents reception of a connection request at AP1 and its subsequent rejection received at AP2:
iAP1_T_CONNECT_req,oAP2_T_CONNECT_ind,
iAP2_T_DISCONNECT_req,oAP1_T_DISCONNECT_ind.

Example 1: Transport Service Specification

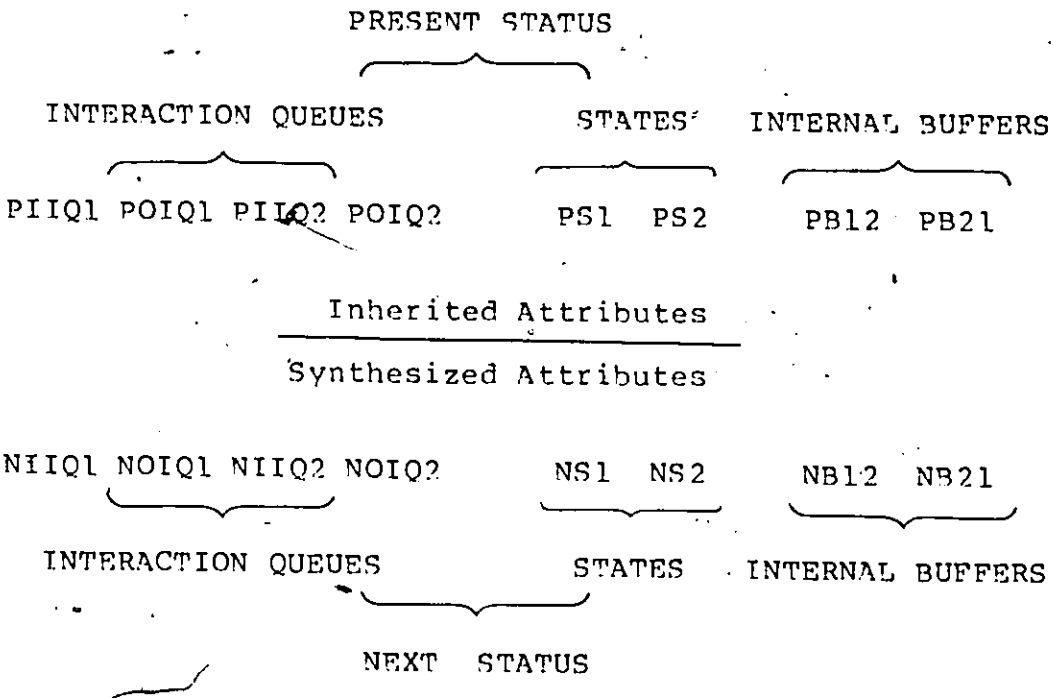
Consider the transport service specification in the extended FSM formalism given in Appendix A. Following the transformation rules in section 4.4.1, the corresponding attributed grammar representation (Appendix B) is constructed as follows:

a) The representation consists of only one process description; namely, process "tservice" corresponding to transport service provider shown in Figure 4.6. This representation specifies all possible orderings of externally observable interactions of the transport service provider via all possible sequences of status changes. On the other hand, the non-terminal "ts" defines all possible single status changes in the transport service provider. Accordingly, representation "tservice" is defined by the following grammar:

```
tservice(INITIALSTATUS) ::=
        ts(INITIALSTATUS, NEXTSTATUS).
```

```
                continue(NEXTSTATUS).
continue(PRESENTSTATUS) ::=
        PRESENTSTATUS EQ "predefined final status"
        empty
      | ts(PRESENTSTATUS, NEXTSTATUS)
        continue(NEXTSTATUS).
```

Here, each attribute PRESENTSTATUS and NEXTSTATUS is in fact a list of attributes:



That is,

1) IIQ1, OIQ1 and IIQ2, OIQ2 are input and output interaction queues associated with the transport service access points AP1 and AP2 respectively,
2) S1 and S2 are major state variables,
3) B12 and B21 are queues corresponding to internal buffers used to transfer user data items from AP1

(AP2) to AP2 (AP1) respectively.

b) Each expansion of a production in the grammar corresponds to the execution of a single transition defined in the corresponding extended FSM representation. It is important to note that there are five possible ways that the contents of the input-output interaction queues may be changed during the execution of a single transition in the given transport service specification. In fact, this is exactly the case in the corresponding extended FSM representation; i.e., each transition is associated with only one of the transport service access points. These cases are enumerated in the following table:

IIQ1	OIQ1	IIQ2	OIQ2
AP1		AP2	
XX	XX	--	--
XX	--	--	--
--	XX	--	--
--	--	XX	--
--	--	--	XX

-- stands for no change

XX stands for a change in the contents of the corresponding queue

Since each transport connection is assumed to be initiated at the access point AP1, we do not have an entry in the table corresponding to the case where there is a change in both IIQ2 and OIQ2. Thus, we can group the transitions (or status changes in the representation) into five classes corresponding to the above cases. Notice that each of these transitions are either input initiated transitions or output generating transitions (cf. section 4.2). If there was an internal transition (which is a spontaneous transition that does not produce any output interaction) then that transition would be considered in a group of transitions which would not cause any change in interaction queues.

The status changes corresponding to the first two transition groups can be written as follows:

```
ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
   NIIQ1,NOIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21) ::=
   PS1 EQ 'idle' AND PS2 EQ 'idle' AND
   is_firstof(PIIQ1,'ap1,tcreq,param') AND congestion
   restof(PIIQ1,NIIQ1) I='ap1,tcreq,param' input(I)
   O='ap1,tdind,provider' lastof(POIQ1,NOIQ1,O) output(O).

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
   NIIQ1,POIQ1,PIIQ2,POIQ2,NS1,NS2,NB12,NB21) ::=
   PS1 EQ 'idle' AND PS2 EQ 'idle' AND
   is_firstof(PIIQ1,'ap1,tcreq,param') AND NOT(congestion)
   restof(PIIQ1,NIIQ1) I='ap1,tcreq,param' input(I)
```

```
NS1=wait_accept  NS2=PS2    actions_1(NB12,NB21)

PS1 EQ 'data_transfer' AND PS2 EQ 'data_transfer' AND
is_firstof(PIIQ1,'ap1,tdatr,DATA') AND
flow_control_to_tpentity_ready
restof(PIIQ1,NIIQ1) I='ap1,tdatr,DATA' input(I)
NS1=PS1 NS2=PS2 actions_2(PB12,PB21,NB12,NB21,DATA)
```

Here, the first transition group contains only one transition which updates both the input and the output interaction queues at AP1. This transition occurs when both major state variables PS1 and PS2 are instantiated to idle, the input interaction at the head of PIIQ1 is tcreq (which is checked by the boolean function is firstof), and the boolean function congestion is true. The actions associated with the execution of the transition are to construct the new contents of input and output queues and to record both input and output interactions. Notice that neither the states nor the contents of internal buffers are changed.

In the first status change defined in the representation, the procedure restof returns the synthesized attribute NIIQ1, which is the new content of the input interaction queue, as the rest of the content of inherited attribute PIIQ1. The procedure lastof adds the output interaction tdind to the rear of the content of POIQ1 to obtain the

content of NOIQ1 and procedures input and output print the corresponding input and output interactions.

The second transition group correspond to transitions that are initiated by an input interaction at AP1 and that do not produce any output interaction. The complete grammar can be found in Appendix B.

Example 2: Refined Transport Service Provider

Consider the refined transport service provider shown in Figure 4.7. An executable representation for this system can be constructed by using the network service specification and two (possibly identical) transport protocol specifications. The construction process involves two steps: The first step is to construct definitions of status changes in each process in the system by utilizing the descriptions of processes in the extended FSM formalism. The second step is to interrelate these three sets of definitions (call them "process descriptions") through attributes corresponding to input and output interaction queues.

STEP_1:

For the sake of simplicity, the construction of process descriptions are based on the following assumptions:

a) The network service specification is given similar to the transport service specification in Appendix A.

Hence, the process description nsp which corresponds to the definitions of changes in the status of the network service provider can be derived similar to non-terminal ts given in Appendix B. Accordingly, in process description "nsp", the status of the corresponding network service provider is determined by the contents of the input and output interaction queues IIQ1, OIQ1 and IIQ2, OIQ2 at two network service access points, by the values of two major state variables S1 and S2, and by the contents of internal buffers B12 and B21. Hence, each status change is represented in the process description "nsp" by a production rule as it was done in the previous example. For example, a typical nonterminal at the left hand side of these rules is:

nsp(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21)

b) The transport protocol specification is given in the extended FSM formalism [ISO 83a] (cf. section 4.3.2). However, the transport protocol specification is rather detailed to be included in this section. Therefore, we briefly outline a process description corresponding to an oversimplified version of the transport service specification. The process, denoted tpe, is assumed to support only one transport connection at a time and to be associated with only two interaction points; namely, a transport service

access point (TSAP) and a network service access point (NSAP). Furthermore, process "tpe" is assumed to involve a major state variable, two pairs of input-output interaction queues (one pair at the TSAP and another at the NSAP), and two internal buffers (to store user data to or from the local transport service user). Accordingly, the status of process "tpe" is determined by the contents of the input and output interaction queues TIQ,TOQ and NIQ,NOQ at the TSAP and NSAP respectively, the value of the major state variable T, and the contents of internal buffers BFU (buffer to store data from user) and BTU (buffer to store data to user). Again, as in the transport service specification, one can write the status changes in process "tpe" in terms of production rules in the grammar. These production rules can be classified in terms of the changes made to the input and output interaction queues during the evaluation of these rules. Considering the case where the input interactions are processed sequentially by process "tpe" (i.e., no two input interactions are considered simultaneously) there may be nine possible ways that the contents of the input and output interaction queues may be changed during the expansion of a production rule. These cases are enumerated in the following table:

TIQ	TOQ	NIQ	NOQ
TSAP		NSAP	
XX	XX	--	--
XX	--	--	--
XX	--	--	XX
--	XX	--	--
--	XX	XX	--
--	--	XX	XX
--	--	XX	--
--	--	--	XX
--	--	--	--

-- stands for no change

XX stands for a change in the contents of the corresponding queue

For example, the first entry in the above table stands for all those transitions which involve an input and a corresponding output interaction at the TSAP, whereas the last entry corresponds to internal transitions (i.e., spontaneous transitions that do not produce any output interaction). Thus, all possible single status changes in process "tpe" can be defined by a set of production rules corresponding to a non-terminal "tp". Call this set of production rules as process description "tp".

A typical nonterminal at the left hand side of these production rules is:

tp(PTIQ,PTOQ,PNIQ,PNOQ,PT,PBFU,PBTU,
NTIQ,NTOQ,NNIQ,NNOQ,NT,NBFU,NBTU)

Since two such process descriptions are required in the refined representation, we duplicate the process

description "tp" and call these process descriptions "tp1" and "tp2". Accordingly, the process headers for these two process descriptions become:

```
tp1(PTIQ1,PTOQ1,PNIQ1,PNOQ1,PT1,PBFU1,PBTU1,
    NTIQ1,NTOQ1,NNIQ1,NNOQ1,NT1,NBFU1,NBTU1)
and
tp2(PTIQ2,PTOQ2,PNIQ2,PNOQ2,PT2,PBFU2,PBTU2,
    NTIQ2,NTOQ2,NNIQ2,NNOQ2,NT2,NBFU2,NBTU2)
```

STEP_2:

As stated previously (cf. section 4.4.2), interconnections between process descriptions are provided by attributes corresponding to input and output queues. In our example, process descriptions "tp1", "tp2", and "nsp" are interconnected as shown in Figure 4.8.

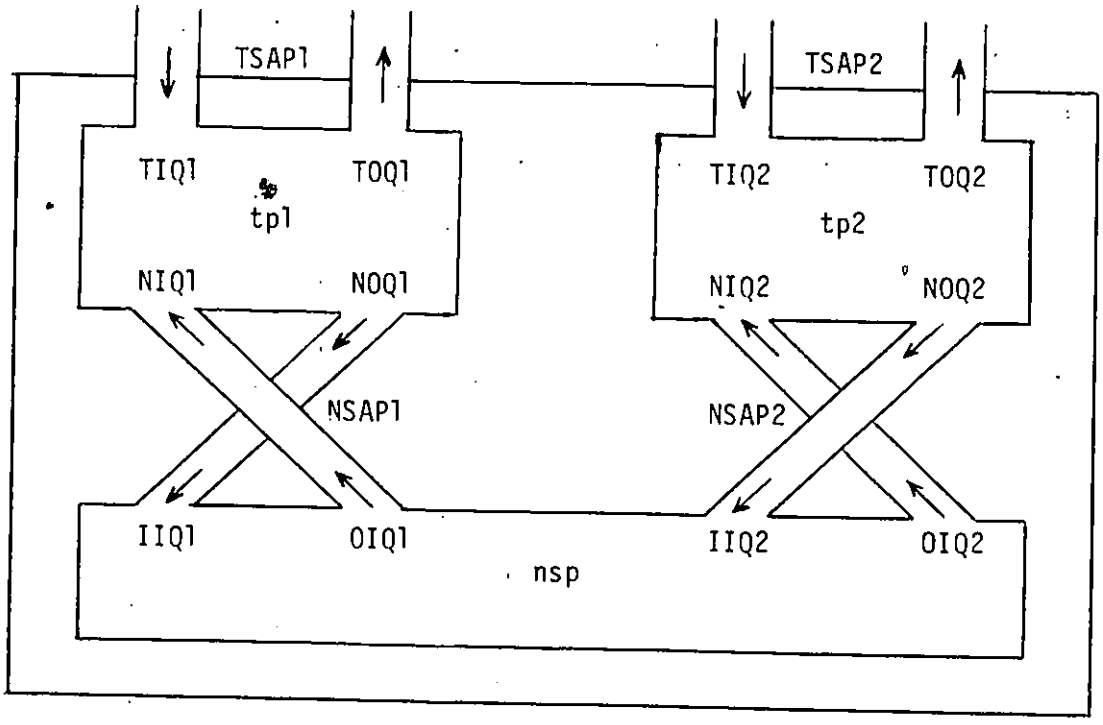


Figure 4.8 Interconnections Between Process Descriptions

It is clear from Figure 4.8 that, the input interaction queues IIQ1 and IIQ2 of process description "nsp" are the output interaction queues NOQ1 and NOQ2 of process descriptions "tp1" and "tp2", respectively. Similarly, the output interaction queues OIQ1 and OIQ2 of process description "nsp" are the input interaction queues NIQ1 and NIQ2 of process descriptions "tp1" and "tp2" at the corresponding NSAP's.

Therefore, one can use either of the equivalent queues. In this example, we choose to use IIQ1,OIQ1,IIQ2, and OIQ2 for each occurrence of NOQ1,NIQ1,NOQ2, and NIQ2, respectively in the process descriptions "tp1" and "tp2". Hence, process headers for these two process descriptions become:

tp1(PTIQ1,PTOQ1,POIQ1,PIIQ1,PT1,PBFU1,PBTU1,
NTIQ1,NTOQ1,NOIQ1,NIIQ1,NT1,NBFU1,NBTU1)

tp2(PTIQ2,PTOQ2,POIQ2,PIIQ2,PT2,PBFU2,PBTU2,
NTIQ2,NTOQ2,NOIQ2,NIIQ2,NT2,NBFU2,NBTU2)

Recall that the global system status is determined in a system functionality representation by the status values of all processes described in the representation (cf. section 4.4.2). Thus, the global status of the refined transport service provider which is denoted the refined tservice is given in terms of contents of all major state variables, input and output interaction queues (without replicating those which are interconnecting two process descriptions),

and internal buffers used in all three process descriptions.

The present global status and the next global status of "refined_tservice" is given in terms of the following attributes:

PRESENTSTATUS

(PTIQ1,PTOQ1,PT1,PBFU1,PBTU1,
PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
PTIQ2,PTOQ2,PT2,PBFU2,PBTU2)

NEXTSTATUS

(NTIQ1,NTOQ1,NT1,NBFU1,NBTU1,
NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21,
NTIQ2,NTOQ2,NT2,NBFU2,NBTU2)

Hence, given the process descriptions "nsp", "tp1", and "tp2" as above, representation "refined_tservice" specifies all possible orderings of externally observable interactions via possible global status tours. The non-terminal "rts" defines all possible single global status changes in the refined transport service provider. Accordingly, representation "refined_tservice" is defined by the following grammar:

```
refined_tservice(INITIALSTATUS) ::=
        rts(INITIALSTATUS,NEXTSTATUS)
            continue(NEXTSTATUS).
continue(PRESENTSTATUS) ::=
        PRESENTSTATUS EQ "predefined final status"
        empty
```

```
      | rts(PRESENTSTATUS,NEXTSTATUS)
                  continue(NEXTSTATUS).
rts(PTIQ1,PTOQ1,PT1,PBFU1,PBTU1,
    PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
    PTIQ2,PTOQ2,PT2,PBFU2,PBTU2,
    NTIQ1,NTOQ1,NT1,NBFU1,NBTU1,
    NIIQ1,NOIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
    PTIQ2,PTOQ2,PT2,PBFU2,PBTU2) ::=
tp1(PTIQ1,PTOQ1,POIQ1,PIIQ1,PT1,PBFU1,PBTU1,
    NTIQ1,NTOQ1,NOIQ1,NIIQ1,NT1,NBFU1,NBTU1).

rts(PTIQ1,PTOQ1,PT1,PBFU1,PBTU1,
    PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
    PTIQ2,PTOQ2,PT2,PBFU2,PBTU2,
    PTIQ1,PTOQ1,PT1,PBFU1,PBTU1,
    PIIQ1,POIQ1,NIIQ2,NOIQ2,PS1,PS2,PB12,PB21,
    NTIQ2,NTOQ2,NT2,NBFU2,NBTU2) ::=
tp2(PTIQ2,PTOQ2,POIQ2,PIIQ2,PT2,PBFU2,PBTU2,
    NTIQ2,NTOQ2,NOIQ2,NIIQ2,NT2,NBFU2,NBTU2).

rts(PTIQ1,PTOQ1,PT1,PBFU1,PBTU1,
    PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
    PTIQ2,PTOQ2,PT2,PBFU2,PBTU2,
    PTIQ1,PTOQ1,PT1,PBFU1,PBTU1,
    NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21,
    PTIQ2,PTOQ2,PT2,PBFU2,PBTU2) ::=
nsp(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
    NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21).
```

Notice that the above grammar includes three alternative production rules for single global status changes. That is, global status of "refined_tservice" changes if the status of any of its component processes changes.

4.5. Logic Programming Implementation

4.5.1. Logic Programming

A logic program P is a set of universally quantified first order axioms in the form of H <-- B1,B2,...,Bm where H and Bi's are primitive goals. Such a clause reads as "H if B1 and B2 and...and Bm". H is called the head of the clause and the conjunction of B's is called the body of the clause. A computation of P is the construction of a proof (succession of deductions) of a given existentially quantified conjunctive goal A1,A2,...,An. The computation can have two results; success or failure. If the computation succeeds the resulting values of attributes in the given conjunctive goal constitute the output of computation.

The computation is thought of as progressing via non-deterministic goal reductions. At each step, the computation has a current goal set A1,A2,...,An. A subgoal Ai (for $1 \leq i \leq n$) is arbitrarily chosen, then a clause H <-- B1,B2,...,Bm (m>0) is arbitrarily selected from P for which Ai and H are unifiable via a substitution S (cf. [Kowa 79]). Finally, this clause is used to reduce the goal set to

(A1,...,Ai-1,B1,...,Bm,Ai+1,...,An)S. The computation terminates when the current goal set is empty.

According to this computation model, different orderings of clauses in P or subgoals in the clauses or in the given goal set do not affect the result of the computation [ApMe 82]. Practical logic programming languages, however, provide only approximations to this computational model. For the sake of simplicity and efficiency, these languages decrease the degree of non-determinism involved in the computation process. This decrease is accomplished by specifying both the order in which goals are reduced and the specific clauses used in the reduction. For example, one logic programming language, sequential PROLOG (e.g., DEC system 10 PROLOG), searches for a proof in a top-down left-to-right manner: The order in which goals are reduced is left-to-right (i.e., the chosen subgoal is always the leftmost in the conjunction of goals), and the order in which specific clauses are considered in the reduction is top-down (i.e., the choice of unifiable clause is accomplished by sequential top-down search and backtracking). [Colm 78, Pere 78].

4.5.2. PROLOG Implementations

Attributed grammar based system functionality representations can be coded as PROLOG procedures by representing each production rule in the grammar by a clause in the PROLOG procedure [UrPr 83b]. Hence, alternative

production rules describing possible expansions of a non-terminal symbol in the grammar are enumarated as alternative clauses in the corresponding PROLOG procedure. This translation allows us to use PROLOG interpreters as "run time" systems for the execution of attributed grammar based representations. Here, we emphasize that PROLOG is employed as a tool providing several capabilities:

a) The top-down left-to-right search mechanism employed by PROLOG interpreters provides a suitable expansion mechanism for executing the attributed grammars corresponding to system functionality representations. That is, during the generation of an instance of the start symbol, a possible expansion of a non-terminal is searched for in a top-down fashion among the alternative production rules, whereas the evaluation of the expression on the right hand side of the chosen production rule progresses in a left-to-right manner (e.g., from the guard to the computation rule and so on).

b) Since PROLOG interpreters augment the search mechanism with a backtracking capability [Colm 78], all possible alternative rules for the expansion of a non-terminal can be tried one after the other in a grammar coded in PROLOG. This capability may be utilized, for example, to generate all possible instances of the start symbol [UrPr 83b]. Notice

that no augmentation of PROLOG encoding of attributed grammars is required in order to implement this sequential search and backtracking capability. Although the mechanisms used by PROLOG interpreters are fixed, there are some provisions to modify the this search and backtracking behavior [Colm 78].

c) Augmentation of attributed grammars with additional attributes, in conjunction with guards and/or computation rules, can be directly implemented in corresponding PROLOG procedures [Prob 83, PrUr 82a, PrUr 82b]. Additional attributes embedded in PROLOG procedures as parameters, provide the capability of guiding the execution of these procedures. That is, one may guide the choice of alternative clauses via the values of attributes. This, in turn, allows searching clauses systematically (e.g., trying each alternative clause one after the other) or selectively (e.g., according to particular requirements of a test generation strategy), and so on.

d) PROLOG supports both forward (from input to output) or backward (from output to input) "execution" of procedures implementing a complete relation between input and output. This capability allows a PROLOG procedure, implementing a process description given in the attributed grammar formalism, to be executed

as a generator, recognizor, or transducer.

As a simple example, consider the following PROLOG encoding of procedure "parameter" (cf. section 4.4):

```
parameter(VALUE,"valid")   :- VALUE = 1
                            | VALUE = 10
                            | VALUE >= 1 , VALUE =< 10.

parameter(VALUE,"invalid") :- VALUE = 0
                            | VALUE = 11
                            | VALUE =< 0
                            | VALUE >= 11.
```

The above PROLOG encoding of procedure "parameter" can be executed:

1) as a transducer

   a) to generate all on-by-one and off-by-one values of parameter X via following invocations:

```
?- parameter(X,"valid").
which yields
X=1;
X=10.

?- parameter(X,"invalid").
which yields
X=0;
X=11
```

b) to check whether a given value of X is in the range of valid values for X. For example, the invocation

```
?- parameter(3,Y).
yields
Y=valid
```

or the invocation

```
?- parameter(15,Y).
yields
Y=invalid
```

2) as a generator to produce, for example, all on-by-one values of parameter X together with the indicator which points out that those values are in the range of valid values of X by the following invocation:

```
?- parameter(X,Y).
which yields
X=1
Y=valid;
X=10
Y=valid
```

3) as a recognizer to check the validity of assertions in the form of "the value A of parameter X is (or is not) in the range of valid values of X". For example, the invocation

```
?- parameter(3,"valid").
```

succeeds, but the invocation

```
?- parameter(15,"valid").
```

fails

or the invocation

```
?- parameter(3,"invalid").
```

fails, but the invocation

```
?- parameter(15,"invalid").
```

succeeds.

Here, "succeeds" and "fails" refer to the cases where the procedure "parameter(X,Y)" recognizes the given combination of X and Y values as an acceptable combination or not acceptable, respectively. We make use of this capability in constructing trace or trajectory generators and validators (cf. section 5.3 and 6.1 respectively).

4.5.2.1. PROLOG Encoding of Executable Representations

A PROLOG procedure, denoted tservice, implementing the executable representation developed in section 4.4.3 (given in Appendix B), is presented in Appendix C. The one-to-one correspondence between the attributed grammar representation and the associated PROLOG encoding is obvious. Therefore, we will not attempt to describe this PROLOG encoding. Instead, we will point out various ways that this implementation can be utilized in our validation approach:

a) Procedure "tservice" can be executed as a generator for all possible interaction sequences represented by

the corresponding grammar: These interaction sequences may include both input and output interactions or input interactions alone. Recall that if an interaction sequence obtained from a representation consists of both input and output interactions, then that sequence is called either a trajectory or a trace, depending on whether the corresponding representation has already been validated or not, respectively. As well, an interaction sequence consisting of input interactions alone is generally called a test sequence [BoCe 82]. This completely automated execution of procedure "tservice" requires that all predicates such as "congestion", "internal_problem", "no_internal_problem", etc. should hold whenever encountered. That is, all such predicates should be declared as clauses in part of procedure "tservice", as given in Appendix C. Moreover, predicates "restof" and "lastof" are not required for generation of interaction sequences. (Recall from section 4.4 that these operations are required when "tservice" is interconnected to other PROLOG procedures implementing some process descriptions such as service layer entities). Therefore, definitions of "restof" and "lastof" in Appendix C are such that they hold whenever they are encountered regardless of the contents of input and output interaction queues.

Furthermore, predicates "input" and "output" simply print the corresponding input or output interaction. The definitions of predicates "input" and "output" in Appendix C correspond to the case where "tservice" is executed to generate all possible input and output interaction sequences (i.e., traces or trajectories).

Notice that if it is desired to generate all possible input interaction sequences alone, the only modification in procedure "tservice" is to avoid printing output interactions. This can be achieved by changing the definition of predicate "output" such that it succeeds whenever it is encountered without printing the output interaction. Thus, the definition of predicate "output":

```
output(O) :- print("o"), print(O).
```

is modified as:

```
output(_).
```

which succeeds whenever it is called.

It is important to note that procedure "tservice", as it stands in Appendix C, generates all possible interaction sequences starting from a given initial system status and ending at a predefined final system status (i.e., when both state variables PS1 and PS2 are assigned to "idle"). However, procedure "tservice" can be easily generalized to generate interaction sequences between any two system status

values as follows:

```
tservice(STATUS, STATUS).
tservice(PRESENTSTATUS, FINALSTATUS):-
        ts(PRESENTSTATUS, NEXTSTATUS),
        tservice(NEXTSTATUS, FINALSTATUS).
```

where predicate "ts" defines all possible single changes in the the status of the corresponding transport service provider. Thus, the invocation "tservice("a system status","another system status")" generates all possible interaction sequences between the two given system status values.

Notice that the new definition of "tservice" can be simply added to the corresponding PROLOG program instead of replacing the existing definition of predicate "tservice". This provides the capability of utilizing the rest of the predicates in the program by either definition of predicate "tservice". Hence, the PROLOG program can be executed to generate all possible interaction sequences that occur either between a given initial system status and a predefined final system status or between any two given system status values. However, if it is so desired, one can easily remove the previous definition of predicate "tservice" and have only a unique definition of this predicate which is the definition given above.

It is obvious that the new definition of predicate "tservice" can also be used to analyse whether a given system status can be reached from another given system status. However, the reachability analysis related issues are outside the scope of this thesis. Interested reader may refer to [Logr. 84].

b) Procedure "tservice" can be augmented with additional predicates and attributes to selectively generate either input or both input and output interaction sequences:

As an example, consider the following predicates:

```
atservice(IP,FCTPR,FCUR,C_NC,
          PRESENTSTATUS,FINALSTATUS) :-
          set_params(IP,FCTPR,FCUR,C_NC)
          tservice(PRESENTSTATUS,FINALSTATUS).

set_params(IP,FCTPR,FCUR,C_NC) :-
    (A=internal_problem, B=no_internal_problem,
        (IP=0   ,(A -> retract(A); true)
                ,(not(B) -> assert(B); true)
        ;IP=1   ,(not(A) -> assert(A) ; true)
                ,(B -> retract(B); true))),

    (C=flow_control_to_tpentity_ready,
        (FCTPR=0,(C -> retract(C); true)
        ;FCTPR=1,(not(C) -> assert(C); true))),

    (D=flow_control_to_user_ready,
        (FCUR=0 ,(D -> retract(D); true)
        ;FCUR=1 ,(not(D) -> assert(D); true))),

    (E=congestion, F=no_congestion,
        (C_NC=0 ,(E -> retract(E); true)
                ,(not(F) -> assert(F); true)
        ;C_NC=1 ,(not(E) -> assert(E); true)
                  ,(F -> retract(F); true))).
```

In this example, an interface to procedure tservice is provided by predicate atservice which allows user guidance in generation of interaction sequences. Users may specify whether predicates:

"internal-problem", "flow-control-tpentity-ready"

"flow-control-user-ready", and "congestion"

should hold or not via the values of attributes IP, FCTPR, FCUR, and C_NC respectively. Depending on the choices specified by the user, predicate "set-params" either adds definitions of these predicates into the PROLOG program or removes (if they exist) from the program. Accordingly, for example, set-params(0,1,1,0) will add the following predicate definitions into the PROLOG program:

```
no_internal_problem.
flow_control_tpentity_ready.
flow_control_user_ready.
no_congestion.
```

Other examples on augmenting executable representations can be found in section 5.2.

c) Excluding the definitions of predicates "tservice" and "continue", the rest of the PROLOG program in Appendix C, denoted procedure ts, describes all possible changes in the status of the corresponding transport service provider. An important property of procedure "ts" is that it can be interconnected to

other process descriptions via its input and output interaction queues. An example of interconnecting process descriptions is given in section 4.4.3.

4.5.2.2 Modifications of PROLOG Encodings

As stated previously, the main function of input and output interaction queues is to interconnect process descriptions. That is, these queues provide required interfaces between process descriptions in a system functionality representation as well as system interfaces <1> between the resulting system functionality representation and its environment. For example, the queues IIQ1, OIQ1, IIQ2, and OIQ2 provide interprocess interfaces between process descriptions "tp1", "tp2", and "nsp", and the queues TIQ1, TOQ1, TIQ2, TOQ2 provide a system interface between the resulting refined description of transport service provider (i.e., procedure "rts") and its environment (Figure 4.8). The queues providing a system interface which may in turn be used to interconnect the system functionality representation with other process descriptions in its environment. For instance, continuing with the above example, the queues TIQ1, TOQ1, TIQ2, TOQ2 can be used to interconnect the refined representation of transport service provider with PROLOG procedures implementing session layer

---

<1> A system interface is an interface where externally observable interactions take place.

entities to form a refined session service provider.

However, if a system functionality representation is executed without being interconnected to other process descriptions (in its environment), a simpler system interface may suffice. Such a system interface may be provided by a pair of attributes corresponding to input and output interactions for each interaction point between the representation and its environment. A direct implementation of this type of interface is accomplished by inserting additional attributes to the list of attributes in clauses corresponding to status changes in the representation. For example, consider the first ts clause in Appendix C. The additional system interface may be implemented as follows:

```
ts(I,O,void,void,
   PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
   PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21):-
   PS1=idle, PS2=idle,
   is_firstof(PIIQ1,"ap1,tcreq,parameters"),
   congestion,
   restof(PIIQ1,NIIQ1), I=[ap1,tcreq,parameters]
   O=[ap1,tdind,provider], lastof(POIQ1,NOIQ1,O)
   input(I), output(O).
```

Here note that, this clause is related to an input and a corresponding output interaction at TSAP "ap1" while no interaction, denoted by "void" (or "v" in Appendix C), is occurring at the other TSAP (i.e., "ap2"). When this new system interface is employed during the execution of the

corresponding system functionality representation (i.e., procedure "ts"), predicates "is_firstof", "restof", and "lastof" should not perform their usual functions. Their definitions should be adjusted accordingly; i.e., should succeed whenever encountered (cf. section 4.5.2.1).

The same type of system interface may also be implemented by replacing the input and output interaction queues at each interaction point between a system functionality representation and its environment by a pair of attributes corresponding to single input and output interactions. Considering the same example as in the previous case, the ts clause is modified as follows:

```
ts(I,O,void,void,
   PS1,PS2,PB12,PB21,PS1,PS2,PB12,PB21):-
   PS1=idle, PS2=idle, congestion,
   I=[ap1,tcreq,parameters]
   O=[ap1,tdind,provider].
```

Here note that all occurrences and definitions of predicates "is_firstof", "restof", "lastof", "input", and "output" are no longer needed.

This simpler interface provides a capability of executing system functionality representations with a given sequence of input (and output) interactions. Via this interface, system functionality representations may be embedded into higher level PROLOG procedures implementing

trace or trajectory generators and validators. These procedures apply a given interaction sequence (e.g., a trace, a trajectory, or a test sequence) to the representation. The details of trace or trajectory generators and validators are given in section 5.3 and 6.1 respectively. In fact, examples of trace and trajectory generation and validation are given using procedure "ts" with this simpler interface. Note that, procedure "ts" which represents all possible status changes in the above transport service provider is given in Appendix D.

5. GENERATION OF REPRESENTATIVE SYSTEM BEHAVIOR

Consider an executable system functionality representation which describes the system functionality in terms of possible types and execution sequences of interactions exchanged between the system and its environment. The system functionality captured by such a representation may be revealed by observing its external behaviour (i.e., sequences of interactions) during its execution.

Potentially, all possible externally observable behaviour of representations may be generated by executing PROLOG programs implementing the corresponding attributed grammars (cf. section 4.5.2.1). Often, the set of all possible interaction sequences is intractably large due to the existence of

a) infinitely many possible iterations of certain loops in the corresponding representation such as the one in data transfer phase in the transport service specification [ISO 81],

b) large number of combinations of valid values for parameter fields in input and output interactions,

c) variations of parameter fields in input and output interactions, etc.

Hence, checking the consistency of representations using all possible interaction sequences (corresponding to all possible externally observable behaviour patterns of these representations) is practically infeasible [Piat 80]. A practical compromise which attempts to reduce the size of the set of interaction sequences used in consistency checking is that of selecting interaction sequences randomly (e.g., [LiMc 83]). However, random (test sequence) selection has been criticized from the viewpoints of completeness [Prob 82] and effectiveness [Myer 79] for software testing purposes.

Thus, a practical consistency checking exercise in protocol development environments is confined to detecting particular instances of inconsistencies between representations by using a manageably sized representative and discriminating set of interaction sequences. In such a set each interaction sequence is considered to correspond to a number of functionally equivalent sequences in the set of all possible interaction sequences and is aimed to check whether the corresponding function is correctly implemented or not.

In this chapter, we discuss approaches to generating representative and discriminating sets of interaction sequences (e.g., test sequences, traces, or trajectories). In this discussion, we first focus on some automated generation techniques which are based on state transition

information or grammatical structure of executable system functionality representations. Then, we concentrate upon user and input directed trace and trajectory generation issues.

5.1 Systematic System Behaviour Generation

Disregarding values and variations of parameter fields of interactions referred to in extended FSM based representations, relatively simple sequential automata describing the "control structure" of those representations can be constructed. Based on the state transition information of such automata, some sequences of (possibly only input) interactions may be constructed systematically. Such sequences are called transition tours [NaTs 81], characterizing sequences [Chow 78], and distinguishing sequences [Gone 70]. These interaction sequences are guaranteed to reveal either all operation errors (e.g., wrong output), transfer errors (e.g., wrong state transition), or all missing states provided certain assumptions hold. For example, characterizing sequences reveal all occurrences of all the above error types if the finite state automata

a) is completely specified, minimal, and strongly connected and

b) starts at a fixed initial state.

More details and the application of these three systematic methods of construction of interaction sequences to communication protocols are reported in [Boch 82, SaBo 82, SaBo 83].

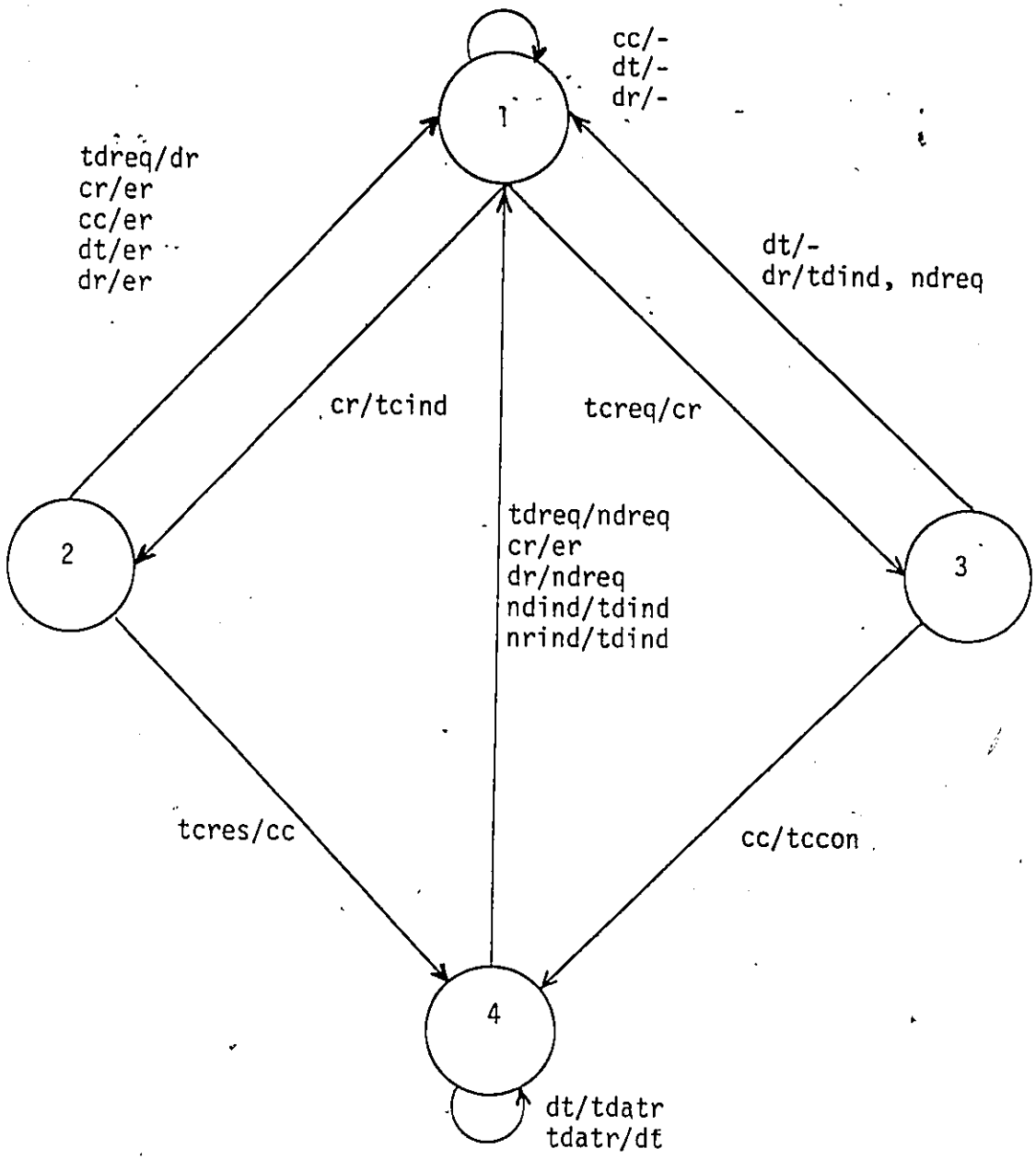
In the following section, we present a fully automated method of generating a representative set of interaction sequences based on a regular grammar corresponding to the control structure of a given system functionality representation.

5.1.1 Construction of Minimum Covers

Consider a regular grammar G(VN,VT,P,S)<1> describing the control structure of a given system functionality representation. Such a grammar can be obtained in the same way that a simpler sequential automata is obtained from a given extended FSM based representation; i.e., by disregarding parameter fields of interactions referred to in the given representation. In fact, if a finite state automata describing the control structure of a given representation exists, the construction of a corresponding regular grammar is straightforward [Brzo 62]. An equivalent regular grammar for the finite state automata (FSA) in Figure 5.1 is given in Figure 5.2.

<1> where VN=Non-terminals, VT=Terminals

P =Productions , S =Start symbol



LEGEND

tcreq	T_Connect_Request	er	Error TPDU
tdreq	T_Disconnect_Request	cr	Connect_Requst TPDU
tcres	T_Connect_Response	cc	Connect_Confirm TPDU
tdatr	T_Data_Request	dt	Data_Request TPDU
tcind	T_Connect_Indication	dr	Disconnect_Request TPDU
tdind	T_Disconnect_Indication	ndind	N_Disconnect_Indication
tccon	T_Connect_Confirm	nrind	N_Reset_Indication
tdati	T_Data_Indication	ndreq	N_Disconnect_Request

Figure 5.1 Finite State Machine for the Class 0 Transport Protocol.

We call a string of the least number of input interactions which covers all the production rules in grammar G at least once, a minimum cover. It is obvious that some production rules must be covered (expanded) more than once in order to cover some others at least once. Hence, for the construction of a minimum cover, the exact number of expansions of each production rule should be determined.

```
s1 --> cr      s2
   |   tcreq   s3
   |   cc      s1
   |   dt      s1
   |   dr      s1
   |   empty   .

s2 --> tcres   s4
   |   tdreq   s1
   |   cr      s1
   |   cc      s1
   |   dt      s1
   |   dr      s1 .

s3 --> cc      s4
   |   dt      s1
   |   dr      s1 .

s4 --> tdatr   s4
   |   dt      s4
   |   tdreq   s1
   |   cr      s1
   |   dr      s1
   |   ndind   s1
   |   nrind   s1 .
```

Figure 5.2 Corresponding Regular Grammar

We reduce the problem of determining the number of expansions of production rules to a special case of the max-flow min-cut problem [FoFu 62]. That is, the problem is reduced to a minimum flow assignment in the corresponding network with lower bounds on the flow in all the edges. This problem can be solved via max-flow min-cut theorem [FoFu 62] and thus the exact number of expansions of each production can be determined. In the following subsections, we give an algorithm to construct minimum covers for regular grammars. Rather than giving a lengthy proof of correctness by tedious examination of cases, we illustrate the use of the algorithm by a detailed example. The illustration can be expanded into a proof in a straight forward way by generalizing it and including a proof of termination. Note that the algorithm has been implemented, tested, and applied to the grammar presented in the illustration.

An overview of the automated construction of a minimum cover for a regular grammar G is given in Figure 5.3. Accordingly, given a regular grammar G, the minimum cover construction process includes the following steps:

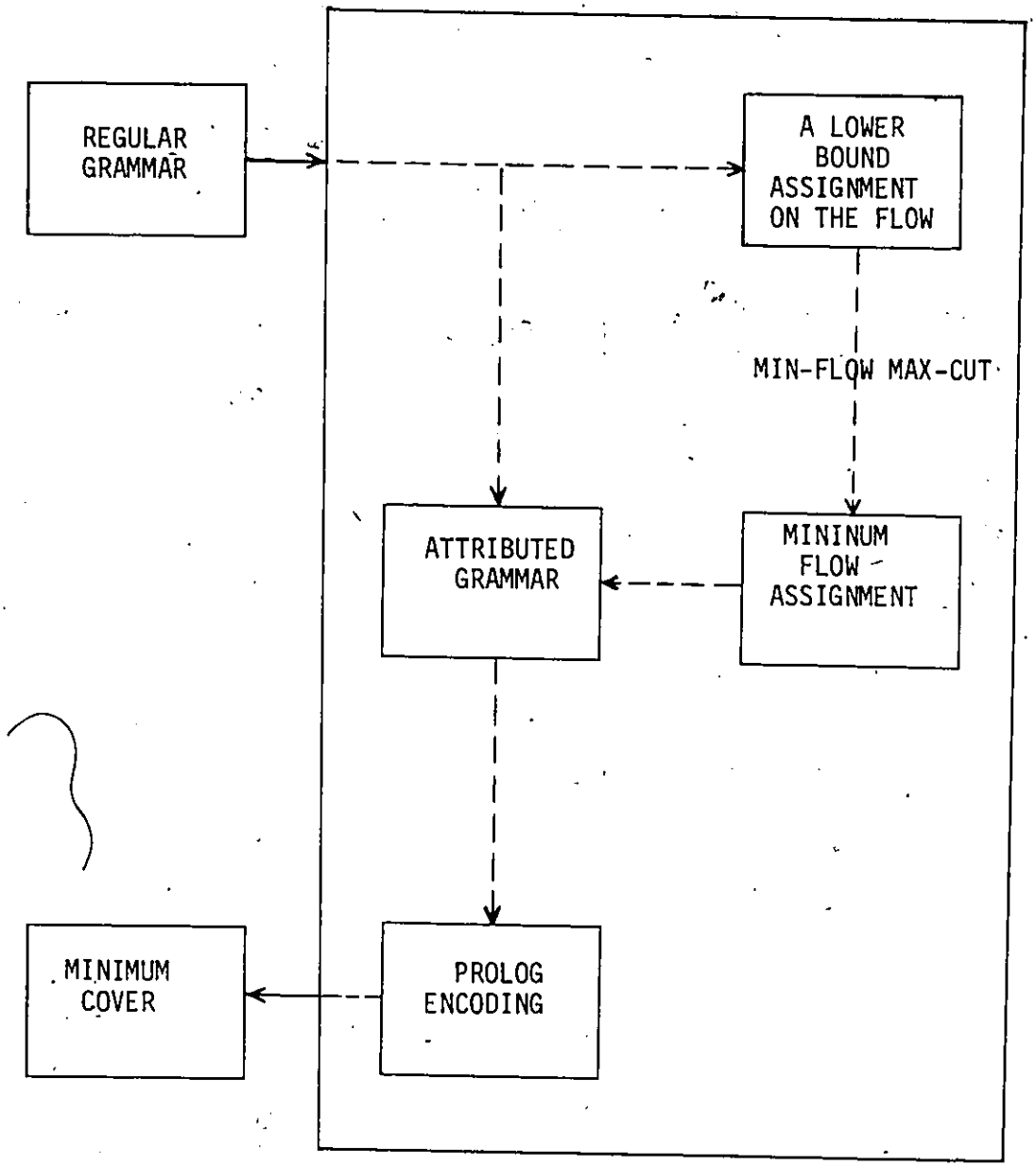


Figure 5.3 Automated Minimum Cover Generation Process

Step 1)

Construct a digraph DG(V,E) from the grammar G(VN,VT,P,S) where V is the set of vertices, E is the set of edges and

a) V = VT and E = P

b) There is a designated vertex in V called the source which is the start symbol S in G.

Here, what we want is to assign some weights on the edges of the digraph DG which indicate the number of times the corresponding production rules are expanded in order to construct a minimum cover. In order to determine these weights, the process continues as follows:

Step 2)

Construct a network N'(V',E') from DG(V,E) satisfying the following conditions:

a) N'(V',E') is a digraph DG'(V',E')

b) There is a designated vertex v in V' called source s which is the source of DG(V,E)

c) V' is defined as V' = {t} U V U Z where t is a designated vertex called sink to which all edges going into the source in DG(V,E) are directed in DG'(V',E'), and Z is the set of vertices which are inserted in the middle of parallel edges or self-loops in DG(V,E).

d) Each parallel edge or self-loop e in E is replaced by a pair of edges in $DG'(V',E')$ which are obtained by inserting a vertex v in Z in that parallel edge or self-loop.

e) If the number of parallel edges and self-loops in $DG(V,E)$ is m then $|V'| = |V| + m + 1$ and $|E'| = |E| + m$.

f) There is a lower bound $lb(e) = 1$ on the flow in every edge e in E'

g) There is no upper bound ub(e) on the flow in any edge e in E'.

At this step, a network with lower bounds on the flow in every edge is constructed with no self-loops and parallel edges.

Step 3)

Find a minimum flow assignment in the network $N'(V',E')$.

S3.1) Determine whether a legal flow F exists in $N'(V',E')$. Note that a legal flow F exists in $N'(V',E')$ if, for every e in E', there exists a directed cycle including edge e or a directed path from source to sink including e. Also note that a flow function f is an assignment of a real number f(e) to each edge e, such that the following two conditions hold:

a) For every edge e in E', $lb(e) \leq f(e)$

b) For every vertex v in V'-{s,t},

$$0 = \sum_{e \text{ in } a(v)} f(e) - \sum_{e \text{ in } b(v)} f(e)$$

where a(v) and b(v) are sets of edges incoming to vertex v and outgoing from v, respectively.

thus, a total flow F of f is defined by

$$F = \sum_{e \text{ in } a(t)} f(e) - \sum_{e \text{ in } b(t)} f(e)$$

where F is the net sum of flow into sink t.

A legal flow F in N'(V',E') may be found as follows [Even 79]:

S3.1.1) Construct an auxiliary network N"(V",E") by modifying N'(V',E') as follows:

a) V" = V' U {s',t'} where s' and t' are called auxiliary source and sink, respectively

b) For every v in V', construct an edge e' from v to t' with an upper bound

$$ub'(e') = \sum_{e \text{ in } b(v)} lb(e)$$

where b(v) is the set of edges which emanate from v in N', and a lower bound $lb'(e') = 0$

c) For every v in V', construct an edge e' from s' to v with an upper bound

$$ub'(e') = \sum_{e \text{ in } a(v)} lb(e)$$

where a(v) is the set of edges which enter v in N', and with a lower bound $lb'(e') = 0$

d) for every edge e in E', construct an edge e' in N" with a lower bound $lb'(e') = 0$ and an upper bound $ub'(e') = \infty$

e) construct two new edges from s to t and from t to s in N" with upper bounds equal to infinity and lower bounds equal to zero.

S3.1.2) Apply a max-flow min-cut algorithm [Dini 70 or FoFu 62] to find a maximum flow from s' to t' in N". A maximum flow in N"(V",E") saturates all the edges in E" which emanate from s' and thus, it can be concluded that N'(V',E') has a legal flow [Even 79].

S3.1.3) Define a legal flow in N'(V',E') by assigning a value to the flow function f'(e) for each edge e in E' which is equal to the sum of the flow function f'(e) in N" and the lower bound lb(e) in N'. That is, for

each edge e in E', f(e) = f'(e) + lb(e).

S3.2) Minimize the legal flow in N'(V',E') found in S3.1) by applying a max-flow min-cut algorithm while exchanging the roles of the sink t' and the source s. Clearly, a min-flow from source to sink is a max-flow from sink to source [Even 79].

Step 4)

Assign weights on edges in DG(V,E) by using the minimum flow assignment in N'(V',E') in the following manner:

a) The weight on a parallel edge e in E between vertices vi and vj in V is equal to the flow entering the auxiliary vertex vij in V' which was inserted on the parallel edge e to obtain a pair of edges in N'(V',E') to replace the parallel edge.

b) The weight on every other edge e in E is equal to the flow on the corresponding edge in E'.

The weights on the edges in DG(V,E) state exactly how many times each alternative production rule must be expanded to obtain a minimum cover. For example, the exact number of expansions of production rules in the grammar in Figure 5.2 is given in Figure 5.4 as determined by this approach.

```
s1  ::= gs11(=6  inc(gs11)  i(cr)     s2
    |   gs12(=6  inc(gs12)  i(tcreq)  s3
    |   gs13(=1  inc(gs13)  i(cc)     s1
    |   gs14(=1  inc(gs14)  i(dt)     s1
    |   gs15(=0  inc(gs15)  i(dr)     s1 .

s2  ::= gs21(=1  inc(gs21)  i(tcres)  s4
    |   gs22(=1  inc(gs22)  i(tdreq)  s1
    |   gs23(=1  inc(gs23)  i(cr)     s1
    |   gs24(=1  inc(gs24)  i(cc)     s1
    |   gs25(=1  inc(gs25)  i(dt)     s1
    |   gs26(=1  inc(gs26)  i(dr)     s1 .

s3  ::= gs31(=4  inc(gs31)  i(cc)     s4
    |   gs32(=1  inc(gs32)  i(dt)     s1
    |   gs33(=1  inc(gs33)  i(dr)     s1 .

s4  ::= gs41(=1  inc(gs41)  i(tdatr)  s4
    |   gs42(=1  inc(gs42)  i(dt)     s4
    |   gs43(=1  inc(gs43)  i(tdreq)  s1
    |   gs44(=1  inc(gs44)  i(cr)     s1
    |   gs45(=1  inc(gs45)  i(dr)     s1
    |   gs46(=1  inc(gs46)  i(ndind)  s1
    |   gs47(=1  inc(gs47)  i(nrind)  s1 .
```

Figure 5.5 Corresponding Attributed Grammar

```
s1 --> cr      s2      6
   |   tcreq   s3      6
   |   cc      s1      1
   |   dt      s1      1
   |   dr      s1      1
   |   empty   .

s2 --> tcres   s4      1
   |   tdreq   s1      1
   |   cr      s1      1
   |   cc      s1      1
   |   dt      s1      1
   |   dr      s1 .    1

s3 --> cc      s4      4
   |   dt      s1      1
   |   dr      s1 .    1

s4 --> tdatr   s4      1
   |   dt      s4      1
   |   tdreq   s1      1
   |   cr      s1      1
   |   dr      s1      1
   |   ndind   s1      1
   |   nrind   s1 .    1
```

Figure 5.4 Exact Number of Expansions of Production Rules

Step 5)

Construct an attributed grammar by augmenting the FSA-equivalent regular grammar G with guards and computation rules controlling the number of expansions of production rules. The notation for attributed grammars is the same as the one described in section 4.4. As an example, consider the attributed grammar in Figure 5.5. Each production rule in this grammar is a production rule in the grammar given in Figure 5.2, preceded by a guard and a computation rule. The function of each guard and computation rule pair is to make sure that the associated production rule expands exactly as many times as specified by the minimum flow assignment.

Step 6)

Construct a PROLOG program implementing the attributed grammar formed in Step 5) and execute this program to obtain a minimal cover. The PROLOG program in Figure 5.6.a corresponds to the attributed grammar in Figure 5.5. The output of this program in Figure 5.7.a is a minimal cover for the regular grammar in Figure 5.2.

5.1.2 Some Properties and Limitations of Systematic Generation

A minimum cover, by definition, corresponds exactly to a transition tour which starts with an initial state and covers all the transitions in a FSM at least once. Assuming

that the FSM is minimal, strongly connected, and fully specified, a transition tour reveals all operation errors (i.e., errors in the ouput function) [SaBo 82] but it can reveal only some of the transfer errors (i.e., errors in the nextstate function).

```
cr
tcres
tdatr
dt
tdreq
cr
tdreq
cr
cr
cr
cc
cr
dt
cr
dr
tcreq
cc
cr
tcreq
cc
dr
tcreq
cc
ndind
tcreq
cc
nrind
tcreq
dt
tcreq
dr
cc
dt
dr
```

Figure 5.7.a A Minimum Cover

```
s1 :-
      gs11(A), A=<6, increment(gs11), input(cr),    s2
    ; gs12(B), B=<6, increment(gs12), input(tcreq), s3
    ; gs13(C), C=<1, increment(gs13), input(cc),    s1
    ; gs14(D), D=<1, increment(gs14), input(dt),    s1
    ; gs15(E), E=<1, increment(gs15), input(dr),    s1
    .

s2 :-
      gs21(A), A=<1, increment(gs21), input(tcres), s4
    ; gs22(B), B=<1, increment(gs22), input(tdreq), s1
    ; gs23(C), C=<1, increment(gs23), input(cr),    s1
    ; gs24(D), D=<1, increment(gs24), input(cc),    s1
    ; gs25(E), E=<1, increment(gs25), input(dt),    s1
    ; gs26(E), E=<1, increment(gs26), input(dr),    s1
    .

s3 :-
      gs31(A), A=<4, increment(gs31), input(cc),    s4
    ; gs32(B), B=<1, increment(gs32), input(dt),    s1
    ; gs33(C), C=<1, increment(gs33), input(dr),    s1
    .

s4 :-
      gs41(A), A=<1, increment(gs41), input(tdatr), s4
    ; gs42(B), B=<1, increment(gs42), input(dt),    s4
    ; gs43(C), C=<1, increment(gs43), input(tdreq), s1
    ; gs44(D), D=<1, increment(gs44), input(cr),    s1
    ; gs45(E), E=<1, increment(gs45), input(dr),    s1
    ; gs46(F), F=<1, increment(gs46), input(ndind), s1
    ; gs47(G), G=<1, increment(gs47), input(nrind), s1
    .

input(X) :- write(X), nl.
increment(X) :- T=..[X,I], T, II is I+1,
                TT=..[X,II], retract(T), assert(TT),!.
gs11(1).
gs12(1).
gs13(1).
gs14(1).
gs15(1).
gs21(1).
gs22(1).
gs23(1).
gs24(1).
gs25(1).
gs26(1).
gs31(1).
gs32(1).
gs33(1).
gs41(1).
gs42(1).
gs43(1).
gs44(1).
gs45(1).
gs46(1).
gs47(1).
```

Figure 5.6.a Corresponding PROLOG Implementation

A minimum cover may be specified by either input interaction sequences or input and output interaction sequences. The construction of minimum covers consisting of input interaction sequences alone is described in the previous subsection. This construction process can be easily augmented to generate minimum covers consisting of sequences of input and output interactions. The required augmentation is accomplished by providing the ouput interactions associated with input interactions and by inserting an output generating action in each alternative production rule. For example, when the attributed grammar and the associated PROLOG encoding are augmented with output interactions, given in Figure 5.1, the minimum cover constructed by the automated process (which generates the PROLOG encoding in Figure 5.6.b) becomes as shown in Figure 5.7.b. Notice that output interactions are given in parentheses following the associated input interactions.

Hence, the automated minimum cover construction process can be used to generate a representative and discriminating set of test sequences, traces, and trajectories. The programs used in the automation of the minimum cover construction process are given in Appendix E.

Recall that, in the construction of minimum covers, interaction parameters have been disregarded. Therefore, minimum covers can be used only to check whether all basic protocol (control) functions have been implemented

correctly. Similar to distinguishing and characterizing sequences, when interaction parameters are taken into consideration, the length of minimum covers may grow very rapidly and their usage may become infeasible.

```
cr      (tcind)
tcres   (cc)
tdatr   (dt)
dt      (tdati)
tdreq   (ndreq)
cr      (tcind)
tdreq   (dr)
cr      (tcind)
cr      (er)
cr      (tcind)
cc      (er)
cr      (tcind)
dt      (er)
cr      (tcind)
dr      (er)
tcreq   (cr)
cc      (tccon)
cr      (er)
tcreq   (cr)
cc      (tccon)
dr      (ndreq)
tcreq   (cr)
cc      (tccon)
ndind   (tdind)
tcreq   (cr)
cc      (tccon)
nrind   (tdind)
tcreq   (cr)
dt      (-)
tcreq   (cr)
dr      (tdind & ndreq)
cc      (-)
dt      (-)
dr      (-)
```

Figure 5.7.b A Minimum Cover (With Outputs)

```
s1 :-
      gs11(A), A=<6, increment(gs11), input(cr),    output(tcind), s2
    ; gs12(B), B=<6, increment(gs12), input(tcreq), output(cr),    s3
    ; gs13(C), C=<1, increment(gs13), input(cc),    output(-),     s1
    ; gs14(D), D=<1, increment(gs14), input(dt),    output(-),     s1
    ; gs15(E), E=<1, increment(gs15), input(dr),    output(-),     s1
    .

s2 :-
      gs21(A), A=<1, increment(gs21), input(tcres), output(cc),  s4
    ; gs22(B), B=<1, increment(gs22), input(tdreq), output(dr),  s1
    ; gs23(C), C=<1, increment(gs23), input(cr),    output(er),  s1
    ; gs24(D), D=<1, increment(gs24), input(cc),    output(er),  s1
    ; gs25(E), E=<1, increment(gs25), input(dt),    output(er),  s1
    ; gs26(E), E=<1, increment(gs26), input(dr),    output(er),  s1
    .

s3 :-
      gs31(A), A=<4, increment(gs31), input(cc), output(tccon),          s4
    ; gs32(B), B=<1, increment(gs32), input(dt), output(-),              s1
    ; gs33(C), C=<1, increment(gs33), input(dr), output('tdind & ndreq'), s1
    .

s4 :-
      gs41(A), A=<1, increment(gs41), input(tdatr), output(dt),    s4
    ; gs42(B), B=<1, increment(gs42), input(dt),    output(tdati), s4
    ; gs43(C), C=<1, increment(gs43), input(tdreq), output(ndreq), s1
    ; gs44(D), D=<1, increment(gs44), input(cr),    output(er),    s1
    ; gs45(E), E=<1, increment(gs45), input(dr),    output(ndreq), s1
    ; gs46(F), F=<1, increment(gs46), input(ndind), output(tdind), s1
    ; gs47(G), G=<1, increment(gs47), input(nrind), output(tdind), s1
    .

input(X) :- nl, write(X).
output(X) :- put(9), write('('), write(X), write(')').
increment(X) :- T=..[X,I], T, II is I+1,
                TT=..[X,II], retract(T), assert(TT),!.
gs11(1).
gs12(1).
gs13(1).
gs14(1).
gs15(1).
gs21(1).
gs22(1).
gs23(1).
gs24(1).
gs25(1).
gs26(1).
gs31(1).
gs32(1).
gs33(1).
gs41(1).
gs42(1).
gs43(1).
gs44(1).
gs45(1).
gs46(1).
gs47(1).
```

Figure 5.6.b Corresponding PROLOG Implementation (With Outputs)

In the following section, we present a complementary approach, namely, a semi-automatic methodology which supports human involvement in the selection of representative sets of interaction sequences. It is our feeling that a tester's understanding of protocol semantics should be integrated into the test construction process as early in the life-cycle as possible. This methodology is aimed at constructing test (sequence) specifications in an attributed grammar formalism such that the resulting grammar can be executed in a user-controlled manner.

5.2 User Directed System Behaviour Generation

In general, a given system functionality representation describes several major functions which can be decomposed into a number of sub-functions and so on. Almost all of these functions and sub-functions are related to some sort of processing based on various interactions and interaction parameters. Such processing include sequencing interactions, encoding and decoding interactions and their parameter fields, etc. These interactions and their parameters vary in both type and format, and sometimes take on a very large number of possible values.

For example, consider the Class 0 Transport protocol specification [ISO 82]. This specification describes three major protocol functions; namely, connection establishment, data transfer, and connection clearing. According to a

functional decomposition [Boch 83], these functions may be decomposed into a number of sub-functions. Some sub-functions are common to all three major functions. These sub-functions are decoding and encoding of interactions, sequencing of interactions, handling multiple connections, and reacting to unexpected input interactions. The remaining sub-functions are uniquely related to one of the major functions. That is, sub-functions such as encoding and decoding of interaction parameters (including addresses of connection endpoints), handling default values of options, interpreting service quality parameters, and detecting local congestions are particular to connection establishment. As well, sub-functions such as preserving user data delivery order and content, handling various lengths of user data units, etc. are particular to data transfer, and sub-functions such as freeing local references and network channels are related to connection clearing. These sub-functions process three types of interactions, namely, transport service primitives, transport protocol data units, and network service primitives. The parameters of these interactions, their values, and variations can be found in related service and protocol specifications [ISO 82].

It is often desired to generate a representative set of interaction sequences

a) reflecting those functions and sub-functions captured

by system functionality representations, and

b) containing representative values for interaction parameters.

Since system functionality representations generally describe only typical and valid system functionality, representative sets of interaction sequences (generated from these representations) normally do not include sequences corresponding to invalid system behaviour. For trace generation, this may be tolerated. However, robust validation of refined representations such as protocol implementations require generation of interaction sequences (i.e., test sequences or trajectories) reflecting hostile and stress cases where the representation under test is pushed to violate the constraints imposed by the specifications. For example, determining the degree of robustness of a protocol implementation is an integral part of protocol implementation assessment [Rayn 81]. In this case, the aim is to generate certain sequences of interactions containing certain parameter values which will potentially fail the protocol implementation and/or will test the error recovery procedures, etc. Thus, a more complete set of representative interaction sequences may be constructed by considering both typical and hostile interaction sequences containing valid and invalid parameter values.

Selection of interaction sequences (i.e., test sequences or trajectories) representing both typical and hostile behaviour of the environment of a system functionality representation may be based on various testing heuristics such as boundary value analysis [Myer 79], functional testing [Howd 80], error-based testing [Weyu 81], etc. For example, such testing heuristics may be used to specify interaction sequences to test the functional behaviour of the representation under test on the boundaries of domains over which the required functions are defined. Such interaction sequences may be designed to uncover incorrect implementations of functions, limits, or constraints, or to test the defensive behaviour of the representation under test. Moreover, in order to enhance the effectiveness of specification-based testing strategies, an analysis of the experience gained during implementing or testing similar systems can be taken into account during the construction of a representative set of interaction sequences. This analysis may involve an enumeration of typical causes for implementation errors, interface and interaction problems among functions, and identification of implicit requirements for the representation under test. An example of the latter is identification of additional characteristics of the representation under test which are not specified in its specification.

In the following section, we propose a methodology which involves user (tester) guidance in specifying and generating

semantically meaningful test sequences or trajectories.

5.2.1 User Guidance in System Behaviour Generation

Consider a system functionality representation, other than the actual implementation of the system, which is given in the extended FSM based formalism described in section 4.2. Assume that this representation includes the definitions of ranges of valid values for the interaction parameters referred to in the representation. In generating test sequences (or trajectories) from the above representation, user guidance in implementing testing heuristics and in selecting representative test sequences may be realized in two distinct manners:

a) A corresponding representation is constructed in the attributed context free grammar based formalism described in section 4.4. Then, the tester may prepare invocations of the PROLOG encoding of the attributed grammar based representaion satisfying a set of constraints imposed by particular requirements of various testing heuristics. A natural extension of this is to design a test specification language which may be used as an interface between the tester and an invocation generator program. This program may enforce the constraints on the generated sequence in accordance to the requirements of encoded testing heuristics selected by the tester. If the reader is

interested, issues related to test specification languages are discussed in [PrUr 83].

b) A corresponding attributed context-free grammar specifying test sequences is constructed from the given extended FSM based representation. We call this grammar a test sequence grammar. A PROLOG encoding of a test sequence grammar can then be executed to selectively generate representative test sequences in a user-controlled manner [UrPr 83b].

Both of the above approaches support human involvement in test sequence construction. The first approach is somewhat similar to trajectory generation and partially covered in section 5.3. In the following, we concentrate on the construction of test sequence grammars.

A test sequence grammar G is defined as G(VN,VT,P,S) where VN,VT,P are finite sets of non-terminals, terminals, and productions, respectively, and S is the start symbol which is in the set VN. The intersection between VN and VT is an empty set and the union of VN and VT is called the alphabet V. A sentence s over the alphabet V is any string of finite length composed of the symbols from V. The empty sentence, denoted e, is defined as the sentence consisting of no symbols. If the set of all sentences over the alphabet V includes the empty sentence e, it is denoted by V*. Otherwise, it is denoted by V+. That is, V*=V++e and V+=V*-e.

Each production rule in P is in the form "a-->b" which reads as " a generates b" where a is in VN and b is in V*. A sentence is defined to be in the "language" generated by the grammar G if the sentence consists solely of terminals and if it can be derived from the start symbol (through the application of an arbitrary number of production rules) [HoUl 69].

In fact, a test sequence grammar is intended to generate all such test sequences as strings of terminals which can be derived from the start symbol. Given a system functionality representation, a test sequence grammar (denoted "tsg") specifying test sequences can be constructed in the following manner:

a) Assuming that each input and output interaction referred in the given representation consist of two fields, denoted interaction-id and interaction-parameters, and that the field interaction-parameters in each interaction consists of specific subfields corresponding to the parameters of that interaction, we define the following:

1) A is the set of all the values of interaction-id fields

2) B is the set of some selected values of all the distinctly identifiable interaction parameter fields

3) C is the set consisting of symbols "i" and "o" (Recall from section 4.3.3 that an "i" ("o") immediately preceding an interaction indicates that the interaction is an input (output) interaction),

4) D is the set of names of all distinctly identifiable fields of each input and output interaction format,

5) E is the set of names of all the states defined in the given representation,

6) F is the set consisting of symbols input, output, and s0.

Based on the above definitions, grammar "tsg(S,VN,VT,P)" is defined as follows:

i) The start symbol S is the initial state in the given representation.

ii) The set of non-terminals VN is the union of sets D, E, and F

iii) The set of terminals VT is the union of sets A, B, and C.

iv) The set of productions P is the union of sets P1 and P2 where P1 is the set of productions which correspond to the state transitions in the given

representation, and P2 is the set of all other productions which involve the symbols in all the sets except the set E.

That is,

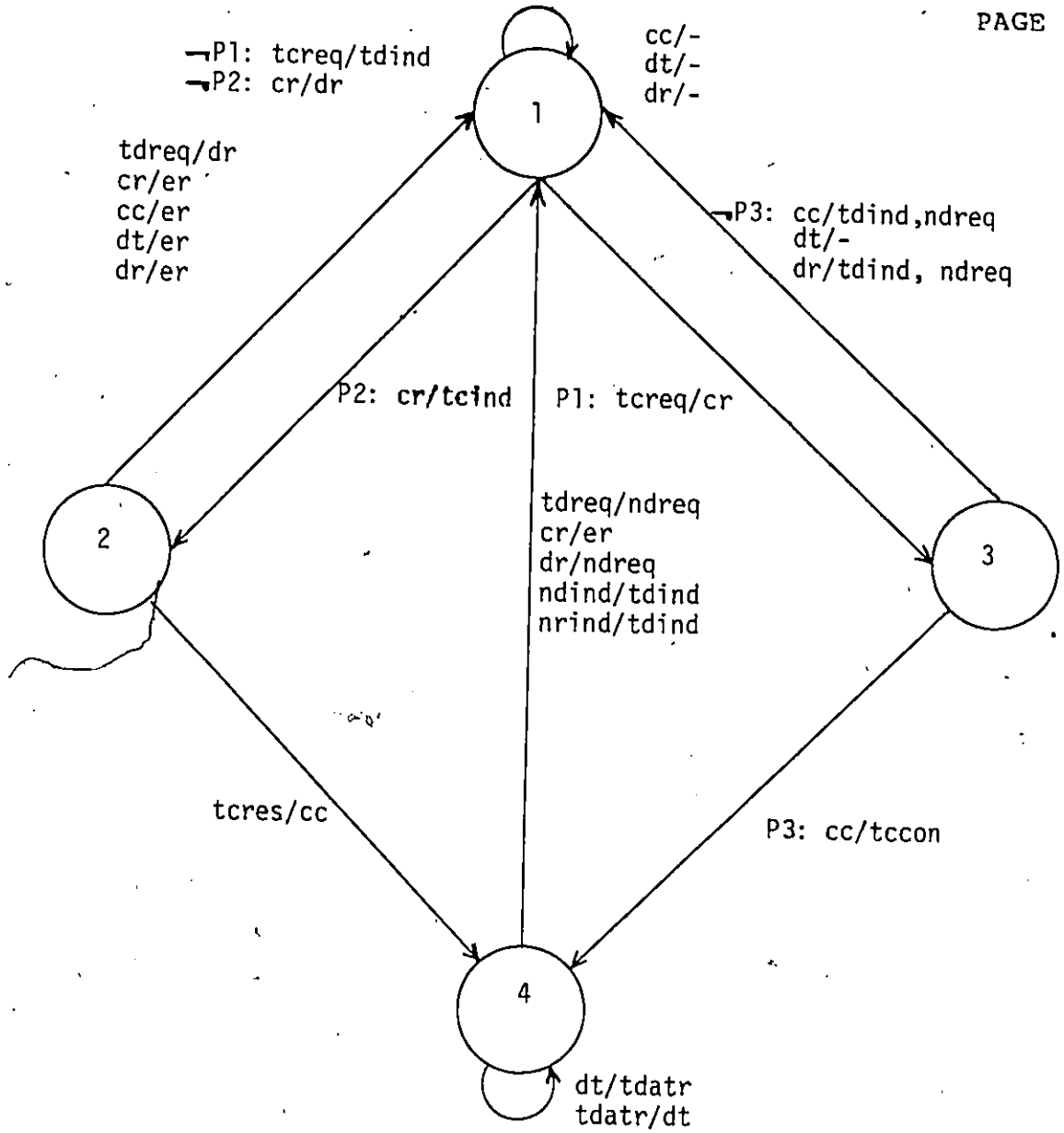
$$P1=\{\ a \rightarrow xb \mid a \in E \ \&\ b \in E \cup \{s0\}\}$$

$$P2=\{\ a \rightarrow x \mid a \in D \cup F\}$$

where x is

a string of symbols $A \cup B \cup C \cup D \cup \{input, output\}$.

Thus, each test sequence is specified as a string of terminals which can be derived from the start symbol by applying an arbitrary number of productions. It is important to note that the set of nonterminals includes the symbol s0 which replaces any occurrence of the start symbol on the right hand side of productions. In fact, this symbol replaces each occurrence of a non-terminal on the right hand side of productions which corresponds to a final state in the given extended FSM in order to relate each specific test sequence to a path from the initial state to a final state. Also, note that we allow the grammar tsg to include inherited and synthesized attributes, guards, and computation rules as described in section 4.4.



LEGEND

tcreq	T_Connect_Request	er	Error TPDU
tdreq	T_Disconnect_Request	cr	Connect_Requst TPDU
tcres	T_Connect_Response	cc	Connect_Confirm TPDU
tdatr	T_Data_Request	dt	Data_Request TPDU
tcind	T_Connect_Indication	dr	Disconnect_Request TPDU
tdind	T_Disconnect_Indication	ndind	N_Disconnect_Indication
tccon	T_Connect_Confirm	nrind	N_Reset_Indication
tdati	T_Data_Indication	ndreq	N_Disconnect_Request

P1 (¬ P1) : tcreq is (not) acceptable
P2 (¬ P2) : cr is (not) acceptable
P3 (¬ P3) : cc is (not) acceptable

Figure 5.8 A Simplified Extended FSM for the Class 0 Transport Protocol

As an example, the reader may consider the simplified extended FSM for the Class 0 Transport Protocol given in Figure 5.8 and a corresponding but thus far imcomplete test sequence grammar in Figure 5.9. Here, the non-terminal input in each production expands into a corresponding input interaction whose "id" is specified by the value of the first attribute. The second attribute attached to the non-terminal input specifies whether valid or invalid (or valid but unacceptable) values are assigned to interaction parameters. Notice that the unacceptable values of parameters correspond to the reception of unacceptable "tcreq", "cr", or "cc" which are indicated by negations of predicates P1, P2, and P3 in Figure 5.8, respectively.

In each production rule the non-terminal output expands into the expected output interaction with valid parameter values. ~~In~~ some cases when the expected output is not defined in the extended FSM representation, the non-terminal output expands into an empty string which is indicated by "nil" in the attribute list. Also notice that the non-terminal s0 replaces all occurrences of s1 on the right hand side of productions which expands into an empty string.

```
s1 --> input (tcreq,0) output (cr   ,0)    s3
     | input (tcreq,1) output (tdind,0)    s0
     | input (cr   ,0) output (tcind,0)    s2
     | input (cr   ,1) output (dr   ,0)    s0
     | input (cc   ,0) output (nil)        s0
     | input (dt   ,0) output (nil)        s0
     | input (dr   ,0) output (nil)        s0.

s2 --> input (tcres,0) output (cc   ,0)    s4
     | input (dr   ,0) output (er   ,0)    s0
     | input (cr   ,0) output (er   ,0)    s0
     | input (cc   ,0) output (er   ,0)    s0
     | input (dt   ,0) output (er   ,0)    s0
     | input (tdreq,0) output (dr   ,0)    s0.

s3 --> input (cc   ,0) output (tccon,0)    s4
     | input (cc   ,1) output (tdind,0)
                       output (ndreq,0)    s0
     | input (dr   ,0) output (tdind,0)
                       output (ndreq,0)    s0
     | input (dt   ,0) output (nil)        s0

s4 --> input (dt   ,0) output (tdati,0)    s4
     | input (tdatr,0) output (dt   ,0)    s4
     | input (tdreq,0) output (ndreq,0)    s0
     | input (dr   ,0) output (ndreq,0)    s0
     | input (cr   ,0) output (er   ,0)    s0
     | input (ndind,0) output (tdind,0)    s0
     | input (nrind,0) output (tdind,0)    s0.

s0 --> empty.
```

Figure 5.9 Corresponding Incomplete Test Sequence Grammar

b) The construction of the grammar "tsg" is completed by augmenting the grammar with additional attributes, guards, and computation rules. The aim of augmentation is:

1) to restrict the generation of test sequences to a relatively small, managable, and representative set of test sequences

2) to encode adaptations of various appropriate testing heuristics

3) to incorporate context sensitive information into the grammar

4) to provide the capability of selectively specifying test sequences.

The augmentation is based on specifying conformance, consistency, and/or robustness criteria, identifying protocol functions and their functional dependencies, defining representative parameter values and so on. These augmentations are implemented via additional attributes which provide the capability of guiding the test sequence generation process.

c) The augmented test sequence grammar "tsg" is then implemented as a test sequence generator in PROLOG.

In the following section, we illustrate the user guided test sequence generation methodology by continuing with the example given above.

5.2.2. An Illustrative Example

Consider the partially constructed test sequence grammar "tsg" in Figure 5.9. The construction of this grammar can be completed by augmenting the grammar according to various testing heuristics, conformance criteria, and functional decompositions. Here, in order to simplify the illustration, the augmentation will be limited to a basic

functional decomposition which divides the major protocol functions into two phases; namely connection establishment and data transfer. We assume that for each test sequence it is desired to specify

1) in either phase

   whether the local session protocol entity is

   a) the initiator or the responder for the connection establishment request

   b) the one which issues the disconnect request or not (denoted by disconnector or not-disconnector, respectively)

2) in connection establishment phase whether the received tcreq, cr, and cc are acceptable or unacceptable or whether unspecified outputs are allowed

3) in data transfer phase

   a) whether the local session protocol entity is the sender or receiver during data transfer

   b) whether network-reset is allowed to take place

   c) whether network-disconnect is allowed to take place

   d) whether unspecified or error outputs are allowed to take place

e) the number of data items to be sent or to be received by the local session protocol entity

Thus, the first step of augmentation is to insert guards in productions in order to implement the above factorization. A possible implementation is given in Figure 5.10 where each guard is a boolean expression consisting of predicates:

initiator (or responder)	IRB
connect-only (or data-transfer-only)	CDTB
acceptable_tcreq (or unacceptable_tcreq)	P1
acceptable_cr_tpdu (or unacceptable_cr_tpdu)	P2
unspecified (or error) output	UEB
disconnector (or not_disconnector)	DNB
acceptable_cc_tpdu (or unacceptable_cc_tpdu)	P3
sender (or receiver)	SRB
network_reset (or not_network_reset)	NRB
network_disconnect (or not_network_disconnect)	NDB

Note that these predicates will be set to .true. or .false. via the values of attributes shown at the right hand side in the above table during the invocation of corresponding PROLOG procedure. In fact, there are two other attributes, namely, "S" and "R" which are used to specify the number of data fragments that the local session protocol entity sends and receives, respectively during a transport connection (in data transfer phase).

The above predicates indicate the following points:

a) In addition to the specifications of test sequences reflecting typical, normal operating conditions, some robustness tests such as testing defensive behavior of the protocol entity when subjected to unspecified receptions, etc. are included in the test sequence grammar.

b) combinations of predicates provide a wide variety of selection criteria for the tester. These include selecting all those test sequences involving connection establishment phase alone when the local session protocol entity is the initiator and when there are unspecified receptions; or all those involving data transfer phase alone when the local session protocol entity is both sending and receiving data; etc.

```
tptest(IRB,CDTB,SRB,S,R,P1,P2,P3,DNB,NRB,NDB,UEB) :-
    set_params(IRB,CDTB,SRB,S,R,P1,P2,P3,DNB,NRB,NDB,UEB),
    setof(X,s1(X,[]),L), outprnt(L), !.

s0 --> [], [nl].

s1 -->
 ((initiator),
    ((connect_only; data_transfer_only),
     (acceptable_tcreq),
      input(tcreq,0), output(cr,0), s3
    ;(connect_only, unacceptable_tcreq),
      input(tcreq,1), output(tdind,0), s0))

;((responder),
    ((connect_only; data_transfer_only),
     (acceptable_cr_tpdu),
      input(cr,0), output(tcind,0), s2
    ;(connect_only, unacceptable_cr_tpdu),
      input(cr,1), output(dr,0), s0))
```

```
;((unspecified, connect_only),
    (input(cc,0), output(nil)
    ;input(dt,0), output(nil)
    ;input(dr,0), output(nil))).

s2 -->
 ((connect_only),
   (((disconnector), input(tdreq,0), output(dr,0), s0)
   ;((error),
      (input(dr,0), output(er,0), s0
      ;input(cr,0), output(er,0), s0
      ;input(cc,0), output(er,0), s0
      ;input(dt,0), output(er,0), s0 ))))

;((connect_only; data_transfer_only),
    *input(tcres,0), output(cc,0), (getval(S,R)), s4(S,R)).

s3 -->
 ((connect_only),
   (((not_disconnector),
      input(dr,0), output(tdind,0), output(ndreq,0), s0)
   ;((unspecified),
      input(dt,0), output(nil), s0)
   ;((unacceptable_cc_tpdu),
      input(cc,1), output(tdind,0), output(ndreq,0), s0)))

;((connect_only; data_transfer_only),
     (acceptable_cc_tpdu),
      input(cc,0), output(tccon,0), (getval(S,R)), s4(S,R)).


s4(0,0) -->
   ((connect_only; (data_transfer_only, (sender; receiver) )),
     ((disconnector), input(tdreq,0), output(ndreq,0), s0)
     ;((not_disconnector),
        ((                    input(dr,0),    output(ndreq,0), s0)
        ;((network_reset),    input(nrind,0), output(tdind,0), s0)
        ;((network_disconnect), input(ndind,0), output(tdind,0), s0)
        ;((error),            input(cr,0),    output(er,0),    s0)))).

s4(S,R) -->
   ((data_transfer_only),
      ((receiver, R>0, rbase(BR), SN is BR-R+1, RR is R-1),
        input(dt,0), output(tdati,SN), s4(S,RR)),

      ;((sender, S>0, sbase(BS), SN is BS-S+1, SS is S-1),
        input(tdatr,SN), output(dt,0), s4(SS,R))).
```

Figure 5.10 An Augmented Test Sequence Grammar

The second step of augmentation is to specify some selected values for the parameter fields of interactions. Here, assuming that the tester is interested to specify only those marginally valid and invalid values, the augmentation will be based on boundary value analysis [Myer 79]. In the following, we present an example of specifying an input interaction; namely, the transport protocol data unit "dr". In fact, the specification of this input interaction starts with the definition of the non-terminal input :

```
input(ID,TYPE) ::= ID EQ "dr"
                   "i"
                   "dr"
                   tpdu_param("dr",TYPE)
                 | *DEFINITIONS OF OTHER INTERACTIONS*
```

According to the above definition, the construction of the input interaction "dr" involves the expansion of the non-terminal input into terminals "i" and "dr" and the non-terminal tpdu_param . A possible definition of the non-terminal tpdu_param may be given as follows:

```
tpdu_param("dr",0) ::= length_of_dr(0,1)
                       code_of_dr(0,1)
                       dst_ref(0,1)
                       src_ref(0,1)
                       reason_of_dr(0,1)
                       information(0,1).
```

The above definition specifies a valid (i.e., TYPE = 0) interaction with a set of valid parameters. In this

definition, each reference to a parameter field carries values of two attributes; namely, E and N. The following table summarizes the values of E:

E=0 : A valid parameter value is requested

E=1 : The lower off-by-one value is requested

E=2 : The upper off-by-one value is requested

E=3 : Both lower and upper off-by-one values are requested

On the other hand, if the value of N > 0, it indicates the relative position of the requested value in the valid range of values (where E=0). If an invalid value is requested (0<E<4), N is set to zero.

As an example, consider the following definition of parameter "reason_of_dr":

```
reason_of_dr(E,N) ::=
   E EQ 0 AND N EQ 1  "0"      *Reason not specified*
 | E EQ 0 AND N EQ 2  "1"      *Congestion at TSAP*
 | E EQ 0 AND N EQ 3  "2"      *Session entity not attached*
 | E EQ 0 AND N EQ 4  "3"      *Address unknown*
 | E EQ 1 AND N EQ 0  "-1"     *Lower off-by-one value*
 | E EQ 2 AND N EQ 0  "4"      *Upper off-by-one value*
 | E EQ 3 AND N EQ 0  "-1" | "4". *Both off-by-one's *
```

In order to implement our assumed test strategy, we could specify all other parameter fields of the input interaction "dr" and all other interactions and their parameter fields in a similar way.

Once the construction of a test sequence grammar is completed, it can be coded in PROLOG as a generator program. A PROLOG encoding of this test sequence grammar "tsg", constructed in this section, is given in Appendix F together with some examples of generated test sequences. Note that in order to simplify setting the truth values of predicates used in guards and facilitating the generation of more than one test sequence satisfying given selection criteria, a simple procedure, denoted "tptest", is added to the beginning of the test sequence grammar "tsg".

The definition of procedure "tptest" is as follows;

```
tptest(IRB,CDTB,SRB,S,R,P1,P2,P3,DNB,NRB,NDB,UEB) :-
        set_params(IRB,CDTB,SRB,S,R,P1,P2,P3,DNB,NRB,NDB,UEB)
        setof(X,s1,L)
        outprnt(L).
```

Via the parameters of procedure "tptest", the tester may select a desired set of test sequences. When procedure "tptest" is invoked it passes the values of its parameters to procedure "set_params" which sets the truth values of predicates described in this section. Then, the built-in predicate "setof" is invoked by procedure "tptest" which generates a set L of all distinct instances of the start symbol s1. Each instance of s1 corresponds to a test sequence which is printed by the procedure "outprnt".

Notice that values of parameter fields of transport protocol data units defined in Appendix F are taken from the

transport protocol specification [ISO 83c]. Many of these values are given as 8 or 4 bits long binary numbers in the above specification. When the grammar was constructed each 4 bit binary number was converted into the corresponding hexadecimal digit in order to conserve space. These hexadecimal digits are enclosed into parentheses and stand for the values of corresponding parameter fields. Thus, for example, (2D) corresponds to an 8 bit binary number which is "00111101".

The interested reader may refer to [UrPr 84b] for a more detailed and complete test sequence grammar (generator) which is constructed from the Class 0 transport protocol specification [ISO 83c].

5.3. Input Directed System Behavior Generation

In previous sections, we have discussed systematic and user directed execution of system functionality representations. Possible outcomes of these executions include representative sets of input interaction sequences such as minimum covers, distinguishing sequences, characterizing sequences, user selected sequences, etc. These input interaction sequences can be used to guide the generation of representative actual or intended behavior (i.e., traces or trajectories) of system functionality representations. That is, sequences of input interactions may be applied to representations and during the application

of each input sequence, the responses (i.e., output interactions) of representations may be recorded together with the applied input interactions in the order of their occurrences.

During input directed trace and trajectory generation, each input stimulus (i.e., input interaction) is applied to the system functionality representation (say representation K) at the designated interaction point. Each applied input stimulus is recorded at the end of the trace or trajectory generated thus far following a prefix consisting of the symbol "i" and the identifier of the associated interaction point (e.g., apl or ap2). Each response of the representation K is recorded similarly following a prefix consisting of the symbol "o" and the identifier of the interaction point at which the response is observed.

In general, there may not be a corresponding response for each input stimulus applied to representation K and if there is, it may not be initiated at the same interaction point at which the input stimulus was applied. Moreover, a response corresponding to an input stimulus may not necessarily occur immediately after the application of that input stimulus. Hence, since the interactions are recorded in the order of their occurrences, there may be some stimuli and/or responses recorded in between a particular input stimulus and its corresponding response. This situation does not affect trace or trajectory generation. However, it presents

some problems during trace or trajectory checking which will be addressed in section 6.1.

In the following sections, we discuss trace and trajectory generation issues.

5.3.1. Input Directed Trace Generation

Traces of an executable representation may be mechanically generated by using a higher level PROLOG procedure, denoted the trace generator. Briefly, a trace generator reads a sequence of input stimuli, applies each input stimulus in this sequence to the executable representation, records both input stimuli and responses, and outputs the resulting observed behaviour of the representation.

During the application of a sequence of input stimuli, it may not be possible to invoke the executable representation any longer, and the trace generation process discontinues. This occurs when the invocation of the PROLOG procedure (implementing the executable representation) with a particular input stimulus is not successful. This failure usually results from an occurrence of an unspecified reception (i.e., the reception of the input interaction involved in the invocation is not specified at the current state of the executable representation). In turn, an unspecified reception may be an indication of incompleteness

in the specification, a potential deadlock, or a late indication that an inconsistency occurred earlier in the application of the sequence of input stimuli.

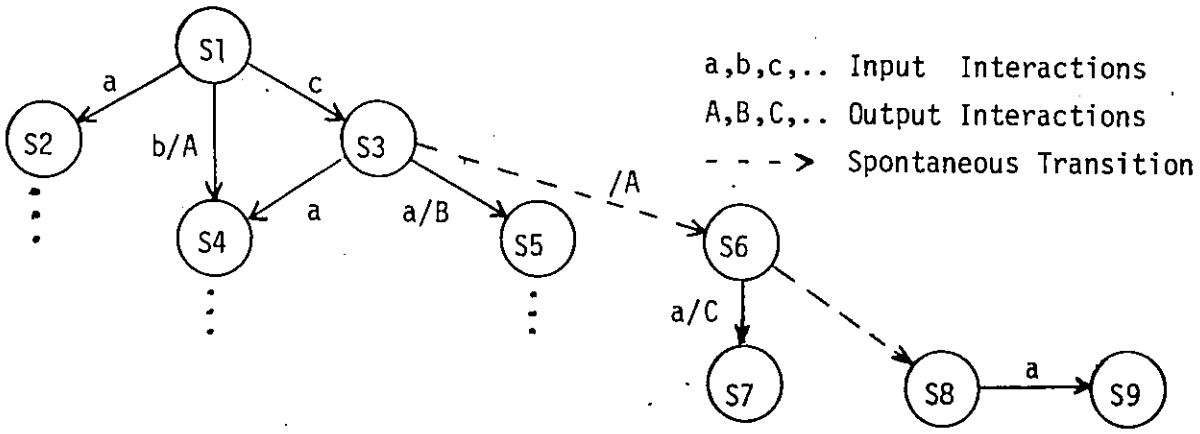
Hence, there can be at most, one trace resulting from the application of a sequence of input stimuli to a deterministic executable representation. However, this may not be the case for a non-deterministic representation due to the existence of internal transitions or more than one possible input-initiated transition for a specific input interaction at some states of the representation. Two possible approaches (among others) may be taken for generating traces of a non-deterministic representation:

a) specify a priori the choice of transition to be executed at each state

b) try each possible transition at each state, one after the other.

The first approach may be implemented by augmenting the attributed grammar with additional attributes to guide the choice of transition to be executed and generating sequences of input stimuli augmented with values for those attributes. However, this approach requires identifying and selecting potential transitions, setting attributes values either manually or mechanically via a testing heuristic which may be rather complex.

The second approach may be implemented by coding a trace generator in PROLOG and using the standard PROLOG backtracking mechanism to generate all possible traces for a given sequence of input stimuli. Notice that each trace is a result of traversal of a possible transition path through the complete state-space of the nondeterministic representation. The set of possible transition paths is determined by the applied sequence of input stimuli and the set of possible spontaneous transitions at the states reached during the application of that sequence of input stimuli. As an example, consider the following partial non-deterministic finite state machine:



Assume that the sequence of input stimuli is ca... The application of input c causes a transition from s1 to s3. But the set of possible states that the machine can be in after the application of input c includes s6 and s8 as well. Hence, the application of the next input stimulus (i.e., a) may cause any one of four possible transitions; namely transitions from s3 to s4, from s3 to s5, from s6 to s7, or from s8 to s9. Thus, the possible partial transition paths

after the application of the second input stimulus are: s1,s3,s4; s1,s3,s5; s1,s3,s6,s7; and s1,s3,s6,s8,s9.

Therefore, a trace generator should generate a set (SSi) of all possible status values that the representation can assume after the application of an input stimulus (Ii) and apply the next stimulus (Ii+1) to each status in that set and so on. The construction of set SSi can be performed in three steps:

i) The input stimulus Ii is applied to each status in SSi-1 (i.e., the set of all possible status values that the representation can assume after the application of the input stimulus Ii-1). This results in a set SSINi which consists of all possible status values that can be reached by the application of Ii to each status in SSi-1, without letting the representation perform any spontaneous transitions.

ii) Let the representation execute each spontaneous transition possible at each status in SSINi and at all the status reachable from those in SSINi by spontaneous transitions. One possible way of accomplishing this is to apply a depth-first search; that is, for each status Sj in SSINi, find all such paths that start from status Sj and end at a status from which no other status can be reached by a spontaneous transition. All the status values

reached via spontaneous transitions from status Sj form the set SSSTj.

iii) Take the union of sets SSSTj's. The resulting set is the set SSi.

It is important to note that non-deterministic representations may involve cycles consisting of only spontaneous transitions. Several such cycles may be formed in the above example by adding spontaneous transitions from s3 to s3, from s8 to s6, or from s8 to s3. The existence of a cycle of this type in a non-deterministic representation results in an infinite idle loop during the construction of set SSi and hence, neither the construction of set SSi nor the trace generation process terminates. In fact, the existence of a cycle of spontaneous transitions may be considered as a potential "tempo blocking" (also called infinite idle looping) in which a representation may execute without any effective progress. That is, although it may initiate some output interactions, it may not accept any input interaction.

Notice that if there is such a cycle in the representation, it appears in a path described in step ii) above. Thus, a cycle can be easily detected by checking whether the most recently reached status has already been visited in the path being formed. If a cycle is detected during the construction of set SSi, the trace generator

should report the existence of the cycle by documenting the sequence of input stimuli, the relative position of input stimulus Ii currently being applied, the status Sj from which the path containing the cycle originates, the immediately visited status SVj on the path, and the spontaneous transition executed at status SVj.

In the remaining of this section, we present an algorithm for trace generators. The algorithmic language that we employ includes the following constructs:

a) The two iteration constructs used in algorithms are:

for each s in S do
[a group of statements]
repeat

which is used to repeat a specific set of actions for each element s in a set S.

for i=m to n do
[a group of statements]
repeat

which is used to repeat a specific set of actions for a specific number of times (i.e., n-m+1 times).

b) The assignment operator "<--" is used to assign the value of an expression to an identifier; (i.e., i<-- expression),

c) The set union operator "U" is used to take the union

of two sets (i.e., set-a U set-b),

d) A procedure is "called" by simply referring the procedure (i.e., procedure-name(arguments)),

e) A special procedure denoted setof(X,P(X),L) finds a set L of all distinct instances of X which satisfies the procedure P. Note that a single call to setof(X,P,L) potentially backtracks and generates alternative but distinct values for X. In fact, after each successful call to procedure P, "setof" forces procedure P to backtrack and generate another instance of X until no other alternative is possible.

f) Each control construct "if-then-else" and each procedure definition are terminated with "endif" and "end "procedure-name"", respectively.

A trace generator may be based on the following algorithm:

```
procedure:  trace_generator (SEQFILE, TRACES)

*SEQFILE is the set of input stimuli sequences*
*TRACES  is the set of resulting traces*

for each SIS (sequence of input stimuli) in SEQFILE do
* initialization *
    SS0 <-- {initial_status}
    PART I:  CONSTRUCTION OF SET SSi
    * consider each Ii (input stimulus) in SIS *
```

```
for i=1 to |SIS| do
    SSi <-- empty
    get Ii
    * apply input stimulus Ii to each status in SSi-1 *
    SSINi <-- empty
    for each PS (present status) in SSi-1 do
        * define SL to be the set of all NS's (next
          status values) reachable from PS when Ii
          is applied to the representation *
        setof (NS, representation(Ii,PS,NS), SL)
        SSINi <-- SSINi U SL
    repeat
    * check whether there is an unspecified reception *
    if SSINi = empty then report_unspecified_reception
                     else continue
    * find all status values that can be reached by
      spontaneous transitions from each status Sj
      in SSINi *
    for each Sj in SSINi do
        k <-- 1
        SSSTk <-- empty
        * PATHS is the set of all paths starting at
          Sj and ending at a status from which no
          other spontaneous transition is possible *
        setof (PATH, paths(Sj,Sj,PATH), PATHS)
        * do not include in SSSTk multiple
          occurrences of status values in PATHS *
```

```
    for m=1 to |PATHS| do
        SSSTk <-- SSSTk U PATHm in PATHS
    repeat
    k=k+1
repeat
* obtain set SSi by taking the union of all SSSTk's *
for m=1 to k do
    SSi <-- SSi U SSSTm
repeat
repeat
**********
```

Strictly speaking, in the first part of the algorithm, each set SSi, SSINi, SSTSjk, etc. consists of elements of the following type: [I,O,SBT,SAT] where

I : Input Interaction

O : Output Interaction

SBT : Status of the representation before the transition

SAT : Status of the representation after the transition

Recall that in section 4.4, a status of a representation is defined as the contents of major state variables, additional state variables, internal buffers, and input, output queues. In fact, each element of the above sets is the parameter list of PROLOG procedure corresponding to the representation. For example, when input stimulus Ii (say input-9) is applied to an element of SSi-1 (say [input-2,output-3,status-3,status-4]), the representation is

invoked by

representation ([input-9,OUTPUT,status-4,NEXTSTATUS]) which returns the values for OUTPUT and NEXTSTATUS (say "output-5" and "status-6", respectively). Keeping these in mind, the rest of the algorithm is given as follows:

****

```
    PART II:  CONSTRUCTION OF TRACES FOR EACH SIS
    * construct an adjacency list from the sets SSi's *
    for each SSi do
        for each [I,O,SBT,SAT] in SSi do
            link [I,O,SAT] into the adjacency list of SBT
        repeat
    repeat
    * construct all possible traces for SIS *
    setof (TRACE, allpaths(SSi,TRACE), TRACES)
repeat
end trace_generator

    procedure:  allpaths(S1,TRACE)
    If there is a transition from S1 to S2 with I and O
       then paths(S2,T)
            TRACE <-- I U O U T
       else TRACE <-- empty
    end if
    end allpaths
    procedure : paths(CS,PSF,PATH)
    * CS is current status
      PSF is the path formed so far
```

```
    PATH is the resulting path  *
  * find a path from CS to a status from which
    no other spontaneous transition is possible *
  representation (CS,NS)
  If there is a NS
     then if NS is_a_member_of PSF
             then report_infinite_loop
             else PATHSF <-- PSF U NS
                  paths(NS,PATHSF,PATH)
          end if
     else PATH <-- PATHSF
  end if
  end paths
```

Since an occurrence of an infinite idle looping (if any) is detected and documented, the algorithm terminates with a finite set of traces, unless there is an unspecified reception. Thus, a trace generator based on the above algorithm generates a set of all possible irredundant traces of a representation corresponding to a given sequence of input stimuli. One then can select a representative subset of traces by considering only the shortest among all of those demonstrating cyclic behaviour.

The above algorithm has been implemented and extensively tested. A PROLOG encoding of this algorithm is given in Appendix G. This PROLOG procedure, denoted "totgen", receives an input interaction sequence and by applying this

sequence of input interactions to representation K (e.g., procedure "ts" or procedure "rts") generates all corresponding traces or trajectories as sequences of input and output interactions. Each interaction sequence is represented as a list whose elements are in the form of:

[Interaction-Type,[TSAP-Id,TS-Primitive-Type,TS-Primitive-Parameters]]

where Interaction-Type is either "i" (stands for input)

or "o" (stands for output)

TSAP-Id is the identifier for the access point at which the interaction occurs.

Note that if the interaction sequence consists of only input interactions then the Interaction-Type is dropped for simplicity.

Three examples of generating traces of the abstract TS provider are given in Appendix G. In these examples, "totgen" is invoked via procedure "run-totgen" which includes a group of sample input interaction sequences. Each invocation of procedure "run-totgen" contains the sequence number of the input interaction sequence to be passed to "totgen".

5.3.2. Input Directed Trajectory Generation

Since input directed trajectory generation is basically the same as input directed trace generation, the algorithm for trace generation may also be used for trajectory generation. That is, the algorithm given in the previous

section generates both traces of representation K and trajectories of representation K-1. Note that trajectory generation involves a representation which has already been validated. Thus, one normally would not encounter the problems such as unspecified receptions, deadlocks, infinite idle loopings that might occur during trace generation. Therefore, the control steps included in the algorithm to check the possible occurrences of such problems may not be necessary during trajectory generation. However, the control steps may be preserved in order to permit the algorithm to be employed for both trace and trajectory generation.

It is important to note that once representation K has been validated, certified traces obtained during its validation can be considered as trajectories for (or potential traces of) representation K+1.

6. CONSISTENCY CHECKING

The two-way complementary consistency checking, namely trace and trajectory checking is a critical component of the life-cycle testing approach. In this chapter, we first present algorithms for invocation routines which automatically invoke PROLOG encodings of executable representations in order to facilitate trace and trajectory checking. Then, we present test architectures for local and remote testing of protocol implementations being developed.

6.1. Trace and Trajectory Checking

In this section, we demonstrate how PROLOG procedures corresponding to an abstract representation K (e.g., the detailed design of a system) are used to check:

a) whether traces obtained by observing the execution of a less abstract representation K+1 (e.g., an implementation of the system) are permissible execution histories and

b) whether trajectories obtained by executing a more abstract representation K-1 (e.g., a functional specification of the system) are achievable.

Recall that a trace or a trajectory is expressed in terms of input and output interactions of the representation that

it is obtained from: A trace of representation K+1 is a sequence of interactions observed during its execution whereas a trajectory obtained from representation K-1 corresponds to an intended sequence of interactions of representation K. Thus, checking both whether a trace of representation K+1 is a permissible execution history or whether a trajectory of representation K-1 is achievable may be performed by representation K in the same manner: In each case, the interaction sequence is read sequentially from left to right and a match for the interaction sequence is searched. For each input and output interaction all possible states that the implemented process can assume and all possible transitions (including internal transitions) allowed at those states are investigated. The states and transitions which do not follow from or lead to the interaction being analyzed are not further considered in the analysis. Therefore, all subsequences of interactions which do not lead to a match for the given interaction sequence are eliminated.

The search for a match may be guided by carefully constructing sequences of invocations of PROLOG procedures corresponding to executable representation K from a given trace or a trajectory. In fact, assuming that no two input or no two output interactions may occur simultaneously at the interaction points of a representation and a representation may have only two interaction points (say channel a and channel b), the invocation sequence

construction process can be mechanized in the following manner:

Step 1) Start with the first interaction in the given interaction sequence (a trace or a trajectory) to be validated;

if the interaction is an input interaction Ia (stands for an input interaction at channel a) and if it is followed by:

a) another input interaction at channel a, take Ia

b) an input interaction at channel b, take Ia

c) an output interaction Oa at channel a, take Ia and Oa as a pair and later (if the previous invocation fails to lead to a match for the interaction sequence) take Ia alone

d) an output interaction Ob at channel b take Ia and Ob as a pair and later (if the previous invocation fails to lead to a match for the interaction sequence) take Ia alone

if the interaction is an output interaction Oa then take Oa alone

Step 2) Consider the input interaction, the output interaction, or the input-output interaction pair obtained in step 1), and prepare the invocation for the PROLOG procedure corresponding to executable representation K. Since each representation is assumed to have two channels, each invocation for the

corresponding PROLOG procedure is a four-tuple containing the values of input and output interactions occurring at these channels. Based on the above assumptions, the following table summarizes the possible invocations for representation K:

Interaction Point A		Interaction Point B	
INPUT	OUTPUT	INPUT	OUTPUT
Ia	Oa	void	void
Ia	void	void	void
Ia	void	void	Ob
void	Oa	void	void
void	Oa	Ib	void
void	void	Ib	Ob
void	void	Ib	void
void	void	void	Ob
void	void	void	void

Notice that it may be required to invoke representation K by the last invocation in the table in order to let the representation execute any internal transition.

Step 3) Consider the next interaction in the given interaction sequence following the interaction pair or single interaction obtained in step 2), and continue likewise from Step 1).

Notice that the above assumptions are made for illustration purposes only. In fact, this algorithm can be generalized for cases where the representation has more than two interaction points easily. Both the above algorithm and several of its variations have been implemented in PROLOG

and extensively tested.

An invocation sequence can be applied to representation K via a higher level PROLOG procedure, denoted the validator. This higher level PROLOG procedure can also control and monitor the matching process and generate discrepancy reports. The efficient use of such high-level PROLOG control procedures has been reported in [Prob 83].

Depending on whether or not a given PROLOG implementation corresponds to a deterministic or a non-deterministic system functionality representation, different algorithms for validators may be designed. In the following subsections, we present two algorithms for validators of traces and trajectories. In order to simplify the presentation we assume that the invocation sequence corresponding to a trace or trajectory is provided to the algorithm.

6.1.1. Validators for Deterministic Representations

Since in a deterministic representation there are no internal transitions and not more than one input initiated transition at any state for a given input interaction, the validation of a trace or trajectory is relatively simple. A general form for validators for deterministic representations is as follows:

```
procedure: validator(INITIALSTATUS, IS, FINALSTATUS)
    PRESENTSTATUS <-- INITIALSTATUS
```

```
    for each invocation I in IS (invocation sequence) do
        representation(I, PRESENTSTATUS, NEXTSTATUS)
        if NEXTSTATUS is undefined
           then report('sequence is invalid')
           else PRESENTSTATUS <-- NEXTSTATUS
        endif
    repeat
    if PRESENTSTATUS ≠ FINALSTATUS
      then report('sequence is invalid')
      else report('sequence is  valid ')
    endif
end validator
```

The algorithm determines whether the trace or the trajectory corresponding to the given invocation sequence is a valid or an invalid interaction sequence with respect to representation K. Accordingly, starting from an initial status, the algorithm applies invocations one at a time to the representation. Each invocation takes the representation from one status (i.e., PRESENT STATUS) to the next (i.e., NEXT STATUS). The latter is assigned as the new present status of the representation when the next invocation is applied.

However, if the current invocation fails to yield a next status (i.e., NS is undefined), the algorithm can no longer continue with the application of the invocation sequence and terminates by declaring the interaction sequence invalid.

This is the case when the representation can not legitimately change its status using the current invocation. That is, there are no production rules corresponding to a status change with those values of input and output interaction variables or queues involved in the invocation. On the other hand, a given trace or trajectory is recognized as a permissible interaction sequence if the status of the representation after the application of the last invocation is the same as the given final status. If this is not the case, then the algorithm declares the given interaction sequence invalid.

Note that this algorithm can also be used to check whether a given status can be reached from another given status by the application of the invocation sequence IS. This may be particularly useful for debugging purposes.

6.1.2. Validators for Non-deterministic Representations

As stated previously, a non-deterministic representation contains internal transitions and/or more than one possible input-initiated transition for a given input interaction at some states. The existence of such transitions require a more careful analysis during the validation of a trace or trajectory. The following algorithm is designed particularly for applying invocation sequences for a given trace or a trajectory to a non-deterministic representation in order to check whether the given trace or trajectory is permitted by

the representation.

Due to the non-deterministic nature of the representation, there may be more than one possible status that the representation may assume after the application of an invocation. For example, from a given status, there may be more than one possible input initiated transition (and hence, more than one possible status that the representation may assume) with an invocation consisting of an input interaction at one of the interaction points of the representation. As well, there may be internal transitions causing status changes from any status reached after the application of the above invocation. Therefore a validator should first obtain a set (SNS) of all possible next status values that the representation may assume after the application of an invocation (Ii) and then apply the next invocation (Ii+1) to each possible status in set SNS, and so on.

The construction of set SNS at step i (where the invocation Ii is applied) can be performed in three steps:

a) The invocation Ii is applied to each status in set SPS (i.e., the set of all possible status values that the representation can assume after the application of the invocation Ii-1). This results in a set SSNS which consists of all possible status that can be reached by the application of Ii to each status in SPS, without letting the representation change any

status via internal transitions.

b) Let the representation execute each internal transition possible at each status in SSNS and all others reachable from those in SSNS by internal transitions. One way of accomplishing this is to apply a depth first search; that is, for each status Sj in SSNS, find all such paths that start from status Sj and end at a status from which no other status can be reached by an internal transition. The set of status reached on such a path (including status Sj) is the set LPSjk.

c) Take the union of sets LPSjk. The resulting set is the set SNS.

These steps are described in detail in the following algorithm:

```
procedure: validator(INITIALSTATUS, IS, FINALSTATUS)
    SPS <-- INITIALSTATUS
    for each invocation Ii in IS (invocation sequence) do
        SSNS <-- empty
        for each status PS in SPS (set of present status) do
        * invoke representation K by Ii*
            setof(NS, representation(Ii, PS, NS), LS)
            * LS is the set of all such next status NS
              reachable from PS with application of Ii*
            SSNS <-- SSNS U LS
```

```
repeat
* SSNS is the set of all such next status NS's
  that can be reached from each status in SPS
  when Ii is applied to representation K    *

* check whether there is an unspecified reception *
if SSNS = empty
   then report('sequence is invalid') exit
   else continue
* find all such status values that can be  reached
  by internal transitions from each status in SSNS *
for each PSj in SSNS do
    k <-- 1
    LPSjk <--  empty
    * PATHS is the set of all paths starting at PSj
      and ending at a status from which no other
      internal transition is possible                *
    setof(PATH, paths (PSj, PSj, PATH), PATHS)
    * remove multiple occurrences of status in PATHS *
    for m=1 to |PATHS| do
        LPSjk <-- LPSjk U PATHm in PATHS
    repeat
    k=k+1
repeat
* obtain set SNS by taking the union of all LPSjk's *
for m=1 to k do
    SNS <-- SNS U LPSjm
repeat
```

```
    * SNS is the set of all possible  status that
      representation K can be after invocation. Ii
      has been applied to representation K        *
    SPS <-- SNS
repeat
* When there is no  more invocation in IS; check
  whether the set  SPS obtained at the last step
  contains the FINALSTATUS or not                 *
for each status PS in SPS do
    if PS = FINALSTATUS
       then report('sequence is valid') exit
       else continue
    endif
repeat
report('sequence is invalid')

procedure : paths(CS, PSF, PATH)
* find a path from CS to a status from which
  no other internal transition is possible *
     representation(empty, CS, NS)
If there is a NS
   then if NS is_a_member_of PSF
           then report_error('infinite-loop')
           else PATHSF <-- PSF U NS
           paths(NS,PATHSF,PATH)
        end if
   else PATH <-- PATHSF
end if
```

end paths.

end validator.

The algorithm employs two procedures, namely, "setof" and "paths" which were discussed in section 5.3. Notice that the representation which is used to check whether a given trajectory is allowed, is the representation under test. Thus, cycle detection is preserved in procedure "paths" in order to permit the algorithm to be employed for both trace and trajectory checking.

An implementation of this algorithm in PROLOG (denoted "checker") is given in Appendix H, together with some examples of trace and trajectory checking. Procedure "checker" has been integrated with an invocation sequence generator (denoted "invoker") corresponding to the algorithm given in section 6.1. This composite validator (denoted "totval") is invoked by an input/output interaction sequence (a trace or trajectory): The given interaction sequence is passed to "invoker" which instead of preparing the whole invocation sequence and then calling "checker", prepares each individual invocation and calls "checker" with that invocation. This process continues until the input/output interaction sequence is found to be either valid or invalid.

In the examples included in Appendix H, procedure "totval" is invoked via procedure "run-totval" which contains a group of sample interaction sequences. Each

invocation of procedure "run-totval" carries an invocation sequence number which identifies the sequence to be passed to "totval". In these examples, traces of the refined TS provider is checked by the abstract TS provider. That is, "totval" is used to invoke procedure "ts" to check whether traces of procedure "rts" are permissible interaction sequences. Notice that when the given interaction sequence is not achievable or permissible an error message is printed.

Thus far in this chapter, algorithms which are used to invoke executable system functionality representations for checking their traces and trajectories have been presented. During the presentation of algorithms, the focus has been on validating service and protocol specifications via their executable representatives constructed specifically for validation purposes. However, the algorithms can also be employed for the validation of protocol implementations (i.e., checking the conformance of a protocol implementation with the protocol specification): Traces of an EUT (protocol implementation under test) can be checked by employing the executable protocol specification (referred process "tp" in section 4.4.3) via a procedure implementing the algorithm given in section 6.1.2. Similarly, a procedure implementing the algorithm given in section 6.1.1 can be used to check whether the trajectories obtained from the protocol specification are realizable by the EUT. In fact, this algorithm may also be implemented as a "test bed"

(or as proposed by Bochmann, may be employed by an "interactive tester" [Boch 83].) for "unit testing" [Myer 79] purposes.

Unit testing of protocol implementations is an important validation activity which is performed by the implementor of the EUT, generally for debugging purposes. In unit testing, the EUT is tested in isolation from the rest of the system via controlling and observing both its upper and lower interfaces.

Besides unit testing, protocol implementations may go through performance tests, certification tests (i.e., demonstrating that the EUT conforms to the reference protocol specification), acceptance tests etc.. These tests are generally performed when the EUT is integrated into the rest of the system. In the following section, we present test architectures that are based on a model which can be incrementally expanded into local and remote testing of protocol implementations.

6.2. Architectures for Protocol Implementation Testing

In this section, we present logical test architectures for local (laboratory) and remote testing of protocol implementations. The architectures are based on a model which consists of two open systems interacting over a network connection. The model can be expanded incrementally from a laboratory environment to an environment for remote testing of protocol implementations. This incremental expansion offers a variety of test configurations for testing and debugging protocol implementations during both pre-release and post-release validation and maintenance phases of the protocol development process. In the following sections, we present the proposed model for test architectures, define the testing components, and describe the logical architectures for local and remote testing.

6.2.1. A Model for Test Architectures

The model is intended specifically for testing OSI protocol implementations for conformance to particular protocol specifications that they implement. We assume that entity under test, abbriviated as EUT, (Figure 6.1.) is a protocol implementation corresponding to one or more layers in the Reference Model of OSI [Zimm 81]. If the EUT is intended to implement more than one layer, the service that the EUT provides to the user layer above is understood to be the service provided by the uppermost layer implemented by

the EUT. To provide this service, the EUT utilizes the service provided by the next layer below its lowermost layer. For example, if the EUT corresponds to both session and transport layers, the service it provides and the service it utilizes will be the session service and the network service respectively.

The proposed model for test architectures is intended to provide a deterministic testing capability whenever possible. In other words, except for certification testing, either the test components in the architectures behave predictably or their unpredictable (unexpected) behavior can be detected and corrected before the effect of this arbitrary behavior contaminates the application of the rest of the test sequence. This capability enables the tester to recreate error conditions and to repeat the tests involving those conditions.

However, the recreation of error conditions and/or correction of unpredictable (unexpected) behaviour of the components of a test architecture (e.g., underlying communication medium) may not always be possible. For example, test architectures for certification testing are based on the assumption that only the upper layer interface of the EUT is directly accessible [Boch 83]. Thus, during certification testing, neither the unexpected failures and abnormal behaviour of the underlying communication medium can be corrected nor those error conditions can be created

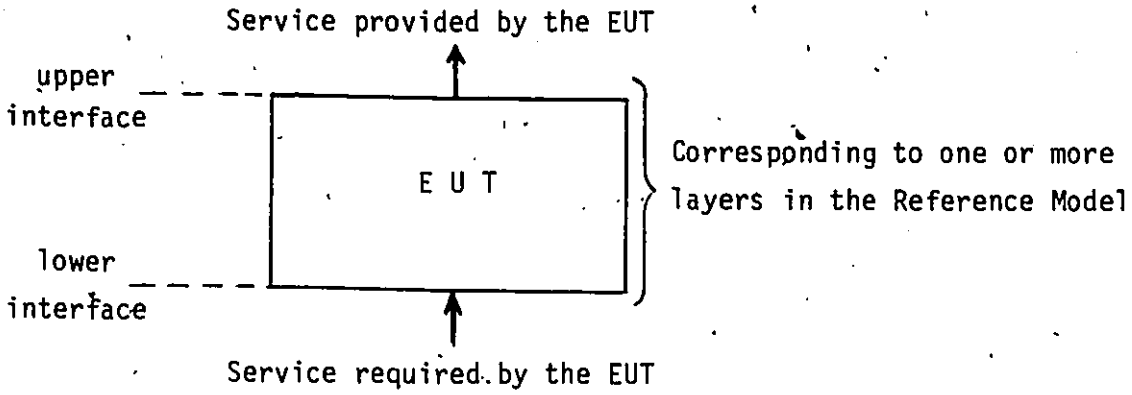


Figure 6.1 Entity Under Test

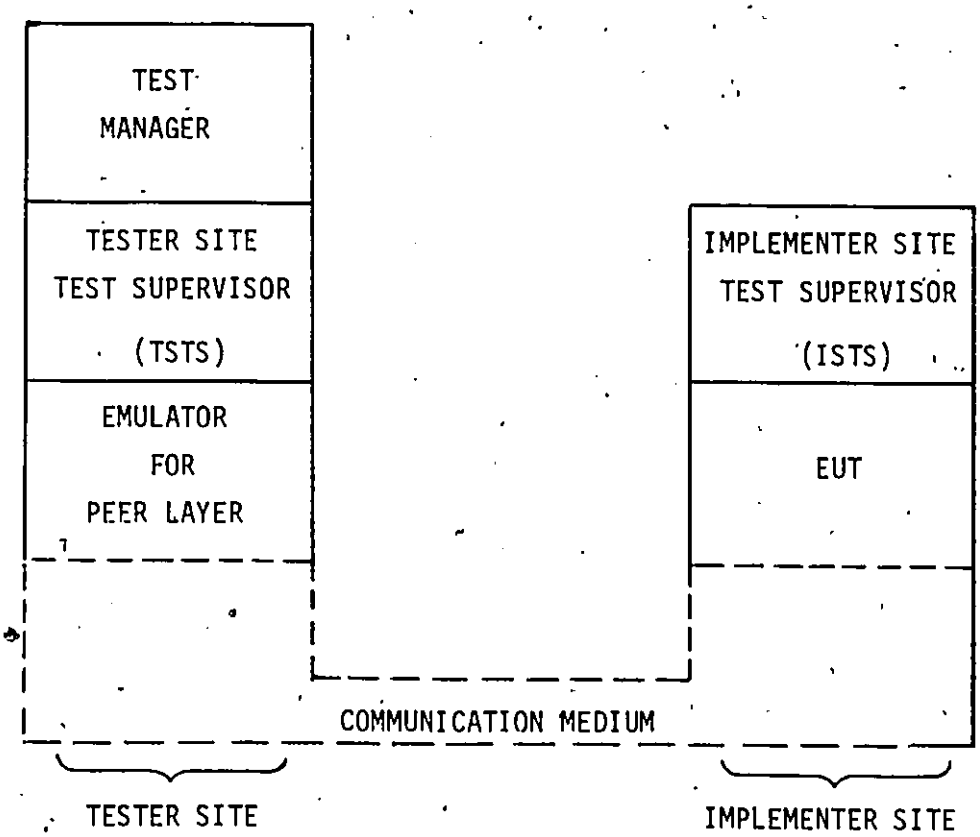


Figure 6.2 A Model for Test Architectures

intentionally [UrPr 83a]. Hence, deterministic testing capability is particularly intended for the case where testing is collaborative with the EUT developer, i.e., both upper and lower interfaces of the EUT are directly accessible. In fact, this is the main focus of the following discussion.

Figure 6.2 describes the model for the proposed architectures. The model consists of two open systems; namely the tester site and the implementer site. The names of the sites are intended to reflect the view that testing is performed by a group other than the implementing group. The implementer site configuration consists of a test supervisor (ISTS) and the EUT. The tester site configuration consists of a test manager, a test supervisor (TSTS), and an emulator for the peer of the EUT. The remaining layers under the EUT and its peer form the underlying communication medium.

In this model the test management function (possibly except for local testing) is implemented in a distributed fashion. The distributed test management is intended to allow for rapid test recovery and restart procedures and to avoid the need for a "tester-responder protocol" (cf. [Rayn 82]). The distributed management is facilitated by locally storing the roles of each test supervisor corresponding to the test sequences to be applied and by employing the same algorithm at both sites for determining the next test

sequence.

It is assumed that test sequences are stored in role files in a predetermined order according to their coverage of protocol functions. Thus, the synchronization between tester site and implementor site is enforced

a) by specific ordering of service primitives in each test sequence during the application of that sequence and

b) by specific ordering of test sequences which is taken into account by both copies of the stored algorithm at the end of the execution of each test sequence or in the case of a loss of synchronization due to a failure of either the EUT or the communication medium during the application of a test sequence.

Moreover, the model allows a variety of possible designs for the communication medium. One example is communication via an emulator for the protocol entity below the peer of the EUT in the tester site and another emulator for the rest of the medium (including all the layers up to the EUT in the implementer site). Another possibility involves using an emulator for the protocol entity below the EUT, another for the one below the peer of the EUT, and representing the rest of the medium with a single emulator or with the actual lower-layer protocol implementations and a physical link.

In the following section we describe the test components employed in the proposed logical architectures in detail.

6.2.2. Components of the Model

During testing, the EUT is subjected to controlled sequences of service primitives across both of its interfaces. The sequences of service primitives correspond to both normal (valid) and abnormal (invalid, exceptional, or error) cases. The service primitives corresponding to normal or abnormal behavior (e.g., undefined or out-of-sequence requests) of the user of the EUT can be generated and applied across its upper layer interface by the ISTS (see Figure 6.2). The service primitives which are applied across the lower layer interface of the EUT correspond to the cases related to normal or abnormal behaviors of the peer user, the peer protocol entity, and the underlying communication medium. Some of the exceptional and error cases in the test sequences should correspond to peer protocol errors and exceptions, some to invalid lower layer service primitives, and some others to failures of the underlving communication medium.

In the proposed model, the mechanisms required to generate (and recognize) some of those cases in a controlled fashion are provided by the emulators for the corresponding protocol entities or layers under the control of the test management mechanism. For example, the emulator for the peer

protocol entity is used to generate on demand peer protocol errors and exceptions in terms of invalid protocol data units (PDU) as well as to provide the normal required service of its layer. Similarly, the emulator for the next lower layer of the EUT is used to generate invalid service primitives as well as valid ones as required by the test sequence script. Note that a basic difference between emulators and enhanced reference implementations" (cf. [Rayn 82]) of the protocol layers is that emulators can be developed from scratch, designed to be independent of implementation-specific features of the EUT, and thus become more flexible and less error-prone than "enhancing" already existing implementations.

As well, an emulator for the communication medium may be used to deliberately generate exceptional and error cases corresponding to failures of the underlying communication medium (i.e., connection resets and clears, etc.) in a controlled fashion. This emulator can be a separate module providing the capability of generating error and exceptional cases as well as providing the required services at the lower layer interfaces of both the EUT and its peer. Alternatively, the emulator can be constructed by using emulators for the protocol entities in the layer below the EUT and its peer and representing the rest of the communication medium by a transformation module. This transformation module provides the required transformation of the service primitives and introduces the controlled

errors and exceptions.

The emulators for both protocol entities and protocol layers (or the communication medium) are directed either by the test manager or by the test supervisors. Basically, the test supervisors (TSTS and ISTS) can be considered as the users of the service provided by the EUT and the emulator of its peer in the tester site. They take part in the application of the test sequence being executed by the test manager. They act as communicating entities in the layer above (next to the EUT) according to their roles which are specified in test sequences. In fact, they may read their roles from role files, initiate or respond to service primitives, set timers, initiate time-out mechanisms, control and direct the emulators in their sites to generate PDU's and service primitives according to their roles, and record observed interactions onto log files. Note that the need for a "Test Driver-Responder Protocol" (cf. Rayn 82) or a "Testing Protocol" (cf. Boch 83) between TSTS and ISTS is avoided by supplying the roles of the ISTS corresponding to the test sessions to be applied in advance. The synchronization between the tester site and the implementer site is enforced by the specific orderings of the service primitives received and transmitted during the application of a particular test sequence. As well, the test sequences are stored in a predetermined order in each test file. Hence, in the case of a loss of synchronization between the sites due to an error in the EUT, the selection of the next

test sequence is performed by the copies of the same algorithm implemented by both test supervisors. The next test sequence to be applied is determined by this algorithm according to the test parameters and the results of execution of previous test sequences.

The primary function of the test manager is to assist test personnel to run the test procedures. A test procedure contains one or more test sessions involving the interaction sequences observed and initiated by test supervisors, test setup parameters, timing parameters, test reporting parameters all coded in a formal test procedure language [PrUr 83]. The test manager either automatically or after prompting by the tester initiates a test session in the test environment. The initiation and management of a test session involves constructing test sequence scripts defining the roles of the test supervisors, initiating procedures (i.e., timers, logging), passing the roles and test parameter values to test supervisors, and invoking the test supervisor responsible for initiating connection establishment for the required connection in the current test session (or for the first connection in a test session involving multi-connections).

Each test session is realized by the mutual interaction of the ISTS and TSTS. Reports on the actual test session can be generated from the log files at the end of the session (or whenever it is no longer possible to continue with the

session). The actual test session concludes with the determination of whether the test has failed, succeeded, or should be re-applied. In addition, a decision needs to be made as to whether to attempt to recover from a failed, incomplete session and to continue the session. These decisions are based on the comparison of the expected interaction sequence to the observed interaction sequence. We call the module in the test manager which performs this comparison the test consultant (another common term is "test oracle").

The complexity of the comparison function performed by the test consultant varies with the predictability of the behavior of the communication medium. This function is trivial when the behavior of the communication medium is predictable (i.e., when an emulator for the communication medium is being employed). Since any anomalies introduced by the emulator are deliberately generated (under the supervision of the TSTS or the test manager), a direct comparison of the test sequence to the observed interaction sequence is sufficient to decide whether the test failed or succeeded.

On the other hand, when the behavior of the communication medium is arbitrary (i.e., when a real connection is being employed) the test consultant provides a more complex comparison function. First it should check whether the arbitrary behavior of the communication medium (reported by

probes) has altered the way that the test sequence should have been applied. If it has been altered then the decision should be either to reapply the test sequence or to recover and continue. If the validity of the test has not been compromised, then the decision should be either to accept the test or to reject it.

As stated previously, a requirement for deterministic testing is to control and monitor the interactions taking place across both interfaces of the EUT. We propose using probes at interfaces which are not directly controlled by test supervisors. Particularly, it may be useful to place a probe at the lower layer interface of the EUT in a test environment containing a real connection between sites to ensure that messages are not altered while they are in transit due to failures of the lower layers. It is also possible to embed a timing function into a probe to time stamp the information obtained as a snapshot. Such information may be used to differentiate between the errors caused by the EUT and the errors introduced by the communication medium.

We call a probe which merely takes snapshots of the interactions across an interface a passive probe. For example, a passive probe may be integrated into the emulator for the lower layer of the EUT or can be a separate module writing snapshots onto the log file primarily used by the ISTS. On the other hand, we call a probe which modifies

interactions across an interface in addition to monitoring them an active probe. For example, the emulator for the protocol entity in the layer below the EUT can be considered to be an active probe directed by the ISTS to perform the required modifications (i.e., corrections) on certain interactions. A natural extension of utilizing probes on the test environments is to provide the probes the ability to access the role files directly to make them independent of test supervisors. We call such probes intelligent probes. Intelligent probes perform the required functions according to the control information they obtain from role files combined with information describing the test context.

It is advantageous to design the components of a test architecture to be as independent of the implementation specific features of the real communication medium and/or the EUT as possible. One such feature is the type and format of the service primitives and protocol data units of protocol layers. It is possible to design the test components by assuming a specific convention for the associated message primitives. Once these message primitives are specified, it is rather easy to develop Encoders-Decoders (ED) to bridge the syntactic gap between these primitives and the given implementation-specific primitives employed by the EUT and the real communication medium. The encoders/decoders may then be coded either as separate units or as integral parts of the emulators for certain layers which have an interface with the EUT or the

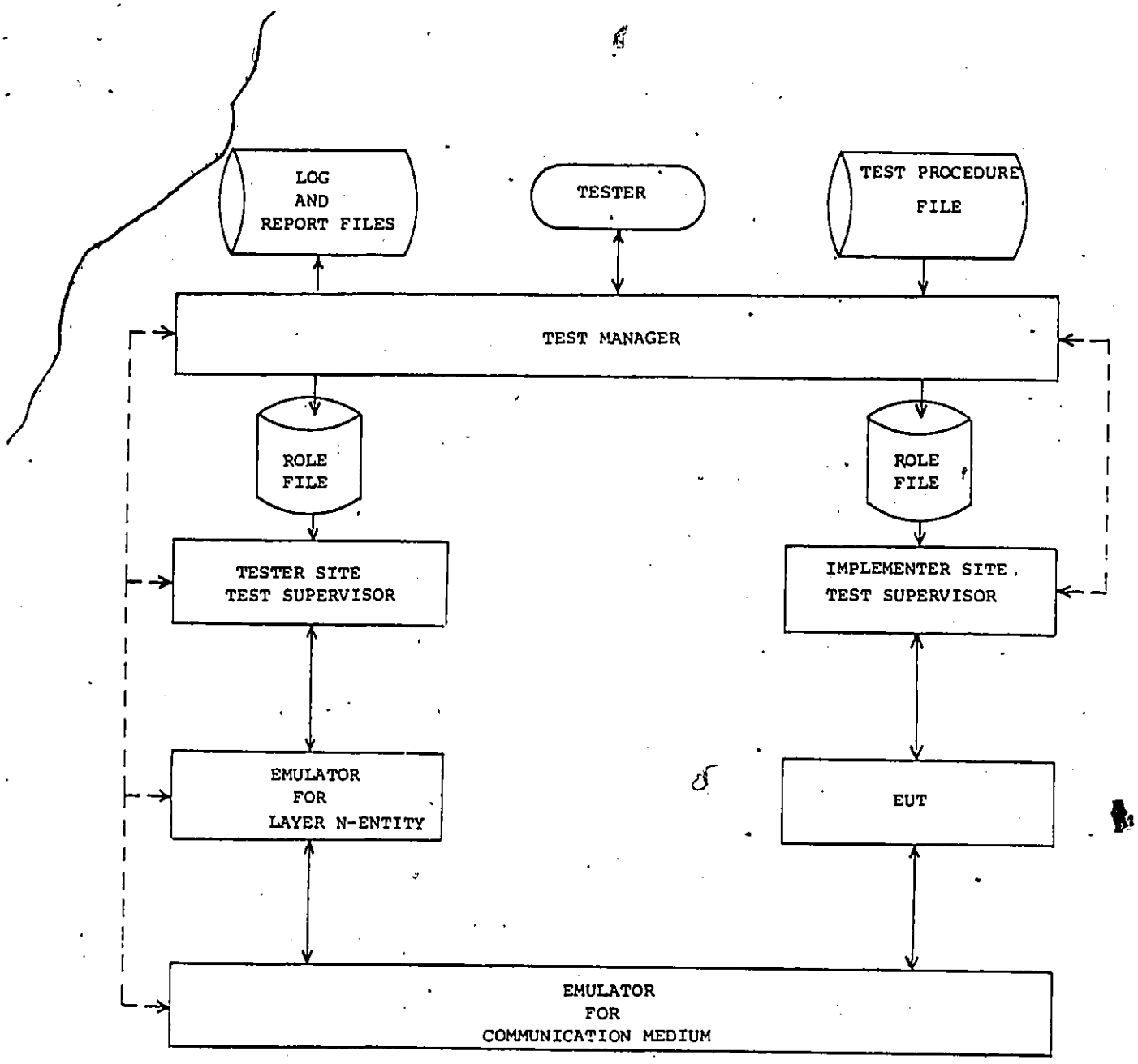


Figure 6.3 A Logical Architecture for Local Testing

actual communication medium.

In the next section, we describe two possible logical test architectures, one for local testing and the other for remote testing.

6.2.3. A Logical Test Architecture for Local Testing

Generally, the messages exchanged between communicating entities in computer communication networks are subject to arbitrary transmission delays, distortions, reorderings over connections which themselves may arbitrarily be reset, cleared, or disconnected. This arbitrary behavior of the communication medium and the resultant anomalies should be taken into account during protocol implementation testing.

In the test architectures employing an actual connection between two open systems, there is no control over the behavior of the communication medium. In other words, the anomalies introduced by the communication medium can not directly be generated in a controlled fashion. They occur, although rarely, in an unpredictable way and their occurrences have to be revealed during testing and should not be considered as errors caused by the EUT. In addition to this, certain exception and error conditions cannot be induced at the implementer site by any action in the tester site due to the limitations imposed by the actual communication medium (c.f. [Rayn 81]).

On the other hand, test architectures employing an emulator for the communication medium provide the capability of creating a controlled hostile environment for the EUT. In fact, the logical architecture given in Figure 6.3 describes a local testing facility where the behavior of the communication medium and hence the lower layer interface of the EUT are directly controlled by the test manager. The emulator can be viewed as a software module designed to perform required mapping functions, to augment, alter, or modify selected service primitives, to generate certain exception indications and errors, and to record the interactions at its interfaces. This module can be constructed by using emulators for the corresponding lower layer protocol entities and service providers and a control sub-module interacting with the test manager.

The functions of the other test components in the architecture given in Figure 6.3 are the same as described previously. Note that the test manager communicates directly with both the TSTS and the ISTS. This provides an alternative synchronization mechanism between the two (virtual) sites in addition to the one provided by the ordering of tests.

The emulator for the communication medium enables the test personnel to apply selected stimuli at the lower layer interface of the EUT, particularly those which correspond to tests to determine the defensive behavior of the EUT under

abnormal and hostile conditions. Examples for these tests include out-of-sequence or unspecified service primitives, loss or misorderings of PDU's due to failures of the layers below the EUT for testing error recovery procedures, or transmission delays for testing time-out mechanisms, etc.

The architecture for local testing can be used for performance testing and for debugging purposes, particularly later in the development process to facilitate regression testing which is performed on a modified implementation with the previously applied test procedures. The aim of regression testing is to verify that the modifications had only the intended local effects and nothing more.

6.2.4. A Logical Architecture for Remote Testing

Another logical architecture that can be derived from the proposed model for test architectures is given in Figure 6.4. This architecture is intended specifically for remote testing which is particularly suitable for performance, robustness, and acceptance testing of protocol implementations. In remote testing the EUT is tested over an actual connection between the tester site and the implementer site. The actual connection is realized, in general, through some real networks. Accordingly, we assume that the layers below the lower layer of the EUT and its peer are represented by actual protocol implementations over a physical link.

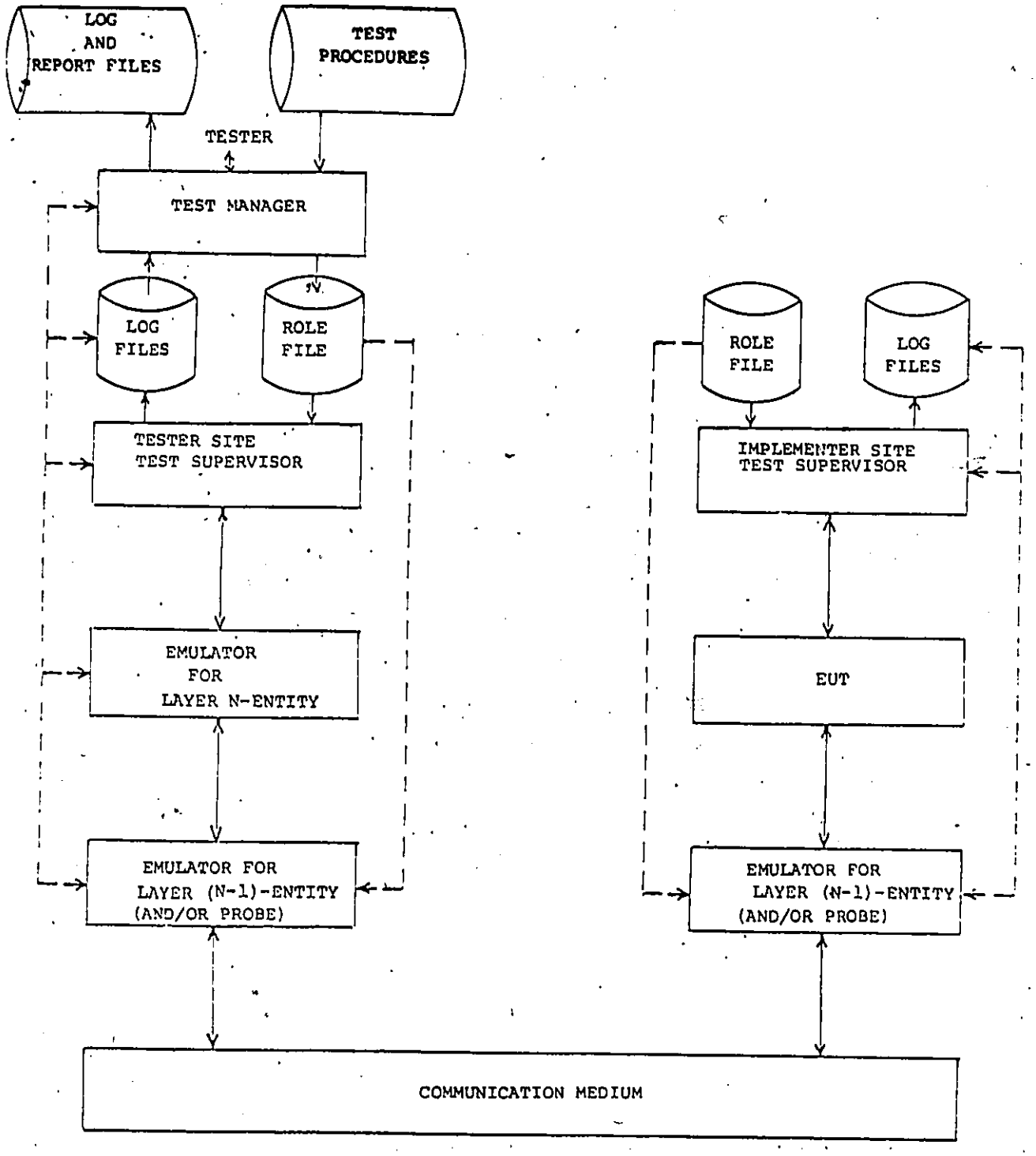


Figure 6.4 A Logical Architecture for Remote Testing

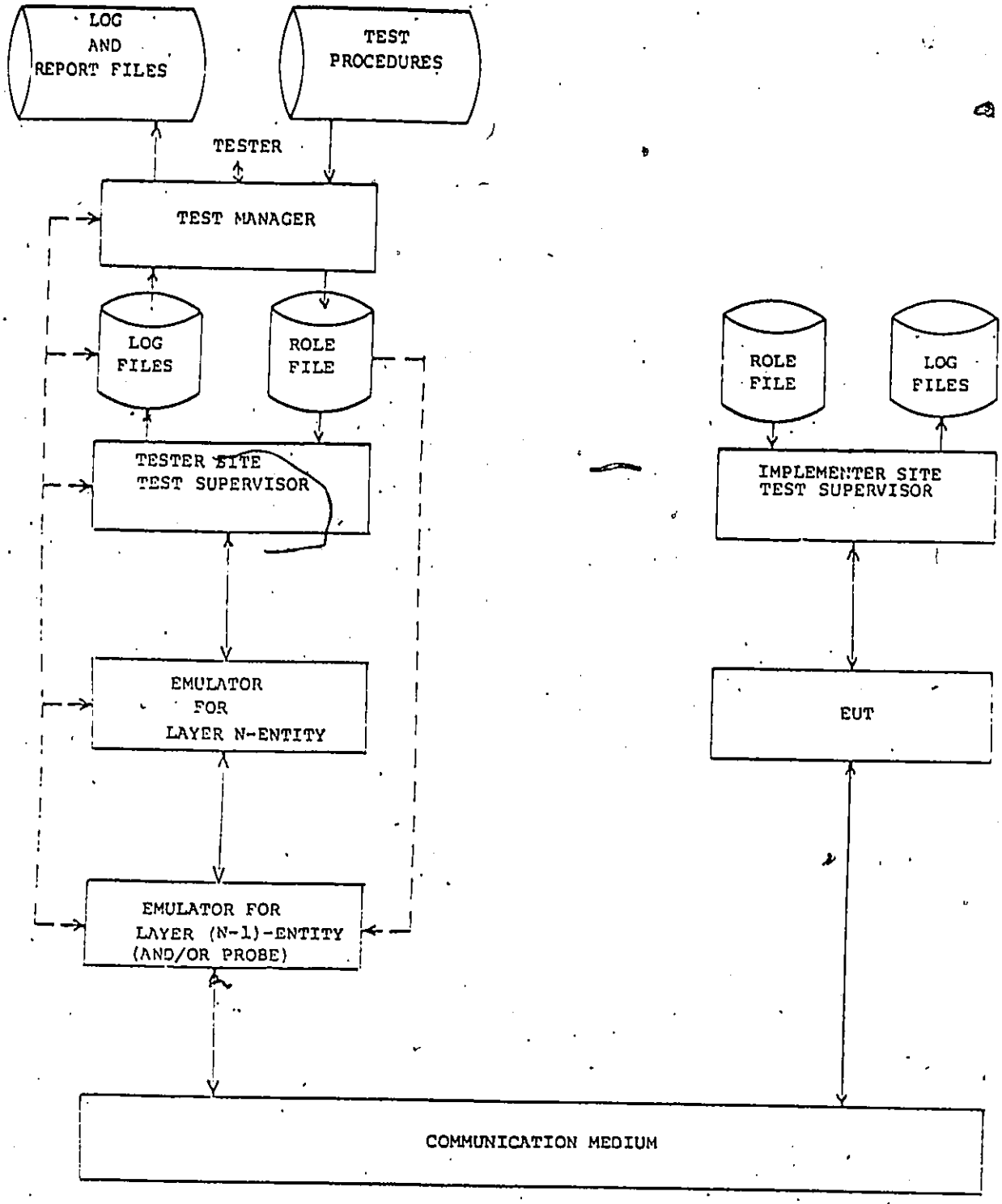


Figure 6.5 A Logical Architecture for Certification Testing

The other two components of the communication medium for layer N (the layer corresponding to the EUT) are represented by two emulators. Both of these emulators may perform the functions of probes (active, intelligent) in addition to the functions described in section 6.2.3. Specifically, the emulator for the N-1 protocol entity in the tester site may be directed by the test manager, by the TSTS, or alternatively may have access to the role file which contains specific control instructions for the activation of certain functions during the application of each test session. It can be used to provide N-1 service, perform additional mapping functions (i.e., as Encoders/Decoders (ED)), generate N-1 exceptions and errors, introduce deliberate delays, record the activity across its lower layer interface into a log file, and report any errors encountered to the TSTS and/or the test manager.

On the other hand, the emulator for the N-1 protocol entity in the implementer site has additional functions relative to its peer in the tester site. These additional functions are similar to those of the Portable Testing Unit (PTU) (cf. [Rayn 82]). However, in our proposed architecture a control protocol such as "PTU Control Protocol" is not required between this emulator and the emulator for N protocol entity or N-1 protocol entity in the tester site. Instead, the emulator operates independently (e.g., as an intelligent probe) and its operation is synchronized with the rest of the environment through its direct access to the

role files containing control information as probe scripts. In addition to the functions described for its peer in the tester site, the emulator uses the role file to generate those conditions which cannot be induced at the lower layer interface of the EUT by any action taking place in the tester site. Moreover, it can be used to maintain the synchronization between the tester site and the implementer site as well as to recover from errors and exceptions introduced by the lower layers. The functions of the remaining components in the proposed remote testing architecture are the same as described in the previous section.

A variation of this architecture (see Figure 6.5) which does not include the emulator for the N-1 protocol entity in the implementor site can be used for certification testing. Notice that in Figure 6.5 the lower layer interface of the EUT is no longer directly accessible. Certification testing related issues are covered extensively elsewhere [Rayn 82, Boch 83]. Therefore, we do not attempt to elaborate on the possible use of this architecture for certification purposes.

7. SUMMARY and CONCLUSIONS

The validation approach presented in this thesis employs testing as a means of checking the consistency of system functionality representations produced throughout a typical protocol development life-cycle. This life-cycle testing approach is realized by

i) contructing executable representations

ii) revealing the system functionality captured by these representations in terms of their externally observable behaviour (i.e., sequences of input and output interactions)

iii) checking the mutual consistency of these representations via their externally observable behaviour.

Executable representations are constructed in an attributed context-free grammar formalism and encoded in sequential PROLOG. These representations describe the system functionality in terms of externally observable behaviour of the system (i.e., in terms of possible types and orderings of interactions that the system exchanges with its environment). In this formalism, a grammar corresponding to a representation defines all possible orderings of input and output interactions via the possible tours of the state space of global system status. An

integral part of such a grammar is the definition of all possible single global status changes. Assuming that each process is described by an extended FSM, each possible global status change is caused by a state transition in one of the processes in the system. In fact, it is demonstrated that the FDT language of ISO Subgroup B can be naturally encoded in this attributed grammar formalism. At present, a prototype translation system has been developed to carry out translations automatically from the extended FSM into the attributed grammar representation and from attributed grammar representations into interpretable PROLOG encodings. It is also observed that similar construction strategies can be devised for other specification techniques such as "Horn-Clauses" [Sidh 83] and "Abstract Data Types" [Logr 83], to obtain executable specifications in attributed grammars encoded in PROLOG.

The PROLOG procedures implementing distinct representations of the same system functionality produced at each (successive) phase(s) of the protocol development life-cycle can be executed as generators and validators of both actual and potential execution histories which are observed and intended sequences of input and output interactions, respectively. Briefly, a PROLOG encoding of representation K (implementing all possible single global status changes described in that representation) can be executed

i) as a generator of its own actual execution

histories (called traces) or alternatively of potential traces (called trajectories) of representation K+1,

ii) as a validator in acceptor or in recognizer mode to check whether

(a) traces of representation K+1 are permissible and

(b) trajectories implied by representation K-1 are achievable.

This capability allows a two-way consistency checking scheme in which the consistency of representation K can be checked against an already validated representation K-1 by checking whether

(a) traces of representation K are allowed by representation K-1 (called trace checking) and

(b) trajectories dictated by representation K-1 are achievable by representation K (called trajectory checking).

It is our tendency to regard trace and trajectory checking as being complementary validation approaches. However, the degree of overlap between these two approaches is for further study.

An important requirement of both trace and trajectory checking is the generation of a restricted set of representative tests as traces and trajectories (or test sequences). In this thesis, three different test generation modes are identified: namely, systematic, user-guided, and input-directed generation. For systematic generation, an automated technique is presented for generating interaction sequences syntactically defined by the grammatical structure underlying executable representations. Like products of other systematic test generation techniques, the resulting tests are restricted to checking the syntax ("control structure") of executable representations. In order to assign semantics to tests, a user-guided generation methodology which aims at integrating the tester's understanding of communication protocols semantics into the test generation process is proposed. This methodology involves constructing specifications of semantically meaningful tests in an attributed grammar formalism. The resulting test grammar is encoded in PROLOG as a generator program which can be executed in a user controlled manner. The usefulness of this methodology is demonstrated by constructing a test specification from a simplified class 0 transport protocol specification. The development of such test specifications early in the protocol development life-cycle may provide a mechanism for reconciliation of communication service and protocol specifications.

Possible outcomes of both systematic and user-directed test generation are generally either sequences of input interactions alone (i.e., test sequences) or of both input and output interactions (e.g., trajectories or traces). Another way of generating traces and trajectories of executable representations is executing these representations using a set of selected sequences of input interactions. This corresponds to input-directed generation which is facilitated by a set of input interaction sequences selected either systematically or via user guidance. Notice that input-directed generation is not a new test generation approach, rather it is an effective and controlled way of generating externally observable behavior of representations.

Algorithms for invocation routines which automatically invoke a PROLOG procedure corresponding to a system functionality representation in order to generate or validate traces and trajectories are presented in the thesis. These algorithms have been implemented, executed, and extensively tested. Some example runs together with codes implementing these algorithms which invoke an executable transport service specification are given in appendices.

It is important to note that these algorithms are intended to avoid searching through the entire state space of global system status during generation and validation of

traces and trajectories. Thus, the algorithms generate and evaluate only those paths which are implied by the given interaction sequence. As well, the algorithms avoid employing depth-first search through the state space whenever possible in order to limit the amount of back-tracking to those absolutely necessary cases, e.g., to generate alternative traces.

Our experience with this approach has been limited to validating transport service and (class 0) transport protocol specifications via their executable representatives. The proposed algorithms can potentially be employed for debugging protocol implementations during unit testing. In addition, test architectures for both local and remote testing are proposed for other protocol implementation testing practices, such as performance, acceptance, certification testing where the EUT is integrated to the rest of the communication system. The architectures are particularly intended for collaborative testing where the EUT developer permits direct access to both upper and lower layer interfaces of the EUT. The model underlying test architectures employs a distributed test management function and thus avoids the need for a tester-responder protocol.

Since the existence of non-determinism in the specifications allows certain choices for an implementation, in general dynamic testing and in particular trajectory

checking (which are based on predetermined test cases or sequences) both encounter some problems (cf. sections 2.1 and 3.1.3). In trajectory checking, these problems appear only when the EUT (entity under test) is unable to provide capabilities to undo the choices that it has taken which do not lead to the expected response included in the applied test sequence. In this respect, there is a need for an adaptive testing strategy which does not rely on the predetermined expected behavior of the EUT but rather adapts to the unpredictable choices of the EUT in responding to applied input stimuli. A possible and convenient way of adapting to responses of the EUT seems to be employing the executable specification as an on-line test oracle. Further research should be carried out along these lines.

It is our belief that the generality of the underlying models (e.g., extended FSM) and the use of attributed grammar formalisms extend the applicability of the approach to other software development environments progressing via step-wise refinements where system specifications are amenable to representations via Finite State based formalisms. At the same time, it should be noted that the use of PROLOG interpreters as run time systems for attributed grammar based representations results in simpler algorithms for trace and trajectory generators and checkers. Thus, PROLOG provides a significant assist to implementing our approach.

REFERENCES

[AnDa 82] Ansart, J.P. and J. Damidau, "CERBERE: A tool to keep an eye on high level protocols", in Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp.529-537.

[Ansa 81a] Ansart, J.P., "Development and certification of standardized protocols", in Networks From the User's Point of View, (ed. L. Csaba et al), North-Holland, 1981, pp.477-487.

[Ansa 81b] Ansart, J.P., "Test and certification of standardized protocols", in Protocol Testing - Towards Proof? Vol.2: Testing and Certification, (ed. D. Rayner et al), NPL, 1981, pp.119-126.

[Ansa 81c] Ansart, J.P., "Tools for the certification of standardized protocols: Cerbere and Genepi", in Protocol Testing - Towards Proof? Vol.2: Testing and Certification (ed. D. Rayner et al), NPL, 1981, pp.127-130.

[Ansa 82] Ansart, J.P., "GENEPI/A: a protocol independent system for testing protocol implementation", in Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp.523-528.

[ApMe 82] Apt, K.R. and M.H. van Emden, "Contributions to the Theory of Logic Programming", Journal of the ACM, Vol.29, No.3, 1982, pp.841-863.

[Bart 69] Bartlett, K.A. et al, "A note on reliable full-duplex transmission over half-duplex links", CACM, Vol.12, No.5, 1969, pp.260-261.

[BeLy 77] Belsnes, D. and E. Lynning, "Some problems with the X.25 packet level protocol", ACM Computer Communication Review, Vol.7, No.4, 1977, pp.41-51.

[BoCe 82] Bochmann, G.V. and E. Cerny, "Protocol Assessment", Report for Dendronics Decisions Ltd., Feb 1982.

[Boch 75] Bochmann, G.V., "Logical verification and implementation of protocols", Proc. of 4th Data Communications Symposium, Quebec, 1975, pp.8.15-8.20.

[Boch 77] Bochmann, G.V., "Notes on the X.25 procedures for virtual call establishment and clearing", ACM SIGCOMM Computer Communication Review, Vol.7, No.4, 1977, pp.53-59.

[Boch 78] Bochmann, G.V., "Finite state description of communication protocols", Computer Networks, Vol.2, No.4/5, 1978, pp.361-372.

[Boch 80] Bochmann, G.V., "A general transition model for

protocols and communication services", IEEE Trans. on Communications, Vol.28, No.4, 1980, pp.643-650.

[Boch 82] Bochmann, G.V. et al, "Some Experience with the use of formal specifications", in Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp.171-185.

[Boch 83] Bochmann, G.V. et al, "Testing transport protocol implementations", Proc. of CIPS Conference 1983, Ottawa, May 1983, pp.123-129.

[BoGe 77] Bochmann, G.V. and J. Gecsei, "A unified model for the specification and verification of protocols", Proc. of IFIP Congress, 1977, pp.229-234.

[BoRa 82] Bochmann G.V. and M. Raynal, "Structured specification of communicating systems", Publ. 428, Dept. d'IRO, Universite de Montreal, 1982.

[BoSu 80] Bochmann, G.V. and C.A. Sunshine, "Formal methods in communication protocol design", IEEE Trans. on Communications, Vol.28, No.4, 1980, pp.624-631.

[BrJo 78] Brand, D. and W.H. Joyner, "Verification of protocols using symbolic execution", Computer Networks, Vol.2, No.4/5, 1978, pp.351-360.

[BrJo 79] Brand, D. and W.H. Joyner, "Verification of HDLC", IBM Research Report RC 7779, July 1979.

[Brzo 62] Brzozowski, J., "A survey of regular expressions and their applications", PGEC, 11, 3, 1962, pp. 324-335.

[Chow 78] Chow, T.S., "Testing software design modeled by finite state machines", IEEE Trans. on Soft. Eng., Vol. SE-4, No.3, 1978.

[Colm 78] Colmerauer, A., "Natural language communication with computers", in Lecture Notes in Computer Science (L. Bolc), Springer-Verlag, New York, 1978.

[DaBr 75] Danthine, A.A.S. and J.J. Bremer, "Communications protocols in a network context", Proc. of Interprocess Communication Workshop, Santa Monica, 1975, pp.87-92.

[DaBr 78] Danthine, A.A.S. and J.J. Bremer, "Modelling and verification of end-to-end transport protocols", Computer Networks, Vol.2, No.4/5, 1978, pp.381-395.

[Dant 80] Danthine, A.A.S., "Protocol representation with finite state models", IEEE Trans, on Communications, Vol.28, No.4, 1980, pp.632-642.

[Dini 70] Dinic, E.A., "Algorithm for solution of a problem of maximum flow in a network with power estimation", Soviet Math. Dokl., Vol. 11, 1970, pp. 1277-1280.

[DuHu 81] Duncan, A.G. and J.S. Hutchison, "Using attributed grammars grammars to test designs and implementations", Proc. of 5th Int. Conf. on Software Engineering, 1981, pp.170-178.

[Even 79] Even, S., Graph Algorithms, Computer Science Press, 1979.

[Floy 67] Floyd, R.W., "Assigning meaning to programs", in Mathematical Aspects of Computer Science (ed. J.T. Schwartz), American Mathematical Society, 1967, pp. 19-32.

[FoFu 62] Ford, L.R. Jr., and D.R. Fulkerson, Flows in Networks, Princeton University Press, 1962.

[Fost 80] Foster, K.A., "Error sensitive test case analysis (ESTCA)", IEEE Trans. on Soft. Eng., Vol. SE-6, No. 3, 1980, pp. 258-264.

[GoCo 78] Good, D.I. and R.M. Cohen, "Verifiable communications processing in GYPSY", Proc. of 17th IEEE COMPCON, Sep 1978, pp.28-35.

[Gogu 80] Goguen J., "Thoughts on specification, design, and verification", ACM Soft. Eng. Notes, Vol. 5, No. 3, 1980, pp. 29-33.

[Gone 70] Gonenc, G., "A method for the design of fault detection experiments", IEEE Trans. on Comp., Vol. C-19, No. 6, 1970.

[Good 77] Good, D.I., "Constructing verified and reliable communications processing systems", ACM SIGSOFT, Software Engineering Notes, Vol. 2, No.5, 1977, pp.8-13.

[Hail 81a] Hailpern, B.T., "Specifying and verifying protocols represented as abstract programs", IBM Research Report RC 8674, (137908), Feb 1981.

[Hail 81b] Hailpern, B.T. and S.S. Owicki, "Modular verification of computer communication protocols", IBM Research Report RC 8726, (138174), March 1981.

[Hail 81c] Hailpern, B.T., "A simple protocol whose proof isn't", IBM Research Report RC 8800, (138567), April 1981.

[Haje 78] Hajek, J., "Automatically verified data transfer protocols", Proc. of 4th Int. Computer Communication Conf., Kyoto, Japan, Sep 1978, pp.749-756.

[Hara 77] Harangozo, J., "An approach to describing a link level protocol with a formal language", Proc. of 5th Data

Communication Symposium, Utah, 1977, pp.4.37-4.49.

[Hara 78] Harangozo, J., "Protocol definition with formal grammars", Proc. of Symposium on Computer Communication Protocols, Liege, Belgium, Feb 1978, pp.F.6.1-6.10.

[Henl 81] Henley, R.F.L., "Implementation assessment of transport and network services: The test responder specification", NPL Report DNACS 46/81, July 1981.

[HoUl 69] Hopcraft, J.E., and J.D. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley, 1969.

[Howd 80] Howden, W.E., "Functional program testing", IEEE Trans. on Soft. Eng., Vol.SE-6, No.2, 1980, pp.162-169.

[IFIP 82] Proc. of 2nd IFIP/WG 6.1 Int. Workshop on Protocol Specification, Verification, and Testing, (ed. C. Sunshine) North-Holland, 1982.

[IFIP 83] Proc. of 3rd IFIP/WG 6.1 Int. Workshop on Protocol Specification, Verification, and Testing (ed., H. Rudin and C West), North-Holland, 1983.

[ISO 81] "Formal Specification of a Transport Service", ISO/TC 97/SG16/WG1, Rapporteurs Group on FDT, 1981.

[ISO 82] "A FDT Based on an Extended State Transition Model", ISO/TC 97/SG16/WG1, Subgroup B on FDT, Nov 1982.

[ISO 83a] "Example of a Transport Protocol Specification", ISO/TC 97/SG16/WG1, Subgroup B on FDT, Feb 1983.

[ISO 83b] "A FDT Based on an Extended State Transition Model", ISO/TC 97/SG16/WG1, Subgroup B on FDT, May 1983.

[ISO 83c] "OSI Connection Oriented Transport Protocol Specification", ISO/TC 97/SG16 N 1576, (ISO/DP 8073), 1983.

[JaBo 83] Jard, C. and G.V. Bochmann, "An approach to testing specifications", Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Pacific Grove, Cal., March 1983, pp. 53-59.

[Kell 76] Keller, R.M., "Formal verification of parellel programs", CACM, Vol.19, No.7, 1976, pp.371-384.

[Kowa 79] Kowalski, R., Logic For Problem Solving, New York, North-Holland, 1979.

[Lai 81] Lai, W.S., "Protocol traps in computer networks: a catalog", Bell-Northern Research Ltd., 1981.

[Lamp 80] Lamport, L., "'Sometime' is sometimes 'not never': on

the temporal logic programs", Proc. of 7th Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Jan 1980, pp:174-185.

[LeMo 73] LeMoli, G., "A theory of colloquies", Proc. of 1st European Workshop on Computer Networks, Arles, France, April 1973, pp.153-173. (Also in Alta Frequenza, Vol. 42, 1973, pp. 493-223E, -500-230E).

[Lewi 76] Lewis, P.M., et al, Compiler Design Theory, Addison-Wesley, 1976.

[LiMc 83] Linn, R.J. and W.H. McCoy, "Producing tests for implementations of OSI protocols", Proc. of 3rd IFIP/WG 6.1 Int. Workshop on Protocol Specification, Verification, and Testing, Zurich, May 1983, pp.505-520.

[Logr 83] Logrippo, L., "Specification of transport service using finite-state transducers and abstract data types", Proc. of 3rd Int. Workshop on Protocol Specification, Testing, and Verification, Zurich, May 1983, pp:111-124.

[Logr 84] Logrippo, L., D. Simon and H. Ural, "Executable description of the OSI transport service in PROLOG", Proc. of 4th IFIP/WG 6.1 Int. Workshop on Protocol Specification, Verification, and Testing, Mt. Pocono Penn, June 1984.

[MeFa 76] Merlin, P.M. and D.J. Farber, "Recoverability of communication protocols: implementations of a theoretical study", IEEE Trans. on Communications, Vol.24, No.9, 1976, pp.1036-1043.

[Merl 74] Merlin, P.M., "A study of the recoverability of computing systems", Ph.D. Thesis, University of California, Dept. of Information and Computer Sciences, Dec 1974.

[Merl 76] Merlin, P.M., "A methodology for the design and implementation of communication protocols", IEEE Trans. on Communications, Vol.24, No.6, 1976, pp.614-621.

[Merl 79] Merlin, P.M., "Specification and validation of protocols", IEEE Trans. on Communications, Vol.27, No.11, 1979, pp.1671-1680.

[Mier 79] van-Mierop, D., "Design and verification of distributed interacting processes", Ph.D. Thesis, University of California, Los Angelos, Computer Science Dept., March 1979.

[MiFi 79] Milton, D.R. and C.N. Fisher, "LL(k) parsing for attributed grammars", Proc. of 6th Int. Colloqium on Automata, Languages, and Programming, Graz, July 1979, pp. 422-430.

[MiHo 81] Miller, E. and W.E. Howden, Tutorial: Software Testing

and Validation Techniques, IEEE Computer Society, 1981.

[Myer 79] Myers, G.J., The Art Of Software Testing, Wiley, 1979.

[NaTs 81] Naito, S., and M. Tsunoyama, "Fault detection for sequential machines by transition tours", Proc. of IEEE Conference on Fault Tolerant Computing, 1981.

[Nigh 82] Nightingale, J.S., "Protocol testing using reference implementation", in Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp.513-522.

[Over 81] Overman, W.T., "Verification of concurrent systems: function and timing", Ph.D. Thesis, University of California Los Angelos, 1981.

[Panz 81] Panzl, D., "A method for evaluating software development techniques", ACM SIGSOFT STME, 1981.

[Pere 78] Pereira, L.M. et al, User's Guide to DEC system-10 PROLOG, Dept. of AI, Univ. of Edinburg, Sept 1978.

[Petr 62] Petri, K.A., "Communications with automata", Ph.D. Thesis, Darmstast Institute of Technology, Bonn, Germany, 1962.

[Piat 80] Piatkowski, T.F., "Remarks on the feasability of validating and testing ADCCP implementations", Proc. of Trends and Applications Symposium ,(NBS), 1980,Gaithersburg, MD.

[Pnue 77] Pnueli, A., "The temporal logic of programs", Proc. of 18th Symposium on Foundations of Computer Science, Providence, Rhode Island, Oct 1977, pp.46-57.

[Post 74] Postel, J.B., "A graph model analysis of computer communication protocols", Ph.D. Thesis, University of California Los Angelos, Computer Science Dept., Jan 1974.

[Prob 82] Probert, R.L., "Life-Cycle/Grey-Box Testing", Congresus Num, Vol.34, 1982, pp.87-117.

[Prob 83] Probert, R.L., D.R. Skuce, and H. Ural, "Specification of representative test cases", Proc. of 16th Hawaii Int. Conf. on System Sciences, Jan 1983, pp.190-196.

[PrUr 82a] Probert, R.L. and H. Ural, "Incremental improvement of specifications by testing", Proc. of Workshop on Effectiveness of Testing and Proving Methods, Avolon, California, May 1982, pp.37-49.

[PrUr 82b] Probert, R.L. and H. Ural, "High-level evaluation and improvement of software functionality representations", Proc. of 5th Minnowbrook Workshop on Software Performance

Evaluation, July 1982.

[PrUr 83] Probert, R.L. and H. Ural, "Requirements for a test specification language for protocol implementation testing", Proc. of 3rd Int. Workshop on Protocol Specification, Testing, and Verification, Zurich, May 1983, pp:437-443.

[Rayn 81] Rayner, D., "Protocol implementation assessment: philosopy and architecture", Proc. of National Telecommunication Conf., New Orleans, Dec 1981, pp.F8.4.1-5.

[Rayn 82] Rayner, D., "A system for testing protocol implementations", NPL Report DITC 9/82, August 1982, (also in Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp.539-553).

[RaEs 80] Razouk, R. and G. Estrin, "Validation of X.21 interface specification using SARA", Proc. of Trends and Application Symposium, NBS, May 1980, pp.155-167.

[Rudi 78] Rudin, H. et al, "Automated protocol validation: one chain of development", Computer Networks, Vol.2, No.4/5, 1978, pp.373-380.

[SaBo 82] Sarikaya, B. and G.V. Bochmann, "Some experience with test sequence generation for protocols", in Protocol Specification, Testing, and Verification, (ed. C. Sunshine), North-Holland, 1982, pp.555-567.

[SaBo 83] Sarikaya B. and G.V. Bochmann, "Synchronization issues in protocol testing", Proc. of ACM SIGCOMM '83 Symp. on Communications Architectures and Protocols, March 1983.

[Schi 80] Schindler, S., "Algebraic and model specification techniques", Proc. of 13th Hawaii Int. Conf. on System Sciences, Jan 1980.

[Schu 80] Schultz, G.D. et al, "Executable description and validation of SNA", IEEE Trans. on Communications, Vol.28, No.4, 1980, pp.661-677.

[Schw 81] Schwabe, D., "Formal techniques for the specification and verification of protocols", Ph.D Thesis, University of California, Los Angelos, April 1981.

[ScMe 81] Schwartz, R.L. and P.M. Melliar-Smith, "Temporal logic specification of distributed systems", Proc. of 2nd Int. Conf. on Distributed Systems, INRIA, Paris, France, April 1981, pp.446-454.

[Sidh 83] Sidhu, D.P., "Protocol verification via executable logic specifications", Proc. of 3rd IFIP/WG 6.1 Int. Workshop on Protocol Specification, Verification, and Testing, Zurich, May 1983, pp:237-248.

[Sten 76] Stenning, N.V., "A data transfer protocol", Computer Networks, Vol.1, No.2, 1976, pp.99-110.

[Suns 75] Sunshine, C.A., "Interprocess communication protocols for computer networks", Digital System Lab, Stanford University, Technical Report 1105, Dec 1975, (Ph.D. Thesis).

[Suns 76] Sunshine, C.A., "Survey of communication protocol verification techniques", Proc. of Trends and Application Symposium, NBS, Maryland, Nov 1976, pp.24-26.

[Suns 78a] Sunshine, C.A., "Survey of protocol definition and verification techniques", Computer Networks, Vol.2, No.4/5, 1978, pp.346-350.

[Suns 78b] Sunshine, C.A. and Y.K. Dalal, "Connection management in transport protocols", Computer Networks, Vol.2, No.4/5, 1978, pp.454-473.

[Suns 79] Sunshine, C.A., "Formal techniques for protocol specification and verification", Computer Magazine, Vol.12, No.9, 1979, pp.20-27.

[Suns 81] Sunshine, C.A., "Formal Modelling of communication protocols", University of Southern California, Information Sciences Institute, Technical Report RR-81-89, March 1981.

[Symo 80a] Symons, F.J.W., "Representation, analysis, and verification of communication protocols", Telecom Australia, Technical Report 7380, 1980.

[Symo 80b] Symons, F.J.W., "Introduction to numerical petri nets, a general graphical model of concurrent processing systems", Australian Telecommunication Research, Vol. 14, No. 1, 1980, pp.28-32.

[Symo 80c] Symons, F.J.W., "The verification of communication protocols using numerical petri nets", Australian Telecommunication Research, Vol. 14, No. 1, 1980, pp.34-38.

[TeLi 78] Teng, A.Y. and M.T. Liu, "A formal model for automatic implementation and logical validation of networks communication protocols", Proc. of Computer Networking Symposium, NBS, Dec 1978, pp.114-123.

[Teng 80] Teng, A.Y., "Protocol construction for communication networks", Ph.D Thesis, The Ohio State University, 1980.

[Thar 79] Thareja, A.K. et al, "Formal approaches to protocol design and implementation: a survey", Report.TR840, Dept. of Computer Science, University of Maryland, Dec 1979.

[Thom 81] Thompson, D.H. et al, "Specification and verification of communication protocols in AFFIRM using state transition

models", University of Southern California, Information Sciences Institute Report RR-81-88, 1981.

[UrPr 83a] Ural, H. and R.L. Probert, "Architectures for testing protocol implementations", Proc. of CIPS Conference 1983, Ottawa, May 1983, pp:130-136.

[UrPr 83b] Ural, H. and R.L. Probert, "Specification-Based test scenario generation", Proc. of 3rd Int. Workshop on Protocol Specification, Testing, and Verification, Zurich, May 1983, pp:421-436.

[UrPr 84a] Ural, H. and R.L. Probert, "Automated testing of protocol specifications and their implementations", Proc. of ACM SIGCOMM 84, Montreal, June 1984.

[UrPr 84b] Ural, H. and R.L. Probert, "Systematic and selective generation of test sequences", TR-84-16, Dept. of CSI, University of Ottawa, 1984.

[VuCo 82] Vuong, S.T. and D.D. Cowan, "A decomposition method for the validation of structured protocols", Proc. of INFOCOM 82, Las Vegas, March 1982, pp.209-220.

[West 78a] West, C.H. and P. Zafiropulo, "Automated validation of a communications protocol: the CCITT X.21 recommendations", IBM J. of Reseach and Dev. Vol.22, No.1, 1978, pp.60-71.

[West 78b] West, C.H., "General technique for communications protocol validation", IBM J. of Research and Dev., Vol.22, No.4, 1978, pp.393-404.

[West 78c] West, C.H., "An automated technique of communications protocol validation", IEEE Trans. on Communications, Vol.26, No.8, 1978, pp.1271-1275.

[Weyu 81] Weyuker, E.J., "An error-based testing strategy", Report No.027, Dept. of Computer Science, New York University, 1981.

[Zafi 78] Zafiropulo, P., "Protocol validation by duologue matrix analysis", IEEE Trans. on Communications, Vol.26, No.8, 1978, pp.1187-1194.

[Zafi 80] Zafiropulo, P. et al, "Towards analyzing and synthesizing protocols", IEEE Trans. on Communications, Vol.28, No.4, 1980, pp.651-660.

[Zimm 80] Zimmermann, H., "OSI reference model-the ISO model of architecture for open system interconnection", IEEE Trans. on Communications, Vol.28, No.4, 1980, pp.425-432.

[Zimm 81] Zimmermann, H., "Progression of the OSI reference model and its applications", Proc. of National Telecommunication Conference, New Orleans, Dec 1981, pp.F8.1.1-6.

APPENDIX A : TRANSPORT SERVICE SPECIFICATION IN EFSM FORMALISM

interactions

TS_access_point(TS_user,TS_provider) is

by TS user:

```
   T_CONNECT_req(TCEPI:                 : TCEP_identifier_type;
                 to_T_address           : T_address_type;
                 from_T_address         : T_address_type;
                 QOTS_request           : quality_of_TS_type;
                 options                : option_type;
                 TS_connect_data        : TS_connect_data_type);

    T_ACCEPT_req(TCEPI                  : TCEP_identifier_type;
                 QOTS_request           : quality_of_TS_type;
                 options                : option_tpe);
                 TS_accept              : TS_accept_data_type;

T_DISCONNECT_req(TCEPI                  : TCEP_identifier_type;
                 TS_user_reason         : TS_user_reason_type);

      T_DATA_req(TCEPI                  : TCEP_identifier_type;
                 TSDU_fragment          : data_fragment_type);
```

by TS provider:

```
   T_CONNECT_ind(TCEPI                  : TCEP_identifier_type;
                 to_T_address           : T_address_type;
                 from_T_address         : T_address_type;
                 QOTS_request           : quality_of_TS_type;
                 options                : option_type;
                 TS_connect_data        : TS_connect_data_type);

    T_ACCEPT_ind(TCEPI                  : TCEP_identifier_type;
                 QOTS_request           : quality_of_TS_type;
                 options                : option_type);
                 TS_accept              : TS_accept_data_type;

T_DISCONNECT_ind(TCEPI                  : TCEP_identifier_type;
                 TS_disconnect_reason   : TS_disconnect_reason_type;
                 TS_user_reason         : TS_user_reason_type);

      T_DATA_ind(TCEPI                  : TCEP_identifier_type;
                 TSDU_fragment          : data_fragment_type;

end TS_access_point;
```

Transitions

```
from (state1=idle) and (state2=idle)
     when AP1.T_CONNECT_req(TCEPI,to_T_address, from_T_address,
                            QOTS_request, requested_options,
                            TS_connect_data )
       provided (no_congestion)
          to (state1=wait_for_acc) and (state2=idle)
          begin
            TCEP1:=TCEPI;
            caller:=from_T_address;
            called:=to_T_address;
            options:=requested_options;
            connect_data:=TS_connect_data;
            buffer21.clear;
            buffer12.clear;
          end;
       provided (congestion)
          to (state1=idle) and (state2=idle)
          begin
            out AP1.T_DISCONNECT_ind(TCEPI,empty);
          end;

from (state1=wait_for_acc) and (state2=idle)
          provided (connection_request_reaches_the_called_user)
          to (state1=wait_for_acc) and (state2=wait_for_acc);
          begin
            TCEP2:=...; (* some unique identifier *)
            TCEP2_QOTS_estimate:=...; (* QOTS_request *)
            out AP2.T_CONNECT_ind(TCEP2,called,caller,
                                  TCEP2_QOTS_estimate,
                                  connect_data,options);
          end;
       provided (internal_problem)
          to (state1=idle) and (state2=idle)
          begin
            out AP1.T_DISCONNECT_ind(TCEP1,empty);
          end;

from (state1=wait_for_acc) and (state2=wait_for_acc)
     when AP2.T_ACCEPT_req(TCEPI,QOTS_request,
                           requested_option,TS_accept_data)
       provided (TCEPI=TCEP2)
          to (state1=wait_for_acc) and (state2=data_transfer)
          begin
            options:=requested_options;
            accept_data:=TS_accept_data;
          end;
     when AP2.T_DISCONNECT_req(TCEPI,TS_user_reason)
       provided (TCEPI=TCEP2)
          to (state1=wait_for_acc) and (state2=disconnect)
          begin
            TS_reason:=TS_user_initiated_termination;
            user_reason:=TS_user_reason
          end;
       provided (internal_problem)
          to (state1=wait_for_acc) and (state2=disconnect)
          begin
```

```
            TS_reason:=...;
            user_reason:=empty;
            out AP2.T_DISCONNECT_ind(TCEP2,TS_reason,user_reason);
          end;
        provided (internal_problem)
          to (state1=disconnect) and (state2=wait_for_acc)
          begin
            TS_reason:=...;
            user_reason:=empty;
            out AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason);
          end;

from (state1=wait_for_acc) and (state2=data_transfer)
      provided (accept_indication_reaches_the_caller)
          to (state1=data_transfer) and (state2=data_transfer)
          begin
            TCEP2_QOTS_estimate:=...;
            out AP1.T_ACCEPT_ind(TCEP1,TCEP1_QOTS_estimate,
                                 options,accept_data);
          end;
    when AP2.T_DATA_req(TCEPI,TSDU_fragment)
      provided (flow_control_to_Transport_Entity_is_ready)
                and (TCEPI=TCEP2)
          to (state1=wait_for_acc) and (state2=data_transfer)
          begin
            buffer21.append(TSDU_fragment);
          end;
      provided (internal_problem)
          to (state1=disconnect) and (state2=data_transfer)
          begin
            TS_reason:=...;
            user_reason:=empty;
            out AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason);
          end;
      provided (internal_problem)
          to (state1=wait_for_acc) and (state2=disconnect)
          begin
            TS_reason:=...;
            user_reason:=empty;
            out AP2.T_DISCONNECT_ind(TCEP1,TS_reason_reason);
          end;

from (state1=data_transfer) and (state2=data_transfer)
    when AP1.T_DATA_req(TCEPI,TSDU_fragment)
      provided (flow_control_to_Transport_Entity_is_ready)
                and (TCEPI=TCEP2)
          to (state1=data_transfer) and (state2=data_transfer)
          begin
            buffer12.append(TSDU_fragment);
          end;
      provided buffer12.get_next(data_fragment)=/ empty
                and (flow_control_to_user_is_ready)
          to (state1=data_transfer) and (state2=data_transfer)
          begin
            AP2.T_DATA_ind(TCEP2,data_fragment);
          end;
```

```
      when AP2.T_DATA_req(TCEP1,TSDU_fragment)
        provided (flow_control_to_Transport_Entity_is_ready)
                 and (TCEPI=TCEP1)
           to (state1=data_transfer) and (state2=data_transfer)
           begin
             buffer21.append(TSDU_fragment);
           end;
        provided buffer21.get_next(data_fragment) =/ empty
                 and (flow_control_to_user_is_ready)
           to (state1=data_transfer) and (state2=data_transfer)
           begin
             out AP1.T_DATA_ind(TCEPI,data_fragment);
           end;
      when AP1.T_DISCONNECT_req(TCEPI,TS_user_reason)
        provided (TCEPI=TCEP1)
           to (state1=disconnect) and (state2=data_transfer)
           begin
             TS_reason:=TS_user_initiated_termination;
             user_reason:=TS_user_reason;
           end;
      when AP2.T_DISCONNECT_req(TCEPI,TS_user_reason)
        provided (TCEPI=TCEP2)
           to (state1=data_transfer) and (state2=disconnect)
           begin
             TS_reason:=TS_user_initiated_termination;
             user_reason:=TS_user_reason;
           end;
        provided (internal_problem)
           to (state1=data_transfer) and (state2=disconnect)
           begin
             TS_reason:=...;
             user_reason:=empty;
             out AP2.T_DISCONNECT_ind(TCEP2,TS_reason,user_reason);
           end;
        provided (internal_problem)
           to (state1=disconnect) and (state2=data_transfer)
           begin
             TS_reason:=...,
             user_reason:=empty;
             out AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason);
           end;

    from (state1=disconnect) and ((state2=data_transfer) or
         (state2=wait_for_acc))
        provided (disconnect_reaches_the_called_user)
           to (state1=idle) and (state2=idle;)
           begin
             out AP2.T_DISCONNECT_ind(TCEP2,TS_reason,user_reason);

           end;

    from ((state1=data_transfer) or (state1=wait_for_acc))
         and (state2=disconnect)
        provided (disconnect_reaches_the_calling_user)
           to (state1=idle) and (state2=idle)
           begin
             out AP1.T_DISCONNECT_ind(TCEP1,TS_reason,user_reason);
           end;

    end; (* transitions*)
```

APPENDIX B : TRANSPORT SERVICE SPECIFICATION IN AG FORMALISM

```
tservice(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21) ::= ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
                                                          NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21)
                                                        continue(NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21).

continue(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21) ::= PS1 EQ 'idle' AND  PS2 EQ 'idle' empty
                                                      |
                                                        ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
                                                          NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21)
                                                        continue(NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21).

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,NIIQ1,NOIQ1,PIIQ2,POIQ2,NS1,NS2,PB12,PB21) ::=

    PS1 EQ 'idle' AND  PS2 EQ 'idle' AND is_firstof(PIIQ1,'apl,tcreq,P') AND  congestion
    restof(PIIQ1,NIIQ1) I='apl,tcreq,P'  input(I)
    NS1='idle'  NS2='idle'  O='apl,tdind,provider' lastof(POIQ1,NOIQ1,O) output(O) .

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,NIIQ1,POIQ1,PIIQ2,POIQ2,NS1,NS2,NB12,NB21) ::=

    PS1 EQ 'idle' AND PS2 EQ 'idle' AND is_firstof(PIIQ1,'apl,tcreq,P') AND no_congestion
    restof(PIIQ1,NIIQ1) I='apl,tcreq,P' input(I)
    NS1='wait_accept'  NS2='idle'  actions_1(NB12,NB21)
|
    PS1 EQ 'data_transfer' AND PS2 EQ 'data_transfer' AND is_firstof(PIIQ1,'apl,tdatr,DF') AND
    flow_control_to_tpentity_ready restof(PIIQ1.NIIQ1) I='apl,tdatr,DF'  input(I)
    NS1='data_transfer'  NS2='data_transfer'  actions_2(PB12,PB21,NB12,NB21,DF)
|
    PS1 EQ 'data_transfer' AND PS2 EQ 'data_transfer' AND is_firstof(PIIQ1,'apl.tdreq.user')
    restof(PIIQ1,NIIQ1) I='apl,tdreq,user'  input(I)
    NS1='disconnect'  NS2='data_transfer'  actions_3(PB12,PB21.NB12,NB21) .

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,PIIQ1,NOIQ1,PIIQ2,POIQ2,NS1,NS2,NB12,NB21) ::=

    PS1 EQ 'wait_accept'  AND PS2 EQ 'idle'  AND internal_problem
    NS1='idle'  NS2='idle'  actions_4(PB12,PB21,NB12,NB21)  O='apl,tdind,provider' lastof(POIQ1,NOIQ1,O) output(O)
|
    PS1 EQ 'wait_accept'  AND PS2 EQ 'wait_accept' AND internal_problem
    NS1='disconnect'  NS2='wait_accept'  actions_5(PB12,PB21,NB12,NB21)
    O='apl,tdind,provider'  lastof(POIQ1,NOIQ1,O)  output(O)
|
    PS1 EQ 'wait_accept' AND  PS2 EQ 'data_transfer' AND  no_internal_problem
    NS1='data_transfer'  NS2='data_transfer'  actions_6(PB12,PB21,NB12,NB21)
    O='apl,taind,P'  lastof(POIQ1,NOIQ1,O)  output(O)
|
    PS1 EQ 'wait_accept'  AND PS2 EQ 'data_transfer'  AND internal_problem
    NS1='disconnect'  NS2='data_transfer'  actions_7(PB12,PB21,NB12,NB21)
    O='al,tdind,provider'  lastof(POIQ1,NOIQ1,O)  output(O)
|
    PS1 EQ 'data_transfer'  AND PS2 EQ 'data_transfer'  AND not(PB21 EQ empty) AND flow_control_to_user_ready
    NS1='data_transfer'  NS2='data_transfer'  actions_8(PB12,NB12)
    get_next_buffer(PB21,DF,NB21)  O='apl,tdati,DF'  lastof(POIQ1.NOIQ1.O)  output(O)
|
    PS1 EQ 'data_transfer' AND PS2 EQ 'data_transfer'  AND internal_problem
    NS1='disconnect'  NS2='data_transfer'  actions_9(PB12,PB21,NB12,NB21)
    O='apl,tdind,provider'  lastof(POIQ1,NOIQ1,O)  output(O)
|
  (PS1 EQ 'data_transfer' | PS1 EQ 'wait_accept'), AND PS2 EQ 'disconnect'
    NS1='idle'  NS2='idle'  actions_10(PB12,PB21,NB12,NB21)
    O='apl,tdind,P'  lastof(POIQ1,NOIQ1,O)  output(O) .
```

```
ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,PIIQ1,POIQ1,PIIQ2,NOIQ2,NS1,NS2,NB12,NB21) ::=

    PS1 EQ 'wait_accept'  AND PS2 EQ 'idle'  AND no_internal_problem
    NS1='wait_accept'  NS2='wait_accept'  actions_11(PB12,PB21,NB12,NB21)
    O='ap2,tcind,P'  lastof(POIQ2,NOIQ2,O)  output(O)
|
    PS1 EQ 'wait_accept'  AND PS2 EQ 'wait_accept'  AND internal_problem
    NS1='wait_accept'  NS2='disconnect'  actions_12(PB12,PB21,NB12,NB21)
    O='ap2,tdind,provider'  lastof(POIQ2,NOIQ2,O)  output(O)
|
    PS1 EQ 'wait_accept'  AND PS2 EQ 'data_transfer'  AND internal_problem
    NS1='wait_accept'  NS2='disconnect'  actions_13(PB12,PB21,NB12,NB21)
    O='ap2,tdind,provider'  lastof(POIQ2,NOIQ2,O)  output(O)
|
    PS1 EQ 'data_transfer'  AND PS2 EQ 'data_transfer'  AND not(PB12 EQ empty) AND flow_control_to_user_ready
    NS1='data_transfer'  NS2='data_transfer'  actions_14(PB21,NB21)
    get_next_buffer(PB12,DF,NB12)  O='ap2,tdati,DF'  lastof(POIQ2,NOIQ2,O)  output(O)
|
    PS1 EQ 'data_transfer'  AND PS2 EQ 'data_transfer'  AND internal_problem
    NS1='data_transfer'  NS2='disconnect'  actions_15(PB12,PB21,NB12,NB21)
    O='ap2,tdind,provider'  lastof(POIQ2,NOIQ2,O)  output(O)
|
    PS1 EQ 'disconnect'  AND (PS2 EQ 'data_transfer' | PS2 EQ 'wait_accept')
    NS1='idle'  NS2='idle'  actions_16(PB12,PB21,NB12,NB21)
    O='ap2,tdind,R'  lastof(POIQ2,NOIQ2,O)  output(O) .

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,PIIQ1,POIQ1,NIIQ2,POIQ2,NS1,NS2,NB12,NB21) ::=

    PS1 EQ 'wait_accept' AND PS2 EQ 'wait_accept' AND is_firstof(PIIQ2,'ap2,tareq,P')
    restof(PIIQ2,NIIQ2)  I='ap2,tareq,P' input(I)
    NS1='wait_accept' NS2='data_transfer'  actions_17(PB12,PB21,NB12,NB21)
|
    PS1 EQ 'wait_accept' AND PS2 EQ 'wait_accept'  AND is_firstof(PIIQ2,'ap2,tdreq,user')
    restof(PIIQ2,NIIQ2)  I='ap2,tdreq,user' input(I)
    NS1='wait_accept'  NS2='disconnect'  actions_18(PB12,PB21,NB12,NB21)
|
    PS1 EQ 'wait_accept'  AND PS2 EQ 'data_transfer' AND is_firstof(PIIQ2.'ap2,tdatr,DF')  AND
    flow_control_to_tpentity_ready restof(PIIQ2,NIIQ2)  I='ap2,tdatr,DF' input(I)
    NS1='wait_accept'  NS2='data_transfer'  actions_19(PB12,PB21,NB12,NB21,DF)
|
    PS1 EQ 'data_transfer' AND PS2 EQ 'data_transfer' AND is_firstof(PIIQ2,'ap2,tdatr,DF') AND
    flow_control_to_tpentity_ready restof(PIIQ2,NIIQ2)  I='ap2,tdatr,DF' input(I)
    NS1='data_transfer'  NS2='data_transfer'  actions_20(PB12,PB21,NB12,NB21,DF)
|
    PS1 EQ 'data_transfer' AND PS2 EQ 'data_transfer' AND is_firstof(PIIQ2,'ap2,tdreq,user')
    restof(PIIQ2,NIIQ2)  I='ap2,tdreq,user' input(I)
    NS1='data_transfer'  NS2='disconnect'  actions_21(PB12,PB21,NB12,NB21) .

input(I)  ::= print('i,')  print(I).
output(O) ::= print('o,')  print(O).

restof(Q1,Q2) ::= Q2=tail_of(Q1).
is_firstof(X,Y) ::= Y EQ head_of(X).
lastof(Q1,Q2,O)  ::= NO=list(O)  Q2=append(Q1,NO).
```

APPENDIX C : PROLOG ENCODING OF TRANSPORT SERVICE SPECIFICATION

```
tservice(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21) :- ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
                                                          NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21),
                                                       continue(NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21).

continue(PIIQ1,POIQ1,PIIQ2,POIQ2,idle,idle,PB12,PB21).
continue(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21) :- ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,
                                                          NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21),
                                                       continue(NIIQ1,NOIQ1,NIIQ2,NOIQ2,NS1,NS2,NB12,NB21).

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,NIIQ1,NOIQ1,PIIQ2,POIQ2,NS1,NS2,PB12,PB21) :-

   (PS1=idle, PS2=idle, is_firstof(PIIQ1,[apl,tcreq,P]), congestion,
    restof(PIIQ1,NIIQ1), I=[apl,tcreq,P], input(I),
    NS1=idle, NS2=idle, O=[apl,tdind,provider], lastof(POIQ1,NOIQ1,O), output(O)).

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,NIIQ1,POIQ1,PIIQ2,POIQ2,NS1,NS2,NB12,NB21) :-

   (PS1=idle, PS2=idle, is_firstof(PIIQ1,[apl,tcreq,P]), no_congestion,
    restof(PIIQ1,NIIQ1), I=[apl,tcreq,P], input(I),
    NS1=wait_accept, NS2=idle,  actions_1(PB12,PB21,NB12,NB21))
;
   (PS1=data_transfer, PS2=data_transfer, is_firstof(PIIQ1,[apl,tdatr,DF]), flow_control_to_tpentity_ready,
    restof(PIIQ1,NIIQ1), I=[apl,tdatr,DF], input(I),
    NS1=data_transfer, NS2=data_transfer, actions_2(PB12,PB21,NB12,NB21,DF))
;
   (PS1=data_transfer, PS2=data_transfer, is_firstof(PIIQ1,[apl,tdreq,user]),
    restof(PIIQ1,NIIQ1), I=[apl,tdreq,user], input(I),
    NS1=disconnect, NS2=data_transfer, actions_3(PB12,PB21,NB12,NB21)).

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,PIIQ1,NOIQ1,PIIQ2,POIQ2,NS1,NS2,NB12,NB21) :-

   (PS1=wait_accept, PS2=idle, internal_problem,
    NS1=idle, NS2=idle, actions_4(PB12,PB21,NB12,NB21), O=[apl,tdind,provider],
    lastof(POIQ1,NOIQ1,O), output(O))
;
   (PS1=wait_accept, PS2=wait_accept, internal_problem,
    NS1=disconnect, NS2=wait_accept, actions_5(PB12,PB21,NB12,NB21),
    O=[apl,tdind,provider], lastof(POIQ1,NOIQ1,O), output(O))
;
   (PS1=wait_accept, PS2=data_transfer, no_internal_problem,
    NS1=data_transfer, NS2=data_transfer, actions_6(PB12,PB21,NB12,NB21),
    O=[apl,taind,P], lastof(POIQ1,NOIQ1,O), output(O))
;
   (PS1=wait_accept, PS2=data_transfer, internal_problem,
    NS1=disconnect, NS2=data_transfer, actions_7(PB12,PB21,NB12,NB21),
    O=[al,tdind,provider], lastof(POIQ1,NOIQ1,O), output(O))
;
   (PS1=data_transfer, PS2=data_transfer, not(PB21=[]), flow_control_to_user_ready,
    NS1=data_transfer, NS2=data_transfer, actions_8(PB12,NB12),
    get_next_buffer(PB21,DF,NB21), O=[apl,tdati,DF], lastof(POIQ1,NOIQ1,O), output(O))
;
   (PS1=data_transfer, PS2=data_transfer, internal_problem,
    NS1=disconnect, NS2=data_transfer, actions_9(PB12,PB21,NB12,NB21),
    O=[apl,tdind,provider], lastof(POIQ1,NOIQ1,O), output(O))
;
  ((PS1=data_transfer ;PS1=wait_accept), PS2=disconnect,
    NS1=idle, NS2=idle, actions_10(PB12,PB21,NB12,NB21),
    O=[apl,tdind,R], lastof(POIQ1,NOIQ1,O), output(O)).
```

```
ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,PIIQ1,POIQ1,PIIQ2,NOIQ2,NS1,NS2,NB12,NB21) :-

    (PS1=wait_accept, PS2=idle, no_internal_problem,
     NS1=wait_accept, NS2=wait_accept, actions_11(PB12,PB21,NB12,NB21),
     O=[ap2,tcind,P], lastof(POIQ2,NOIQ2,O), output(O))
;
    (PS1=wait_accept, PS2=wait_accept, internal_problem,
     NS1=wait_accept, NS2=disconnect, actions_12(PB12,PB21,NB12,NB21),
     O=[ap2,tdind,provider], lastof(POIQ2,NOIQ2,O), output(O))
;
    (PS1=wait_accept, PS2=data_transfer, internal_problem,
     NS1=wait_accept, NS2=disconnect, actions_13(PB12,PB21,NB12,NB21),
     O=[ap2,tdind,provider], lastof(POIQ2,NOIQ2,O), output(O))
;
    (PS1=data_transfer, PS2=data_transfer, not(PB12=[]), flow_control_to_user_ready,
     NS1=data_transfer, NS2=data_transfer, actions_14(PB21,NB21),
     get_next_buffer(PB12,DF,NB12), O=[ap2,tdati,DF], lastof(POIQ2,NOIQ2,O), output(O))
;
    (PS1=data_transfer, PS2=data_transfer, internal_problem,
     NS1=data_transfer, NS2=disconnect, actions_15(PB12,PB21,NB12,NB21),
     O=[ap2,tdind,provider], lastof(POIQ2,NOIQ2,O), output(O))
;
    (PS1=disconnect, (PS2=data_transfer ;PS2=wait_accept),
     NS1=idle, NS2=idle, actions_16(PB12,PB21,NB12,NB21),
     O=[ap2,tdind,R], lastof(POIQ2,NOIQ2,O), output(O)).

ts(PIIQ1,POIQ1,PIIQ2,POIQ2,PS1,PS2,PB12,PB21,PIIQ1,POIQ1,NIIQ2,POIQ2,NS1,NS2,NB12,NB21) :-

    (PS1=wait_accept, PS2=wait_accept, is_firstof(PIIQ2,[ap2,tareq,P]),
     restof(PIIQ2,NIIQ2), I=[ap2,tareq,P], input(I),
     NS1=wait_accept, NS2=data_transfer, actions_17(PB12,PB21,NB12,NB21))
;
    (PS1=wait_accept, PS2=wait_accept, is_firstof(PIIQ2,[ap2,tdreq,user]),
     restof(PIIQ2,NIIQ2), I=[ap2,tdreq,user], input(I),
     NS1=wait_accept, NS2=disconnect, actions_18(PB12,PB21,NB12,NB21))
;
    (PS1=wait_accept, PS2=data_transfer, is_firstof(PIIQ2,[ap2,tdatr,DF]), flow_control_to_tpentity_ready,
     restof(PIIQ2,NIIQ2), I=[ap2,tdatr,DF], input(I),
     NS1=wait_accept, NS2=data_transfer, actions_19(PB12,PB21,NB12,NB21,DF))
;
    (PS1=data_transfer, PS2=data_transfer, is_firstof(PIIQ2,[ap2,tdatr,DF]), flow_control_to_tpentity_ready,
     restof(PIIQ2,NIIQ2), I=[ap2,tdatr,DF], input(I),
     NS1=data_transfer, NS2=data_transfer, actions_20(PB12,PB21,NB12,NB21,DF))
;
    (PS1=data_transfer, PS2=data_transfer, is_firstof(PIIQ2,[ap2,tdreq,user]),
     restof(PIIQ2,NIIQ2), I=[ap2,tdreq,user], input(I),
     NS1=data_transfer, NS2=disconnect, actions_21(PB12,PB21,NB12,NB21)).

input(I) :- print('i,'), print(I).
output(O) :- print('o,'), print(O).

restof(_,_).
is_firstof([X|Y],X).
lastof(_,_,_).
```

```
congestion.
no_congestion.
internal_problem.
no_internal_problem.
flow_control_to_tpentity_ready.
flow_control_to_user_ready.

actions_1(X,Y,X,Y).
actions_2(B1,X,NB1,X,DF) :- buffer12_append(B1,[DF],NB1).
actions_3(X,Y,X,Y).
actions_4(X,Y,X,Y).
actions_5(X,Y,X,Y).
actions_6(X,Y,X,Y).
actions_7(X,Y,X,Y).
actions_8(X,X).
actions_9(X,Y,X,Y).
actions_10(X,Y,X,Y).
actions_11(X,Y,X,Y).
actions_12(X,Y,X,Y).
actions_13(X,Y,X,Y).
actions_14(X,X).
actions_15(X,Y,X,Y).
actions_16(X,Y,X,Y).
actions_17(X,Y,X,Y).
actions_18(X,Y,X,Y).
actions_19(X,B2,X,NB2,DF) :- buffer21_append(B2,[DF],NB2).
actions_20(X,B2,X,NB2,DF) :- buffer21_append(B2,[DF],NB2).
actions_21(X,Y,X,Y).

buffer12_append(B1,DF,NB1) :- append(B1,DF,NB1).
buffer21_append(B2,DF,NB2) :- append(B2,DF,NB2).

get_next_buffer12([DF|NB1],DF,NB1).
get_next_buffer21([DF|NB2],DF,NB2).

append([],X,X).
append([H|T1],L,[H|T2]) :- append(T1,L,T2).
```

APPENDIX D : ABSTRACT TRANSPORT SERVICE PROVIDER : PROCEDURE "TS"

```
ts([[I,O],[v,v]],[PS1,PS2,PB12,PB21],[PS1,PS2,PB12,PB21]) :-
     PS1=idle, PS2=idle, I=[ap1,tcreq,p], congestion,
     no_action, O=[ap1,tdind,provider].

ts([[I,v],[v,v]],[PS1,PS2,PB12,PB21],[NS1,NS2,NB12,NB21]) :-
    (PS1=idle,          PS2=idle, I=[ap1,tcreq,p], no_congestion,
     NS1=wait_accept,   NS2=idle, actions_1(PB12,NB12,PB21,NB21))
;
    (PS1=data_transfer, PS2=data_transfer, I=[ap1,tdatr,DF], flow_control_to_tpentity_ready,
     NS1=data_transfer, NS2=data_transfer, actions_2(PB12,NB12,PB21,NB21,DF))
;
    (PS1=data_transfer, PS2=data_transfer, I=[ap1,tdreq,user],
     NS1=disconnect,    NS2=data_transfer, actions_3(PB12,NB12,PB21,NB21)).

ts([[v,O],[v,v]],[PS1,PS2,PB12,PB21],[NS1,NS2,NB12,NB21]) :-
    (PS1=wait_accept,   PS2=idle, internal_problem,
     NS1=idle,          NS2=idle, actions_4(PB12,NB12,PB21,NB21), O=[ap1,tdind,provider])
;
    (PS1=wait_accept,   PS2=wait_accept, internal_problem,
     NS1=disconnect,    NS2=wait_accept, actions_5(PB12,NB12,PB21,NB21), O=[ap1,tdind,provider])
;
    (PS1=wait_accept,   PS2=data_transfer, no_internal_problem,
     NS1=data_transfer, NS2=data_transfer, actions_6(PB12,NB12,PB21,NB21), O=[ap1,taind,p])
;
    (PS1=wait_accept,   PS2=data_transfer, internal_problem,
     NS1=disconnect,    NS2=data_transfer, actions_7(PB12,NB12,PB21,NB21), O=[ap1,tdind,provider])
;
    (PS1=data_transfer, PS2=data_transfer,not(PB21=[]), flow_control_to_user_ready,
     NS1=data_transfer, NS2=data_transfer, actions_8(PB12,NB12),
     get_next_buffer21(PB21,DF,NB21), O=[ap1,tdati,DF])
;
    (PS1=data_transfer, PS2=data_transfer, internal_problem,
     NS1=disconnect,    NS2=data_transfer, actions_9(PB12,NB12,PB21,NB21), O=[ap1,tdind,provider])
;
   ((PS1=data_transfer ;PS1=wait_accept), PS2=disconnect,
   ((internal_problem, R=provider) ;(no_internal_problem, R=user)),
    NS1=idle,          NS2=idle, actions_10(PB12,NB12,PB21,NB21), O=[ap1,tdind,R]).

ts([[v,v],[v,O]],[PS1,PS2,PB12,PB21],[NS1,NS2,NB12,NB21]) :-
    (PS1=wait_accept,   PS2=idle, no_internal_problem,
     NS1=wait_accept,   NS2=wait_accept, actions_11(PB12,NB12,PB21,NB21), O=[ap2,tcind,p])
;
    (PS1=wait_accept,   PS2=wait_accept, internal_problem,
     NS1=wait_accept,   NS2=disconnect,  actions_12(PB12,NB12,PB21,NB21), O=[ap2,tdind,provider])
;
    (PS1=wait_accept,   PS2=data_transfer, internal_problem,
     NS1=wait_accept,   NS2=disconnect,  actions_13(PB12,NB12,PB21,NB21), O=[ap2,tdind,provider])
;
    (PS1=data_transfer, PS2=data_transfer, not(PB12=[]), flow_control_to_user_ready,
     NS1=data_transfer, NS2=data_transfer, actions_14(PB21,NB21),
     get_next_buffer12(PB12,DF,NB12), O=[ap2,tdati,DF])
;
    (PS1=data_transfer, PS2=data_transfer, internal_problem,
     NS1=data_transfer, NS2=disconnect,    actions_15(PB12,NB12,PB21,NB21), O=[ap2,tdind,provider])
;
   (PS1=disconnect,   (PS2=data_transfer ;PS2=wait_accept),
   ((internal_problem, R=provider) ;(no_internal_problem, R=user)),
    NS1=idle,          NS2=idle,  actions_16(PB12,NB12,PB21,NB21), O=[ap2,tdind,R]).
```

```
ts([[v,v],[I,v]],[PS1,PS2,PB12,PB21],[NS1,NS2,NB12,NB21]) :-
    (PS1=wait_accept,   PS2=wait_accept, I=[ap2,tareq,p],
     NS1=wait_accept,   NS2=data_transfer, actions_17(PB12,NB12,PB21,NB21))
;
    (PS1=wait_accept,   PS2=wait_accept, I=[ap2,tdreq,user],
     NS1=wait_accept,   NS2=disconnect, actions_18(PB12,NB12,PB21,NB21))
;
    (PS1=wait_accept,   PS2=data_transfer, I=[ap2,tdatr,DF], flow_control_to_tpentity_ready,
     NS1=wait_accept,   NS2=data_transfer, actions_19(PB12,NB12,PB21,NB21,DF))
;
    (PS1=data_transfer, PS2=data_transfer, I=[ap2,tdatr,DF], flow_control_to_tpentity_ready,
     NS1=data_transfer, NS2=data_transfer, actions_20(PB12,NB12,PB21,NB21,DF))
;
    (PS1=data_transfer, PS2=data_transfer, I=[ap2,tdreq,user],
     NS1=data_transfer, NS2=disconnect, actions_21(PB12,NB12,PB21,NB21)).


no_action.
actions_1(B1, B1,B2, B2).
actions_2(B1,NB1,B2, B2,DF) :- buffer12_append(B1,[DF],NB1).
actions_3(B1, B1,B2, B2).
actions_4(B1, B1,B2, B2).
actions_5(B1, B1,B2, B2).
actions_6(B1, B1,B2, B2).
actions_7(B1, B1,B2, B2).
actions_8(B,B).
actions_9(B1, B1,B2, B2).
actions_10(B1, B1,B2, B2).
actions_11(B1, B1,B2, B2).
actions_12(B1, B1,B2, B2).
actions_13(B1, B1,B2, B2).
actions_14(B,B).
actions_15(B1, B1,B2, B2).
actions_16(B1, B1,B2, B2).
actions_17(B1, B1,B2, B2).
actions_18(B1, B1,B2, B2).
actions_19(B1, B1,B2,NB2,DF) :- buffer21_append(B2,[DF],NB2).
actions_20(B1, B1,B2,NB2,DF) :- buffer21_append(B2,[DF],NB2).
actions_21(B1, B1,B2, B2).

buffer12_append(B1,DF,NB1) :- append(B1,DF,NB1).
buffer21_append(B2,DF,NB2) :- append(B2,DF,NB2).

get_next_buffer12([DF|NB1],DF,NB1).
get_next_buffer21([DF|NB2],DF,NB2).

:- ['setparam_tsp'].
:- set_params(0,1,1,0).
```

APPENDIX E : MINIMUM COVER CONSTRUCTION FOR FSM SPECIFICATIONS

```
GRAM : PROC OPTIONS (MAIN);

   DCL FILE1 FILE INPUT
       ENV(F BLKSIZE (80));
   DCL (REG , TSG) FILE OUTPUT
       ENV(F BLKSIZE(80));
   DCL (PRODUCTION , STRING) CHAR(80) VARYING;
   DCL (STATE , NEXTSTATE , IN , OUT) CHAR(20) VARYING;
   DCL NOTEOF BIT(1) INITIAL('1'B);
   DCL (N , M , IS , MAX , LAST)  FIXED INITIAL(0);
   ON ENDFILE (FILE1) NOTEOF = '0'B;
   PUT FILE(REG) SKIP(10);
   PUT FILE(REG) EDIT ('REGULAR  GRAMMAR')(COL(30),A(16));


   GET FILE(FILE1) LIST(STATE , IN , OUT , NEXTSTATE);
   IS = SUBSTR(STATE , 2 , 1);
   DO WHILE (NOTEOF);
      IF (INDEX(STRING , STATE) = 0) THEN DO;
         IF (LAST = 1) THEN DO;
            PRODUCTION = '       | EMPTY';
            PUT FILE(REG) SKIP EDIT(PRODUCTION) (COL(2),A);
         END; /* IF */
         PUT FILE(REG) SKIP(5);
         PRODUCTION = STATE || ' --> ' || IN || ' ' || NEXTSTATE;
         PUT FILE(REG) SKIP EDIT(PRODUCTION) (COL(2),A);
         N = N + 1;
         STRING = STRING || STATE;
         IF (M > MAX)THEN
             MAX = M;
         M = 1;
         LAST = LAST + 1;
      END;
      ELSE DO;
         PRODUCTION ='       | ' || IN || ' ' || NEXTSTATE;
         PUT FILE(REG) SKIP EDIT(PRODUCTION) (COL(2),A);
         M = M + 1;
      END; /* ELSE */
      GET FILE(FILE1) LIST(STATE , IN , OUT , NEXTSTATE);
   END; /* DO WHILE */
   NOTEOF = '1'B;
   MAX = MAX + 1;
   CLOSE FILE (FILE1);
   OPEN FILE (FILE1) INPUT;
   BEGIN;
      DCL 1   ADJ(N , MAX) ,
              2 GUARD FIXED(2),
              2 PSTATE FIXED(1),
              2 INP CHAR(20) VARYING,
              2 OUTP CHAR(20) VARYING,
              2 NSTATE FIXED(1);
      DCL SAVE(N);
      DCL (P , I , J , K , L , SUB) FIXED INITIAL(0);
      DCL PRESENT CHAR(20) VARYING INITIAL('');
```

```
ON ENDFILE(FILE1) NOTEOF = '0'B;
GET FILE(FILE1) LIST(STATE , IN , OUT , NEXTSTATE);
I = 0;
DO WHILE(NOTEOF);
   J = 1;
   I = I + 1;
   IF (PRESENT ~= STATE) THEN DO;
      CALL INITMAT;
      PRESENT = STATE;
      SUB = ADJ(I , J).NSTATE;
      IF (SUB = IS | SUB = I) THEN
         P = P + 1;
      J = 2;
      GET FILE(FILE1) LIST(STATE , IN ,OUT ,NEXTSTATE);
      DO WHILE((PRESENT = STATE) & (NOTEOF));
         CALL INITMAT;
         SUB = ADJ(I , J).NSTATE;
         IF (SUB = IS | SUB = I) THEN
            P = P + 1;
         J = J + 1;
         GET FILE(FILE1) LIST(STATE , IN , OUT ,NEXTSTATE);
      END; /* DO WHILE */
   END; /* IF */
END; /* DO WHILE */
PUT FILE(REG) SKIP(10);
PUT FILE(REG) EDIT('MINIMAL COVER')(COL(30),A);
CALL AUXNT;
PUT FILE(REG) SKIP(5);
DO I = 1 TO N;
   PUT FILE(TSG) EDIT ('P',I,'(') (COL(2),A(1),F(1),A(1));
   DO J = 1 TO (SAVE(I) - 1);
      PUT FILE(TSG) EDIT ('1,') (A(2));
   END; /* DO J */
   PUT FILE(TSG) EDIT ('1).') (A(3));
END; /* DO I */
PUT FILE(REG) SKIP(10);
PUT FILE(REG) EDIT ('ATTRIBUTED GRAMMAR')(COL(30),A(18));
PUT FILE(REG) SKIP(5);

DO I = 1 TO N;
   K = 1;
   DO J = 1 TO MAX;
      IF (J <= SAVE(I)) THEN DO;
         PUT FILE(REG) EDIT (' S') (COL(2),A(2));
         PUT FILE(REG) EDIT (ADJ(I,J).PSTATE) (F(1));
         PUT FILE(REG) EDIT ('     = GS') (A(9));
         PUT FILE(REG) EDIT (ADJ(I,J).PSTATE,K) (F(1),F(1));
         PUT FILE(REG) EDIT (' LE ') (A(4));
         PUT FILE(REG) EDIT (ADJ(I,J).GUARD) (F(1));
         PUT FILE(REG) EDIT (' : I INC(GS') (A(11));
         PUT FILE(REG) EDIT (ADJ(I,J).PSTATE,K) (F(1),F(1));
         PUT FILE(REG) EDIT (') I(') (A(5));
         PUT FILE(REG) EDIT (ADJ(I,J).INP) (A);
         PUT FILE(REG) EDIT (') O(') (A);
```

```
                PUT FILE(REG) EDIT (ADJ(I,J).OUTP)'(A);
                PUT FILE(REG) EDIT (')  S') (A);
                PUT FILE(REG) EDIT (ADJ(I,J).NSTATE) (F(1));
                PUT FILE(REG) SKIP;
                PUT FILE(TSG) EDIT ('S',ADJ(I,J).PSTATE)
                                   (COL(2),A(1),F(1));
                PUT FILE(TSG) EDIT ('  <-  AXN(P',ADJ(I,J).PSTATE)
                                   (A(11),F(1));
                PUT FILE(TSG) EDIT(',',SAVE(I)') (A(1),F(1));
                PUT FILE(TSG) EDIT (',P',ADJ(I,J).PSTATE,'(')
                                   (A(2),F(1),A(1));
                PRODUCTION = '(';
                DO L = 1 TO (SAVE(I) - 1);
                   PRODUCTION = PRODUCTION || '*,';
                END; /* DO L */
                PRODUCTION = PRODUCTION || '*)';
                LAST = 1;
                CALL STROUT;
                PUT FILE(TSG) EDIT (')  &  LE(') (A(9));
                PUT FILE(TSG) EDIT ('*GS',ADJ(I,J).PSTATE,K)
                                   (A(3),F(1),F(1));
                PUT FILE(TSG) EDIT (',',ADJ(I,J).GUARD,')')
                                   (A(1),F(1),A(1));
                PUT FILE(TSG) EDIT ('&  DELAX(P',ADJ(I,J).PSTATE)
                                   (COL(30),A(10),F(1));
                PUT FILE(TSG) EDIT (PRODUCTION,')') (A,A(1));
                PUT FILE(TSG) EDIT ('&  SUM(*GS') (COL(30),A(10));
                PUT FILE(TSG) EDIT (ADJ(I,J).PSTATE,K) (F(1),F(1));
                PUT FILE(TSG) EDIT (',1,*G)') (A(6));
                PUT FILE(TSG) EDIT ('&  ADDAX(P',ADJ(I,J).PSTATE,'(')
                                   (COL(30),A(10),F(1),A(1));
                LAST = 0;
                CALL STROUT;
                PUT FILE(TSG) EDIT (')') (A(1));
                PUT FILE(TSG) EDIT ('&  IN(''',ADJ(I,J).INP,''')')
                                   (COL(30),A(7),A,A(2));
                PUT FILE(TSG) EDIT ('&  OUT(''',ADJ(I,J).OUTP,''')')
                                   (COL(30),A(8),A,A(2));
                PUT FILE(TSG) EDIT ('&  S',ADJ(I,J).NSTATE,'.')
                                   (COL(30),A(4),F(1),A(1));
                K = K + 1;
             END; /* IF */
          END; /* DO */
       END; /* DO */
       PUT FILE(TSG) EDIT ('IN(*X)  <-  WRITECH(*X).')
                          (COL(2),A(24));
       PUT FILE(TSG) EDIT ('OUT(*X)  <-  WRITECH(''('')')
                          (COL(2),A(25));
       PUT FILE(TSG) EDIT ('  &  WRITECH(*X)  &  WRITECH(')'').')
                          (A(34));
```

```
STROUT : PROC;
   DCL A FIXED;

   DO A = 1 TO (SAVE(I) - 1);
      IF (A=K) & (LAST = 0) THEN
         PUT FILE(TSG) EDIT ('*G,') (A(3));
      ELSE DO;
         PUT FILE(TSG) EDIT ('*GS',ADJ(I,J).PSTATE,A,',')
                           (A(3),F(1),F(1),A(1));
      END; /* ELSE */
   END; /* DO A */
   IF (K = SAVE(I) & LAST = 0) THEN
      PUT FILE(TSG) EDIT ('*G)') (A(3));
   ELSE
      PUT FILE(TSG) EDIT ('*GS',ADJ(I,J).PSTATE,SAVE(I),')')
                        (A(3),F(1),F(1),A(1));
END STROUT;



INITMAT : PROC;
   ADJ(I , J).PSTATE = SUBSTR(STATE , 2 , 1);
   ADJ(I , J).INP = IN;
   ADJ(I , J).OUTP = OUT;
   ADJ(I , J).NSTATE = SUBSTR(NEXTSTATE , 2 , 1);
   SAVE(I) = SAVE(I) + 1;
END INITMAT;
AUXNT: PROCEDURE;
   DCL (K,L,M) FIXED BIN(31);
   DCL (MAT(1:99,1:99,4),STA(1:99,4),LAB(1:99)) FIXED BIN(31);
   M = N + 1;
   DO I = 1 TO N BY 1;
      L = I + 1;
      J = 1;
      DO WHILE (J <= SAVE(I));
         IF (ADJ(I,J).NSTATE = IS)
            THEN
            DO;
              M = M + 1;
              MAT(L,M,1) = 0;
              MAT(L,M,2) = 1;
              MAT(L,M,3) = 999;
              MAT(M,N+P+2,1) = 1;
              MAT(M,N+P+2,2) = 1;
              MAT(M,N+P+2,3) = 999;
            END;
            ELSE
            DO;
              IF (ADJ(I,J).NSTATE = I)
                 THEN
                 DO;
                   M = M + 1;
                   MAT(L,M,1) = 0;
```

```
                        MAT(L,M,2) = 1;
                        MAT(L,M,3) = 999;
                        MAT(M,L,1) = 1;
                        MAT(M,L,2) = 1;
                        MAT(M,L,3) = 999;
                     END;
                     ELSE
                     DO;
                        K = ADJ(I,J).NSTATE + 1;
                        MAT(L,K,1) = 1;
                        MAT(L,K,2) = 1;
                        MAT(L,K,3) = 999;
                     END;
                  END;
               J = J + 1;
            END;
         END;
         DO I = 2 TO N + P + 1 BY 1;
            M = 0;
            DO J = 2 TO N + P + 2 BY 1;
               M = M + MAT(I,J,2);
            END;
            MAT(I,N + P + 3,3) = M;
         END;
         DO J = 2 TO N + P + 2 BY 1;
            M = 0;
            DO I = 2 TO N + P + 2 BY 1;
               M = M + MAT(I,J,2);
            END;
            MAT(1,J,3) = M;
         END;
         MAT(2,N + P + 2,3) = 999;
         MAT(N + P + 2,2,3) = 999;
         N = N + P + 2;
      DO I = 1 TO N BY 1;
         LAB(I) = 0;
         DO J = 1 TO 4 BY 1;
            STA(I,J) = 0;
         END;
      END;
      CFB = 1;
      STA(1,1) = 1;
      STA(1,4) = 999;
      L = 1;
      LAB(1) = 1;
      I = 1;
      J = N + 1;
      DO WHILE (CFB > 0);
         IF (CFB = 1)
            THEN
            DO;
               DO WHILE (J > 0);
                  IF (MAT(I,J,3) > 0 & LAB(J) = 0 & MAT(I,J,3)>MAT(I,J,4))
                     THEN
```

```
                DO;
                  LAB(J) = 1;
                  L = L + 1;
                  STA(L,1) = J;
                  STA(L,2) = I;
                  STA(L,3) = 0;
                  M = MAT(I,J,3) - MAT(I,J,4);
                  IF (M < STA(L-1,4))
                     THEN STA(L,4) = M;
                     ELSE STA(L,4) = STA(L-1,4);
                  I = J;
                  IF (I ~= N + 1) THEN J = N + 2;
                END;
              J = J - 1;
            END;
            IF (I = N + 1)
               THEN
               DO;
                 M = STA(L,4);
                 DO K = L TO 2 BY -1;
                   IF (STA(K,3) = 0)
                      THEN MAT(STA(K,2),STA(K,1),4) =
                           MAT(STA(K,2),STA(K,1),4) + M;
                      ELSE MAT(STA(K,1),STA(K,2),4) =
                           MAT(STA(K,1),STA(K,2),4) - M;
                 END;
                 DO K = 2 TO N + 1 BY 1;
                    LAB(K) = 0;
                    DO M = 1 TO 4 BY 1;
                       STA(K,M) = 0;
                    END;
                 END;
                 I = 1;
                 J = N + 1;
                 L = 1;
               END;
               ELSE
               DO;
                 IF (I = 1) THEN CFB = 0;
                            ELSE DO;
                                   J = I;
                                   I = N;
                                   CFB = 2;
                                 END;
               END;
         END;
      END;
   IF (CFB = 2)
      THEN
      DO;
        DO WHILE (I > 1 & J > 1);
           IF (MAT(I,J,3) > 0 & LAB(I) = 0 & MAT(I,J,4)>MAT(I,J,2))
              THEN
              DO;
                LAB(I) = 1;
```

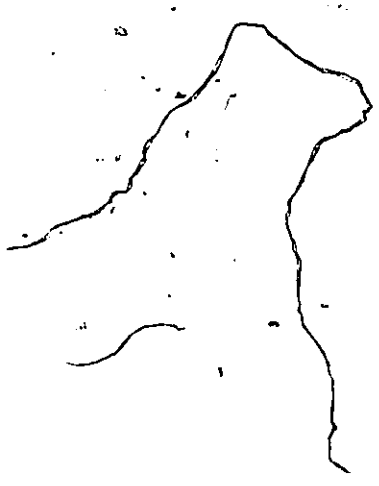
```
                        L = L + 1;
                        STA(L,1) = I;
                        STA(L,2) = J;
                        STA(L,3) = 1;
                        M = MAT(I,J,4) - MAT(I,J,2);
                        IF (M < STA(L-1,4))
                           THEN STA(L,4) = M;
                           ELSE STA(L,4) = STA(L-1,4);
                        J = 1;
                        I = I + 1;
                     END;
                  I = I - 1;
               END;
               IF (J = 1 & I > 1)
                  THEN
                  DO;
                    J = N + 1;
                    CFB = 1;
                  END;
                  ELSE
                  DO;
                    LAB(STA(L,1)) = 0;
                    M = STA(L,3);
                    L = L - 1;
                    IF (STA(L,1) = 1)
                       THEN CFB = 0;
                       ELSE
                       DO;
                         IF (M = 0)
                            THEN
                            DO;
                              I = STA(L,1);
                              J = J - 1;
                              CFB = 1;
                            END;
                            ELSE
                            DO;
                              I = J - 1;
                              J = STA(L,1);
                              CFB = 2;
                            END;
                       END;
                  END;
            END;
      END;
      N = N - 2;
      DO I = 2 TO N + 1 BY 1;
         DO J = 2 TO N + 2 BY 1;
            MAT(I,J,4) = MAT(I,J,4) + MAT(I,J,2);
         END;
      END;
      DO I = 1 TO N BY 1;
         LAB(I) = 0;
         DO J = 1 TO 4 BY 1;
```

```
      STA(I,J) = 0;
   END;
END;
CFB = 2;
STA(1,1) = 1;
STA(1,4) = 999;
L = 1;
LAB(N + 2) = 1;
J = N + 2;
I = N + 1;
DO WHILE (CFB > 0);
   IF (CFB = 2)
      THEN
      DO;
        DO WHILE (I > 1);
           IF (MAT(I,J,2) > 0 & LAB(I) = 0 & MAT(I,J,4)>MAT(I,J,2))
              THEN
              DO;
                 LAB(I) = 1;
                 L = L + 1;
                 STA(L,1) = I;
                 STA(L,2) = J;
                 STA(L,3) = 1;
                 M = MAT(I,J,4) - MAT(I,J,2);
                 IF (M < STA(L - 1,4))
                    THEN STA(L,4) = M;
                    ELSE STA(L,4) = STA(L - 1,4);
                 J = I;
                 IF (J ~= 2) THEN I = N + 2;
              END;
           I = I - 1;
        END;
        IF (J = 2)
           THEN
           DO;
             M = STA(L,4);
             DO K = L TO 2 BY -1;
                IF (STA(K,3) = 0)
                   THEN MAT(STA(K,2),STA(K,1),4) =
                        MAT(STA(K,2),STA(K,1),4) + M;
                   ELSE MAT(STA(K,1),STA(K,2),4) =
                        MAT(STA(K,1),STA(K,2),4) - M;
             END;
             DO K = 2 TO N + 1 BY 1;
                LAB(K) = 0;
                DO M = 1 TO 4 BY 1;
                   STA(K,M) = 0;
                END;
             END;
             L = 1;
             J = N + 2;
             I = N + 1;
           END;
           ELSE
```

```
            DO;
              IF (J = N + 2) THEN CFB = 0;
                 ELSE
                 DO;
                   I = J;
                   J = N + 2;
                   CFB = 1;
                 END;
            END;
      END;
   IF (CFB = 1)
      THEN
      DO;
        DO WHILE (J > 1 & I > 1);
          IF (MAT(I,J,2) > 0 & LAB(J) = 0)
             THEN
             DO;
               LAB(J) = 1;
               L = L + 1;
               STA(L,1) = J;
               STA(L,2) = I;
               STA(L,3) = 0;
               STA(L,4) = STA(L - 1,4);
               I = 1;
               J = J + 1;
             END;
          J = J - 1;
        END;
        IF (I = 1 & J > 1)
           THEN
           DO;
             I = N + 1;
             CFB = 2;
           END;
           ELSE
           DO;
             LAB(STA(L,1)) = 0;
             M = STA(L,3);
             L = L - 1;
             IF (STA(L,1) = 1)
                THEN CFB = 0;
                ELSE
                DO;
                  IF (M = 0)
                     THEN
                     DO;
                       J = I - 1;
                       I = STA(L,1);
                       CFB = 1;
                     END;
                     ELSE
                     DO;
                       I = I - 1;
                       J = STA(L,1);
```

```
                        CFB = 2;
                        END;
                    END;
                END;
            END;
        END;
        N = N - P;
        DO I = 2 TO N + 1 BY 1;
            K = I - 1;
            L = 0;
            DO J = 2 TO N + P + 2 BY 1;
                IF (MAT(I,J,4) ^= 0)
                THEN
                DO;
                    L = L + 1;
                    IF (MAT(I,J,1) = 1)
                        THEN ADJ(K,L).GUARD = MAT(I,J,4);
                        ELSE
                        DO;
                            IF (MAT(I,J,4) = MAT(J,N+P+2,4) |
                                MAT(I,J,4) = MAT(J,I,4))
                                THEN ADJ(K,L).GUARD = MAT(I,J,4);
                                ELSE ADJ(K,L).GUARD = 0;
                        END;
                END;
            END;
        END;
    DO I = 1 TO N;
        DO J = 1 TO MAX;
            IF (J <= SAVE(I)) THEN
                PUT FILE(REG) SKIP LIST (ADJ(I,J).GUARD);
        END;
    END;
    END AUXNT;
        END; /* BEGIN */

    END GRAM;
```



APPENDIX F : A TEST SEQUENCE GRAMMAR FOR TRANSPORT PROTOCOL CLASS 0

```
tptest(IRB,CDTB,SRB,S,R,P1,P2,P3,DNB,NRB,NDB,UEB) :-
     set_params(IRB,CDTB,SRB,S,R,P1,P2,P3,DNB,NRB,NDB,UEB),
     setof(X,s1(X,[]),L), outprpt(L), !.

s0 --> [], [nl].

s1 -->
 ((initiator),
     ((connect_only; data_transfer_only),
      (acceptable_tcreq),
       input(tcreq,0),
       output(cr,0),
       s3

     ;(connect_only, unacceptable_tcreq),
       input(tcreq,1),
       output(tdind,0),
       s0
     )
 )
;((responder),
     ((connect_only; data_transfer_only),
      (acceptable_cr_tpdu),
       input(cr,0),
       output(tcind,0),
       s2

     ;(connect_only, unacceptable_cr_tpdu),
       input(cr,1),
       output(dr,0),
       s0
     )

;((unspecified, connect_only),
    (input(cc,0), output(nil)
    ;input(dt,0), output(nil)
    ;input(dr,0), output(nil)
    )
 ).

s2 -->
 ((connect_only),
   (((disconnector), input(tdreq,0), output(dr,0), s0)
   ;((error),
      (input(dr,0), output(er,0), s0
      ;input(cr,0), output(er,0), s0
      ;input(cc,0), output(er,0), s0
      ;input(dt,0), output(er,0), s0
      )
```

```
    ))
  )
 ;((connect_only; data_transfer_only),
     input(tcres,0),
     output(cc,0),
     (getval(S,R)), s4(S,R)
 ).

 s3 -->
  ((connect_only),
    (((not_disconnector),
       input(dr,0),
       output(tdind,0),
       output(ndreq,0),
       s0
     )
    ;((unspecified),
       input(dt,0),
       output(nil),
       s0
     )
   ;((unacceptable_cc_tpdu),
       input(cc,1),
       output(tdind,0),
       output(ndreq,0),
       s0
     )
    )
  )
 ;((connect_only; data_transfer_only),
     (acceptable_cc_tpdu),
      input(cc,0),
      output(tccon,0),
      (getval(S,R)), s4(S,R)
 ).

 s4(0,0) -->
  ((connect_only; (data_transfer_only, (sender; receiver) )),
    ((disconnector), input(tdreq,0), output(ndreq,0), s0)
   ;((not_disconnector),
      ((                        input(dr,0),    output(ndreq,0), s0)
      ;((network_reset),        input(nrind,0), output(tdind,0), s0)
      ;((network_disconnect),   input(ndind,0), output(tdind,0), s0)
      ;((error),                input(cr,0),    output(er,0),    s0)
      ))).
 s4(S,R) -->
  ((data_transfer_only),
     ((receiver, R>0, rbase(BR), SN is BR-R+1, RR is R-1),
       input(dt,0),
       output(tdati,SN),
       s4(S,RR)
```

```
    )
   ;((sender, S>0, sbase(BS), SN is BS-S+1, SS is S-1),
      input(tdatr,SN),
      output(dt,0),
      s4(SS,R)
    )
  ).

 input(I,P) -->
  ((I=tcreq), ['i '], [tcreq], ['('], tsp_param(tcreq,P), [')'], [nl])
 ;((I=tcres), ['i '], [tcres], ['('], tsp_param(tcres,P), [')'], [nl])
 ;((I=tdatr), ['i '], [tdatr], ['('], tsp_param(tdatr,P), [')'], [nl])
 ;((I=tdreq), ['i '], [tdreq], ['('], tsp_param(tdreq,P), [')'], [nl])
 ;((I=cr   ), ['i '], [cr],    ['('], tpdu_param(cr,P),   [')'], [nl])
 ;((I=cc   ), ['i '], [cc],    ['('], tpdu_param(cc,P),   [')'], [nl])
 ;((I=dt   ), ['i '], [dt],    ['('], tpdu_param(dt,P),   [')'], [nl])
 ;((I=dr   ), ['i '], [dr],    ['('], tpdu_param(dr,P),   [')'], [nl])
 ;((I=nrind), ['i '], [nrind], ['('], nsp_param(nrind,P), [')'], [nl]).
 ;((I=ndind), ['i '], [ndind], ['('], nsp_param(ndind,P), [')'], [nl]).

 output(nil) --> [], [nl].
 output(O,P) -->
  ((O=tcind), ['o '], [tcind], ['('], tsp_param(tcind,P), [')'], [nl])
 ;((O=tccon), ['o '], [tccon], ['('], tsp_param(tccon,P), [')'], [nl])
 ;((O=tdati), ['o '], [tdati], ['('], tsp_param(tdati,P), [')'], [nl])
 ;((O=tdind), ['o '], [tdind], ['('], tsp_param(tdind,P), [')'], [nl])
 ;((O=cr   ), ['o '], [cr],    ['('], tpdu_param(cr,P),   [')'], [nl])
 ;((O=dt   ), ['o '], [dt],    ['('], tpdu_param(dt,P),   [')'], [nl])
 ;((O=dr   ), ['o '], [dr],    ['('], tpdu_param(dr,P),   [')'], [nl])
 ;((O=cc   ), ['o '], [cc],    ['('], tpdu_param(cc,P),   [')'], [nl])
 ;((O=er   ), ['o '], [er],    ['('], tpdu_param(er,P),   [')'], [nl])
 ;((O=ndreq), ['o '], [ndreq], ['('], nsp_param(ndreq,P), [')'], [nl]).

 tsp_param(tcreq,0) -->
 tcepi(0), to_addr(0), from_addr(0), qots_req(0), options(0), tscon_data(0).

 tsp_param(tcreq,1) -->
  tcepi(0), to_addr(0), from_addr(0), qots_req(1), options(0), tscon_data(0)
 ;tcepi(0), to_addr(0), from_addr(0), qots_req(0), options(1), tscon_data(0).

 tsp_param(tcind,0) --> tcepi(0), to_addr(0), from_addr(0), qots_req(0),
       options(0), tscon_data(0).
 tsp_param(tcres,0) --> tcepi(0), qots_req(0), options(0), tsacc_data(0).
 tsp_param(tccon,0) --> tcepi(0), qots_req(0), options(0), tsacc_data(0).
 tsp_param(tdatr,P) --> tcepi(0), tsdu_fragment(P).
 tsp_param(tdati,P) --> tcepi(0), tsdu_fragment(P).
 tsp_param(tdreq,0) --> tcepi(0), discon_reason(0), user_reason(0).
 tsp_param(tdind,0) --> tcepi(0), discon_reason(0), user_reason(0).

 tsdu_fragment(X) --> [X].
 calculate(_,_) --> [].
```

```
tpdu_param(cr,0) --> length(0,1), crcdt(0,1), dst_ref(0,1), src_ref(0,1), class_option(0,1),
       tsap_id1(0,1), tsap_id2(0,1), tpdu_size(0,1), throughput(0,1), transmit_delay(0,1),
       calculate(0,1).
tpdu_param(cr,1) --> length(0,1), crcdt(0,1), dst_ref(0,1), src_ref(0,1), class_option(0,1),
       tsap_id1(0,1), tsap_id2(0,1), tpdu_size(1,0), throughput(0,1), transmit_delay(0,1),
       calculate(0,1).
tpdu_param(cc,0) --> length(0,1), cccdt(0,1), dst_ref(0,1), src_ref(0,1), class_option(0,1),
       tsap_id1(0,1), tsap_id2(0,1), tpdu_size(0,1), throughput(0,1), transmit_delay(0,1),
       calculate(0,1).
tpdu_param(cc,1) --> length(0,1), cccdt(0,1), dst_ref(0,1), src_ref(0,1), class_option(0,1),
       tsap_id1(0,1), tsap_id2(0,1), tpdu_size(1,0), throughput(0,1), transmit_delay(0,1),
       calculate(0,1).
tpdu_param(dr,0) --> length(0,1), dr_code(0,1), dst_ref(0,1), src_ref(0,1),
       dr_reason(0,1), info(0,1), calculate(0,1).
tpdu_param(dt,0) --> length(0,1), dt_code(0,1), eot_seqnp(0,1), user_data(0,1), calculate(0,1).
tpdu_param(er,0) --> length(0,1), er_code(0,1), dst_ref(0,1), reject_cause(0,1),
       invalid_tpdu(0,1), calculate(0,1).

nsp_param(ndreq,0) --> ncepi(0), discon_reason(0), user_reason(0).
nsp_param(ndind,0) --> ncepi(0), discon_reason(0), user_reason(0).
nsp_param(nrind,0) --> ncepi(0), reason(0).

ncepi(0) --> ['(0)'].
tcepi(0) --> ['(1)'].
to_addr(0) --> ['(2)'].
from_addr(0) --> ['(1)'].
qots_req(0) --> ['(+)'].
qots_req(1) --> ['(-)'].
options(0) --> ['(%)'].
options(1) --> ['($)'].
tscon_data(0) --> ['(&)'].
tsacc_data(0) --> ['(#)'].
user_reason(0) --> ['(^)'].
user_reason(1) --> ['(^^)'].
discon_reason(0) --> ['(@)'].
discon_reason(1) --> ['(?)'].
reason(0) --> ['(!!)'].

getval(S,R) :- data_sent(S), data_received(R).

decrement(Header) :- F=..[Header,Count], F, Newcount is Count - 1,
    NF=..[Header,Newcount], retract(F), assert(NF).

outprnt([]).
outprnt([H|T]) :- writer(H), outprnt(T).
writer([]) :- !.
writer(nl) :- nl.
writer([nl|T]) :- nl, !, writer(T).
writer([X|T]) :- write(X), !, writer(T).
```

```
length(E,N) -->
        (E=0, N>0, N<5), ['(2B)']
       ;(E=1, N=0), ['(2A)']
       ;(E=2, N=0), ['(2C)'].

crcdt(E,N) --> cr_code(E,N), cr_cdt_code(E,N).
cr_code(E,N) -->
        (E=0, N>0, N<5), ['(E)'].

cr_cdt_code(E,N) -->
       (E=0, N>0, N<5), ['(0)']
       ;(E=2, N=0), ['(1)'].

dst_ref(E,N) -->
        (E=0, N>0, N<5), ['(1)']
       ;(E=1, N=0), ['(FFFF)']
       ;(E=2, N=0), ['(0001)'].

src_ref(E,N) -->
        (E=0, N>0, N<5), ['(1)']
       ;(E=1, N=0), ['(08AD)']
       ;(E=2, N=0), ['(08AF)'].

class_option(E,N) -->
        (E=0, N>0, N<5), ['(00)']
       ;(E=2, N=0), ['(01)'].

tsap_id1(E,N) -->
        tsap_id1_code(E,N),
        tsap_id1_length(E,N),
        tsap_id1_value(E,N).

tsap_id1_code(E,N) -->
        (E=0, N>0, N<5), ['(C1)']
       ;(E=1, N=0), ['(BF)']
       ;(E=2, N=0), ['(C3)'].

tsap_id1_length(E,N) -->
        (E=0, N>0, N<5), ['(03)']
       ;(E=1, N=0), ['(02)']
       ;(E=2, N=0), ['(04)'].

tsap_id1_value(E,N) --> (E=0, N>0, N<5), ['(1)'].

tsap_id2(E,N) -->
        tsap_id2_code(E,N),
        tsap_id2_length(E,N),
        tsap_id2_value(E,N).

tsap_id2_code(E,N) -->
        (E=0, N>0, N<5), ['(C2)']
```

```
        ;(E=1, N=0), ['(BF)']
        ;(E=2, N=0), ['(C3)'].

tsap_id2_length(E,N) -->
        (E=0, N>0, N<5), ['(03)']
       ;(E=1, N=0), ['(02)']
       ;(E=2, N=0), ['(04)'].

tsap_id2_value(E,N) -->  (E=0, N>0, N<5), ['(2)'].

tpdu_size(E,N) -->
        tpdu_size_code(E,N),
        tpdu_size_length(E,N),
        tpdu_size_value(E,N).

tpdu_size_code(E,N) -->
        (E=0, N>0, N<5), ['(C0)']
       ;(E=1, N=0), ['(BF)']
       ;(E=2, N=0), ['(C3)'].

tpdu_size_length(E,N) -->
        (E=0, N>0, N<5), ['(01)']
       ;(E=1, N=0), ['(00)']
       ;(E=2, N=0), ['(02)'].

tpdu_size_value(E,N) -->
        (E=0, N>0, N<5), ['(07)']
       ;(E=1, N=0), ['(06)']
       ;(E=2, N=0), ['(08)'].

throughput(E,N) -->
        throughput_code(E,N),
        throughput_length(E,N),
        throughput_value(E,N).

throughput_code(E,N) -->
        (E=0, N>0, N<5), ['(89)']
       ;(E=1, N=0), ['(87)']
       ;(E=2, N=0), ['(8A)'].

throughput_length(E,N) -->
        (E=0, N>0, N<5), ['(0C)']
       ;(E=1, N=0), ['(0B)']
       ;(E=2, N=0), ['(0D)'].

throughput_value(E,N) -->
       (E=0, N>0, N<5), ['(', [34], 222111444333, [34], ')'].

transmit_delay(E,N)-->
        transmit_delay_code(E,N),
        transmit_delay_length(E,N),
```

```
        transmit_delay_value(E,N).

transmit_delay_code(E,N) -->
        (E=0, N>0, N<5), ['(88)']
      ;(E=1, N=0), ['(87)']
      ;(E=2, N=0), ['(8A)'].

transmit_delay_length(E,N) -->
        (E=0, N>0, N<5), ['(08)']
      ;(E=1, N=0), ['(07)']
      ;(E=2, N=0), ['(09)'].

transmit_delay_value(E,N) -->
        (E=0, N>0, N<5), ['(', [34], 55447766, [34], ')'].

cccdt(E,N) --> cc_code(E,N), cc_cdt_code(E,N).
cc_code(E,N) --> (E=0, N>0, N<5), ['(D)'].

cc_cdt_code(E,N) -->
        (E=0, N>0, N<5), ['(0)']
      ;(E=2, N=0), ['(1)'].

dr_code(E,N) -->
        (E=0, N>0, N<5), ['(80)']
      ;(E=1, N=0), ['(7F)']
      ;(E=2, N=0), ['(81)'].

dr_reason(E,N) -->
        (E=0, (N=1; N=4)), ['(00)']
      ;(E=0, (N=2; N=7)), ['(03)']
      ;(E=0, N=3), (['(00)']; ['(01)'] ; ['(02)'] ; ['(03)'])
      ;(E=0, N=5), ['(01)']
      ;(E=0, N=6), ['(02)']
      ;(E=2, N=0), ['(04)'].

info(E,N) -->
        info_code(E,N),
        info_length(E,N),
        info_value(E,N).

info_code(E,N) -->
        (E=0, N>0, N<5), ['(E0)']
      ;(E=1, N=0), ['(DF)']
      ;(E=2, N=0), ['(E1)'].

info_length(E,N) -->
        (E=0, N>0, N<6), ['(?)']
      ;(E=1, N=0), ['(00)']
      ;(E=2, N=0), ['(7A)'].

info_value(E,N) -->  (E=0, N>0, N<6), ['(1 to 119 bytes internal info)'].
```

```
dt_code(E,N) -->
        (E=0, N>0, N<5), ['(F0)']
       ;(E=1, N=0), ['(EF)']
       ;(E=2,, N=0), ['(F1)'].

eot_seqno(E,N) -->
        (E=0, (N=1; N=3; N=4)), ['(00)']
       ;(E=0, (N=2; N=3; N=5)), ['(80)']
       ;(E=1, N=0), ['(FF)']
       ;(E=2, N=0), ['(81)'].

user_data(E,N) -->  (E=0, N>0, N<6), ['(1 to 125 bytes user data)'].

er_code(E,N) -->
        (E=0, N>0, N<5), ['(70)']
       ;(E=1, N=0), ['(6F)']
       ;(E=2, N=0), ['(71)'].

reject_cause(E,N) -->
        (E=0, (N=1; N=4)), ['(00)']
       ;(E=0, (N=2; N=7)), ['(03)']
       ;(E=0, N=3), (['(00)'] ; ['(01)'] ; ['(02)'] ; ['(03)'])
       ;(E=0, N=5), ['(01)']
       ;(E=0, N=6), ['(02)']
       ;(E=2, N=0), ['(04)'].

invalid_tpdu(E,N) -->
        invalid_tpdu_code(E,N),
        invalid_tpdu_length(E,N),
        invalid_tpdu_value(E,N).

invalid_tpdu_code(E,N) -->
        (E=0, N>0, N<5), ['(C1)']
       ;(E=1, N=0), ['(C0)']
       ;(E=2, N=0), ['(C2)'].

invalid_tpdu_length(E,N) -->
        (E=0, N>0, N<6), ['(?)']
       ;(E=1, N=0), ['(00)']
       ;(E=2, N=0), ['(7A)'].

invalid_tpdu_value(E,N) -->
        (E=0, N>0, N<6), ['(1 to 121 bytes from erred TPDU)'].
```

```
set_params(IRB,CDTB,SRB,S,R,P1,P2,P3,DNB,NRB,NDB,UEB) :-
    (A=initiator, B=responder,
        (IRB=1,  (not(A) ->    assert(A);  true)
                ,(    B  ->   retract(B);  true)
        ;IRB=2,  (    A  ->   retract(A);  true)
                ,(not(B) ->    assert(B);  true)
        ;IRB=3,  (not(A) ->    assert(A);  true)
                ,(not(B) ->    assert(B);  true))),
    (C=connect_only, D=data_transfer_only,
        (CDTB=1, (not(C) ->    assert(C);  true)
                ,(    D  ->   retract(D);  true)
        ;CDTB=2, (    C  ->   retract(C);  true)
                ,(not(D) ->    assert(D);  true)
        ;CDTB=3, (not(C) ->    assert(C);  true)
                ,(not(D) ->    assert(D);  true))),
    (E=sender, F=receiver,
        (SRB=0,  (    E  ->   retract(E);  true)
                ,(    F  ->   retract(F);  true)
        ;SRB=1,  (not(E) ->    assert(E);  true)
                ,(    F  ->   retract(F);  true)
        ;SRB=2,  (    E  ->   retract(E);  true)
                ,(not(F) ->    assert(F);  true)
        ;SRB=3,  (not(E) ->    assert(E);  true)
                ,(not(F) ->    assert(F);  true))),
    (data_sent(S) -> true;((retract(data_sent(_));true),assert(data_sent(S)))),
    (sbase(S)     -> true;((retract(sbase(_))    ;true),assert(sbase(S)))),
    (data_received(R) -> true;((retract(data_received(_)) ; true)
                              ,assert(data_received(R)))),
    (rbase(R)     -> true;((retract(rbase(_)); true),assert(rbase(R)))),
    (G=acceptable_tcreq, H=unacceptable_tcreq,
        (P1=1,   (not(G) ->    assert(G);  true)
                ,(    H  ->   retract(H);  true)
        ;P1=2,   (    G  ->   retract(G);  true)
                ,(not(H) ->    assert(H);  true)
        ;P1=3,   (not(G) ->    assert(G);  true)
                ,(not(H) ->    assert(H);  true))),
    (I=acceptable_cr_tpdu, J=unacceptable_cr_tpdu,
        (P2=1,   (not(I) ->    assert(I);  true)
                ,(    J  ->   retract(J);  true)
        ;P2=2,   (    I  ->   retract(I);  true)
                ,(not(J) ->    assert(J);  true)
        ;P2=3,   (not(I) ->    assert(I);  true)
                ,(not(J) ->    assert(J);  true))),
    (K=acceptable_cc_tpdu, L=unacceptable_cc_tpdu,
        (P3=1,   (not(K) ->    assert(K);  true)
                ,(    L  ->   retract(L);  true)
        ;P3=2,   (    K  ->   retract(K);  true)
                ,(not(L) ->    assert(L);  true)
        ;P3=3,   (not(K) ->    assert(K);  true)
                ,(not(L) ->    assert(L);  true))),
    (M=disconnector, N=not_disconnector,
```

```
        (DNB=1,  (not(M) ->   assert(M); true)
                ,(   N  ->  retract(N); true)
        ;DNB=2,  (   M  ->  retract(M); true)
                ,(not(N) ->   assert(N); true)
        ;DNB=3,  (not(M) ->   assert(M); true)
                ,(not(N) ->   assert(N); true))),
    (O=network_reset, P=not_network_reset,
        (NRB=1,  (not(O) ->   assert(O); true)
                ,(   P  ->  retract(P); true)
        ;NRB=2,  (   O  ->  retract(O); true)
                ,(not(P) ->   assert(P); true)
        ;NRB=3,  (not(O) ->   assert(O); true)
                ,(not(P) ->   assert(P); true))),
    (AA=network_disconnect, BB=not_network_disconnect,
        (NDB=1,  (not(AA) ->  assert(AA); true)
                ,(   BB  -> retract(BB); true)
        ;NDB=2,  (   AA  -> retract(AA); true)
                ,(not(BB) ->  assert(BB); true)
        ;NDB=3,  (not(AA) ->  assert(AA); true)
                ,(not(BB) ->  assert(BB); true))),
    (CC=unspecified, DD=error,
        (UEB=1,  (not(CC) ->   assert(CC); true)
                ,(   DD  ->  retract(DD); true)
        ;UEB=2,  (   CC  ->  retract(CC); true)
                ,(not(DD) ->   assert(DD); true)
        ;UEB=3,  (not(CC) ->   assert(CC); true)
                ,(not(DD) ->   assert(DD); true))),
(A -> write(A), nl;true),
(B -> write(B), nl;true),
(C -> write(C), nl;true),
(D -> write(D), nl;true),
(E -> write(E), nl;true),
(F -> write(F), nl ; true),
(data_sent(S) -> write(data_sent(S)), nl;true),
(data_received(R) -> write(data_received(R)), nl;true),
(G -> write(G), nl;true),
(H -> write(H), nl;true),
(I -> write(I), nl;true),
(J -> write(J), nl;true),
(K -> write(K), nl;true),
(L -> write(L), nl;true),
(M -> write(M), nl;true),
(N -> write(N), nl;true),
(O -> write(O), nl;true),
(P -> write(P), nl;true),
(AA -> write(AA), nl;true),
(BB -> write(BB), nl;true),
(CC -> write(CC), nl;true),
(DD -> write(DD), nl;true), nl.
```

```
pro
C-Prolog version 1.4a
[ Restoring file /.plstartup ]
| ?- [wk81,wk81d].
setpwk8 reconsulted 6236 bytes 2.75 sec.
wk81 consulted 22892 bytes 14.0833 sec.
wk81d consulted 13008 bytes 8.06667 sec.

yes
| ?- tptest(1,1,0,0,0,1,1,1,1,2,2,2).
initiator
connect_only
data_sent(0)
data_received(0)
acceptable_tcreq
acceptable_cr_tpdu
acceptable_cc_tpdu
disconnector
not_network_reset
not_network_disconnect
error

i tcreq((1)(2)(1)(+)(1)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tccon((1)(+)(1)(#))
i tdreq((1)(2)(^))
o ndreq((0)(2)(^))


yes
| ?- tptest(1,1,0,0,0,1,1,1,2,1,1,1).
initiator
connect_only
data_sent(0)
data_received(0)
acceptable_tcreq
acceptable_cr_tpdu
acceptable_cc_tpdu
not_disconnector
network_reset
network_disconnect
unspecified

i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))

i dr((2B)(80)(1)(1)(00)(E0)(2)(1 to 119 bytes internal info))

i dt((2B)(F0)(00)(1 to 125 bytes user data))

i tcreq((1)(2)(1)(+)(1)(&))
```

```
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tccon((1)(+)(*)(#))
i dr((2B)(80)(1)(1)(00)(E0)(?)(1 to 119 bytes internal info))
o ndreq((0)(@)(^))

i tcreq((1)(2)(1)(+)(*)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tccon((1)(+)(*)(#))
i ndind((0)(@)(^))
o tdind((1)(@)(^))

i tcreq((1)(2)(1)(+)(*)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tccon((1)(+)(*)(#))
i nrind((0)(!!))
o tdind((1)(@)(^))

i tcreq((1)(2)(1)(+)(*)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i dr((2B)(80)(1)(1)(00)(E0)(?)(1 to 119 bytes internal info))
o tdind((1)(@)(^))
o ndreq((0)(@)(^))

i tcreq((1)(2)(1)(+)(*)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i dt((2B)(F0)(00)(1 to 125 bytes user data))



yes
| ?- tptest(3,1,0,0,0,2,1,1,1,2,2,2).
initiator
responder
connect_only
data_sent(0)
data_received(0)
unacceptable_tcreq
acceptable_cr_tpdu
acceptable_cc_tpdu
disconnector
not_network_reset
not_network_disconnect
error

i cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tcind((1)(2)(1)(+)(*)(&))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o er((2B)(70)(1)(00)(C1)(?)(1 to 121 bytes from erred TPDU))
```

```
i cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tcind((1)(2)(1)(+)($)(&))
i cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o er((2B)(70)(1)(00)(C1)(?)(1 to 121 bytes from erred TPDU))

i cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tcind((1)(2)(1)(+)($)(&))
i dr((2B)(80)(1)(1)(00)(E0)(?)(1 to 119 bytes internal info))
o er((2B)(70)(1)(00)(C1)(?)(1 to 121 bytes from erred TPDU))

i cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tcind((1)(2)(1)(+)($)(&))
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o er((2B)(70)(1)(00)(C1)(?)(1 to 121 bytes from erred TPDU))

i cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tcind((1)(2)(1)(+)($)(&))
i tcrest((1)(+)($)(#))
o cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i tdreq((1)(@)(^))
o ndreq((0)(@)(^))

i cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tcind((1)(2)(1)(+)($)(&))
i tdreq((1)(@)(^))
o dr((2B)(80)(1)(1)(00)(E0)(?)(1 to 119 bytes internal info))

i tcreq((1)(2)(1)(+)($)(&))
o tdind((1)(@)(^))

i tcreq((1)(2)(1)(-)($)(&))
o tdind((1)(@)(^))


yes
| ?- tptest(1,2,3,2,3,1,1,1,1,2,2,2).
initiator
data_transfer_only
sender
receiver
data_sent(2)
data_received(3)
acceptable_tcreq
acceptable_cr_tpdu
acceptable_ct_tpdu
disconnector
not_network_reset
not_network_disconnect
error
```

```
i tcreq{(1)(2){1}(+)(#)(&)}
o cr{(2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
i cc{(2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
o tccon{(1)(+)(#)(#)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)1}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)2}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)3}
i tdatr{(1)1}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i tdatr{(1)2}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i tdreq{(1)(@)(^)}
o ndreq{(0)(@)(^)}

i tcreq{(1)(2)(1)(+)(#)(&)}
o cr{(2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
i cc{(2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
o tccon{(1)(+)(#)(#)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)1}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)2}
i tdatr{(1)1}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)3}
i tdatr{(1)2}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i tdreq{(1)(@)(^)}
o ndreq{(0)(@)(^)}

i tcreq{(1)(2)(1)(+)(#)(&)}
o cr{(2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
i cc{(2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
o tccon{(1)(+)(#)(#)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)1}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)2}
i tdatr{(1)1}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i tdatr{(1)2}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)3}
i tdreq{(1)(@)(^)}
o ndreq{(0)(@)(^)}
```

```
i tcreq{(1)(2)(1)(+)($)(&)}
o cr{(2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
i cc{(2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
o tccon{(1)(+)($)(#)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)1}
i tdatr{(1)1}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)2}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)3}
i tdatr{(1)2}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i tdreq{(1)(@)(^)}
o ndreq{(0)(@)(^)}

i tcreq{(1)(2)(1)(+)($)(&)}
o cr{(2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
i cc{(2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
o tccon{(1)(+)($)(#)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)1}
i tdatr{(1)1}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)2}
i tdatr{(1)2}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)3}
i tdreq{(1)(@)(^)}
o ndreq{(0)(@)(^)}

i tcreq{(1)(2)(1)(+)($)(&)}
o cr{(2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
i cc{(2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34])}
o tccon{(1)(+)($)(#)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)1}
i tdatr{(1)1}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i tdatr{(1)2}
o dt{(2B)(F0)(00)(1 to 125 bytes user data)}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)2}
i dt{(2B)(F0)(00)(1 to 125 bytes user data)}
o tdati{(1)3}
i tdreq{(1)(@)(^)}
o ndreq{(0)(@)(^)}
```

```
i tcreq((1)(2)(1)(+)(#)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tccon((1)(+)(#)(#))
i tdatr((1)1)
o dt((2B)(F0)(00)(1 to 125 bytes user data))
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)1)
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)2)
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)3)
i tdatr((1)2)
o dt((2B)(F0)(00)(1 to 125 bytes user data))
i tdreq((1)(2)(^))
o ndreq((0)(2)(^))

i tcreq((1)(2)(1)(+)(#)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tccon((1)(+)(#)(#))
i tdatr((1)1)
o dt((2B)(F0)(00)(1 to 125 bytes user data))
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)1)
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)2)
i tdatr((1)2)
o dt((2B)(F0)(00)(1 to 125 bytes user data))
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)3)
i tdreq((1)(2)(^))
o ndreq((0)(2)(^))

i tcreq((1)(2)(1)(+)(#)(&))
o cr((2B)(E)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
i cc((2B)(D)(0)(1)(1)(00)(C1)(03)(1)(C2)(03)(2)(C0)(01)(07)(89)(0C)([34]2.22111e+11[34])(88)(08)([34]55447766[34]))
o tccon((1)(+)(#)(#))
i tdatr((1)1)
o dt((2B)(F0)(00)(1 to 125 bytes user data))
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)1)
i tdatr((1)2)
o dt((2B)(F0)(00)(1 to 125 bytes user data))
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)2)
i dt((2B)(F0)(00)(1 to 125 bytes user data))
o tdati((1)3)
i tdreq((1)(2)(^))
o ndreq((0)(2)(^))
```

```
i tcreq({1}{2}{1}{+}{$}{&})
o cr({2B}{E}{0}{1}{1}{00}{C1}{03}{1}{C2}{03}{2}{C0}{01}{07}{89}{0C}{[34]2.22111e+11[34]}{88}{08}{[34]55447766[34]})
i cc({2B}{D}{0}{1}{1}{00}{C1}{03}{1}{C2}{03}{2}{C0}{01}{07}{89}{0C}{[34]2.22111e+11[34]}{88}{08}{[34]55447766[34]})
o tccon({1}{+}{$}{#})
i tdatr({1}1}
o dt({2B}{F0}{00}{1 to 125 bytes user data})
i tdatr({1}2}
o dt({2B}{F0}{00}{1 to 125 bytes user data})
i dt({2B}{F0}{00}{1 to 125 bytes user data})
o tdati({1}1}
i dt({2B}{F0}{00}{1 to 125 bytes user data})
o tdati({1}2}
i dt({2B}{F0}{00}{1 to 125 bytes user data})
o tdati({1}3}
i tdreq({1}{@}{^})
o ndreq{10}{@}{^})
```

yes
| ?- halt.

[Prolog execution halted]

APPENDIX G : A TRACE AND TRAJECTORY GENERATOR : PROCEDURE "TOTGEN"

```
totgen(SIS) :- initial(SSi_1), echo(SIS), invoker(SSi_1,SIS), alltrace(SSi_1).

invoker(SSi_1,[]).
invoker(SSi_1,[I|RIS]) :- for_each(I,SSi_1,[],SSINi), (SSINi=[] -> error1(SSi_1,I,RIS)
                                                    ; (for_all(SSINi,[],SSi), invoker(SSi,RIS))).

/* Apply each input stimulus IS to each state CS in set SSi_1 and obtain set SSINi */

for_each(_,[],X,X).
for_each(IS,[CS|RESTSTATUS],LS,SSINi) :- get(IS,CS,LSTATUS), append(LS,LSTATUS.LN), for_each(IS,RESTSTATUS,LN,SSINi).

get(I1,[_,[CN1|PS],_,_],LSTATUS) :- I1=[ap1|R], I2=v, CN is CN1+1,
  T=..[ts,[[I1,O1],[I2,O2]],PS,NS], setof([[CN1|PS],[CN|NS],[I1,I2],[O1,O2]],T,LSTATUS), ! .
get(I2,[_,[CN1|PS],_,_],LSTATUS) :- I1=v, I2=[ap2|R], CN is CN1+1,
  T=..[ts,[[I1,O1],[I2,O2]],PS,NS], setof([[CN1|PS],[CN|NS],[I1,I2],[O1,O2]],T,LSTATUS). ! .
get(_,_,[]).

/* Find all states that can be reached by spontaneous transitions from each state in set SSINi and obtain set SSi */

for_all([],X,X).
for_all([CS|RS],LS,SSi) :- store_s(CS,T), ((T=0, for_all(RS,LS,SSi))
                                         ;(get_all(CS,LS1), !, union(LS,LS1,LN), for_all(RS,LN,SSi))), !.

get_all(CS,LS1):- store_p(CS), not(each_path(CS)) , collect(LS1).

each_path([_,[CN1|PS],_,_])  :- get_each([CN1|PS],[PS]), fail.

get_each([CN1|PS],PSF):- ts([[v,O1],[v,O2]],PS,NS), Y=[[CN1|PS],[CN1|NS],[v,v],[O1,O2]],
                         store_s(Y), check(Y,NS,PSF) ,get_each([CN1|NS],[NS|PSF]).

check(Y,S,X):- not(member(S,X)), store_p(Y),!.
check(Y,S,X):- error2(S,X).

initial([[_,[0,idle,idle,[],[]],_,_]]).
fin([0,idle,idle,_,_]).

alltrace([[_,IS,_,_]]) :- allpaths(IS) , traceout.

allpaths(IS) :- fin(FS), at(IS,FS,T), not(path(T)) , assert(path(T)) , fail.
allpaths(_).

traceout:- retract(path(T)) , traceprint(T) , nl , traceout.
traceout.

at([I|X],[J|X],[]) :- not(I=J).
at(X,Y,Z) :- s([X,X1,I,O]), at(X1,Y,Z1), rmv(I,O,V), append(V.Z1,Z).

rmv([v,v],[v,v],[])      :- !.
rmv([v,v],[v,O],[[o,O]]) :- !.
rmv([v,v],[O,v],[[o,O]]) :- !.
rmv([v,I],[v,v],[[i,I]]) :- !.
rmv([I,v],[v,v],[[i,I]]) :- !.
rmv([I,v],[O,v],[[i,I],[o,O]]) :- !.
rmv([v,I],[v,O],[[i,I],[o,O]]) :- !.
rmv([I,v],[v,O],[[i,I],[o,O]]) :- !.
rmv([v,I],[O,v],[[i,I],[o,O]]) :- !.
```

```
append([],X,X).
append([X|R],Y,[X|Z]) :- append(R,Y,Z).

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

union([],X,X).
union([X|R],Y,Z) :- member(X,Y) ,!, union(R,Y,Z).
union([X|R],Y,[X|Z]) :- union(R,Y,Z).

collect([X|R]) :- retract(p(X)), collect(R), !.
collect([]).

store_s(NS,1) :- not(s(NS)), assert(s(NS)), !.
store_s(NS,0).
store_s(NS) :- not(s(NS)), assert(s(NS)), !.
store_s(NS) :- fail.

store_p(NS) :- not(p(NS)), asserta(p(NS)), !.
store_p(NS) :- fail.

traceprint([]).
traceprint([X|T]) :- write(X), nl, write('    '), !, traceprint(T).

lists :- s(L), write(L), nl, fail.
outprint([]).
outprint([X|T]) :- write(X), nl, !, outprint(T).

echo(SIS) :- nl, write('Input Interaction Sequence is:'), nl, outprint(SIS),
             nl, write('Trace or trajectory is(are):'), nl.

error1(IS,I,RIS) :-
    nl, write('Execution Aborted. Unspecified reception encountered.'),nl,
    nl, write('All possible Status that the representation can be'),
    nl, write('after the application of previous Input Interaction.'),nl,
    outprint(IS),

    nl, write('Current Input Interaction Sequence:'), nl,
    write(I), nl,

    nl, write('Rest of Input Interaction Sequence:'). nl,
    outprint(RIS),

    nl, write('Status seen up to this point:'),
    nl, not(lists),

    !, fail.

error2(Y,PSF) :-
       nl, write('Execution Aborted. Infinite Idle loop encountered'),
       nl, write('The path:'), nl, outprint(PSF),
       nl, write('forms a loop with:'), nl, outprint(Y),
       !, fail.
```

```
C-Prolog version 1.4a
[ Restoring file /.plstartup ]
| ?- [ts, totgen, run_totgen].
setparam_tsp consulted 992 bytes 0.683336 sec.
ts consulted 9208 bytes 5.45 sec.
totgen consulted 7904 bytes 4.6 sec.
run_totgen consulted 996 bytes 0.616667 sec.

yes
| ?- run_totgen(1).
Input Interaction Sequence is:
[ap1,tcreq,p]
[ap2,tareq,p]
[ap2,tdatr,1]
[ap1,tdreq,user]

Trace or trajectory is(are):
[i,[ap1,tcreq,p]]
    [o,[ap2,tcind,p]]
    [i,[ap2,tareq,p]]
    [o,[ap1,taind,p]]
    [i,[ap2,tdatr,1]]
    [o,[ap1,tdati,1]]
    [i,[ap1,tdreq,user]]
    [o,[ap2,tdind,user]]

[i,[ap1,tcreq,p]]
    [o,[ap2,tcind,p]]
    [i,[ap2,tareq,p]]
    [o,[ap1,taind,p]]
    [i,[ap2,tdatr,1]]
    [i,[ap1,tdreq,user]]
    [o,[ap2,tdind,user]]

[i,[ap1,tcreq,p]]
    [o,[ap2,tcind,p]]
    [i,[ap2,tareq,p]]
    [i,[ap2,tdatr,1]]
    [o,[ap1,taind,p]]
    [o,[ap1,tdati,1]]
    [i,[ap1,tdreq,user]]
    [o,[ap2,tdind,user]]

[i,[ap1,tcreq,p]]
    [o,[ap2,tcind,p]]
    [i,[ap2,tareq,p]]
    [i,[ap2,tdatr,1]]
    [o,[ap1,taind,p]]
    [i,[ap1,tdreq,user]]
    [o,[ap2,tdind,user]]


yes
| ?- run_totgen(2).
Input Interaction Sequence is:
[ap1,tcreq,p]
[ap1,tdreq,user]
```

```
Trace or trajectory is(are):

Execution Aborted. Unspecified reception encountered.

All possible Status that the representation can be
after the application of previous Input Interaction.
[[1,wait_accept,idle,[],[]],[1,wait_accept,wait_accept,[],[]],[v,v],[v,[ap2,tcind,p]]]
[[0,idle,idle,[],[]],[1,wait_accept,idle,[],[]],[[ap1,tcreq,p],v],[v,v]]

Current Input Interaction Sequence:
[ap1,tdreq,user]

Rest of Input Interaction Sequence:

Status seen up to this point:
[[0,idle,idle,[],[]],[1,wait_accept,idle,[],[]],[[ap1,tcreq,p],v],[v,v]]
[[1,wait_accept,idle,[],[]],[1,wait_accept,wait_accept,[],[]],[v,v],[v,[ap2,tcind,p]]]

no
| ?- run_totgen(3).
Input Interaction Sequence is:
[ap1,tcreq,p]
[ap2,tdreq,user]

Trace or trajectory is(are):
[i,[ap1,tcreq,p]]
    [o,[ap2,tcind,p]]
    [i,[ap2,tdreq,user]]
    [o,[ap1,tdind,user]]


yes
| ?- halt.
[ Prolog execution halted ]



run_totgen(Z) :- clear, input_interaction_sequence(Z,X) ,totgen(X).
clear :- retract(s(_)), fail.
clear.
input_interaction_sequence(1,[
[ap1,tcreq,p],
[ap2,tareq,p],
[ap2,tdatr,1],
[ap1,tdreq,user]
]).
input_interaction_sequence(2,[
[ap1,tcreq,p],
[ap1,tdreq,user]
]).
input_interaction_sequence(3,[
[ap1,tcreq,p],
[ap2,tdreq,user]
]).
```

APPENDIX H : A TRACE AND TRAJECTORY VALIDATOR : PROCEDURE "TOTVAL".

```
totval(IOS) :- echo(IOS), initial(SPS), invoker(IOS,SPS).

invoker([],SPS) :- checker([],SPS).
invoker([[i,ap1,I,P]],SPS) :- checker([[[[ap1,I,P],v],[v,v]]],SPS,SNS), invoker([],SNS).
invoker([[o,ap1,O,P]],SPS) :- checker([[[v,[ap1,O,P]],[v,v]],SPS,SNS)  , invoker([],SNS).
invoker([[i,ap2,I,P]],SPS) :- checker([[[v,v],[[ap2,I,P],v]],SPS,SNS)  , invoker([],SNS).
invoker([[o,ap2,O,P]],SPS) :- checker([[[v,v],[v,[ap2,O,P]]],SPS,SNS)  , invoker([],SNS).
invoker([[S1,C1,I1,P1]|R1],SPS) :- R1=[Y|R2], Y=[S2,C2,I2,P2],
    ((S1=i, S2=i,
       ((C1=ap1, R=R1, O=[[[C1,I1,P1],v],[v,v]])
       ;(C1=ap2, R=R1, O=[[v,v],[[C1,I1,P1],v]])
     ) )
    ;(S1=i, S2=o,
       ((C1=ap1, C2=ap1,
          (R=R1, O=[[[C1,I1,P1],v],[v,v]]
          ;R=R2, O=[[[C1,I1,P1],[C2,I2,P2]],[v,v]]))
       ;(C1=ap2, C2=ap2,
          (R=R1, O=[[v,v],[[C1,I1,P1],v]]
          ;R=R2, O=[[v,v],[[C1,I1,P1],[C2,I2,P2]]]))
       ;(C1=ap1, C2=ap2,
          (R=R1, O=[[[C1,I1,P1],v],[v,v]]
          ;R=R2, O=[[[C1,I1,P1],v],[v,[C2,I2,P2]]]))
       ;(C1=ap2, C2=ap1,
          (R=R1, O=[[v,v],[[C1,I1,P1],v]]
          ;R=R2, O=[[v,[C2,I2,P2]],[[C1,I1,P1],v]]))
     ) )
    ;(S1=o, C1=ap1, R=R1, O=[[v,[C1,I1,P1]],[v,v]])
    ;(S1=o, C1=ap2, R=R1, O=[[v,v],[v,[C1,I1,P1]]])
    ) ,checker(O,SPS,SNS), invoker(R,SNS).

checker([],SNS) :- final(FS), member(FS,SNS), nl, write('*** VALID ***'),!.
checker(IO,SPS,SNS) :- for_each(IO,SPS,[],SSNS), (SSNS=[] -> error1(IO) ; for_all(SSNS,[],SNS)).

/* Apply each invocation IO to each state CS in set SPS and obtain set SSNS */

for_each(_,[],X,X).
for_each(IO,[CS|RESTSTATUS],LS,SSNS) :- get(IO,CS,LSTATUS), append(LS,LSTATUS,LN),
                                        for_each(IO,RESTSTATUS,LN,SSNS).

get(IO,PS,LSTATUS) :- setof(NS, ts(IO,PS,NS), LSTATUS), !.
get(_,_,[]).

/* Find all states that can be reached by internal transitions from each state in set SSNS and obtain set SNS */

for_all([],X,X).
for_all([CS|RS],LS,SNS) :- get_all(CS,LS1), !, union(LS,LS1,LN), for_all(RS,LN,SNS),!.

get_all(CS,LS1)  :- store_p(CS), not(each_path(CS)), collect(LS1).

each_path(CS)    :- get_each(CS,[CS]), fail.

get_each(PS,PSF) :- ts([[v,v],[v,v]],PS,NS) ,store_s(NS) ,check(NS,PSF), get_each(NS,[NS|PSF]).
get_each(_,_).
```

```
check(S,X) :- not(member(S,X)), store_p(S),!.
check(S,X) :- error2(S,X).

initial([ [idle,idle,[],[]] ]).
final([idle,idle,_,_]).

append([],X,X).
append([X|R],Y,[X|Z]) :- append(R,Y,Z).

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

union([],X,X).
union([X|R],Y,Z) :- member(X,Y) ,!, union(R,Y,Z).
union([X|R],Y,[X|Z]) :- union(R,Y,Z).

collect([X|R]) :- retract(p(X)), collect(R), !.
collect([]).

store_s(NS) :- not(s(NS)) , assert(s(NS)) ,!.
store_s(NS) :- fail.

store_p(NS) :- not(p(NS)) , asserta(p(NS)) ,!.
store_p(NS) :- fail.

echo(IOS)  :- nl, write('Input Output Interaction Sequence'), nl, outprint(IOS).

outprint([]).
outprint([H|I]) :- write(H), nl, !, outprint(I).

error1(IO) :-
       ((is_I(IO);is_O(IO)) -> nl, write('Execution Aborted')),
       (is_I(IO) -> nl, write('Unspecified Reception Encountered');true),
       (is_O(IO) -> nl, write('Unexpected Output Encountered: '), write(IO);true),
       ((is_I(IO);is_O(IO)) -> nl, write('*** NOT VALID ***'), nl), !, fail.

error2(Y,PSF) :-
       nl, write('Infinite idle loop encountered'),
       nl, write('The path:'), nl, outprint(PSF),
       nl, write('forms a loop with:'), nl, outprint(Y),!,fail.

is_I([[I,v],[v,v]]).
is_I([[v,v],[I,v]]).
is_O([[v,O],[v,v]]).
is_O([[v,v],[v,O]]).
```

```
run_totval(X) :- clear, interaction_sequence(X,S), totval(S).
clear :- retract(s(_)), retract(p(_)), fail.
clear.
interaction_sequence(1,[
[i,ap1,tcreq,p],
[o,ap2,tcind,p],
[i,ap2,tareq,p],
[i,ap2,tdatr,1],
[i,ap2,tdatr,2],
[o,ap1,taind,p],
[i,ap1,tdatr,1],
[i,ap1,tdatr,2],
[o,ap1,tdati,1],
[i,ap1,tdatr,3],
[o,ap1,tdati,2],
[o,ap2,tdati,1],
[o,ap2,tdati,3]
]).
interaction_sequence(2,[
[i,ap1,tcreq,p],
[o,ap2,tcind,p],
[i,ap2,tareq,p],
[i,ap2,tdatr,1],
[i,ap2,tdatr,2],
[o,ap1,taind,p],
[i,ap1,tdatr,1],
[i,ap1,tdatr,2],
[o,ap1,tdati,1],
[i,ap1,tdatr,3],
[o,ap1,tdati,2],
[o,ap2,tdati,1],
[i,ap1,tdreq,user],
[o,ap2,tdind,user]
]).
```

```
C-Prolog version 1.4a
[ Restoring file /.plstartup ]
| ?- [ts, totval, run_totval].
setparam_tsp consulted 992 bytes 0.416669 sec.
ts consulted 9208 bytes 3.28333 sec.
totval consulted 8528 bytes 2.78333 sec.
run_totval consulted 1948 bytes 0.550004 sec.

yes
| ?- run_totval(1).
Input Output Interaction Sequence
[i,ap1,tcreq,p]
[o,ap2,tcind,p]
[i,ap2,tareq,p]
[i,ap2,tdatr,1]
[i,ap2,tdatr,2]
[o,ap1,taind,p]
[i,ap1,tdatr,1]
[i,ap1,tdatr,2]
[o,ap1,tdati,1]
[i,ap1,tdatr,3]
[o,ap1,tdati,2]
[o,ap2,tdati,1]
[o,ap2,tdati,3]

Execution Aborted
Unexpected Output Encountered: [[v,v],[v,[ap2,tdati,3]]]
*** NOT VALID ***

no
| ?- run_totval(2).
Input Output Interaction Sequence
[i,ap1,tcreq,p]
[o,ap2,tcind,p]
[i,ap2,tareq,p]
[i,ap2,tdatr,1]
[i,ap2,tdatr,2]
[o,ap1,taind,p]
[i,ap1,tdatr,1]
[i,ap1,tdatr,2]
[o,ap1,tdati,1]
[i,ap1,tdatr,3]
[o,ap1,tdati,2]
[o,ap2,tdati,1]
[i,ap1,tdreq,user]
[o,ap2,tdind,user]

*** VALID ***
yes
| ?- halt.
[ Prolog execution halted ]
```