DEVELOPMENT OF TESTS FOR THE ATM SIGNALING PROTOCOL

by
Dionis Hristov

School of Computer Science
McGill University, Montreal

April, 1998

A Thesis is submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Alexandre Petrenko and Guang Gao, supervisors

0-612-44184-9

Abstract

The work presented here addresses the problem of the development of a conformance test suite and an interoperability test suite for the ATM Signaling protocol based on the finite state machine model (FSM).

The experimental tools developed at the University of Montreal are used to partially automate the process of test development. Tools use the underlying FSM model. Development of a conformance test suite starts from a complete SDL specification of the Signaling protocol. The specification is developed from the standard document prior to test development. Using the tool chain, the FSM model from the SDL specification is extracted, and, based on an FSM test generation method, a test suite from the FSM model is produced. Test cases for the data part of the protocol are derived directly from the text in the standard,. The obtained test suite is validated in the SDL environment against the SDL specification.

The interoperability testing assumes a system of two implementations of the Signaling protocol connected via a network. Tests are represented in terms of protocol services. Test development process is also based on the FSM testing. The behavior of the complete system is represented by a global FSM, and a test suite is developed form the obtained FSM.

Résume

Le travail présenté dans ce document montre les problèmes liés au développement des suites de tests de conformité et des suites de tests d'intéropérabilité pour le protocole de signalisation ATM basé sur le modèle FSM (machines à états finis).

Des outils expérimentales développés à l'université de Montréal ont été utilisés pour automatiser partiellement le processus de développement de test. Ces outils utilisent le modèle FSM. Une suite de tests de conformité est développée à partir d'une spécification complète SDL du protocole de signalisation. La spécification st développé à partir du document standard avant le développement du test. En utilisant la chaîne d'outils, le modèle FSM est extrait à partir de la spécification SDL, et basé sur une méthode de génération de test FSM, une suite de test est produite à partir du modèle FSM. Les cas de tests de la partie données du protocole est dérivée directement du texte dans le standard. La suite de test ainsi obtenue est validée à l'aide d'un environnement SDL contre la spécification SDL.

Le test d'intéropérabilité suppose qu'un système possède deux implantations du protocole de signalisation connectés via un réseau. Les tests sont représentés comme des services de protocoles. Le processus de développement des tests est également basé sur une méthode de test FSM. Le comportement du système complet est représenté par un FSM global, et une suite de tests est développée à partir du FSM obtenu.

Dedicated to my parents who inspired my urge for knowledge

Acknowledgment

I would like to express my gratitude to my supervisors Alexandre Petrenko and Guang Gao. I am deeply grateful to Alexandre Petrenko, for his expertise in the area and his competent advice during the research and the preparation of the text. His suggestions and, especially, criticisms have been of invaluable help. Without his commitment to supervision, this work would have required longer time to complete.

I am grateful to Guang Gao for his brilliant lectures in computer architecture, from which I gained technical knowledge and deep understanding of the professional world.

My special thanks go to Greogor v. Bochmann who gave me an opportunity to join his EPAC group, and do my research project under his team lead. His original ideas, fruitful discussions, and moral support contributed to this thesis. Also, I thank other members of the EPAC team, Roxana Marcocci, Jean Zaidan, Omar Bellal and Daniel Oumiet, who have discussed this work during different stages.

My graduate studies would be much less enjoyable and inspiring without my friends from McGill. In spite of originating from different parts of the world, we share the same ideas and attitudes and we all struggle for the same goal: a meaningful life. Namely, I thank Marcia, Shereen, Madalina, Taha, Darek, Bora and all my dear friends from McGill for the nice atmosphere that we shared together. Also, many thanks to Stefan for the long, inspiring discussions; Saso, Snezana, Gligor and Berta for the immeasurable help and friendship. Also, special thanks to my parents, sister, brother-in-law and Helena for their unwavering support during my graduate studies.

This work was supported by a grant from consortium EPAC (EICON Technologies, Positron Fiber Systems, AIKS and CRIM).

CONTENT

Chapter 1
Introduction

Computer networks offer a number of new, previously unknown services to the residential and business users such as electronic mail, file transfer, video conferencing, electronic commerce, etc. Furthermore, with the appearance of the B-ISDN, which defines a variety of possible broadband services over the integrated network for voice and data transfer, the market for telecommunication products has been growing aggressively, and computer networks have become an important infrastructure for society, bringing immediate commercial potentials.

Recognizing the technology and market potentials, computer and telecommunication vendors have developed and deployed a myriad of different communication solutions and standards. However, it was recognized early (1978) that standards for networks of heterogeneous systems were urgently required in order to ensure interworking of different products. In 1983, the Open System Interconnection (OSI) Reference Model became the international standard and from then on was used as a framework for the development of the OSI communication standards. The idea behind the OSI is the need for global networking solutions to communicate requirements in a multi-vendor environment[Davi89]. To achieve this, the core element is to have the development of the

OSI standards and their acceptance as a relevant specification for products by the vendors. However, today, it is widely agreed that to achieve the OSI goals, conformance testing of the product to the standard must be an integral part of the OSI efforts.

Conformance testing is a process of assessment whether or not the implementation conforms to the requirements of the standard. There are two reasons for the need for conformance testing: in spite of the best efforts, standards may be unclear in stating the requirements and lead to different implementations; and, since implementations that conform to standards are more likely to interoperate, conformance testing increases the confidence in building complex, multi-vendor networks. A majority of the work in testing networks is related to the protocol testing, as protocols are an essential part of the communication systems.

Protocols are sets of rules that govern the interaction of concurrent processes in distributed communication systems, and they are part of the OSI standards. Similar to any other software product, protocol development goes through three phases (also called the protocol life cycle): protocol specification, protocol implementation and protocol testing. A protocol specification is typically broken into its control and data portion, where the control portion is usually modeled by a finite state machine. Most of the formal work on conformance testing addresses the problem of testing the control portion and is based on the well-defined problem of the FSM conformance testing [LeYa96]. However, in practice a protocol specification includes variables and operations based on the variable values; "pure" FSMs are not powerful enough in a succinct way to model the protocols. Extended finite state machines (EFSMs), which are finite state machines with variables, are used to specify protocols. An EFSM with finite variable domains can be looked at as a "compact" representation of an FSM. Thus, testing of the EFSM can be reduced to testing of the "pure" FSM by expanding the EFSM in an equivalent FSM. However, for many protocols, this would lead to the well known problem of "state explosion".

The interest in FSM and EFSM conformance testing grows with the use of Formal Description Techniques (FDT) for protocol development and standardization of formal languages such as SDL. Since the FDTs specify protocols in a precise and machine processable form and model the protocol with a precise mathematical model, communicating EFSMs, they pave the way to automated implementation generation and conformance testing. A number of tools have been developed in recent years to aid the protocol specification and testing.

The topic of this work is test development from the SDL specification of the ATM Signaling protocol. The experimental tools developed at the University of Montreal are used to partially automate the process of test development. Most tools use the underlying FSM model, that is, the behavior of the system to be tested is expressed in terms of states and inputs, and the outputs and state transitions produced by the arrival of a given input in a given state.

The goal of this work is twofold:
(1) to develop a conformance test suite and an interoperability test suite for the ATM Signaling protocol using an experimental tool chain developed at UofM;
(2) to evaluate the effectiveness of the experimental tool chain for test derivation on a real-life protocol and to identify problems and possible areas for its improvement.

The ATM Signaling protocol is a part of the higher layer protocols in the Control plane of the ATM stack. It is responsible for establishing and maintaining a switched connection between the ATM users. The standard specifying the Signaling protocol is ATM Forum UNI 3.1 document [UNI3.1]. For parts of the protocol we also make use of Q2931[Q2931] document. The protocol is specified in the UNI 3.1 document with 9 states and 8 basic messages. However, the behavior of the protocol is described using plain English text.

Development of a conformance test suite starts from a complete SDL specification of the Signaling protocol. Since this work is a part of a bigger EPAC project in which the complete protocol engineering of the ATM Signaling protocol is done, the specification is developed from the standard document prior to the test development. Using the tool chain, the FSM model from the SDL specification is extracted, and, based on an FSM test generation method, a test suite from the FSM model is produced. Test cases for the data part of the protocol are derived directly from the text in the standard. The obtained test suite is validated in the SDL environment against the SDL specification.

As a part of the requirements of the project, an interoperability test suite is also developed. The interoperability testing assumes a system of two implementations of the Signaling protocol connected via a network. Tests are represented in terms of protocol services. Test development process is also based on the FSM testing. The behavior of the complete system is represented by a global FSM, and a test suite is developed from the obtained FSM.

This document is organized in seven chapters. Chapter 2 discusses the protocol life cycle and in particular, conformance testing and interoperability testing from the OSI standards point of view. Chapter 3 presents the formal work related to finite state machine testing and the impact of FDTs on the automation of the test development. Also, the experimental tool chain of UofM is explained in this chapter. ATM Signaling protocol and its SDL specification are described in Chapter 4. In Chapter 5, the experience in developing and validating the conformance test suite is shared. Also, comments related to the evaluation of the tool chain are presented. Chapter 6 explains the development of the interoperablity test suite. Evaluation of the tool chain in terms of advantages and difficulties encountered using it are given in Chapter 7.

Chapter 2

Protocol life cycle

2.1 OSI Reference Model

A protocol is a set of rules governing the exchange of messages between entities in a computer communication network. Protocols are key components of distributed systems and are generally complex. They must work correctly for a system to provide expected services [SiCh89].

The Reference Model for Open System Interconnection (OSI), now an International Standard published by the International Organization for Standardization (ISO) [ISO7498] and the ITU-T (formerly, CCITT) [CCITT83], has evolved over the past years as an architectural framework for the development of communication services and protocol standards. These "OSI standards" are intended to facilitate the interconnection of computer systems considered to be "open" by virtue of their mutual adherence to the standards.

The OSI Reference Model (OSI-RM) consists of seven hierarchical layers, each built on its predecessor layer, as shown on Figure 2.1. The number of layers implemented in a given system may vary depending on the function that the system is intended to perform. At the layer N, the purpose of an (N) - entity is to provide a set of services (called (N) - services) to its upper-layer entity (that is, (N+1) - entity). Similarly, an (N-1) - entity provides (N-1) - services of an (N) - entity. These two (N) - entities are peer entities, operating in the two end systems that communicate.

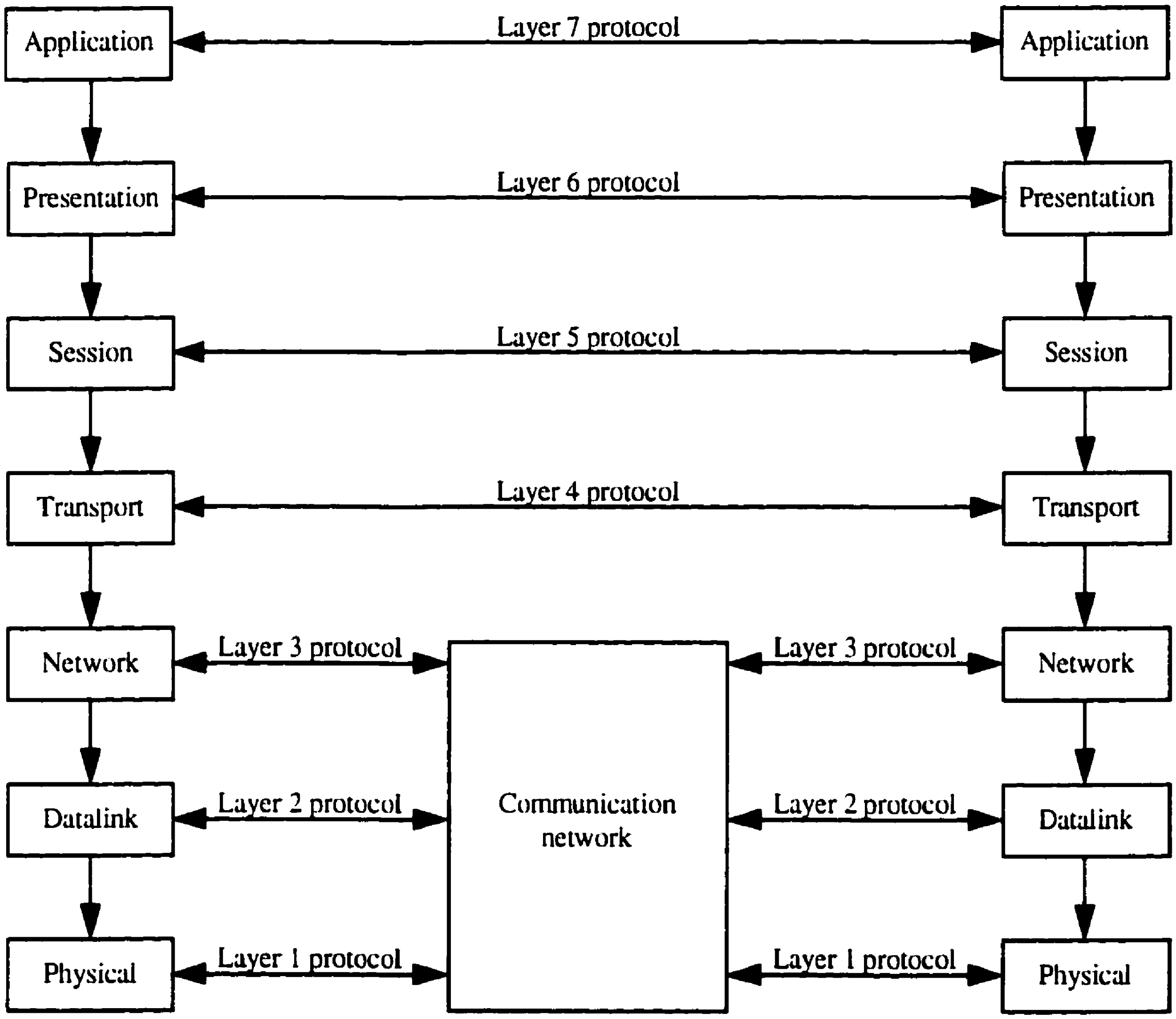

Figure 2.1: Layered structure of the OSI - RM

An (N)-entity provides (N)-services for its user entity by using and enhancing the services of the adjacent lower layer entity. In this model, an (N+1)-entity receives services from an (N)-entity, but the implementation details and the services of the (N-1)-entity are completely isolated from the (N+1)-entity. For an (N+1)-entity, all lower-layer entities are referred to as (N)-service providers.

An (N)-function is a part of the activity of an (N)-entity. Flow control, sequencing, data transformation are all examples of (N)-functions. Cooperation among (N)-entities is governed by one or more (N)-protocols. An (N)-protocol is the set of rules and formats which govern the communication between (N)-entities performing the (N)-functions in different end systems. Peer (N)-entities communicate with each other by exchanging (N)-protocol data units ((N)-PDUs).

Figure 2.2 illustrates the interactions of an (N)-entity with its adjacent upper and lower entities that are defined at (N)- and (N-1)-service access points, respectively. An (N)-service access point (abbreviated as (N)-SAP) is a logical interface between (N)- and (N+1)- entities by which service requests and responses are made.

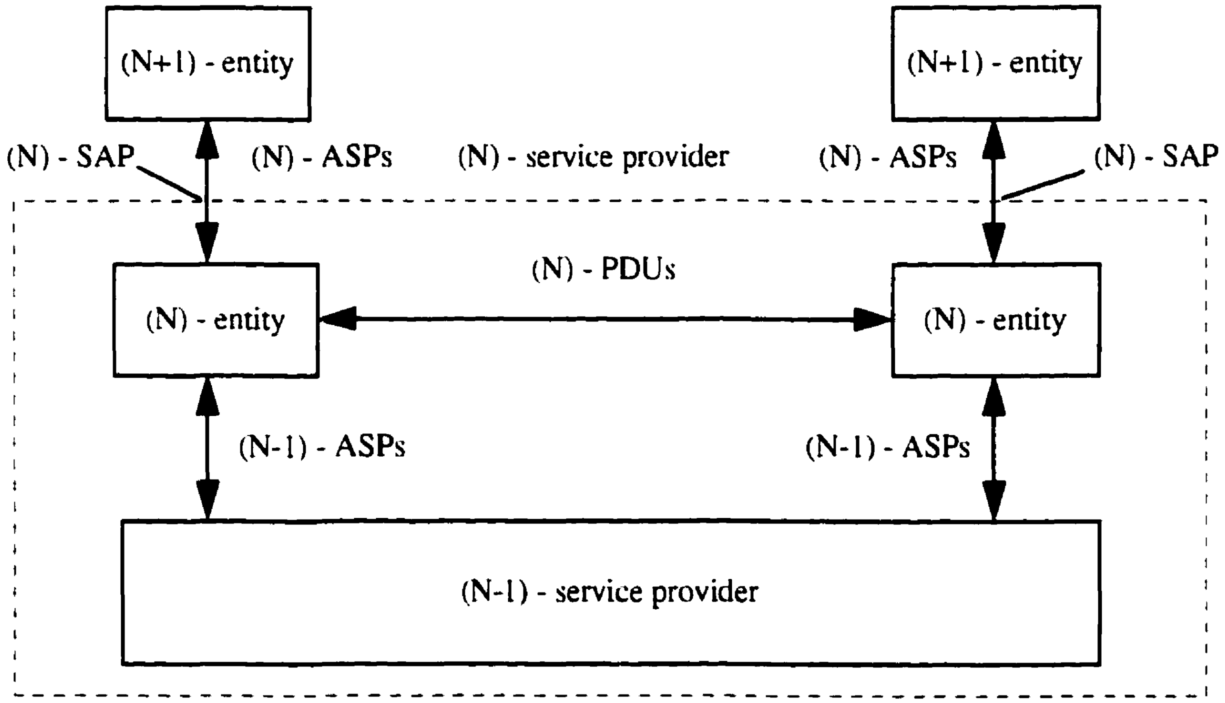

Figure 2.2: Configuration of the (N) - service provider

The (N)-services are offered to the (N+1)-entities at the (N)-SAPs. An (N+1)-entity communicates with an (N)-entity in the same system through an (N)-SAP. An (N)-SAP can be served by only one (N)-entity and used by only one (N+1)-entity, but one (N)-entity can serve several (N)-SAPs and one (N+1)-entity can use several (N)-SAPs (Figure 2.3).

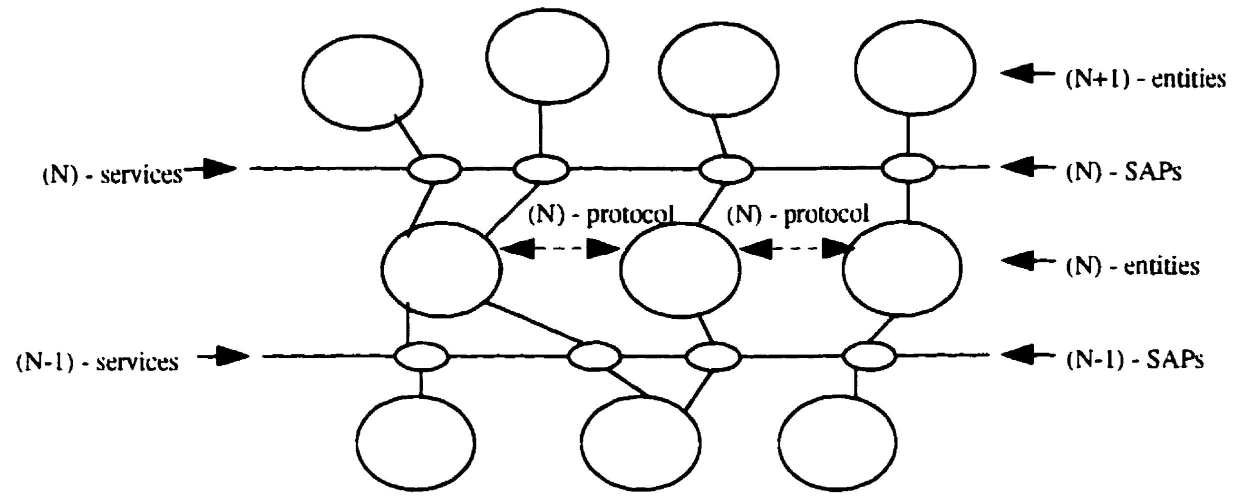

Figure 2.3: Entities, service access points and protocols

The OSI-RM spans from the physical media (layer 1) up to application protocols (layer 7). It accommodates a broad spectrum of media at the bottom (point-to-point such as twisted pair, fiber, and coaxial cable) switched circuits and broadcast media (such as radio, satellite, and coaxial buses). These may be integrated into local, metropolitan, and wide area networks (LAN, MAN, and WAN) by Datalink and Network Layer protocols. Each layer masks the characteristics of lower layer components and leads to composite services employed by successively higher layers, which are largely independent of the characteristics of lower layer components.

However, the OSI-RM does not define details of protocols or services. Details of services and protocols are defined in separate standards for each layer. The OSI-RM does not define programming interfaces to layer services nor does it address issues local to an implementation of a protocol in a particular environment. At present, the OSI-RM does not address broadcast or multipeer services and protocols. However, OSI -RM is not a static entity. Since its adoption as a standard in 1983, the OSI-RM has been the subject of questions raised by working groups within ISO and CCITT, with a set of interpretations and answers published as "final answers to questions." Based on a five-year maintenance cycle established by ISO for its standards, the OSI-RM is due to be reissued with revisions and extensions.

The distinction between services and protocols are fundamental to the OSI-RM. At an (N)-SAP, an (N)-entity interacts with an (N+1)-entity by exchanging (N)-abstract service primitives. At the conceptual level, exchange of service primitives is an indivisible event. As shown iFigure 2.4, service primitives are categorized as follows:

- Request: a primitive issued by an (N)-service user to invoke a particular function (for example, connection establishment, data transfer, connection termination).
- Indication: A primitive issued by the service provider ((N)-layer protocol entity) indicating that the peer (N)-service user has requested a specific service or function (for example, connection establishment, data delivery, connection termination).
- Response: A primitive issued by an (N)-service user confirming that a service function previously requested by its peer (N)-service user and signaled by a service indication may proceed to completion (for example, connection establishment).
- Confirm: A service primitive issued by a service provider to (N)-layer protocol entity indicating that services previously requested by the (N)-service user have been completed or established (for example, connection establishment).

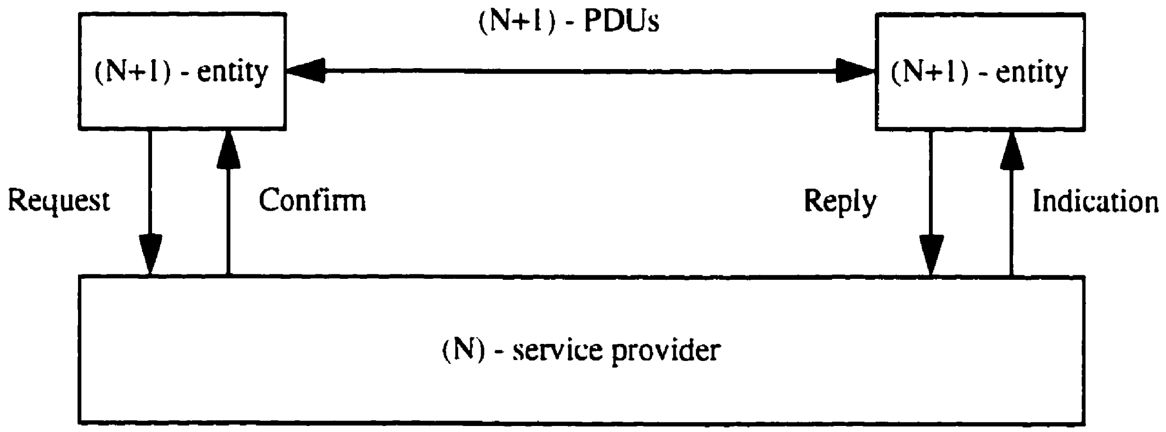

Figure 2.4: Service primitives

Typically, these categories are prefixed by a name denoting the functional aspects of the service (for example, connect request, connect response, connect confirm, data request, data indication). During the service dialog, the underlying protocol entities are responsible for exchange of PDUs to realize the service requests. Conceptually, protocol entities are driven by service requests and responses from adjacent upper layer users and by data indications from the (N-1)-service provider. As discussed earlier in this section, the (N)-layer protocol entities respond to these stimuli by exchanging PDUs via the (N-1)-service provider to communicate with each other and realize the

(N)-service. They deliver information exchanged between peer (N)- entities to an adjacent upper layer via indication and confirm service primitives. Assumptions and rules governing the dynamic exchanges of messages between peer protocol entities are dependent on the functions assigned to the layer. Service primitives and service access points need not be realized, as such, in an implementation of a protocol; i.e., they are conceptual entities which are optional in a product. Thus, their realization is outside the scope of standardization.

A common service offered by all layers consists of providing associations between peer SAPs which can be used in particular to transfer data (as well as for other purposes such as to synchronize the served entities participating in the association). More precisely, the (N) - layer offers (N) - connections between (N) - SAPs as part of the (N) - services. The most usual type of connection is the point-to-point connection, but there are also multi-endpoint connections which correspond to multiple associations between entities. The end of an (N)-connection at an (N)-SAP is called an (N)-connection endpoint or (N)-CEP, for short. Several connections may coexist between the same pair (or n-tuple) of SAPs. If SAP only services a single (N+1)-connection at one time, then the SAP may serve as the connection endpoint identifier. However, if the (N)-layer protocol provides multiplexing of data from more than one (N+1)-entity for each (N)-layer connection, then a connection endpoint identifier is associated with each (N+1)-connection, and the (N)-layer protocol must be able to bind each (N+1)-connection to a connection endpoint identifier within the (N)-SAP. In other words, logically, a SAP is an addressable unit, and connection endpoint identifiers allow distinction between PDUs destined for different users of the same SAP.

In summary, the OSI-RM has defined a conceptual framework for discussion, design, specification, implementation, and testing of protocols. We use the terminology and concepts of the OSI-RM for specifying and testing the ATM Signaling protocol.

2.2 Protocol life cycle

Activities related to a protocol development are basically partitioned into three groups: specification, implementation and testing.

2.2.1 Protocol specification

The following two activities may be distinguished within the specification development phase [Boch87-1]:

(1) Specification creation: These are the activities of creating and updating the representation of the protocol specifications, which may be in the form of text, graphic representations and/or formal language code;

(2) Protocol verification: The goal of protocol verification is to ensure that a protocol specification is free of design errors before it is implemented. As mentioned in Section 2.1, a complete specification of a protocol layer consists of a protocol service specification and a specification of protocol entities in that layer. The protocol service specification forms the standard against which a protocol is verified. The input and output behaviors of a protocol visible at a SAP consist of sequences of events occurring at this SAP. These sequences can be compared to those generated from the protocol entity-to-entity interactions to verify that a protocol is consistent with its service specification. In general, protocol verification involves proving certain general protocol properties such as completeness, deadlock freeness, termination, cyclic behavior, and boundness. Completeness means that a protocol accepts all possible inputs in each system state. Deadlock freeness means that a protocol never gets into a system state where no more transitions are possible and it stays in that state indefinitely. For a terminating protocol, termination means that a protocol always reaches the final state when it started from the initial state. A non-terminating protocol should have the property of a cyclic behavior, which means that the protocol can progress indefinitely. Boundness refers to the property that the total number of messages in the channel is always less than some fixed number [SiCh87].

2.2.2 Protocol implementation

A protocol specification can be very abstract or quite detailed with respect to a possible implementation of the protocol in a high level language. An abstract specification generally hides many implementation level details of a system behavior. A specification of the protocol usually consist of two parts, namely machine-independent and machine-dependent parts. The machine-independent part of the specification includes the rules according to which protocol entities interact in response to incoming communication events and other changes in a system state. This part can be fully specified for a protocol and implemented as a high level programming language source code. This can either be done manually or can be automated if the gap between a specification description and actual implementation is not that large. The machine-dependent part of the implementation includes such things as the mechanisms for causing and detecting events, the means of communication between adjacent protocol layers, and memory management. This part cannot be completely specified for a protocol because it is tied to the machine architecture and the host operating system. However, once the code is written for this part of a protocol, much of it can be reused for another protocol running on the same computer system [SiCh87].

2.2.3 Protocol testing

After the implementation phase of a protocol, a given protocol implementation, usually called an "implementation under test" (IUT), is checked against the protocol specification in order to certify that the IUT conforms to the protocol specification, which acts as a reference. In the conformance testing [Boch87-1], the IUT is stimulated by test inputs which are generated by one or several test modules. The output generated by the IUT in response to the test input must be observed and compared with the protocol specification in order to determine whether the observed output is a possible one according to the specification.

Since the topic of this work is the development of tests, we will discuss the concepts and framework of the conformance testing in more detail in the following sections. The conformance proof is a necessary but not sufficient condition to guarantee interworking capability of a protocol implementation [SiCh87]. Therefore, interoperability testing is often performed to determine whether two conforming implementations can interoperate. Interoperability testing is discussed in Section 2.4.

2.2.4 Formal description techniques (FDT) and protocol development

2.2.4.1 Impact of the FDT on the protocol development

The introduction of a new communication protocol, for proprietary systems, as well as for OSI, requires careful analysis of the proposed protocols and much effort in the development and testing of the new protocol implementations. In this context, the use of the formal methods for the specification of communication protocols and services has received much attention; such methods allow a more systematic approach to protocol validation, implementation and testing, as compared to the traditional use of the protocol specification given in a natural language [Boch87-1]. FDTs have been developed with essentially two objectives [BoPe97]:

(1) encouraging the development of precise specifications which do not allow any ambiguities. A major advantage of using FDTs for protocol specification is that the resulting specifications can be rigorously analyzed for completeness and consistency;

(2) allowing the partial automation of the protocol development activities:

- Verification and evaluation of the protocol specification: FDTs form the basis for formal verification of the protocol specification. Verification techniques can generally be classified into two approaches, synthesis and analysis. The synthesis approach is used when a protocol is constructed from its informal specification by the application of certain design rules. The analysis approach is used when the specification of a communication protocol is given and we analyze the protocol to prove that it satisfies certain desirable properties. State space exploration (also called reachability analysis) and program proving are two common techniques in this approach [SiCh89].
- Implementation process: Semi-automatic code generation for a protocol specification can provide an increased assurance for correct protocol implementations. A suitably defined transformation technique can translate a large portion of formal specification (i.e. the machine-independent part) of a protocol into some high - level language code. Semi - automatic generation has been an active research area [SiB190, PoSm82, Nash83], look for some more recent examples
- Test development process: Formal methods exist for generating protocol test cases directly from a formal specification of a protocol. If a formal test generation technique is

automated, tests with a certain fault detection capability can be generated economically from a protocol specification. In Chapter 3, we will focus on the problem of developing test cases automatically from the formal specification, and we will provide a detailed description of the automated test selection process.

Realizing the advantages of using FDT for protocol development, organizations such as ISO and IUT-T have developed standards for three formal specification languages: SDL [CCITT88], ESTELLE [IS9074], and LOTOS [IS8807]. Also, a number of commercially available tools (predominantly for SDL) have been developed, supporting the protocol development process using FDTs. In the following section we describe SDL because it is used in this project to specify the Signaling protocol. An overview of the SDL tools used is also given.

2.2.4.2 Specification and Description Language (SDL)

SDL is a standard language for the specification and description of systems. The most recent standard document is SDL - 92 [CCITT92], which is an extension of SDL - 88 in the area of object orientation.

SDL has been developed for use in telecommunication systems, including data communications, but actually it can be used in all real-time systems. It has been designed for the specification and description of the behavior of such a system, i.e. the interworking of the system and its environment. It is also intended for the description of the internal structure of a system so that the system can be developed one part at a time.

The underlying model of an SDL specification is an Extended Finite State Machine (EFSM). EFSMs, which are FSMs extended with variables, have emerged from the design and analysis of sequential circuits and communication protocols. Its most general form has not only internal (context) variables, but also input parameters such that a transition can only be executed if its enabling condition (usually in the form of a predicate depending on input parameters and state variables) is satisfied.

The behavior of the SDL system is constituted by a combined behavior of a number of processes in the system. A process is an EFSM, that works autonomously and concurrently with other processes. The cooperation between the processes is performed asynchronously by discrete messages, called signals. A process can also send signals to and receive signals from the environment of the system. It is assumed that the environment acts in an SDL-like fashion, and it must obey the constraints given by the system description. The behavior of a process is deterministic: it reacts to external stimuli (in the form of signals) in accordance with its description. A process has a memory of its own for the storage of variables in addition to the state information, which is not accessible for the user of SDL. A process cannot write in the variables of an other process. A process has an infinite input queue, where incoming signals are queued. A process is either in a waiting state or it performs a transitions between two states. A transition is initiated by the first signal in the input queue. When a signal has initiated a transition, it is removed from the input queue (and is said to be consumed). In a transition, variables can be manipulated, a decision can be made, a new process can be created, signals can be sent (to other processes or to the process itself), etc.

Data types in the SDL are realized as Abstract Data Types. That means that all data types (predefined and user defined) are defined in an implementation-independent way in terms of their properties. The definition of ADT has three components: a set of values, a set of operations on these values, and a set of axioms defining the operations.

An SDL system can be represented in a graphical and textual form. SDL/GR is a standardized graphical representation of the system that is used to give a graphical overview. SDL elements such as signals, processes etc. are drawn using standardized symbols. The graphical representation is augmented with text for concepts that cannot be represented with graphics (such as ADTs). SDL/PR is a textual phrase representation of the SDL system; in other words, it is an SDL "source code" [BeHo88].

2.2.4.3 SDL tools

SDL commercial tools, such as SDT[SDT] [] and GEOD[GEOD][], provide integrated graphical environments for developing an SDL system. Tools consist of a number of functional components:

(1) Graphical editor for creating and editing the SDL system;

(2) SDL checker that finds syntax and static semantic errors in the SDL code;

(3) SDL simulator that simulates the system in the graphical environment;

(4) SDL tools that provide a C code builder that automatically generates a C/C++ code from the SDL specification for different target platforms.

SDL tools (in particular GEOD and SDT) were extensively used in this work for the specification of the Signaling protocol, and validation of the test suite was developed using the tools presented in Section 3.3. Since the GEOD SDL simulator was employed for test validation, in the following paragraph we give a more detailed description of its capabilities.

The GEOD SDL simulator can run in three simulation modes: interactive, random, and exhaustive. These three modes may be mixed during the same simulation session. For each of these modes, the simulator generates scenarios containing the results of the verification (deadlock detection etc). These scenarios can be replayed in interactive mode and expressed graphically in the form of Message Sequence Charts (MSCs) [MSC94]. The three simulation modes are discussed below:

(1) The interactive or step-by-step mode provides a fine-grained simulation. The user is free to decide which parts of the design to execute and to move up and down the simulation process with the Undo and Redo commands. Like conventional debuggers, the simulation offers a graphical view of the design being executed, indicates the current position in the corresponding MSC, and produces results in real-time. This mode was used for the test validation;

(2) The random simulation mode derives a pattern from a number of patterns provided by the developer to explore some of the possible application behaviors;

(3) The exhaustive mode requires the simulator to explore all behavioral paths. In this mode, the simulator checks all verification properties. If a violation is detected, a scenario is created, which describes how to get to the faulty condition.

2.3 Conformance testing

The objective of the OSI is to enable heterogeneous systems, implemented in different and independent ways, to interwork with one another [Knig87]. If this objective is to be met, two issues are of great importance: standards that define the systems should be written in a precise and unambiguous way, and the user should have a certain guarantee that the procured system complies with the standard requirements that the system vendor claims to satisfy. Here we are concerned with the second issue: how to test a protocol implementation in order to determine its conformance to the protocol specification.

As discussed in Section 2.2.3, conformance testing is a third phase of the protocol life cycle. Realizing the importance of the conformance testing for the objectives of the OSI, ISO worked on the standardization of the testing methods and concepts. Standardization efforts, which resulted in a five-part standard [ISO9646], identify the issues regarding the conformance testing process. In the rest of this section, concepts and framework of the conformance testing are introduced as they are defined by ISO.

A standard document defines a protocol by a set of requirements that should be met by a conforming protocol implementation. Conformance requirements fall into two groups [Rayn87]:
(1) Static conformance requirements are those that define the allowed minimum capabilities of an implementation in order to facilitate interworking;
(2) Dynamic requirements are all those requirements (and options) which determine what observable behavior is permitted by the relevant OSI protocol standard(s) in instances of communication.

Before the test laboratory may proceed with the conformance testing of the protocol, the client (in the OSI terminology a vendor of the implementation is a client for the test laboratory performing the testing) should provide or complete two documents. The Protocol Implementation Conformance Statement (PICS) is a statement made by the supplier of an OSI implementation stating the capabilities and options which have been implemented, and any features which have been omitted. It is needed so that the implementation can be tested for conformance against relevant requirements, and against those requirements only [Rayn87]. PICS is a document that should accompany the protocol implementation. In order to provide more information about the particular implementation, the client should supply the Protocol Implementation Extra Information for Testing (PIXIT). PIXIT provides information about the execution environment, addressing information, identification of the implementation, and other information necessary to run the tests.

2.3.1 Points of control and observation and test methods

In the conformance testing of communication protocols, the implementation under test (IUT) is tested as a black box. That is, the implementation details of the IUT are invisible to the external tester, leaving the implementator a freedom to decide about the internal design. As explained in Section 2.1, the OSI protocol standards define allowed behavior of a protocol entity in terms of the PDUs and the ASPs both above and below the entity. Thus the behavior of an (N)-entity is defined in terms of the (N)-ASPs and (N-1)-ASPs (the latter including the (N)-PDUs) [Rayn87]. These ASPs define the externally controllable and observable behavior of the protocol implementation. Therefore, conformance testing of the protocol implemented in the (N) - layer is performed by applying the ASPs that are defined as inputs and observing the ASPs that are defined as outputs at the (N) - and (N-1) - SAPs.

The points where the exchange of the (N)-PDUs, (N)-ASPs, and (N-1)-ASPs can be observed and controlled are called points of control and observation (PCO). In some instances, ASPs above the protocol are neither accessible nor controllable, directly or indirectly. It is assumed that (N-1)-ASPs are directly or indirectly controllable and observable by the tester. In the cases when (N-1)-ASPs are not directly controllable, conformance testing assumes that (N-1) - ASPs are sufficiently reliable to perform the testing remotely. The conformance test methods that are defined by ISO are based on the availability of PCOs in a given protocol IUT [ISO9646].

Conceptually, test methods are realized by a lower and an upper tester. The lower tester (LT) provides control and observation of the (N-1) - ASPs that are exchanged at the local lower IUT interface or at the remote interface of the service provider. The upper tester (UT) provides control and observation of the (N) - ASPs that are exchanged at the upper IUT interface. If LT and UT are realized as distributed entities, a test coordination procedure (TCP) governs the cooperation of the LT and UT during the testing.

Test methods can be grouped into two classes: a local test method that acts on the PCOs that are immediately above and below the IUT, and an external test method that acts on the PCOs that are remote from the IUT, for example, on the other side of the service provider. For single layer testing, there is only one local test method and three external test methods: distributed test method, coordinated test method and remote test method, defined as follows [Linn90]:

- local test method: This method defines the PCOs as being at the service boundaries above and below the (N) - entity under test. The test events are specified in terms of (N) - ASPs above the IUT and (N-1) - ASPs and (N) - PDUs below the IUT, as shown in Figure 2.5;

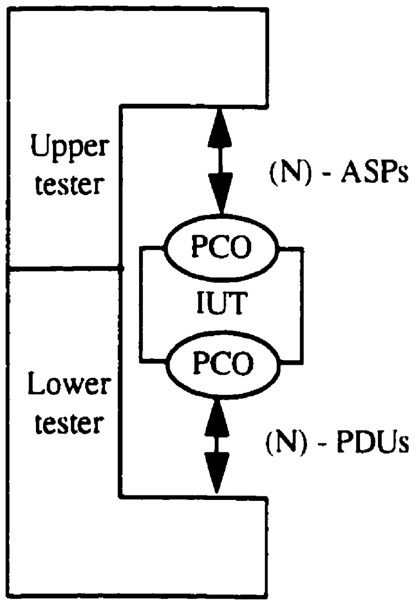

Figure 2.5: Local test method

- distributed test method: the IUT does not have an accessible PCO at the lower interface. This method defines the PCOs as being at the service boundaries above the (N) - entity under test and at the opposite side of the (N-1) - service provider [Rayn87]. (N-1) - ASPs and (N) - PDUs are controlled and observed at the remote end of the (N-1) service provider. LT and UT are realized as separate processes (Figure 2.6);

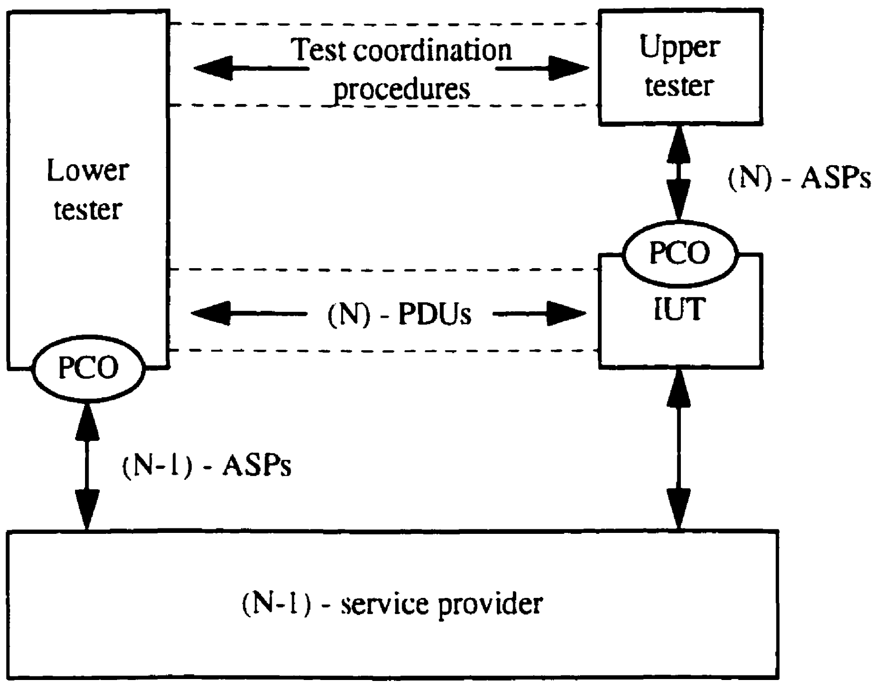

Figure 2.6: Distributed test method

- coordinated test method: this method is similar to the distributed test method. It is distinguished by two features from the distributed test method: no exposed upper interface is necessary within the IUT; and a standardized test management protocol and test management protocol data units are used to automate test management and coordination procedures. Exchange of the test management data units may be in-band, through the same channel as the protocol being tested, or out-band, through a reliable independent channel (Figure 2.7);

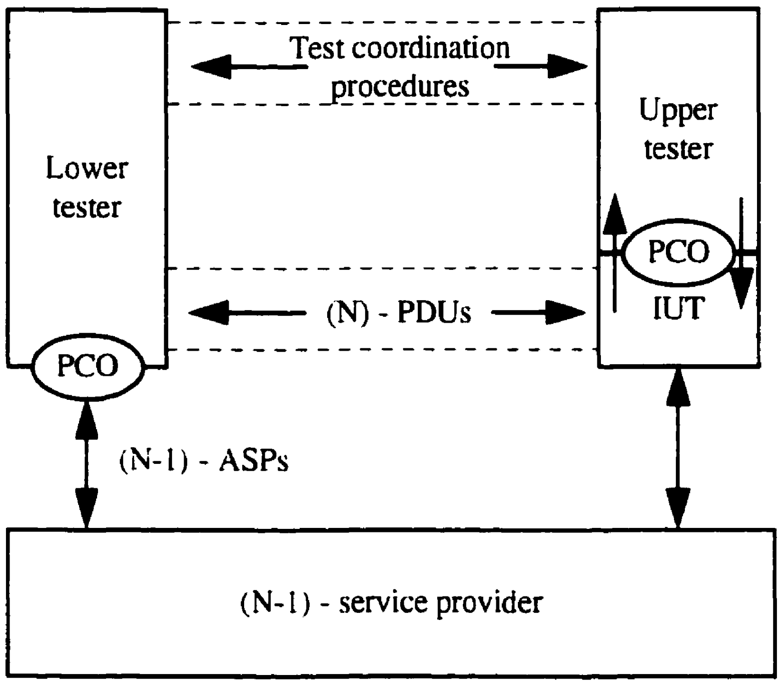

Figure 2.7: Coordinated test method

- remote test method: this method defines the PCO as being on the opposite side of the (N-1) - service provider form the (N) - entity under test [Rayn87]. The IUT does not have any exposed upper PCO and the TCP is used. The LT and UT are synchronized by the protocol being tested (Figure 2.8).

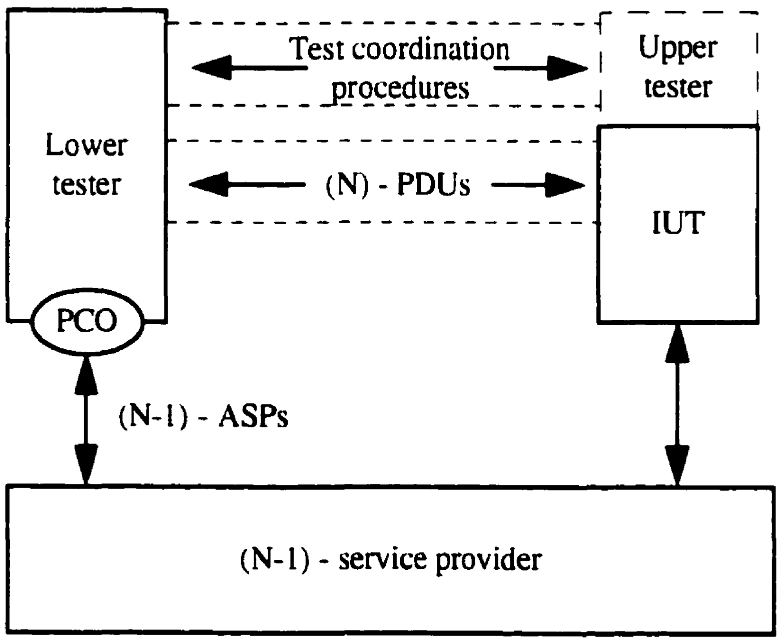

Figure 2.8: Remote test method

2.3.2 Test procedure overview

The test procedure (also called conformance assessment process [ATMF94]) is given in ISO/IEC 9646-1 [ISO9646]. It can be summarized in the flowchart shown in Figure 2.9. There are five main steps in this procedure [Rayn87].

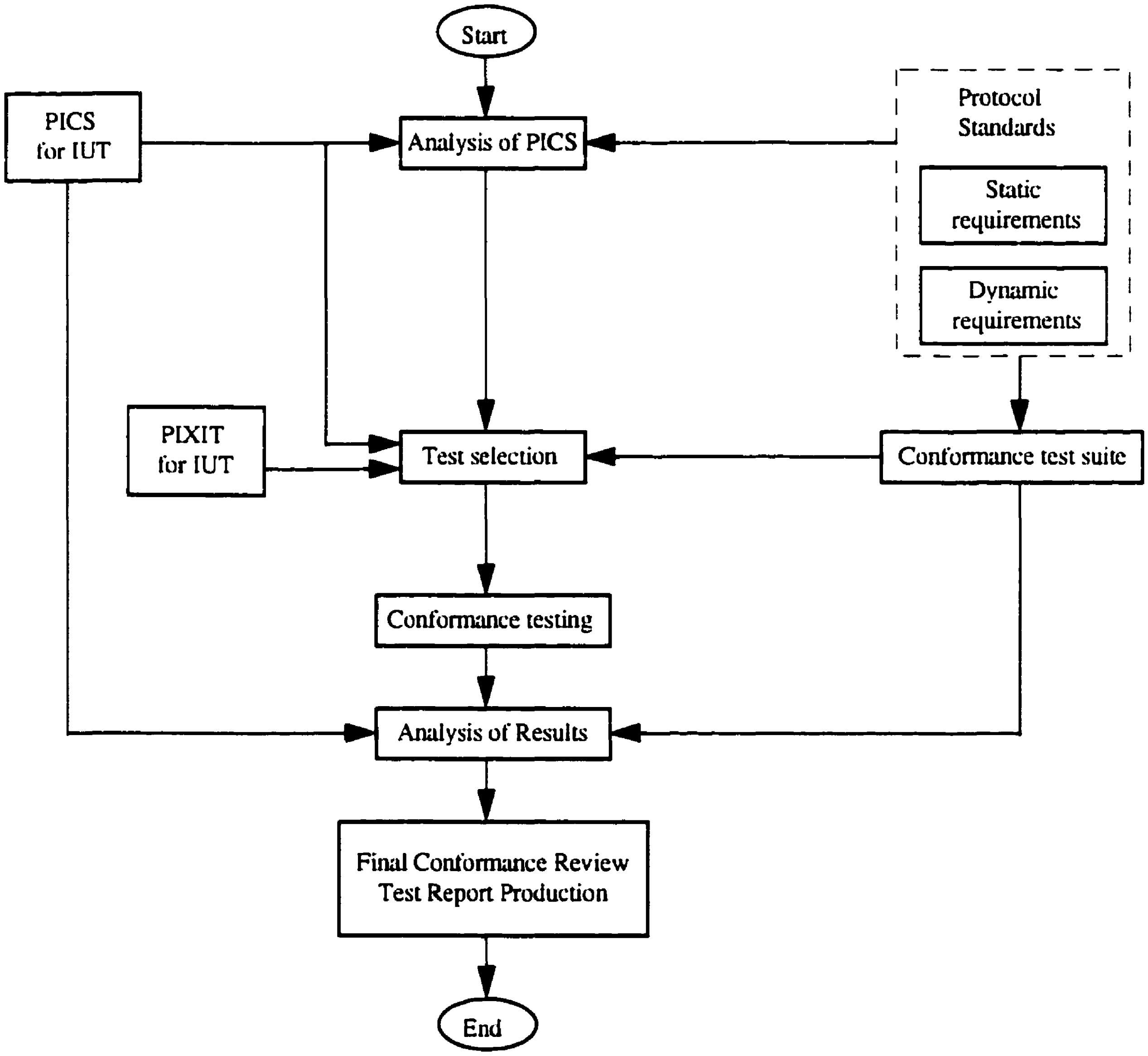

Figure 2.9: Overview of the test process

The first step is the analysis of the PICS accompanying the IUT. The PICS will be analyzed for its own consistency and for its consistency with the static conformance requirements specified in the relevant standard(s).

The second step is test selection. During the test selection, the PICS and PIXIT are used to select the appropriate abstract test cases from the existing conformance test suite and to parameterize them using the PIXIT information. An abstract test case is a test case that defines the sequence of the test events in a form that is independent of the target implementation platform. During the second step, abstract test cases are converted into corresponding executable test cases suitable for the intended real tester. This conversion can be done before or after the selection and parameterization. The result of conversion, selection and parameterization is called a parameterized executable test suite, which is the actual test suite to be run.

The third step is the execution of the parameterized test suite.

The forth step is the analysis of the results. This may in fact be interleaved with the execution of different groups of test cases, but it is easiest to think of it as coming after the test execution step is over. The observed outcome is the series of events which occurred during execution of a test case. The foreseen outcomes are identified and defined by the abstract test case specification taken in conjunction with the protocol standard. A verdict is a statement of pass, fail or inconclusive to be associated with every foreseen outcome in the abstract test suite specification. The analysis of results is performed by comparing the observed outcomes with foreseen outcomes, and a statement of verdict is passed.

The fifth step is the final conformance review, which involves a synthesis of the results of the behavior tests with what has been learned form the analysis of the PICS. The conclusion on the conformance of the IUT to the requirements of the standards(s) can be reached, and results are recorded in standardized Conformance Test Reports [Rayn87].

2.3.3 Development of a test suite

A test suite is the collection of test cases that have narrowly defined purposes, such as to verify that the IUT has a certain required capability (e.g. the ability to support certain packet sizes) or exhibit a certain required behavior (e.g. behaves as required when a particular event occurs in a particular state) [Rayn87]. A test case consists of a sequence of test events, where a test event is an automatic interaction between the IUT and the LT or UT, including the expiration of timer. A test case specifies the input sequence that is "fed" by the UT or LT to the IUT and the expected output sequence that should be observed by the testers. If an IUT produces the expected sequence, it is said that the IUT passed the test case.

A test purpose describes the objective of the corresponding test case. A test purpose can be derived directly by studying the relevant protocol standard, or, as in the cases of the automatic test selection process, test purposes can be associated with the transitions of the FSM model of the protocol.

A test suite is often organized in the collection of nested test groups. Each test group consists of a number of test cases that are related to the same logical grouping of protocol functions. For example, test cases that correspond to the connection establishment functions of the protocol should belong to one test group. A test group can be refined into an unlimited number of nested subgroups which relate to different subsets of a protocol function. Test groups may be used to aid planning, development, understanding and execution of test cases [Rayn87] Figure 2.10).

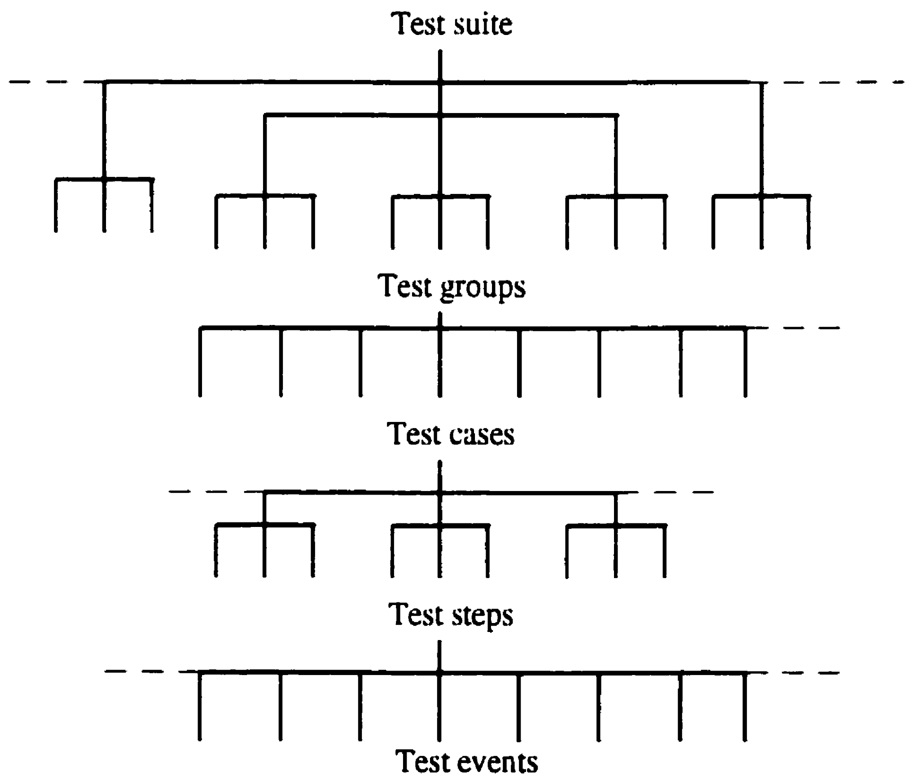

Figure 2.10: Test grouping

Standardized test suites are usually expressed in the form of an abstract test suite. An abstract test case may consist of three components:

(1) Test preamble defines necessary events to bring the IUT into the desired starting state to achieve the purpose of the test case;
(2) Test body defines test events that are needed to achieve the test purpose (in the case when the test purpose corresponds to a transition from the state table, the corresponding transition is realized by the test body) and;
(3) Test postamble that is used to put the IUT into the starting state from where the next test case will start. Some test development methods require a protocol to have a reliable reset function that will bring the protocol to the initial state.

ISO and ITU-T have developed the Tree and Tabular Combined Notation, TTCN, [TTCN] as a standardized test notation. It was originally designed for human readability, with the ability to define all the relevant paths through test case and assign verdicts to each. An abstract test suite can be written also using formal languages or some "ad hoc" techniques. A number of tools exist for translation from one of these "informal" test notations to a TTCN.

The goal of test suite development (called also test suite production [Rayn87]) is to generate a test suite for a particular protocol standard. In Figure 2.9, this process is represented as an edge from the protocol specification to the conformance test suite. The process of test suite development is not standardized by ISO. A test development process is often a manual derivation of a test suite from the standard document. However, automated tools exist today that help the test developer in developing tests. In general, a test development process starts with the study of the relevant standards to determine the conformance requirements. Often, the test developer derives the state table of the protocol as a base for the generation of test sequences. The test developer decides which test groups will be needed to achieve the appropriate coverage of the conformance requirements and refines the test groups into sets of test purposes. Afterwards, tests are derived using some test development method, or, still a common practice, the test developer derives tests based on his/her knowledge about the protocol and experience in testing. The result is a test suite used for conformance testing of the protocol implementations.

In conformance testing, an IUT that fails one of the test cases from the test suite is not conforming to the standard. If, however, the IUT passes all of the test cases, we still cannot claim the conformance unless we are confident that the conformance test suite used covers all possible faults in the IUT. The objective of the test development process is to generate a test suite that will have a complete or almost complete coverage of all possible errors in IUTs. It is obvious that this objective is not easy to realize, as the size of a test suite could be too big to be used in practice. A more pragmatic solution is to perform so-called interoperability testing in addition to conformance testing.

2.4 Interoperability testing

The problem of interoperability arises when end - users need to interconnect equipment from different manufacturers and to have a certain confidence that these pieces of equipment can interoperate [ATMF94]. The purpose of interoperablity testing is to confirm the degree of interoperability.

Interoperability testing is a process supplementary to conformance testing. While conformance testing involves testing of only one IUT, interoperability testing considers a system under test (SUT) of two or more interconnected IUTs. From the point of view of the OSI - RM, interoperability testing determines the functionality of a service provider that consists of two or more connected IUTs with a communication network between them. The generic testing configuration [ATM94] is given in Figure 2.11. The availability of the Monitor points is not guaranteed for every test configuration. Some test configurations may have access to Monitor points C and E. For "third-party" testing, the connections between the two IUTs may be a physical communication line, and, consequently, accessing the Monitor point D would be practically difficult or unfeasible. However, specialized testing equipment may help to make all three points accessible.

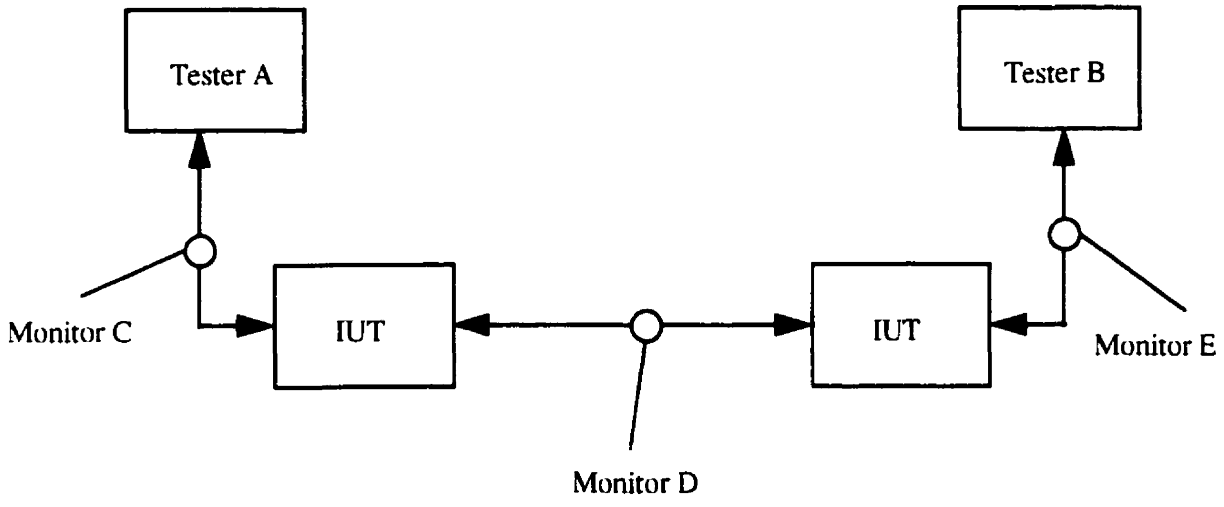

Figure 2.11: Generic testing configuration

Designers of the OSI model claim that OSI protocols are constructed in such a way that conformance implies interoperability [ISO7498]. On the other hand, practical experience has shown that pieces of equipment from different manufacturers, which are claimed to conform to the same protocol standard, may nevertheless be unable to communicate with each other [ArPh92]. For two IUTs to interoperate, two situations can be observed which can impact their ability to do so:

(1) The two IUTs implement the same mandatory features/functions, but differ in regard to optional and unspecified ones. In this case, even two conforming IUTs may not interoperate because their ability to interoperate depends on the optional and/or unspecified features;
(2) The two IUTs implement different mandatory features/functions. It is obvious that these IUTs are not conforming to a standard, but if there is sufficient overlap, the two IUTs may still be able to interoperate.

Interoperability testing does not include assessment of the performance, robustness, or reliability of an implementation. The interoperability testing is of great importance for the user of the IUTs that are realizing the (N) - service provider. The user is interested in the expected behavior of the underlying service provider and not so much in the conformance of the individual implementations of the (N) - protocol entities. However, conformance of the IUTs to the standard will increase the likelihood that they can interoperate.

Chapter 3

Formal methods and tools for protocol testing

It is widely recognized that a test development process should be based on a well-founded theoretical framework in order to produce a test suite of a satisfactory quality. In the previous 10 years, there was an extensive research work related to the process of test development. Most of this work was based on the FSM model-based testing, a topic rooted in problems related to state identification and fault detection for sequential circuits [Henn64]. More recently, the extended FSM model was used for protocol development, verification, and testing. Actually, the use of formal specifications of protocols opens the possibility for automatic or semi-automatic development of a test suite from a protocol specification. A number of different techniques have been proposed for automatic development of tests from the protocol specification expressed in one of the formal languages. Test development methods based on the FSM and the use of FDT in test development are discussed in more detail in the next section, followed by a description of the test development method used in this work.

3.1 FSM-based testing

3.1.1 Finite State Machine

A finite state machine is an abstract model for the description of the behavior of a system as a sequence of events that occur at discrete instants, designated t = 1, 2, 3, ... Let us consider a machine M that has been receiving input signals and has been responding by producing output signals. If now, at time t, we were to apply an input signal X(t) to M, its response Y(t) would depend on X(t), as well as on past inputs to M. Since a given machine M might have an infinite variety of possible histories, it would need an infinite capacity to store them and, consequently, machine M might need an infinite number of states to describe the behavior of the system. However, the past histories of a system described by a finite state machine can affect its future behavior in only a finite number of ways. In other words, a finite state machine can distinguish among a finite number of classes of input histories referred to as the internal states (or simply states) of the machine. Every finite state machine, therefore, contains a finite number of memory devices, which store the information regarding the past input history. The formal definition of a finite state machine, often simply called a machine, is given as follows:

A finite state machine M is a 7-tuple, denoted as $M = <X, Y, S, s_1, D, \delta, \lambda>$, where

X is a finite set of input symbols;

Y is a finite set of output symbols;

S is a finite set of (internal) states;

$s_1 \in S$ is the initial state;

$D \subseteq S \times X$ is a specification domain;

δ: D --> S is the transfer function (also called the next state function);

λ: D --> Y is the output function.

Intuitively, the input symbol set X represents all the possible input values that an input signal X(t) can take at discrete time instants, while the output set Y includes all the possible output values that an output signal Y(t) can take at discrete time instants. The state set S represents all the internal states that the FSM may experience.

The transfer function δ and the output function λ together characterize the behavior of the FSM. As δ and λ are required to be functions, the FSM model defined is deterministic. More specifically, for each $(s_i, x) \in D$, there should exist exactly one state $s_j \in S$ and exactly one output symbol $y \in Y$ such that $\delta(s_i, x) = s_j$ and $\lambda(s_i, x) = y$. In this case, it is said that there is a specified transition from state s_i to s_j with input x and output y. Such a transition is usually written as s_i -x/y-> s_j. Usually, s_i is called the head (or starting) state and s_j is called the tail (or ending) state of the transition.

It should also be noted that if the specification domain $D = S \times X$, then the transfer function δ and the output function λ are defined for all the state-input combinations and accordingly the FSM is said to be completely specified (or completely defined). On the other hand, if the specification domain $D \subset S \times X$, then there should be some state-input combinations for which the transfer function δ and the output function λ are not defined and consequently the FSM is said to be partially specified (or partially defined).

An FSM can be given in the form of a state table. States and input symbols are used to name the rows and columns, respectively. A state/output pair s_j/y_l appeared at the location of row s_i and column x_k implies that there is a transition s_i -x_k/y_l-> s_j. The symbol "-" is used to denote that the transfer function or the output function is not defined. Accordingly, the pair "-/-" is used to represent the case that neither the transfer function nor the output function is defined. A more commonly used approach is to describe an FSM as a directed graph called the state diagram, with the states and transitions of the FSM represented by the vertices and arcs of the graph, respectively.

3.1.2 The FSM conformance testing

In the FSM conformance testing, we have complete information about the specification machine A; we have its state transition and output functions in the form of a transition diagram or state table. The implementation machine B is a "black box" that can be observed only through its I/O behavior. The goal of the FSM conformance testing is to design a test that will determine whether B is a correct implementation of A (B is equivalent to A) by applying the test sequence and observing the outputs [LeYa96]. Obviously, without certain assumptions the problem cannot be solved; for any test sequence we can easily construct a machine B, which is not equivalent to A but produces the same outputs as A for the given test sequence. There is a number of natural assumptions about the specification and implementation FSMs [LeYa96].

Assumptions that are made about the specification machine A (which is deterministic by default) are basically about its following structural properties:
(1) completeness: if A is completely or partially specified;
(2) connectedness: if A is strongly or initially connected;
(3) reducibility: if A is reduced or non-reduced.

Another class of assumptions is about the types of faults (i.e. the fault model [Boch92, More90]) that can be present in an implementation. Implementing a system modeled by the given specification machine A can be considered as a process during which the developer makes various changes to the specification machine A. Such changes may introduce undesired behavior of the system and make the system invalid. For the FSM model presented in Section 3.1.1, we have in general the following four types of changes that can be made by a developer:
Type 1: change the tail state of a transition;
Type 2: change the output of a transition;
Type 3: add a transition; and
Type 4: add an extra state.

Without imposing any restriction on the types of faults, the number of all possible invalid implementations of the given specification A is infinite. A finite set of mutant FSMs which can be constructed by introducing a number of changes (Types 1-4) is called a fault model. A test suite is said to have a complete fault coverage for a given fault model if it gives the fail verdict for any nonconforming implementation from the fault model.

The existing test generation methods, discussed later, are based on the assumptions concerning the specification machine A; a particular test generation method is applicable only to the class of the specification machines that satisfy the assumptions. Similarly, given an implementation machine B, a particular test generation method should usually be capable of detecting errors in the implementation that can be modeled with one or several of the fault types introduced above. However, some of the existing methods detect all four types of errors.

3.1.3 Test generation methods

In the following, we give an informal description of five fundamental test generation methods. Today, a number of modifications, improvements and optimizations for these five methods exist in the literature. For a more complete, updated description of test generation methods, the reader should refer to [PeBoYa96].

Most of the existing test generation methods are based on a certain kind of state identification facility. State identification facilities are certain input/output behaviors that can distinguish a given state from the other states in an FSM at hand. The key point of FSM-based testing is how to use some state identification facility derived from the specification machine A to identify the different states in the implementation machine B.

Most of the methods perform the testing in two phases. In the first phase, the state identification facility derived from the specification machine A should be applied to the implementation machine B to check if it can also properly identify the states in the implementation. If the implementation cannot pass the first phase of testing, then the implementation does not conform to the specification and accordingly no further test is required. On the other hand, if the implementation passes the first phase of testing, then it can be concluded that:
(1) the state identification facility is capable of identifying the state in the implementation; and
(2) each state in the implementation corresponds to a state in the specification.

Once the implementation passes the first phase, the second phase tests if the output and next state of each transition is correctly implemented. In particular, the correctness of the next state of a transition can be verified with the state identification facility as it has already been checked in the first phase that the facility can properly identify all the states in the implementation.

A number of test development methods have been proposed for protocols that are specified with the FSM model. These methods are classified as the Transition Tour method (T - method)

[NaTs81], UIOv-method [Vuon89], D-method [Gone70], W-method [Chow78], and HSI-method [Petr91].

The T-method is relatively simple; the test sequence produced by this method is a transition tour of the FSM for a protocol. This method only requires that the specification machine is initially connected. This method aims at detecting all the output faults in the absence of any transfer faults. Accordingly, it cannot provide complete fault coverage.

The other methods are more sophisticated and implement the two-phase testing approach. They not only check the implemented output of each transition, but also verify whether the tail state of each transition is correct or not. The state is verified using state identification sequences. These are a distinguishing sequence (DS), unique I/O sequence (UIO), characterization set W, and harmonized state identifier (HSI) for the D-method, UIOv-method, W-method, and HSI-method respectively. If the corresponding state identification exists for a given FSM, all methods can provide complete fault coverage.

The D-method assumes that the machine is minimal, strongly connected, completely specified and possesses a distinguishing sequence (DS). An input string α is said to be a distinguishing sequence of a machine A if the output string produced by A in response to α is different for each starting state. D-method is restricted to those FSMs that have a DS.

The UIOv - method also assumes that a machine is minimal, strongly connected, and completely specified. It involves deriving a unique input/output (UIO) sequence for each state of A. A UIO sequence for a state of A is an I/O behavior that is not exhibited by any other state of a machine A.

The W-method is based on a W-set for state identification. A W-set is a characterization set that consists of input sequences that can distinguish between the behaviors of every pair of states in the FSM [Chow78]. The original version of W-method [Vasi73] assumes that the specification machine A is minimal, deterministic, initially connected and completely specified. The W-set exists for any minimal and completely specified FSM.

The HSI-method [Petr91] uses a harmonized state identification facility (HSI) that exists also for partially specified machines that are minimal. An HSI, written as $H = < H_1, H_2, ..., H_n>$ satisfies the following:

(1) H_k is a set of permissible sequences for the specification machine A;

(2) For any two state identifiers H_i and H_j, there exist two sequences, one from H_i and the other from H_j, which have a common prefix that distinguishes state s_i from state s_j.

HSI is the most general state identification facility and includes as special cases DS, UIO and W - sequences. Apparently, if $H_1 = H_2 = ... = H_n$, then HSI is a W-set. If, in addition, all H sets consist of one sequence, then HSI is a DS. When a set H_i consists of only one sequence, it is a UIO.

The above methods are focused on testing FSMs. An FSM models only the control part of a protocol (also called the FSM part). In reality, protocols often have variables and actions that are based on the variable values. Formal techniques, based on EFSM, have emerged as more powerful tools to model the protocols, and, consequently, to facilitate the protocol testing.

3.2 Methods for automatic test development from SDL specification

A formal specification, such as an SDL specification, can be used for the following activities [BoPe97]:

(1) Test suite development: Semi - automatic development of test purposes, an abstract test suite (written in TTCN, SDL or some ad hoc notation, possibly including sequences of API calls) and their automatic translation into executable tests;

(2) Test suite validation: The specification can be used as a "reference implementation" to check the correctness of newly developed test cases using a simulated execution environment provided by existing SDL tools.

Test suite validation can be performed using the commercial SDL tools. However, there is no general solution for automated test development. In the following paragraph, we concentrate on the problem of automated derivation of tests from a given SDL specification, and present the alternative approaches to the problem.

It is a well-known difficult problem to derive a parameterized input sequence which either transfers an EFSM to a desired state or which distinguishes a pair of states [BoPe97]. Compared with the classical FSM model, the EFSM model may provide a very compressed behavioral description of the system, but at the same time, it is much less tractable for verification and test derivation purposes. If certain limiting assumptions are made about the form of the predicates and actions, the analysis of the behavior of the specification and systematic test selection remains decideable [Higa94], but in general, in particular when the actions may include loops, the question of deciding which input parameters should be used to force the execution of a particular transition becomes undecideable, like the question of deciding the executability of a given branch of a program in software testing.

One possible approach to deriving tests from the EFSM is to employ the dataflow analysis. According to this approach, a control flow graph and data flow graph are constructed from the EFSM. Test sequences are generated by constructing subtours of the control flow graph. To make test sequences executable, information from the data flow graph is used to parameterize inputs and to initialize the context variables [SaBo87].

An alternative approach is to view an EFSM as a compressed notation of an FSM. The intention behind this approach is to retain the applicability of the FSM-based methods to generate tests [BoPe97]. With this approach, at least three solutions exist to obtain a more tractable state-oriented specification:
(1) to derive a pure FSM by ignoring all the extensions (parameters, predicates, and actions) to the basic FSM model;
(2) to unfold the EFSM into an FSM by expanding the values of the input parameters and context variables;
(3) to extract an FSM by a partial unfolding of variables of enumerated types, while using enabling conditions as a part of the corresponding FSM inputs.

The main drawback of the first option is that all the tests derived from the obtained FSM should be verified for executability. The second option, a straightforward unfolding of an EFSM, easily leads to an explosion of the number of states and inputs.

The test selection process used in this work is based on the approach that views an EFSM as a compressed notation of an FSM. More precisely, we approximate the behavior of the SDL specification by an FSM (called an approximating machine), where an input of the FSM corresponds to the pair of an input signal and an enabling condition (if any), while states of the FSM mostly correspond to the control states of the SDL specification, except for unfolded states, which are control states augmented with values of enumerative variables [BoPe97]. This approach is implemented in an experimental test derivation tool developed at University of Montreal.

3.3 UofMontreal tool chain for automatic test development

3.3.1 Overview of the tools

A number of different experimental tools [BoPe97] have been developed at the University of Montreal for partially automating the test development process. Most tools use the underlying FSM model, that is, the behavior of the system to be tested is expressed in terms of a number of states and inputs as well as and the outputs and state transitions produced by the arrival of a given input in a given state. These tools are therefore useful for systems that can be characterized by FSM-oriented specifications, such as communication protocols.

In the following section, we focus on a chain of tools for the development of test cases from SDL specifications, as shown inFigure 3.1.

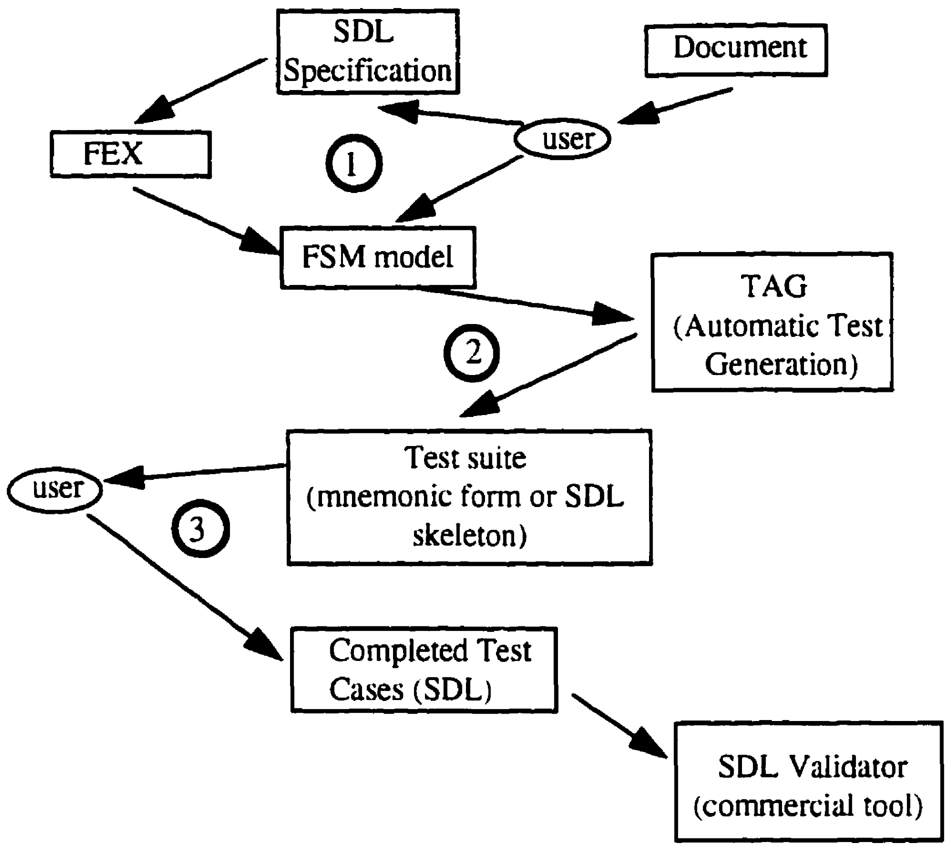
SDL Specification
Document
FEX
user
1
FSM model
TAG (Automatic Test Generation)
2
Test suite (mnemonic form or SDL skeleton)
user
3
Completed Test Cases (SDL)
SDL Validator (commercial tool)

Figure 3.1: Test suite development from SDL specifications

The middle column in Figure 3.1 shows the description of objects leading from the formal specification in SDL to the parameterized, SDL executable test cases. On the left and on the right, the tools shown that can be used during the test development process.

3.3.2 FEX tool

The first tool, called FEX (FSM Extractor), extracts from the SDL specification a partial view of the behavior represented in the form of an FSM (an approximating machine). At the same time, files containing SDL declarations of interactions (called "signals") and channels are created which can later be used to obtain complete test cases written in SDL. The FEX tool generates an FSM transition for each branch of each SDL transition in the specification; thus each branch corresponding to a particular input and particular conditions of the input parameters gives rise to a separate transition (for each state of the specification). We note that the resulting FSM model is quite similar to the "test matrix" which is commonly used for the manual development of protocol conformance test suites.

3.3.3 TAG tool

3.3.3.1 Overview

The TAG (Automatic Test Generation) tool [Tan96] is a generic tool for test suite development based on FSM specifications. The TAG tool implements the HSI method [Petr91], discussed in Section 3.1.3. It accepts as input a partially specified, deterministic FSM and generates test cases according to the options provided by the test designer. The options include the following:

(1) Automatic generation of a complete test suite with guaranteed coverage of output and transfer faults (assuming that the number of states of the implementation under test (IUT) is not larger than the number of states of the specification);

(2) Generation of tests for a specific transition (corresponding to a given "test purpose") selected by the test designer;

(3) The use of state identification sequences for checking transfer faults is optional;

(4) Separate generation of test preambles, postambles and state identification sequences;

(5) Generation of tests for grouped transitions (corresponding to a single SDL transition having several starting states, or several input signals, further discussed in Section3.3.3.2);

(6) Generation of tests related to timers (setting, resetting and time-out transitions, further discussed in Section 3.3.3.3).

The TAG tool supports several output formats for the generated test cases:

(1) I/O sequences (mnemonic form): The mnemonic form represents a test case as a sequence of input and output events from the tester. Each event is represented with a descriptive name and is written in a new line. The output of the tester is distinguished with the prefix sign ! in front of the event name, while the expected input is prefixed with the sign ?. This format is easy to read and relatively condensed.

(2) SDL skeletons: The generated SDL skeletons represent test cases (preambles, postambles and state identification sequences). They are complete SDL procedures, except that the details concerning the signal parameters are not included. (Note: If the SDL signals of the specification have no parameters, the generated SDL skeletons are complete SDL procedures).

(3) TTCN-MP skeletons: The generated skeletons represent test cases, preambles, postambles and state identification sequences. They are complete TTCN dynamic behavior trees, except that the details concerning the signal parameters are not included.

The generated test suite (in SDL or in TTCN) must often be completed by the test developer in order to add the information concerning the signal parameters. Often, when a signal is received by the tester, input parameters have to be checked for correct values. Also, before a signal is send from the tester, correct values of the output parameter have to be determined.

The final development step shown in Figure 3.1 is the validation of the obtained test cases against the original SDL specification, using an existing SDL development environment. This step is not automated by the experimental tools and requires manual modifications of the test cases by the test developer. More details about this step and testing of the data part of the protocol will be given when we discuss conformance testing of the Signaling protocol (Chapter 5).

3.3.3.2 Coherent transitions resulting from SDL specifications

In spite of the fact that state/transition explosion does not usually occur when an approximating machine is derived from an SDL specification, the number of transitions specified in the obtained machine can yet be very high. As a result, the total length of tests derived by means of the TAG tool could be quite large as well. This often happens when a single transition of the given specification yields in the resulting machine multiple similar transitions having the same output. The basic idea is to test only one (or a few) transition among a set of similar ones.

It is well known that a single statement in SDL may be used to describe multiple transitions. For example, the fragment

```
state * (s1, s2, s3);
    input i1, i2;
        output o;
        nextstate s4;
```

corresponds to many transitions from all states, except **s1, s2, s3**, under input i1 or i2. Each of these transitions has the same output **o** and leads to the same next state **s4**. We call such a group of transitions convergent transitions. If the next state is specified as "-" (meaning to remain in the same state), then the statement describes the set of transitions which we call a group of looping transitions. In addition, the symbol "*" may be used to describe the whole set of inputs.

In general, we call a group of convergent or looping transitions a group of coherent transitions. These groups can be distinguished according to the sets of starting states and/or the sets of inputs; all transitions of a group have the same output. The information about coherent transitions may either be deduced automatically from a given SDL specification or given by the test designer in the form of a list of coherent groups (in addition to the list of individual transitions).

The extended TAG [Main96] tests a single representative among convergent transitions. In particular, it is assumed that if a single transition has a fault then all coherent transitions in the group are faulty; all of them have a wrong output and/or wrong tail state. Testing just one among the group would be sufficient. Theoretically speaking, fault detection power of the resulting test suite may not always correspond to what is often called "complete fault coverage" [Boch94]. However, deriving tests only for selected transitions gives a good tradeoff between the length of tests and the fault coverage.

3.3.3.3 Handling timers and related counters

Error-recovery functions of communication protocols often rely on timers which invoke limited retransmission of PDUs. At the expiration of a timer, a specific output is sent and the timer is restarted if the maximum number of retransmissions is not yet attained. If the maximum number is reached, usually a different transition with a different output is taken, for example, to release the current connection. Certain input messages may stop a running timer.

The classical model of an FSM has no notion of time, yet it is quite common to use, in state-oriented specifications, a dummy input T to represent a silent time period which leads to the expiration of timer T. To model the behavior triggered by timers and related counters, one typically augments FSM transitions between (control) states by internal actions start T, stop T and adds transitions guarded by timer expirations (timeouts), as shown in Figure 3.2.

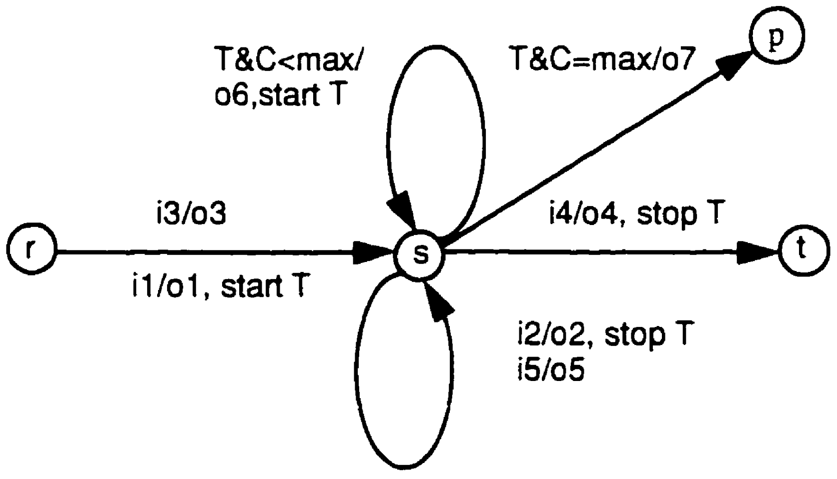

Figure 3.2: The fragment of a machine with the timer

Here, C represents a counter used by the protocol entity to ensure that the number of timer expirations never exceeds a given limit max.

A specification of the timer-regulated behavior should be consistent, in the sense that the presence of a timer should influence the observable behavior of the protocol and should be detectable by an external observer. In particular, as the above fragment shows, if timer T can be active in state s there should be at least one incoming transition labeled with start T as well as at least one outgoing transition labeled with stop T. Once max is reached, a time-out should cause an output different from the one produced by the previous time-outs, i.e. $o_6 \neq o_7$ and, in general, a transition to a next state. In addition, to be consistent, a specification should, in case that several timers can be active at the same state, have no transition simultaneously starting several timers.

Potential implementation errors related to timers may either change the expected behavior or cause a new, unexpected behavior. Faults of the former group may occur in

- transitions labeled with**start T** (expected **start**)
- transitions labeled with**T and [C=max]** (expected **max**)
- transitions labeled with**stop T** (expected **stop**).

Faults of the latter group may create unexpected actions with timers, such as

- new transitions labeled with**start T** (unexpected **start**).

- new transitions labeled with**stop T** (unexpected **stop**).

In the following paragraph, we discuss the structure of test cases which are needed to check the above transitions, using the example shown in Figure 3.2.

Expected **start**: To check whether or not the input $i1$ sets the timer T, we use the test sequences defined by the following expression:

$$\alpha[r].\ i1.\ T.\ W[s],$$

where $\alpha[r]$ is a preamble to bring the machine from the initial state to the state r; T indicates that the tester should have time-out T, $W[s]$ is a set of identification sequences for the state s (optional, in case we wish to confirm the tail state of the transition caused by the first expiration of the timer). Once the IUT passes all these tests, the following tests could be applied.

Expected **max**: To check whether or not the implemented counter reaches the specified limit **max**, we use the test sequences defined by the following expression:

$$\alpha[r].\ i1.\ T(1).\ T(2).\ \ldots T(\mathbf{max}).\ W[p],$$

where **max** consecutive signals T indicate that the tester should have its time-out T expired **max** times observing repeated output $o1$ followed by $o2$. An earlier reception of $o2$ indicates that either the related counter was not properly initialized or the implemented value is less than **max**. In the case when a timer should expire only once (no counter is used), an additional time-out may be included in the test to verify if any unforeseen counter is implemented for this timer.

Expected **stop**: To check whether or not the input $i4$ arrived after $i1$ stops the timer T, we use the test sequences defined by the following expression:

$$\alpha[r].\ i1.\ i4.\ T.\ W[t],$$

where the use of the state identifier $W[s]$ is optional. Any output produced by the IUT during the time-out period indicates that the input $i2$ did not stop the timer T.

Unexpected **start**: To check whether the input $i3$, for example, sets the timer T on, we use the test sequences defined by the following expression:

$$\alpha[r].\ i3.\ T.\ W[s].$$

Any output produced by the IUT during the time-out period indicates that the input $i3$ unexpectedly set the timer T. Tests of this type applied to all states at which the specification has no active timers would reveal an unforeseen timer. Assuming that, in the implementation under test, all timers are placed at the correct states, one may skip many tests related to unexpected **start**.

***Unexpected* stop:** To check whether or not the input $i5$ stops the timer T, we use the test sequences defined by the following expression:

$$\alpha[r].\ i1.\ i5.\ T.\ W[s].$$

The IUT is expected to produce the output $o1$ after the time-out T; the failure to do so signals an error. In Chapter 5, we present the results of using the presented tool chain for test derivation from the ATM Signaling protocol.

Chapter 4
The ATM Signaling protocol and its specification

4.1 ATM protocol architecture

Asynchronous Transfer Mode (ATM) is a telecommunications concept defined by ANSI and ITU standards for carriage of a complete range of user traffic, including voice, data and video signals, on any User-to-Network Interface (UNI). An ATM user represents any device that makes use of an ATM network via an ATM UNI (Figure 4.1).

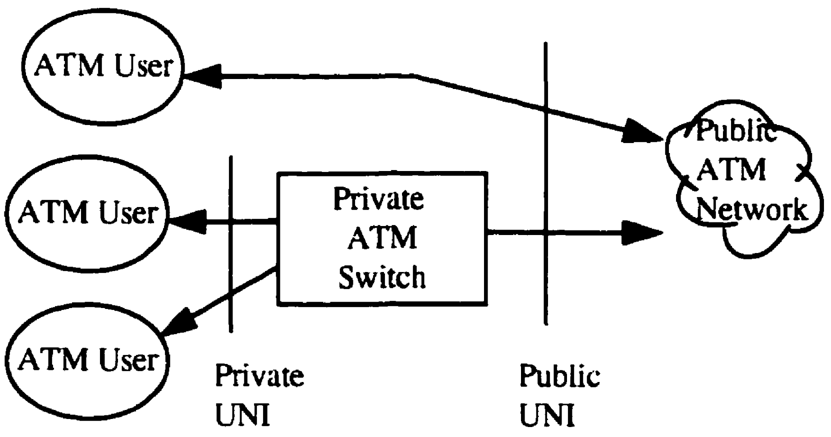

Figure 4.1: Implementations of the ATM UNI

The ATM protocol reference model uses a layered architecture divided into multiples planes (Figure 4.2).

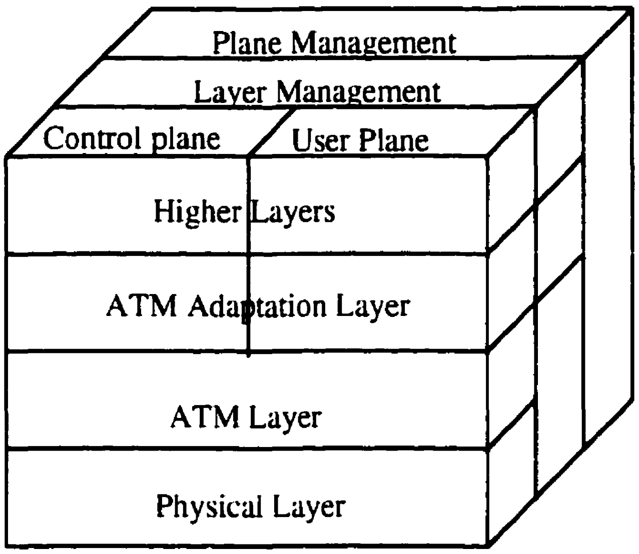

Figure 4.2: ATM protocol stack

The User plane (U-plane) provides services for the transfer of user application information. The Control plane (C-plane) protocols deal with call establishment, release and other connection control functions necessary for providing switched virtual circuits (SVC). The Management plane (M-plane) provides management functions and the capability to exchange information between U- and C-planes.

The UNI specification involves those protocols which are either terminated or manipulated at the user-network interfaces. The protocols that are defined by the UNI specification belong to the Physical and ATM layers, C-plane higher protocol layers for SVC and other protocols required for UNI management. The ATM UNI Signaling protocol, which is one of the UNI protocols, specifies the procedures for dynamically establishing, maintaining and clearing ATM connections at the UNI. It resides in the C-plane and uses the Signaling ATM Adaptation Layer (SAAL) services for message exchange with the peer entity. The formal specification and testing of this protocol in this work is based on the ATM Forum UNI specification, version 3.1.

4.2 An overview of the Signaling protocol

The initial deployment of the ATM technology anticipated only permanent virtual connections (PVC). PVC are connections between the communicating parties that are established via provisioning (usually by the network configuration) at the time of setup of the network. They generally remain established for long periods of time and should automatically be re-established in the event of the network failure. The further deployment of ATM required switched virtual circuits (SVC). SVCs are dynamically established in a real time using signaling procedures. Establishing the ATM SVC is analogue to dialing the telephone number and getting the connection. SVCs are also referred to as "connection on demand".

The Signaling protocol, as defined in ATM Forum UNI 3.1, specifies the procedures for dynamically establishing, maintaining and clearing the SVC at UNI. The Signaling protocol specification, as is the case with most communication protocols, is divided into two parts: procedures and data. The control part of the protocol, as defined in the ATM Forum UNI 3.1, consists of 9 states and 8 messages (PDUs) for point-to-point calls. Additional 4 messages are defined for point-to-multipoint calls. The data part defines the type, structure, fields, and values of the messages and the information elements used to characterize the ATM connection. The messages also include information that defines the characteristics of the connection (peak cell rate, timing, quality of service, etc.).

The end-point (ATM user) that originates the request for connection is the calling party. The end-party that accepts the connection request is the called party. The ATM network identifies the called party by the ATM address in the call request. Each ATM end-point has a unique ATM address, however, more that one connection is possible between two ATM end-points. Each connection is assigned a local unique Call Reference (CR) number. CR is unique inside the local end-point: it is not for the entire ATM network. In our work, we consider protocol functions required for managing point-to-point calls at the user side.

4.2.1 Messages

Messages are exchanged between two peer protocol entities that want to establish or have established a connection. The Signaling protocol makes use of the services of the underlying protocol SAAL for reliable, in-order transport of the messages. The procedures of the ATM UNI Signaling protocol are defined in terms of the following messages (PDUs):

(1) SETUP: initiates a call establishment;

(2) CALL PROCEEDING (CALL_PRO): indicates that the call request arrived at the local UNI;

(3) CONNECT (CONN): indicates to the calling party that the call is accepted by the called user;

(4) CONNECT ACKNOWLEDGE (CON_ACK): acknowledges a CONNECT message;

(5) RELEASE: requests the clearing of a call;

(6) RELEASE COMPLETE (REL_COM): confirms the clearing of a call;

(7) STATUS ENQUIRY (STATUS_ENQ): is send at any time to solicit a STATUS message from the peer entity;

(8) STATUS: is send in response to a STATUS ENQUIRY message or at any time to report certain error conditions.

In the following text, the term PDUs and messages are used interchangeably to refer to the same concept. The message structure consists of a general part (common for all messages) and a sequence of information elements (IE) that convey the data associated with the particular message. The common part of the messages shall always be present, while the IEs are specific to each message type (they may be absent too). Within the ATM Signaling protocol, every message consists of the following four common fields, seeFigure 4.3:

(1) protocol discriminator: distinguishes messages for user-network Signaling protocol from other protocols;

(2) Call Reference (CR): identifies the connection to which a message applies;

(3) message type: each message has a specific message type identifier;

(4) message length: specifies the length of the message content;

(5) variable length IEs: a sequence of one or more IEs that characterize the ATM connection and ensure the interoperability.

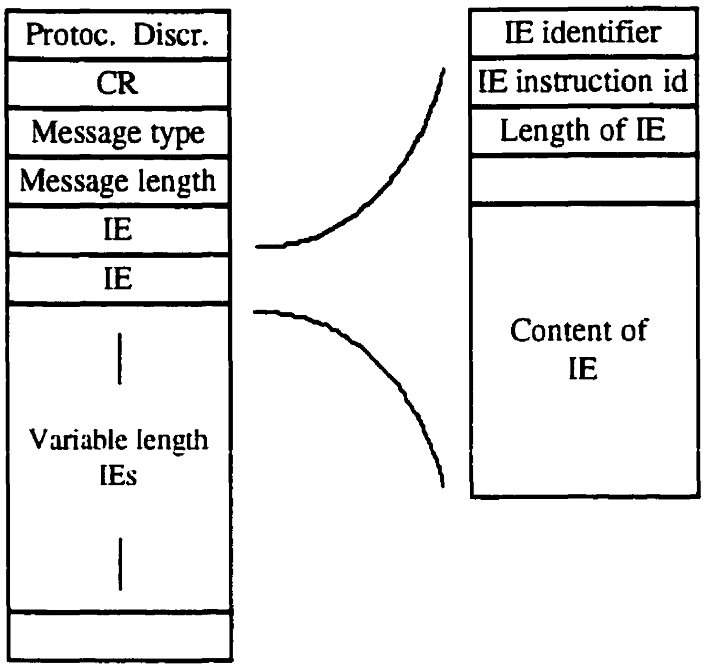

Figure 4.3: General message format

The presence of an IE within a message may be mandatory or optional. For each message, the standard defines the IEs that are mandatory and IEs that are optional as well as the conditions under which a particular IE may be present in the message. By itself, IEs are organized as a sequence of octets (we call them fields) with a defined data type and the range of values. A field of an IE may be:

(1) always present, present only in combination with a value or with the presence of another part of the IE. For example, in the Cause IE, the Value field is always present, while the Diagnostic field is present only for specific values of the Value field (96, 101 etc.);

(2) optionally present, not pertaining to another field or depending on the value or the presence of another field of the IE. Examples: 1. The ATM adaptation parameters IE may contain or not Error correction method field; 2. In Broadband bearer capability IE, Timing Type and Timing Requirements fields may only be present if Bearer Class X is indicated; 3. In B_LLI IE, Packet Window Size field may only be present if Default Packet Size field is present.

The Signaling protocol entity may receive a message with an invalid content. The standard defines five types of possible errors related to a content of IEs in the messages. For each type of error, specific Error Handling procedures are defined by the standard. The message may have the following errors:

(1) Mandatory IE Missing (MIEM) if one or more Mandatory IEs are not present in the message;

(2) Mandatory IE Content Error (MIECE) when one or more of the mandatory IEs has an invalid content value (for example, out-of-range value or a wrong value combination);

(3) Unrecognized IE (UIE), if all mandatory IEs are found and there is an IE that cannot be recognized as a valid IE;

(4) Non Mandatory IE Content Error, when the optional IE has an invalid content value:

(5) Recognized unexpected IE (RUIE), when the IE is correctly recognized (decoded) but is not needed according the standard in that particular message.

4.2.2 Abstract Service Primitives

It should be emphasized that ATM Forum UNI 3.1 does not explicitly specify the Signaling protocol using the ASPs at the upper SAP of the Signaling protocol. It is left to the implementation to complete the protocol in a particular way and to define its interaction with the upper layer. The goal is to standardize the protocol in terms of interworking capabilities between two instances of the protocol and to leave sufficient flexibility for different implementations. However, the FSM model of the protocol without messages on the upper interface would not be complete and some of the protocol states as defined in the standard would not be reachable. The states that are entered by the transitions which are driven by the request or response from the upper layer (as request for the Call Setup) cannot be reached if interactions with the upper layer are not included in the FSM model. To obtain a complete FSM specification of the protocol, we use diagrams from the Q.2931 document [Q2931], which uses both PDUs and ASPs.

Table 4.1 shows messages and corresponding ASPs used to specify the Signaling protocol in this work. In the left part of the table, primitives received from the upper layer are presented and the corresponding messages that are sent in the direction from the protocol to the network. In the right part of the table, messages that are received from the network are presented and the corresponding primitives that are sent to the upper layer.

Primitives from upper layer	Messages user->network direction	Messages network->user direction	Primitives to upper layer
Call Setup		**Call Setup**	
Setup_req	SETUP	SETUP	Setup_ind
Proceeding_req	CALL_PROCEEDING	CALL_PROCEEDING	Proceeding_ind
Setup_resp	CONNECT	CONNECT	Setup_conf
		CONNECT_ACK	Setup_complete_ind
Call Clearing		**Call clearing**	
Release_req	RELEASE	RELEASE	Release_ind
Release_resp	RELEASE_COMPLETE	RELEASE_COMPLETE	Release_conf

Table 4.1: Messages and primitives defined for the Signaling protocol

A simplified version of the resulting FSM is presented inFigure 4.4.

4.2.3 Protocol behavior

The FSM of the Signaling protocol uses 9 states (Figure 4.4). States are named according to the ATM document [UNI31]:

(1) U0: Null state, no connection is in progress or active;

(2) U1: Call Initiated, the ATM user requested the connection establishment;

(3) U3: Call Waiting, the ATM user waits for acceptance of the call from the called party;

(4) U10: Call Active, call is active and data transfer is allowed;

(5) U6: Call Indication, a request for call establishment is received;

(6) U8: Call Accepted, the called party accepts the call, and waits for local confirmation to transfer to U10;

(7) U9: Call Proceeding, the called party proceeds with the call;

(7) U11: Call Release requested, the ATM user requested to release the call; and

(9) U12: Call Release indication, the ATM user is informed that the peer entity requested Call Clearing.

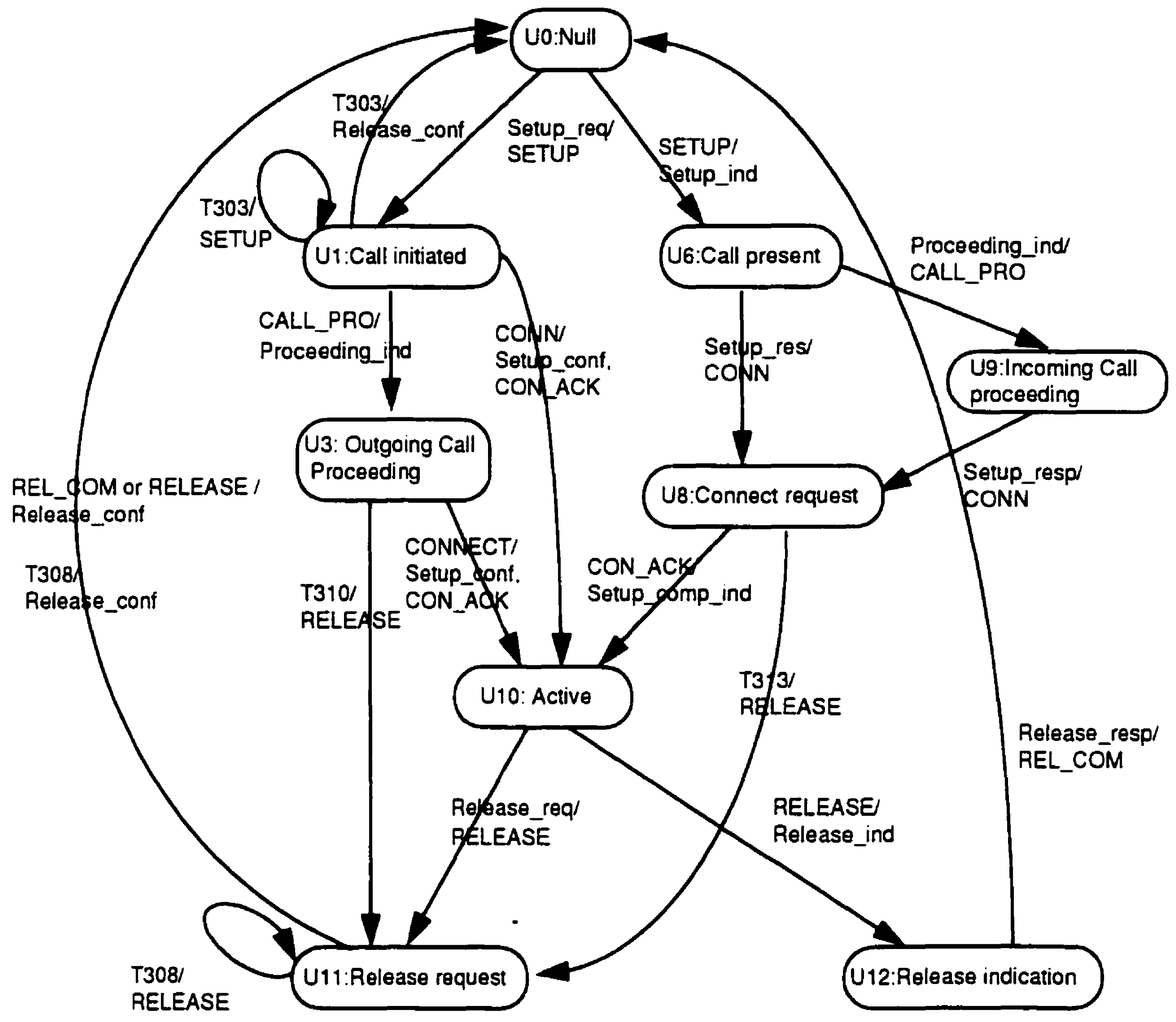

Figure 4.4: Simplified state diagram of the Signaling Protocol

The Signaling protocol supports the following basic functions at the UNI: Connection/Call Setup, Connection/Call Clearing, and Error Handling procedures.

4.2.3.1 Call Setup

The Call Setup is a functionality of the protocol that supports the establishment of connection/call between two different parties. It includes two functions: Connection/Call Request and Connection/Call Answer. (1) Call Request: allows an originating party (calling party) to request the establishment of a connection/call to a certain destination. In this request, the originating party may provide information related to the connection/call. A Call Request is initiated by the primitive Setup_req in U0 state (Figure 4.4). A unique identifier, Call Reference (CR), of the call is

generated, and a SETUP message is transferred on the signaling virtual channel across the interface. Timer T303 is started. The SETUP message shall contain all the information required by the network to process the call. If no response to the SETUP message is received by the user before the first expiration of the timer T303, the SETUP message will be retransmitted and timer T303 restarted. The second expiration of T303 initiates the clearing of the call. If the network can determine that the access to the requested service is authorized and available, the network may send a CALL_PRO message as a response to the SETUP message, indicating that the call is being processed. The user which receives a CALL_PRO message reports it to the upper layer by the primitive Proceeding_ind, stops timer T303, starts T310 and waits for the CONN or RELEASE message. If timer T310 expires before the arrival of the CONN, the call is cleared. The arrival of a CONNECT message, reported to the upper layer by the Setup_complete_ind primitive, indicates to the calling user that a connection has been established through the network. It will stop the timer T303 or T310 and send a CON_ACK message to the network. The connection is established and the protocol is in state U10 ready for data transfer (that actually happens in the U-plane);
(2) Connection/Call Answer: allows the destination party (called user) to respond to an incoming connection/call request. The destination party may include information related to the connection/call. Rejecting the connection/call request is considered as part of the Connection/Call Clearing function. At the destination, the network indicates the arrival of a call at the UNI by a SETUP message which is reported to the upper layer by the primitive Setup_ind. The user which accepts the incoming call responds with a CONN message generated by Setup_resp primitive. The user that wants to prolong the acceptance of the call (because it is busy or because it needs more processing time) eventually sends a CALL_PRO generated by Proceed_req primitive. Upon sending the CONNECT message, the user starts timer T313 waiting for a CON_ACK which indicates the completion of the ATM connection establishment for the interface and stops the timer. The connection is established and the protocol is in state U10 ready for data transfer. The expire of the timer T313 initiates the clearing.

4.2.3.2 Connection/Call Clearing

This function allows any party involved in a connection/call to initiate its removal from an already established connection/call. If the connection/call is between two parties only, then the whole connection is removed. The call clearing is initiated by the Release_req primitive which initiates the sending of a RELEASE message and the start of the timer T308. The receipt of the REL_COM message indicates the release of all resources and stops the timer T308. If the timer expires for the first time the user retransmits the RELEASE message. A second expiration initiates an implementation dependent recovery, like restart procedures, which are not considered in this work. This function also allows a destination party to reject its inclusion in a connection/call. The network or the user rejects a call by sending a REL_COM message at the originating user-network interface.

4.2.3.3 Error Handling procedures

All messages which use the protocol discriminator "Q2931 user - network call control message" must pass the checks described in the standard. Error Handling procedures are defined for each type of error that can occur in the mandatory part of the message or IEs, as anticipated by the standard. The procedures are defined for the following error conditions:

(1) Protocol discriminator error: a message with a protocol discriminator other than "Q2931 user - network call control message" is received;

(2) Message too short: a message with the length shorter than the minimum message length (the common part of the message must be present);

(3) Call Reference Error: a number of CR errors are defined related to the invalid CR value, CR value indicating inactive call, CR with a global CR value, etc.;

(4) Message type or message sequence errors: a message with undefined type is received or an inopportune message is received in some state;

(5) MIEM Error: a message with the MIEM is received (see the previous section);

(6) MIECE Error: a message with the MIECE is received;

(7) UIE Error: a message with the UIE is received;

(8) NMIECE Error: a message with the NMIECE is received;

(9) RUIE Error: a message with the RUIE is received.

The Error Handling procedures depend on the type of error condition, state in which the error occurred and type of message. When a message with the error types (5) or (6) is received, a typical Error Handling procedure is to respond to a peer party with STATUS message having the Cause IE value equal to 99 or 101 respectively. In the case of error conditions (7), (8), and (9), in general, the protocol should ignore the error and try to process the message using the correct IEs. The protocol should process error conditions in the order of precedence: error conditions higher in the list have a higher priority than subsequent conditions. Therefore, when a message with more than one error type is received, only the Error procedure corresponding to an error condition with the highest priority is executed.

4.3 SDL specification of the protocol

The Signaling protocol specification in SDL was developed in two steps:

(1) In the first step, we have constructed a state table based on the standard document. The state table describes, for each state: the valid inputs; the verification made on the input parameters; and depending on the result of verification, the actions to be taken and the next state of the transition;

(2) From a state table, in the second step, an SDL specification is obtained by developing an SDL system. In addition to the FSM part, the SDL system includes appropriate data structures, input parameters, and parameter checking procedures. To obtain a complete SDL specification, the system was enhanced to support multiple connections. Developments in this step were done independently from this work and are explained in more detail in [Marc97].

4.3.1 State table

In the process of state table development, the following assumptions were made:

(1) An assured mode signaling AAL connection is already established between the user and the network prior to the start of sending the Signaling messages. The underlying SAAL layer is assumed to be reliable and no malfunctioning is experienced during the protocol functioning because of the lost SAAL connection;

(2) STATUS ENQUIRY message is initiated by Initiate Status Enquiry SAAL primitive: It is used to simulate the Status inquiry procedure, originally started by the SAAL layer malfunction;

(3) Restart procedures are not included;

(4) Only a point-to-point connection is considered.

The resulting state table has 56 rows and 9 columns. Since the state table is used as a starting point for the design of the SDL system and not for test derivation, we did not include the state table in this text.

4.3.2 SDL system structure

The SDL system representing the Signaling protocol contains one block with two processes: Proc, which models the behavior of a connection and Coord, which manages calls and routes messages and primitives to the corresponding Proc process (Figure 4.5).

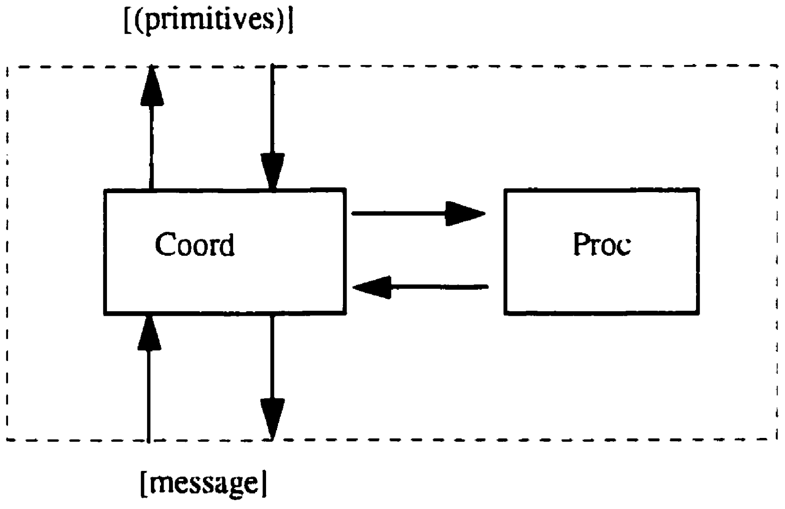

Figure 4.5: Signaling protocol implemented as SDL system

The purpose of the Coord process is to multiplex a number of connections realized with one or more Proc processes. For each initiated incoming or outgoing call, Coordinator assigns a locally unique CR, and creates an instance of the Proc process. An incoming call has a CR already assigned (the flag, a field in the CR, identifies whether the call is incoming or outgoing). The Coordinator realizes functions that are common for all protocol instances. When the Coordinator accepts a call, it creates an instance of the Proc process and associates the newly created process with the CR of the call. When a PDU from the SAAL or ASP from the upper SAP arrives, the Coordinator finds the protocol instance to which PDU or ASP should be sent using the association between the process identifiers and the connection CR. In performing its function, the Coordinator only inspects the first four common fields of the message. The fields are checked for errors and if they are correct, only the CR and the variable length information elements are forwarded to the corresponding Proc process. The primitives are redirected from the Coordinator to a corresponding process instance without any processing.

The Proc process realizes the majority of the Signaling protocol functions. Compared with the Coord process, which is a single state FSM, Proc comprises 9 states (U0,U1,U3,U10,U6,U8, U9, U11, U12). In addition to executing the protocol, Proc verifies each message it receives. It checks for any of the Error types defined in Section 4.2.1 and performs the required procedures if any of the error conditions from the Section 4.2.3.3 are met.

4.3.3 Data type declarations

The message content is represented by a set of complex data types. The complexity comes from the number of IEs and their fields as well as the combination that data types should model. In spite of this, data types cannot be declared as static data types because the size and the structure of the messages and IEs are unknown at the "compile" time. For example, since the size and structure of the SETUP message may vary depending on the information that it conveys (some IEs may be absent, other have flexible structure), the corresponding data types will have to be dynamically modified. In our approach, we defined a "general" message that is a superset of all possible messages. Using the Boolean fields as flags, we include or exclude fields (IEs or parts of the IEs) from the "general" message to represent the message we want. However, since this "general" message is declared statically, each message that is constructed from it will contain all IEs (even IEs that it does not need), which makes it long and difficult to manipulate.

The messages exchanged at the lower SAP between the Coordinator and the adjacent layer are represented with a signal message. The parameter of the signal message is a structure realizing the "general" message. It contains four fields: the first three are the common message fields and the fourth is a structure representing the sequence of IEs, see Example 4.1. The ie_type data type by itself is a structure that contains all possible IEs. Each IE is represented by a data type that has an associated Boolean flag field that indicates whether the IE is present or absent. When an IE is absent, the Boolean flag field has a value True, and, consequently, when IE is absent, the Boolean flag has a value False.

Example 4.1: Definition of the signal message and declaration of the "general" message structure:

```
SIGNAL message(message_content_type)

NEWTYPE message_content_type STRUCT
      pr_dis protocole_descrimator_type;
      CR call_reference_type;
      message_type octet;
      ie ie_type;
ENDNEWTYPE;
```

For each IE, we declare a data structure with fields from corresponding data types, as defined by the standard. The optional fields are declared as structures with a Boolean field indicating the presence or absence of the field in the IE, see Example 4.2.

Example 4.2: Connection identifier IE has one optional field class_options and is declared as:

```
NEWTYPE  B_BC_type  STRUCT
      presence boolean;
      coding_standard natural;
      class natural
      class_options class_options_type;
      clipping natural;
      user_plane natural;
ENDNEWTYPE;

NEWTYPE class_options_type STRUCT
      presence boolean
      traffic_type natural;
      timing_req natural;
ENDNEWTYPE;
```

The IEs that may be repeated within a message are declared as a structure containing two fields (see Example 4.3):

(1) the first one indicates the number of occurrences of the IE within the message;

(2) the second is an array of that IE.

Example 4.3: Broadband low layer information IE may have at maximum three occurrences within the same message. Thus, it is declared as:

```
NEWTYPE  B_LLI_type  STRUCT
      nb_occ max_occ_B_LLI;
      occ B_BLLI_table;
ENDNEWTYPE;

NEWTYPE B_LLI_table
      array(max_occ_B_LLI, B_LLI_ie);
ENDNEWTYPE;

SYNTYPE max_occ_B_LLI = NATURAL
      CONSTANTS 0:3
ENDSYNTYPE;

NEWTYPE  B_LLI_ie  STRUCT
coding_standard int_2;
      info_layer_1 optional_value_type;
      info_layer_2 layer_2_type;
      info_layer_3 layer_3_type;
ENDNEWTYPE;
```

The parameters of the primitives exchanged with the upper layer are CR (except for Setup_req) and a structure containing the IEs of the corresponding message. For example, since the only information element allowed in a RELEASE message is Cause, Release_req and Release_ind have as parameters the CR identifying the call and a structure containing only one field: the Cause IE.

The constructed SDL specification of the Signaling protocol consists of almost 10 000 lines of SDL code [Marc97]. The data declaration part by itself is almost 70% of the code. As explained in Section 2.2.1, a protocol specification should be verified against the service specification of the protocol layer. The UNI 3.1 does not have a protocol service specification, and, therefore, the SDL specification could not be verified in this sense. However, as discussed in Section 5.7, the specification and the test suite developed later are validated to a certain extent one against the other.

Chapter 5

Development of a conformance test suite

5.1 Overview

In this work, we use the experimental tool chain of UdeM (Section 3.3) to automate the development of a conformance test suite. A complete SDL specification of the Signaling protocol has been developed from the standard document prior to the test development. In a first step, the FEX tool is used to extract the FSM model from the SDL specification. The FSM model is described in the form readable by the TAG tool. In a second step, the TAG tool is used to generate a test suite from the FSM model. The output of the tool is a test suite in a mnemonic or SDL skeleton format. In a third step, in order to produce SDL executable test cases, test cases are manually completed with parameters. In this step, PDUs for parameter variation are also constructed and used to complete the corresponding test cases. The obtained test suite is validated in the SDL environment against the SDL specification. The validated test suite is an abstract test suite; it can be used for deriving C executable test cases.

5.2 Extracting an FSM model from the SDL specification

The FSM model of the Signaling protocol used for test development is produced by the FEX tool from the SDL specification. Before we take a closer look at the application of the tool chain, we explain the process of unfolding inputs in the obtained FSM model by the FEX tool. According to the standard, the Signaling protocol has 8 inputs, 8 outputs and 9 states. However, the produced FSM model has 56 inputs, 45 outputs, and 9 states. The main reason for the increase in the number of input/output events is due to the partial unfolding of input messages according to their parameters.

Example 5.1 shows a typical case of generating FSM transitions. The SDL code in the example is translated by the FEX tool into the FSM transitions readable by the TAG tool. The transition `U0 ?SETUP !Setup_ind >U6;` means that when the specification receives the input SETUP in state U0, it sends the output Setup_ind (which is a primitive to the upper layer) and goes to state U6. In state U0 the SDL specification may also receive the input Setup_req (which is a primitive from the upper layer). In this case it sends the output SETUP and transits to state U1; this generates the second transition in the FSM.

Example 5.1:

```
SDL specification:
state U0;
      input SETUP;
      output Setup_ind;
      nextstate U6;

      input Setup_req;
      output SETUP;
      nextstate U1;

FSM transitions generated:
U0 ?SETUP !Setup_ind >U6;
U0 ?Setup_req !SETUP >U1;
```

Receiving the SETUP message in U0 state may result in different protocol behavior depending on the content of the SETUP message (or more precisely, the value of the SETUP parameters). If a SETUP message has valid values of the parameters (a valid SETUP message), the protocol should proceed with a Setup_ind to the upper layer. In the case when SETUP message has an invalid content (as explained in Section 4.2.1), the protocol should determine the error category (MIEM, MIECE, etc.) and perform the correct action as prescribed by the standard document. As Example

5.2 shows, in the SDL specification this behavior is realized using the checking procedure and a decision statement. The check_setup procedure checks the content of the SETUP parameter setup_data and returns the result in the variable error_code. In the FSM model, this SDL code is represented by a number of different transitions, where the inputs are conditioned by a predicate related to the message content. The input SETUP was partially unfolded according to its input parameter values in four distinct inputs: SETUP with a valid content (written simply as SETUP), SETUP with a MIEM (written as SETUP(Mie=1)), SETUP with MIECE (SETUP(Mie=2)), and SETUP with UIT, NMIECE, or RUIE (SETUP(NMie=1 | Nmie=2 | Nmie=3)) (see Section 4.2.1).

TAG treats these inputs as four different inputs, and, consequently, in the derived test cases, they are considered as different test events. Later, when the TAG produces the test cases, the test developer is required to complete test cases in the SDL skeleton form with the corresponding values for the SETUP message. Since test events are conditioned with the predicates, the test developer should determine the values of the parameters that will produce the intended behavior, since the required parameter values are not produced by the tools.

Example 5.2

SDL specification:
```
state U0;
      input SETUP(setup_data);
      check_setup(setup_data, error_code)
      decision  error_code;
            (MIEM): output REL_COM(96);
                  nextstate U0;
            (MIECE): output REL_COM(100);
                  nextstate U0;
            (UIE,RUIE,NMIECE): output Setup_ind(setup_data);
                  nextstate U6;
            (OK): output Setup_ind;
                  nextstate U6;
      enddecision;

      input Setup_req;
      output SETUP;
      nextstate U1;
```

FSM transitions generated:
```
U0 ?SETUP !Setup_ind >U6;
U0 ?SETUP(Mie=1) !REL_COM(ca=96) >U0;
U0 ?SETUP(Mie=2) !REL_COM(ca=101) >U0;
U0 ?SETUP(NMie=1 | Nmie=2 | NMie=3) !Setup_ind >U0;
```

The above process of unfolding is applicable to every input message. The extracted FSM model of the protocol is given in Appendix A.

5.3 Test method

The test development process used in this work assumes a local test method. The UT has control over the upper PCO. The test events exchanged between UT and IUT on the upper PCO are primitives defined in the previous chapter. LT has control over the lower PCO. The test events exchanged between the tester and the IUT are messages (PDUs) as defined in Table 4.1.

The local test method is the most appropriate architecture for the test development method implemented by the tool chain. Since the test development starts from the SDL specification, which describes the protocol behavior using the events at the local interfaces (immediately above and below the Signaling protocol), derived test cases are realized with the same events. Therefore, the tester that implements such a test suite must have access to the upper and lower PCO; in other words, it has to be a local tester.

Because a test suite for the local test method requires control over local PCOs that are not always accessible, the local test method is mostly applicable for in-house testing. In this project, we derive tests for this test method since, according to the technical requirements of the project, tests should be mainly used during the development of an ATM card for PCs.

5.4 Identification of test purposes for the control part of the protocol

To have proper fault coverage, a test suite should check as many properties of a protocol as possible. To test the "control" properties of the Signaling protocol, we assign a test purpose to each transition of the corresponding FSM model. Test purposes are represented by transitions covering the control part of the protocol. Test purposes for the data part of the protocol are identified according to the textual description in the standard.

Figure 5.1 shows an example test purpose expressed as a transition of the FSM as well as the corresponding test case in mnemonic and SDL skeleton form. The mnemonic form represents the test case as a sequence of the input/output events. Events with the ! sign in front of their names are outputs from the tester; events with the ? sign are expected inputs to the tester. A test case consists of a preamble that leads from the initial state to the head state of a transition (in the example in Figure 5.1, there is no need for any preamble because the head state is the initial state), transition under test, and postamble, driving the IUT back to the initial state. State identification of the tail state is optional, once used it provides identification of the tail state and guaranteed fault coverage.

The SDL skeleton is the SDL code representation of the mnemonic form of a test case. The verdicts in the SDL skeleton are generated by the TAG tool. The fail verdict is assigned if the tester does not receive the expected output from the IUT. For the example in Figure 5.1, if on output Setup_req in state wait_Setup_in_U0, the tester receives a signal different from SETUP, it will give a fail verdict. However, if the tester never reaches the fail label, the result of the test run is a pass verdict. To be used as an executable SDL test case, the SDL skeleton in Figure 5.1 has to be extended with procedures for parameter checking and timers that limits a tester waiting time on the IUT output. Introducing procedures and timers, the test developer also augments the test cases with appropriate verdict assignments. A corresponding complete test case in SDL is shown in Figure 5.4.

Test purpose:

U0 ?Setup_req !SETUP >U1;

Test case in mnemonic form:

U0 on input Setup_req */
!Setup_req;
?SETUP:PDU:
/* Transition under test in state
/* Identifying U1 state */
!STATUS_ENQ:PDU;
?STATUS(ca=30, cs=1):PDU;

/* Postamble from U1 State */
!REL_COM:PDU;
?Rel_conf;

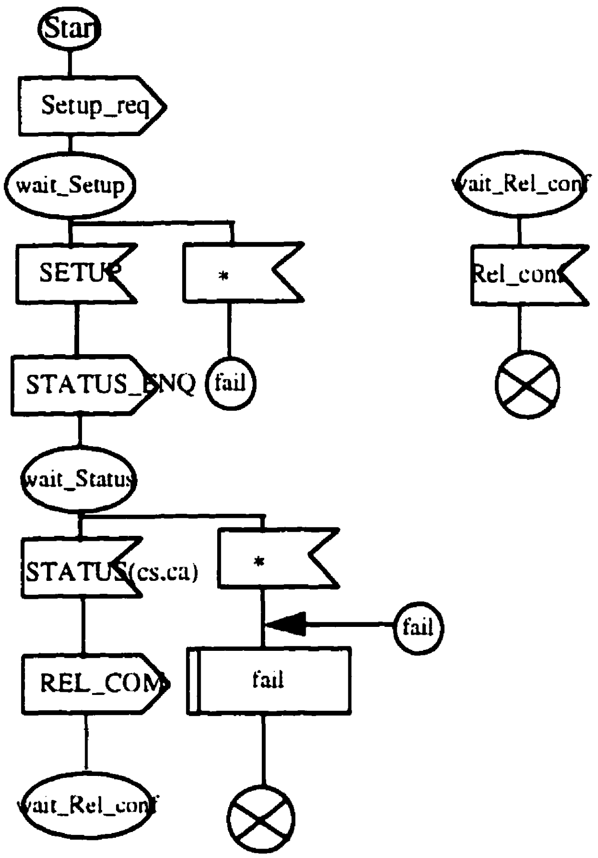

Figure 5.1: A test purpose and test case in mnemonic and SDL skeleton form

5.5 Application of the tool chain

Ideally, it should be possible to derive tests in a completely automatic way using a tool. While, with the existing tool chain at UdeM, we derived the majority of tests automatically, a totally automated process is not yet possible because manual modifications were necessary to the intermediate results. In this section, we present our results of applying the tool chain to the SDL specification of the Signaling protocol.

By a direct application of the tool chain we mean an automated test derivation process where the output of one tool is supplied unchanged by the test developer to the input of another. In this process, there is no manual modification to the SDL specification or to any intermediate results. The goal is to produce a test suite directly from the SDL specification. The direct application follows the methods described in Chapter 3. In this process, the FEX tool is applied to the SDL specification to extract the FSM model of the protocol. Using the extracted FSM, TAG derives preambles and postambules for each state. A complete test suite is developed with the TAG option for complete test suite generation. The resulting test suite - called a basic test suite - has 197 test cases. The derived basic test suite was incomplete in the sense that certain aspects of the protocol were not tested with the test suite derived.

Its incompleteness comes from tool limitations. Three problems have been identified.
(1) The output signals that have parameters in the SDL specification are translated to FSM output events without parameters; in other words, output events of the same type with different parameters are not distinguished in the FSM. While this is not related to any protocol property, it makes states of the protocol indistinguishable and TAG cannot find a state identifier. A state identification sequence for the Signaling protocol is STATUS ENQ message: on input of a STATUS ENQ, protocol replies with a STATUS message whose parameter identifies the current state of the protocol. If the parameter that identifies the current state is not part of the FSM output, no state can be recognized.
(2) FEX does not include timers in the FSM model in the format understandable by TAG, so, no test cases were generated for timers;
(3) FEX accepts an SDL specification that consists only of one process. However, as explained in the previous chapter, our SDL system consists of two processes, Coord and Proc. Since the majority of protocol functions were implemented in the Proc process, FEX was applied to the SDL system consisting of this process only. As a result, the FSM model did not include the Coordinator transitions, therefore, no test cases were generated for them by TAG. As well, without the Coordinator, U0 state cannot be distinguished from other states. The input of STATUS ENQ in U0 is processed by the Coordinator, and, therefore, it is an unspecified input in the FSM model composed only of the Proc functions. The basic test suite does not use a state identification facility. To complete the basic test suite, manual modifications are necessary.

The previous problems are solved by modifying the FSM model produced by the FEX tool. To avoid manual development of tests, we model as much of the missing protocol behavior as possible with the FSM and use the capabilities of the TAG to develop test cases. However, we are still forced to develop a small number of test cases manually.

FEX does not include timer actions in the FSM model. Since TAG is capable (with some restrictions) to generate test cases for the timer actions, timer transitions have been added to the FSM. To test timers, transitions representing start, stop, and final expiration of the timers were added to the FSM. Table 5.1 shows the number of starting and stopping transitions per timer. Each timer has one transition representing the final expiration of the timer. Test cases for each timer were generated using the selective test generation option of TAG, giving in total 100 additional test cases for timers.

Timer	# starting transitions	# stopping transitions
T303	1	21
T310	3	23
T313	2	18
T308	44	11
T322	9	9
Total	50	73

Table 5.1: Timers

When the number of transitions from Table 5.1 is compared with the number of newly generated test cases, it is obvious that there are more transitions than test cases generated. TAG tool does not produce test cases for transition that is starting transition for one timer and stopping transition for another timer at the same time. For example, Figure 5.2 represents a transition that is a stopping transition for timer T303 and a starting transition for timer T310. When a Call_Pro message is received in U1 state, T303 is stopped, and, at the same time, T310 is started. In this case, TAG tool generates a test case for the start of T310, but there is no test case for the stop of T303. This missing test has to be developed manually. In the protocol, there are 26 transitions that are transitions with two timers, and consequently, 26 test cases have to be written manually.

```
U1 ? Call_Pro !Proceeding_ind > U3, stop T303, start T310
```

Figure 5.2: Transition with two timer actions

When tests for timers are developed, TAG assumes that there is only one active timer per state and that the starting transition of the timer is not a looping transition. However, for timer T322 both of these assumptions are violated. T322 is active during the Status Enquiry procedure that can be initiated in any state even when another timer is active. Also, when Status Enquiry message is sent, T322 is started and the starting transition loops once because the protocol remains in the same state. In the Signaling protocol, there are 27 transitions related to T322 timer (since T322 may be active in any state it has 9 transitions related to the final expiration), and we could not use tools to develop test cases for them. We have manually designed test cases for T322 only in states where no other timer is active (U0, U10, U6, U9, U12). Altogether, for 53 transitions out of 153 related to timers, TAG does not produce test cases, and test cases related to some of these transitions are designed manually (26 for transitions with two timers and 15 for T322). Appendix C shows example test cases for timers.

Coordinator transitions, as mentioned previously, cannot be produced by the FEX. From the point of view of the SDL specification, Coordinator is considered as a separate module. The functions it provides are common to all states and all protocol instances. To produce a single FSM that includes the Coordinator, the FSM model of the Coord process is composed manually with the FSM of the Proc process. If a reliable SAAL connection is assumed, Coordinator is a single state machine. This corresponds to composing two FSMs into a global machine. Since the Coordinator FSM has one state, it is reduced to specifying a few transitions in the Proc FSM which were previously undefined ("don't care transitions"). We call these transitions Coordinator transitions. As a result, the augmented FSM has 570 transitions instead of 196 transitions.

Since Coordinator transitions are coherent transitions, it is natural to try to group them and to reduce the number of generated test cases. There are 12 types of Coordinator transitions, and, after grouping, the 374 Coordinator transitions are represented with 12 coherent transitions. Figure 5.3 shows an example of a coherent Coordinator transition. If messages CONN, CON_ACK, CALL_PRO, or RELEASE are received with CR related to an inactive call (parameter ge=4 represents the condition of having CR that refers to a non-active call) in any state, REL_COM with the cause IE value equal to 81 should be returned, and the protocol remains in the same state.

```
* ? {CONN(ge=4), CON_ACK(ge=4), CALL_PRO(ge=4), RELEASE(ge=4)}!
REL_COM(ca=81) > _
```

Figure 5.3: A coherent transition

Twenty four test cases for the coherent transitions have been developed using the TAG selective test derivation option. Coherent Coordinator transitions and corresponding test cases are shown in Appendix B.

When the Coordinator transitions are added to the protocol FSM, U0 becomes distinguishable from other states, and a test suite with state identification can be produced. Using the global FSM model, TAG is used to generate a new test suite with the state identification facility. The existence of the STATUS ENQ message in the protocol helps the testing of the protocol and enhances the fault detection power of the test suite without significantly increasing the cost of the testing. Since the state identification facility is of length one, the test suite has 320 test cases, the same number as the previous one. The complete test suite, together with the test cases for T322 and transitions with two timers that have been developed manually, has in total 320 + 26 + 15= 361 test cases.

An analysis of the obtained FSM has shown that an additional optimization of the test suite length is possible by grouping certain transitions. A number of transitions related to the Setup Inquiry and Call Clearing phase are coherent transitions. For example, when the STATUS message is received in any state with the value of the message parameter Current State equal to U0, the protocol should release the connection, and transfer to U0 state. If these transitions are represented as coherent transitions, the resulting alternative test suite has 274 + 26 +15 = 315 test cases.

In summary, by the direct application of the tool chain, a test suite with 196 test cases is generated automatically. When timer and Coordinator transitions are added to the FSM, 124 additional test cases are developed using the TAG tool. Finally, 26 test cases related to transitions with two timers and 15 test cases for T322 are developed manually. The resulting conformance test suite has a complete fault coverage in terms of the FSM model. The test suite is represented in mnemonic as well as SDL skeleton form (See Appendix C for an example test case in the mnemonic form).

5.6 Test purposes and tests for the data part

Test suites represented in the SDL skeleton form do not have parameters. The need to complete them with parameters is twofold: to produce SDL executable tests and to test the data part of the protocol. The data part of the Signaling protocol specifies the PDU structure and valid values for the PDU fields (parameters). An implementation should accept PDUs with parameters with valid values and should have correct error handling behavior for a PDU with invalid parameters. The exhaustive testing of the data part of the protocol would require generation of all possible valid PDUs and all possible invalid ones. The number of parameters in the ATM Signaling protocol is quite large. Additionally, the parameter value and its existence in a message may depend on the value of other parameters. Therefore, exhaustive testing of the data part of the Signaling protocol is not feasible.

To test the data part, we use an alternative approach by choosing only a representative set of values of parameters including both valid and invalid values. In this respect, two kinds of problems are addressed. The first is related to the generation of sets of representative values from determined data types; the second involves the organization of these sets of values into value tuples (actually PDUs), which can be used in a single test case.

Representative valid and invalid values are determined for each parameter. Parameters that are integers or a range of some data type are represented with three values: minimum, maximum and some random value from the range. Representative invalid values are values immediately outside the range [ISO9646]. Parameters from this type are usually defined as 16-bit or 4-bit integers in the ATM UNI 3.1 document. Parameters that are defined by enumeration (have a set of discrete values) are represented using all of their defined values. Representative invalid values are some random values outside the set of defined values. Most of the parameters in the ATM UNI 3.1 are of the second type.

The second problem is to organize parameters in tuples (PDUs). The question is which kind of PDUs we want to use for testing. Since we cannot test all possible PDUs, we need a criterion according to which we determine the type of PDUs important to test. In our approach, we divide the domain space of all possible PDUs into equivalence classes and try to cover each of the equivalent classes with a number of representative PDUs. The equivalence class is a set of PDUs that have some common properties (they are all valid, or have the same type of error) [Myer79]. We divide the domain space into six classes: valid PDUs, PDUs with MIEM, PDUs with MIECE, PDUs with UIE, PDUs with NMIECE, and PDUs with RUIE. For each of theses classes, a number of PDUs are constructed that are used to test the IUT.

To represent valid P

DUs, a number of PDUs with correct values are constructed. These PDUs are constructed to cover all correct values of the parameters, determined in the previous step. The set of generated PDUs contains all representative values. In order to reduce the number of required PDUs, the values of the parameters are varied in parallel. For example, the ATM Traffic Descriptor (ATMTraff) consists of the following parameters:

Parameter	Parameter Full Name	Values	# representative values
FWD_PCR0	Forward Peak Cell Rate with 0	INTEGER 0 - 2^24	3
FWD_PCR1	Forward Peak Cell Rate with 1	INTEGER 0 - 2^24	3
BWD_PCR0	Backward Peak Cell Rate with 0	INTEGER 0 - 2^24	3
BWD_PCR1	Backward Peak Cell Rate with 1	INTEGER 0 - 2^24	3
FWD_SCR0	Forward Sustainable Cell Rate with 0	INTEGER 0 - 2^24	3
FWD_SCR1	Forward Sustainable Cell Rate with 1	INTEGER 0 - 2^24	3
BAK_SCR0	Backward Sustainable Cell Rate with 0	INTEGER 0 - 2^24	3
BAK_SCR1	Backward Sustainable Cell Rate with 1	INTEGER 0 - 2^24	3
FWD_MBS0	Forward Maximum Cell Rate with 0	INTEGER 0 - 2^24	3
FWD_MBS1	Forward Maximum Cell Rate with 1	INTEGER 0 - 2^24	3
BAK_MBS0	Backward Maximum Cell Rate with 0	INTEGER 0 - 2^24	3
BAK_MBS1	Backward Maximum Cell Rate with 1	INTEGER 0 - 2^24	3
BEST_EFFORT	Best Effor Tag	1	1
FWD_TAG	Forward Tagging	0 or 1	2
BWD_TAG	Backward Tagging	0 or 1	2

Table 5.2: Parameters of the ATM Traffic IE

The standard prescribes the allowed combinations of the parameters. The allowed combinations in the forward direction (they are same for the backward direction) are:

Combi 1	Comb. 2	Comb. 3	Comb. 4	Comb. 5	Comb 6
FWD_PCR0	FWD_PCR0	FWD_PCR1	FWD_PCR1	FWD_PCR0	FWD_PCR1
FWD_PCR1	FWD_PCR1	FWD_SCR0		FWD_PCR1	FWD_SCR0
	FWD_TAG	FWD_MCR0		FWD_TAG	FWD_MCR0
					FWD_TAG

Table 5.3: Allowed ATM Traffic IE combinations in the forward direction

The representative valid values for FWD_PCR0 and FWD_PCR1 are (0, 2^8, 2^24). To test valid values for parameters FWD_PCR0 and FWD_PCR1, the parameters of the ATMTraff IE may be initialized like in Table 5.4. All other parameters that are not shown in Table 5.4 are missing from the IE (they do not exist in the IE). The parameters are varied in parallel, i.e. they are initialized independently. If one of the parameters has more values, the others are initialized with some default values (for example, they retain the value from the last combination).

Parameter	ATMTraff 1	ATMTraff 2	ATMTraff 3
FWD_PCR0	0	2^8	2^24
FWD_PCR1	0	2^8	2^24
BWD_PCR0	0	2^8	2^24
BWD_PCR1	0	2^8	2^24

Table 5.4: Three ATM Traffic instatiations covering values for FWD_PRC0 and FWD_PCR1

To test all representative values in ATMTraff we need:

6 (combinations) X 3 (maximum number of representative values for any combination)

= 18 (instantiations of ATMTraff IE)

Since ATMTraff is part of the message, 18 SETUP PDUs are constructed to include 18 ATMTraff instantiations.

PDUs with invalid parameters are constructed according to the following rules:

(1) PDUs with MIEM: For each message type, PDUs with one missing mandatory IE are generated. For example, a SETUP message has four mandatory IEs, thus four SETUP PDUs are generated with a single mandatory IE missing at a time;

(2) PDUs with MIECE: The representative PDUs from this class have one erroneous parameter at a time. Parameters with invalid values are not varied in parallel. If one parameter in the PDU has content error, all other parameters in the PDU should have valid values. This approach helps us to avoid confusion as to which error value has caused such a behavior;

(3) PDUs with UIE: For each message type, a message with one unrecognized IE is generated;

(4) PDUs with NMIECE: These PDUs are generated in the same way as PDUs with MIECE. The difference is that the IEs having the content error are non mandatory for the message;

(5) PDUs with RUIE: For each message type, a message with one extra valid IE is generated.

We have constructed 25 PDUs to test the ability of an IUT to recognize valid PDUs and 115 PDUs to test the ability of IUT to detect an error in PDUs and the type of the error. Table 5.5

groups the 115 PDUs by the type of message and error. Appendix D contains SDL examples of SETUP message with valid values, with MIEM, and with MICE.

Message	MIEM	MIECE	UIE	NMIECE	RUIE
SETUP	4	15+2+2+5=24	1	13+2+14+2=31	1
CONN	1	2	1	13+2=15	1
CONN_ACK	1	1	1	0	1
CALL_PRO	1	2	1	2	1
RELEASE	1	2	1	0	1
REL_COM	1	2	1	2	1
STATUS ENQ	1	1	1	0	1
STATUS	2	2+1=3	1	0	1
Total:(115)	12	37	8	50	8

Table 5.5: Number of messages representing the five classes of invalid PDUs

Test cases for the data part are constructed by selecting the appropriate test cases from the test suite and completing them with the PDUs from the representative sets. In general, this requires the same test case to be repeated with different values for its parameters. An example of a complete test case is given in Figure 5.4.

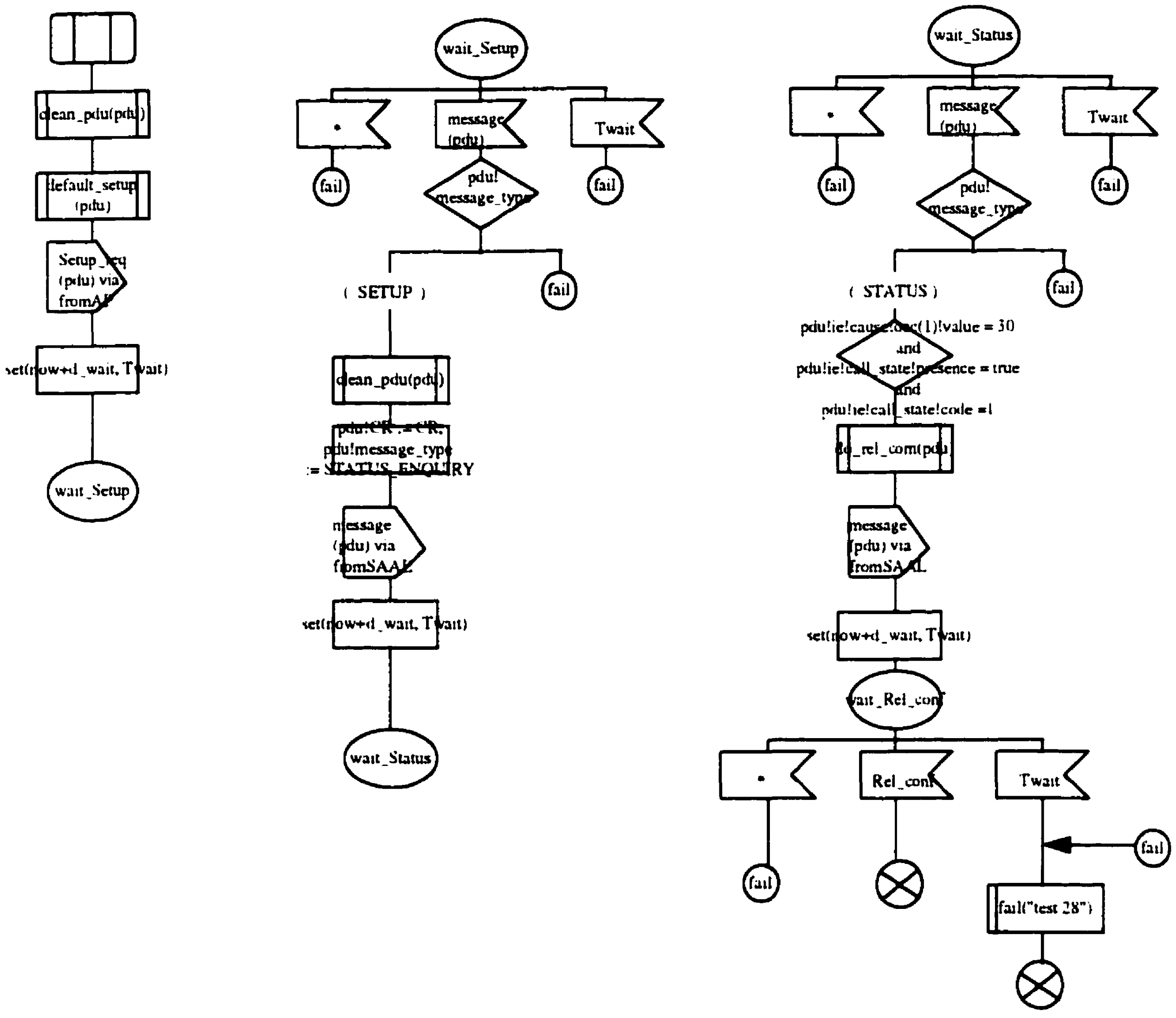

Figure 5.4: A complete test case for the SDL skeleton in Figure 5.1

5.7 Validation of the conformance test suite

5.7.1 Objectives of validation

The last step in the test development process is validation of the developed test suite. The objective of the validation should first be clarified. In the validation process, the complete test suite in SDL is executed against the protocol SDL specification in an SDL simulation environment. This activity is similar to the conformance testing procedure. Instead of having a "black box" IUT, we use the SDL protocol specification as an IUT, and, instead of a real tester, we have an SDL realization of the local tester that implements the test suite as a set of SDL procedures. The "execution" activity is performed by an SDL simulation tool, firing the SDL transitions one by one. For this purpose, we used commercial SDL tools SDT and Geode. A properly designed test suite complete with parameters test suite should give a pass verdict for the specification. In this case, we consider the test suite as validated (or correct). While the objective of the validation process is to check the correctness of the test suite, it may discover some errors in the SDL specification as well. Therefore, the validation process can be considered as a mutual assessment of the correctness of the test suite and the SDL specification.

Two problems concerning the test validation process have to be addressed:
(1) When a test case gives the fail verdict, we have to determine where the error is: in the test suite or in the SDL specification. In the case when the SDL specification can be trusted to be error free, the conclusion is that there is a mistake in the test suite. When problems are detected, errors are located by proofreading the SDL code of the specification as well as the test suite SDL code, and the code or the test case is corrected. The SDL graphical environment provides an excellent means to locate and to resolve these problems.

(2) Error detecting capabilities of the validation process: The test suite is based on the FSM model derived from the SDL specification. If there is an error in the control part (FSM part) of the SDL specification, the same error will be in the test suite, and, consequently, it will never be discovered by the validation process. Using the test development process introduced in this work, we cannot detect such errors in the test suite. In our project, the correctness of the SDL specification is ensured by checking the equivalence of the two FSMs: the one, manually derived directly from the ATM document (see Section 4.3.1) and the other, obtained by the FEX tool (see Section 5.2).

The validation process may also detect errors in the data part of the protocol. PDUs for parameter variation are generated from the standard document, and procedures for constructing PDUs and checking PDU parameters in the SDL specification are developed independently. Besides errors in the data part, there are other errors than could be discovered during the validation process. The SDL specification contains code that performs integration of the Coord process and protocol instances, such as support of multiple connections, creating and terminating new Proc instances (as new connections are opened and closed), dispatching primitives and messages from/to the Coord process and protocol instances, etc. These functions are not part of the standard but are necessary in order to have a complete functional specification of the Signaling protocol.

5.7.2 SDL validation system

The SDL system used for the validation process is shown in Figure 5.5. It consists of a Tester block and a Q2931 block (the Signaling protocol specification). They are connected by four channels.

The Tester process integrates the UT and the LT in a single SDL process TestSuite. Test cases, inside TestSuite are realized as procedures. The completed example test case from Section 5.1 is shown in Figure 5.4. A timer T is added to limit the maximum waiting time of the Tester. For example, if the specification does not reply at all, the Tester will time-out, and send a fail verdict.

5.7.3 Results of the verification

During the validation process, errors in earlier versions of the SDL specification as well some errors in the test cases were discovered. Most of the discovered errors are related to the data part of the protocol (procedures for checking the content of the PDUs). There were few errors in the Coordinator (e.g. one related to Call Reference assignment) and in the integration of the Coordinator and the protocol instances. No error was found in the FSM part of the protocol. Errors that were found during the validation were located and corrected by proofreading the code of the tester and the SDL specification.

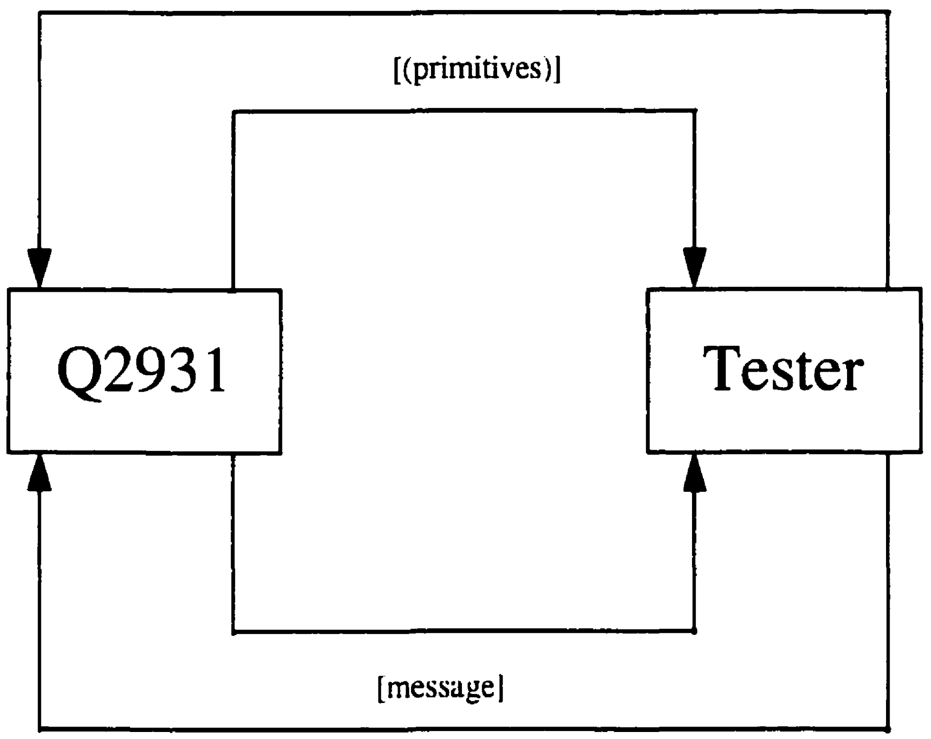

Figure 5.5: Test validation architecture

5.8 Test grouping

It is common practice to organize a test suite in the hierarchical, tree-like structure. Usually, a test suite is organized around the protocol functions or phases in order to facilitate maintaining the test suite and selecting test cases (Section 2.3.3). The test suite produced with the TAG tool is not grouped or structured. It is generated as a "flat" sequence of test cases following the rules of FSM-based testing, as explained in Section 3.1.3. In order to produce a hierarchical structure, test cases are manually grouped in the test groups. Test groups are identified according to the protocol property they focus on.

The test suite structure is shown in Figure 5.4. The leaves and nodes represent the test groups. The numbers in brackets represent the number of test cases in the corresponding group. The test suite is divided into two main groups: Valid and Invalid behavior tests. As explained in Section 4.2.3, the Signaling protocol valid behavior is classified in the following phases: Call Request, Call Answer, Call Clearing. Test cases that test transitions from one of these phases are grouped into corresponding test groups: Call Request test group, Call Answer test group and Call Clearing test group. Together with tests for timers and Status Enquiry Procedure, these tests form the Valid Behavior test group.

Test cases that form the Invalid Behavior test group are composed of cases that test the Error Handling procedures (Section 4.2.3.3). They are organized in groups according to the Error procedures they test. The General Error group consists of cases that test the IUT behavior when one of the error conditions occurs: a message is received with a protocol discriminator error, it has a message length error, or it has a CR error. These procedures are part of the Error Handling, but since procedures associated with these errors are realized by the Coordinator, they are organized as a separate group.

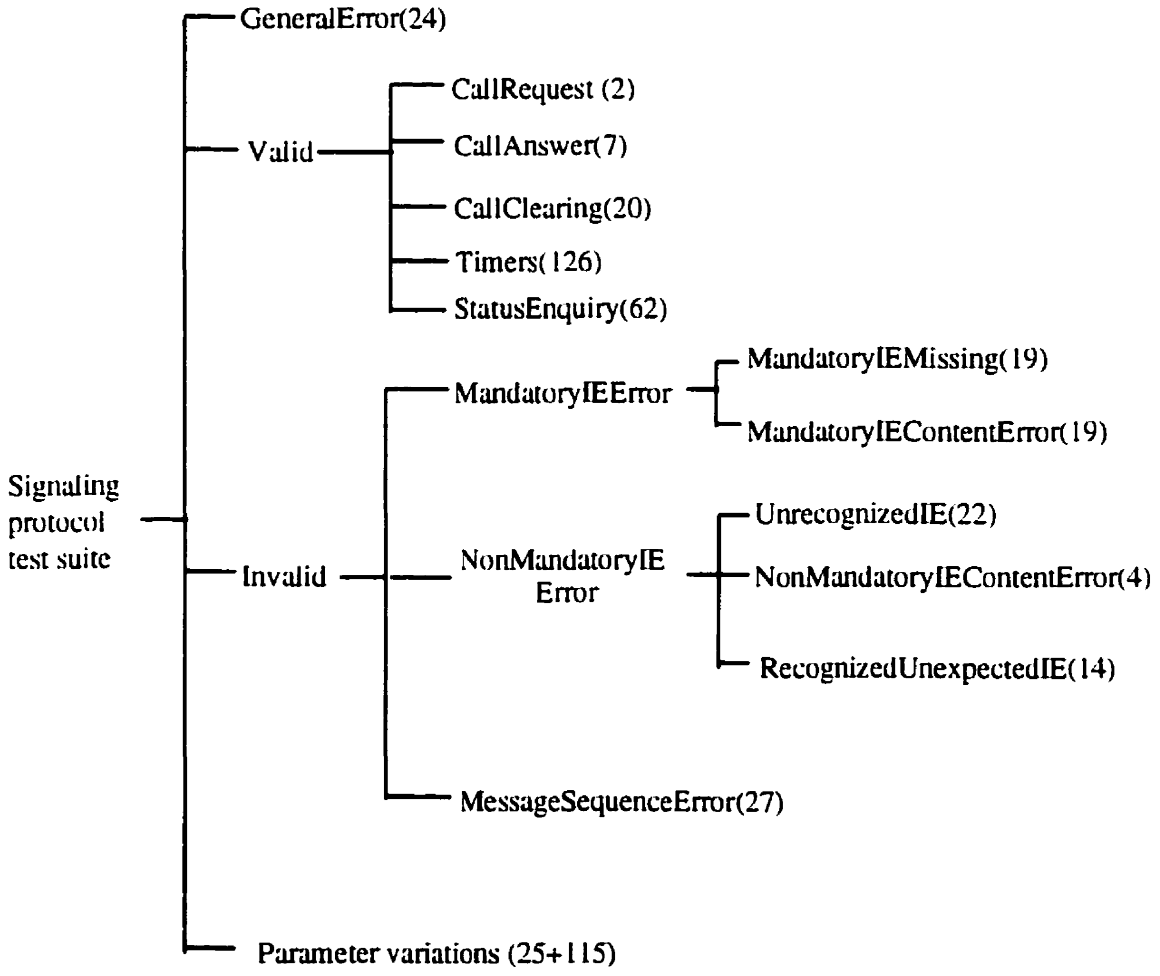

Figure 5.6: Test suite structure

It is worth noting that the test grouping (Figure 5.6) matches the test categories recommended by [ISO9646].

The following table gives the outline of the test categories required by ISO 9646 and the test groups from the test suite that corresponds to them.

5.9 Conclusion

In this chapter, we used the experimental tool chain to develop a conformance test suite for the Signaling protocol. Starting from the SDL specification of the protocol, the FSM model was extracted by the FEX tool. Using the TAG tool, a number of the test cases for the control part of the protocol was developed in an automated way from the FSM model. For protocol properties that could not be tested with the test cases developed by the tool, additional tests were developed manually. The resulting test suite has 320 test cases and complete fault coverage in terms of the FSM model. The test suite has tests for each major protocol function.

Tests for the data part of the protocol were developed manually directly from the standard document. A set of PDUs with different parameter values was constructed, and corresponding test cases were completed with these PDUs. Test cases and the PDUs are represented in SDL. There are 25 test cases checking the protocol behavior on input of valid PDUs and 115 test cases for invalid PDUs.

The SDL specification and the test suite in SDL were validated one against the other in an SDL executable environment. A number of errors were discovered and corrected in the data part of the protocol while no error was found in the control part. To help maintain of the test suite, test cases were organized into test groups.

Our experience with the Signaling protocol has shown that the standard in some places is ambiguous. It is sometimes difficult to determine the conformance requirements from the plain text and few possible interpretations are possible. For example, when receiving STATUS with Cause equal to 30, the standard says that an "appropriate action shall be taken", while not explicitly explaining this "appropriate action". Ideally, the standard should be written using formal methods to avoid any discrepancy. Unfortunately, this was not the case with the UNI 3.1. The Q2931 has SDL diagrams, but at the time of this work it was still a draft version. The results of the test development process are as accurate as the modeling of the protocol with the FSM. Aspects of the protocol that cannot be represented with the FSM transitions have to be tested in addition to the FSM testing. Since the main point of the test development method used is the fact that we can rely on methods for FSMs which have a proven fault coverage and precise mathematical foundation for generating tests for protocols, developing a relevant FSM model for the protocol is crucial. Therefore, specifying protocol standards using FDTs would be a good practice.

Chapter 6

Development of an interoperability test suite

6.1 Test configuration

The conformance test suite developed in the previous section can be used to determine whether or not an implementation of the Signaling protocol conforms to the specification. As discussed in Section 2.4, there is still no guarantee that any two conforming implementations will interoperate.

Interoperability testing assumes the test configuration of Figure 6.1. In Figure 6.1, only the Signaling protocol entities are shown. The rest of the ATM protocol stack is not presented (the lower lever protocols: SAAL, ATM layer, etc.). The system under test (SUT) consists of two Signaling protocol IUTs connected through a network (or a switch). The behavior of the system is modeled using the ASPs that are exchanged at the upper protocol interface of the IUTs. Each of the IUTs is modeled by the FSM of the Signaling protocol developed for conformance testing. We assume that the network is reliable ("null" network): there are no errors introduced by the network, and there is no loss of PDUs.

Since the goal is to test a single point-to-point connection, one IUT is defined as Sender and the other IUT as Receiver. Sender initiates the call and Receiver answers the call. Assigning the role of Sender to one IUT implies that it consists of the part of the FSM that is responsible for the Call Request phase (states U0, U1, U3, U10, U11, U12). Apparently, Receiver consists of the FSM part responsible for Call Answer (states U0, U6, U8, U9, U10, U11, U12). If the IUT that was tested as Receiver has to be tested as Sender, the role of the IUTs should be switched, and the test suite will be reapplied. If both IUTs are allowed to initiate the call, the ASPs received will refer to two different connections, since each connection is represented by a different instance of the protocol (FSM).

In this test configuration, it is assumed that there is no test equipment that monitors the flow of the PDUs between the two IUTs: the only points of observation and control are at upper layer PCOs of the Sender and Receiver. It is obvious from the configuration that:
(1) The internal communication between the two FSMs cannot be observed. Therefore, it is impossible to directly check outputs of the IUT which are inputs to the other IUT;
(2) Some transitions of the IUT always initiate other transitions of the other IUT. These transitions cannot be checked in isolation.
However, in the interoperability testing we are not interested in the internal communication of the IUTs. We are concerned with the services that are offered to the user, and since they are observable only on the upper layer PCOs, monitoring the message exchange between the IUTs reveals no additional information.

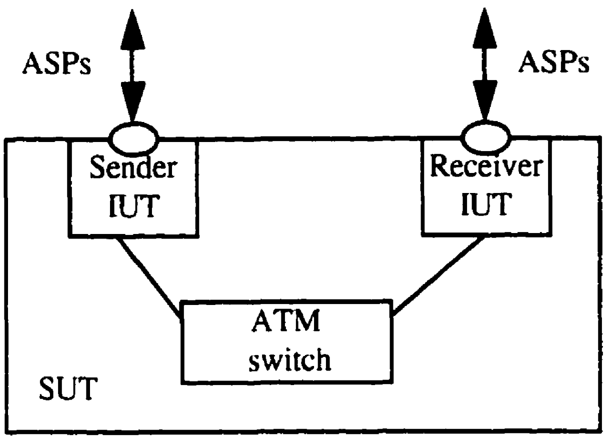

Figure 6.1: SUT used for the Interoperability Testing

6.2 The test development process

The process used to derive an interoperability test suite is similar to that used for conformance testing. The test development process explained in the conformance testing starts with the standard document of the protocol. However, for the interoperability testing there is no document that specifies the behavior of the system shown in Figure 6.1. The global FSM model for the SUT has to be derived manually from the FSMs of the IUTs, by combining the two FSMs. The global FSM is used as input for the TAG tool to generate test cases. Tests for the data part of the protocol are developed from the PDUs constructed in the conformance testing and test cases generated from the global FSM. The crucial point is the generation of the global FSM for the SUT. The Sender and Receiver FSMs are communicating FSMs that exchange messages between each other and the environment. We are required to build a global FSM that represents the composed behavior of the two FSMs. The method used is given in the following section.

6.2.1 Composing two FSMs

The goal of composing two FSMs is to produce a global FSM that describes the joint behavior of the constituent FSMs. Two FSMs that we want to combine are FSMs of the Signaling protocol communicating between each other. They are shown inFigure 6.2.

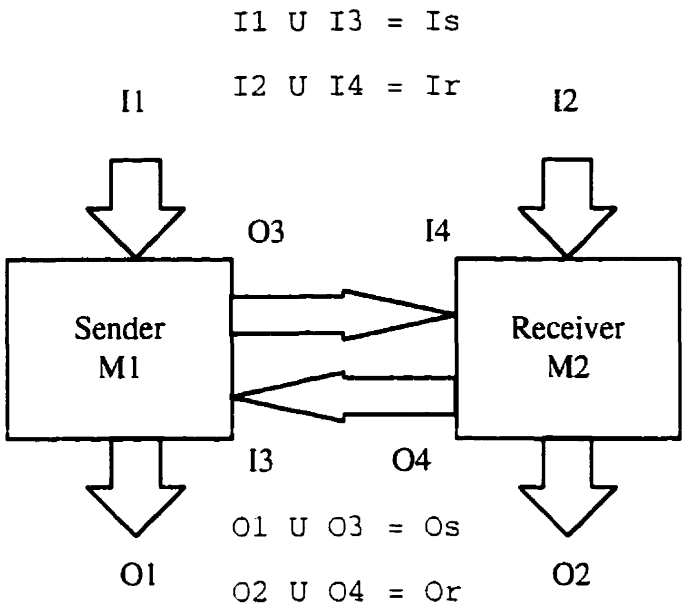

Figure 6.2: Two communicating FSMs

The global FSM has the input set $Ig= I1 \cup I2$ and output set $Og=O1\cup O2$. The events that belong to O3 and O4 are exchanged between M1 and M2 and are not visible at the outputs of the global FSM. The definition of the combining operator of two FSMs and the corresponding method are given in [DaKl89]. In the following, we give an informal explanation of the method.

We define machine M1, M2 and global FSM, gFSM, as : $M1 = \{I_1, O_1, S_1, s_{01}, D_1, \delta_1, \lambda_1\}$, $M2 = \{I_2, O_2, S_2, s_{02}, D_2, \delta_2, \lambda_2\}$, and $gFSM = \{I_g, O_g, S_g, s_{0g}, D_g, \delta_g, \lambda_g\}$. A state of the global FSM M is the ordered pair of states of machine M1 and machine M2 ($S_g = S_1 X S_2$). The global FSM is constructed by applying inputs $i \in I_g$ to M1 and M2, and observing the outputs $o \in O_g$. The output and the next state of the global FSM for the input i in the state (s_1, s_2) are determined according to the following two rules:

(1) If i is an input for machine M1 ($i \in I1$), and an output $o = \lambda_1 (i, s_1) \in O1$, the output of the global FSM on input i is: $\lambda_g(i, (s_1, s_2)) = \lambda_1(i, s_1)$, and the next global state is: $\delta_g(i, (s_1, s_2)) = (\delta_1(i, s_1), s_2)$. In this case, the output o is sent to the environment and no new transition is fired before the input from the environment is applied. The procedure is identical for an input to machine M2;

(2) If i is an input for machine M1 ($i \in I1$), and an output $o = \lambda_1(i, s_1) \in O3$, the output o is sent to M2. The FSMs will relay events between them as long as no output is sent to the environment

(we assume that this process will eventually terminate; in other words, that there is no livelock). During this process, a number of events may be exchanged between the machines, and they may transfer through the number of states. The output of the global FSM is the last event in the sequence of the exchanged events and the next global state is the pair of states in which M1 and M2 rested. As in (1), M1 and M2 will remain in these states as long as there is no input from the environment ($i \in I_g$).

Using the above two rules, for each global state (s_1, s_2) transitions are determined for all inputs $i \in$ Ig. There are states of the global FSM that can be visited during a proper communication at the start state, as well as states that cannot be visited during such a communication. However, some of the visited states are not stable, i.e. the combined FSM will spontaneously transit from those states (without external inputs). Summarizing, we have reachable and non-reachable states of which the reachable states can be divided into stable and non-stable states. We are particularly interested in the stable reachable states because these are states of the global FSM.

6.2.2 A global FSM for interoperability testing

The global FSM for the interoperability testing is derived from the Sender and Receiver FSMs. The Sender FSM has 6 states, and Receiver FSM has 7 states (see Section 6.1). Sender and Receiver inputs and outputs are given inTable 6.1:

	Input (I1,I2)	#	Output (O1,O2)	#	Input(I3,I4) Output(O3,O4)
Sender	Setup_req_s	1	Setup_conf_s	21	all PDUs
	Setup_req_s(Mie=1)	2	Release_conf_s	22	
	Setup_req_s(Mie=2)	3	Release_conf_s(ca=96)	23	
	Setup_req_s(NMie=2)	4	Release_conf_s(ca=100)	24	
	Release_req_s	5	Release_conf_s(ca=31)	25	
	Release_req_s(Mie=1)	6	Proceeding_ind_s	26	
	Release_req_s(Mie=2)	7			
	Release_conf_s	8			
Receiver	Release_resp_r	11	Setup_ind_r	31	all PDUs
	Release_resp_r(Mie=1)	12	Setup_comp_ind_r	32	
	Release_resp_r(Mie=2)	13	Release_ind_r	33	
	Proceeding_req_r	14	Release_ind_r(ca=31)	34	
	Proceeding_req_r(Mie=1)	15	Null	35	
	Setup_resp_r	16			
	Setup_resp_r(NMie=2)	17			

Table 6.1: Inputs and outputs for Sender and Receiver FSMs

Using the method from the previous section, the global FSM with 6 reachable stable states A=(0,0), B=(1,6), C=(3,9), D=(10,10), and E=(11,12) is obtained (Figure 6.3).

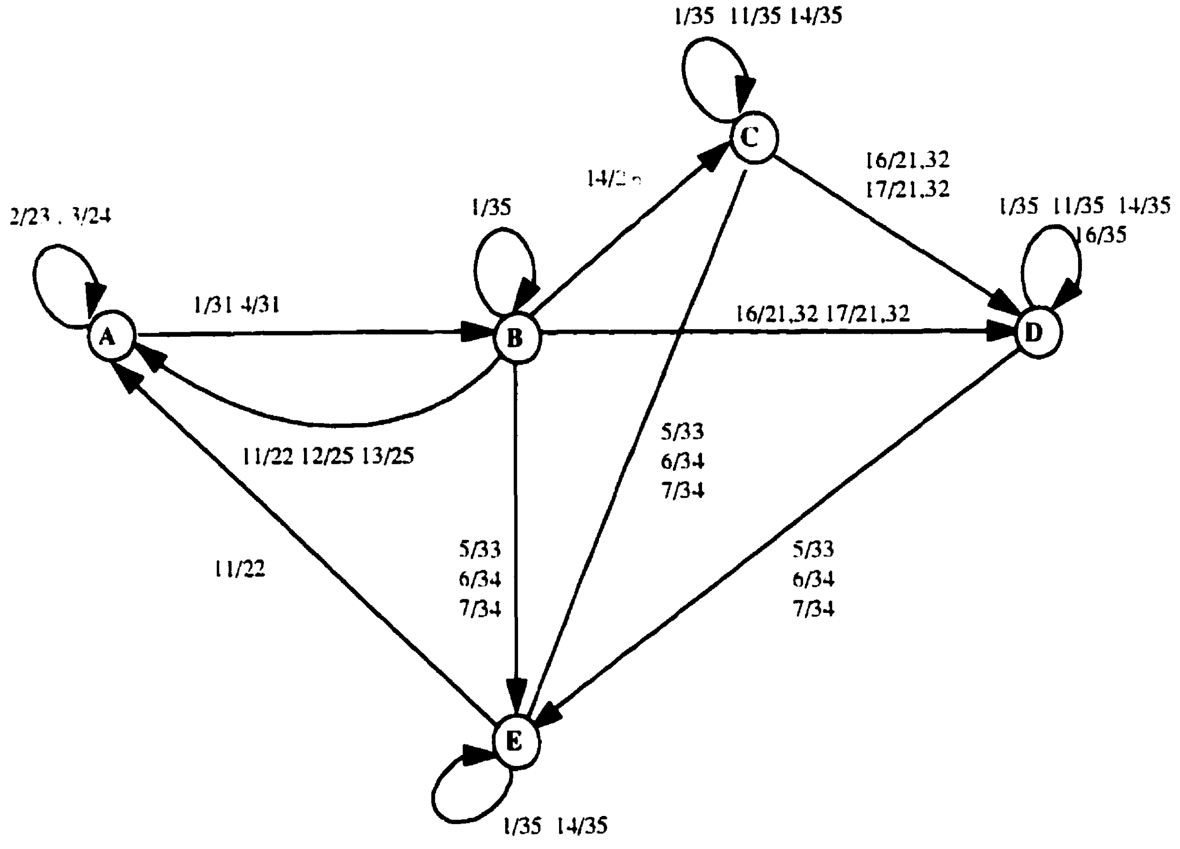

Figure 6.3: The interoperability FSM state diagram

In the following paragraph, an example of a valid call request and error Handling procedure is used to illustrate the exchange of messages in the SUT and explain the transitions in the corresponding global FSM.

In the starting global state A, Sender and Receiver are in Null (U0) state. When Setup_req is received, the Sender initiates the call request by sending a SETUP message and transfers to U1 state. If the network is responding with the CALL_PRO, the Sender will be in the U3 state. If the network does not respond with the CALL_PRO, Sender will remain in the U1 state. The SETUP is received by the Receiver, Setup_ind is sent at the Receiver PCO and the Receiver transfers to

U6 state. At this moment, the global FSM is in B state. The corresponding exchange of the messages is shown in Figure 6.4.

The error Handling procedure is explained below. When the SETUP is received with an error (for example MIEM), the Receiver will respond with the REL_COM (actually the network at the Sender side will respond with the REL_COM but since in our system we are not modeling the network functionality, it will be done by the Receiver) and connection will be cleared, Sender and Receiver will be in U0 and system will remain in A state. The corresponding exchange of the messages is shown in Figure 6.4.

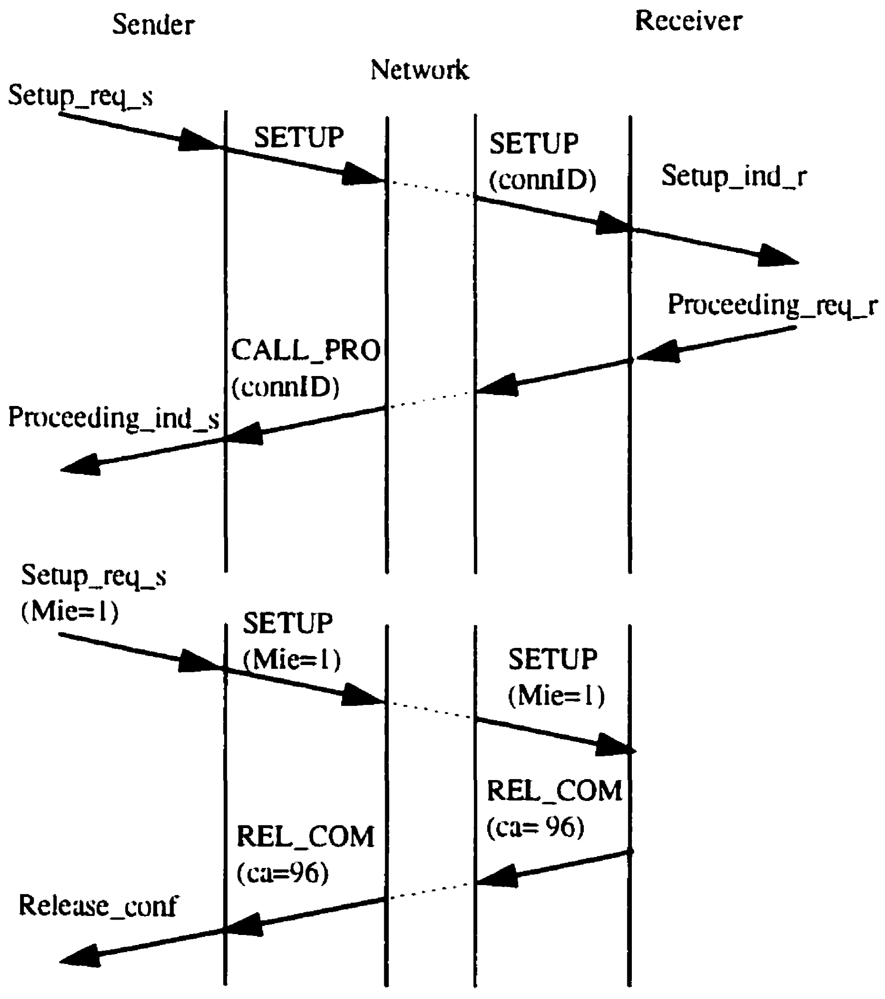

Figure 6.4: Call request and MIEM procedure for the SETUP message

6.2.3 Applying the TAG tool to the global FSM

The global FSM described in a TAG format is used to derive tests with the tool. A complete test suite is generated by the TAG tool; however, the TAG tool cannot find the state identification facility for the global FSM and the test suite has no checking sequences because the FSM has no distinguishable states.

As already discussed, the FSM model is not completely specified for ASPs. As discussed in Section 4.3.1, we interpret the unspecified inputs as "don't care" transitions. This did not pose any problem for the test development in conformance testing because the existing state identification facility makes no use of any ASP. However, for the global FSM, an HSI set does not exist that will distinguish the states. The possible solution to this problem is to relinquish the interpretation of the unspecified inputs using "don't care" transitions. A common approach is to complete the behavior for the unspecified inputs with looping transitions having no output, so called Null transitions. We add Null transitions for as many ASPs as required to distinguish the states. The state identification facility produced by TAG is as follows:

```
State identification sequences:
WA={Setup_req_s}
WB={Setup_req_s.Release_resp_r, Proceeding_req_r}
WC={Setup_req_s.Release_resp_r.Setup_resp_r}
WD={Setup_req_s.Release_resp_r.Setup_resp_r}
WE={Setup_req_s.Release_resp_r, Proceeding_req_r}
```

The generated interoperability test suite has 44 test cases. Example test cases are given in Appendix E.

As discussed in Section 6.1, the test configuration in Figure 6.1 has certain limitations. Some protocol properties cannot be tested because communication between the Sender and Receiver is not visible and some transitions are "coupled". The following procedures of the Signaling protocol cannot be tested with the interoperability test suite:

(1) Error Handling procedures for Messages type and Message sequence errors (see Section 4.2.3.3): using only ASPs, we cannot generate any message in any state or a wrong message;

(2) Status Inquiry procedure: this procedure is initiated only by the STATUS ENQ message that is sent upon indication of the SAAL failure. In the conformance testing (Section 4.3.1), we modeled the Status Inquiry procedure with the primitive from SAAL, InitiateStatus Enquiry; the STATUS

ENQ is sent when InitiateStatusEnquiry is received on the lower PCO. Since we do not have access to the lower PCO, we cannot test this procedure in the interoperability testing;
(3) Coordinator functions: the previous two cases are related to the limitations of the test configuration. The ability to test the Coordinator functions depends on the availability of the common message parameters that trigger the Coordinator functions (Protocol discriminator, message length or CR). Since we assume that these parameters are not associated with the ASPs (as explained in the next section), we cannot produce tests for the Coordinator functions.

However, the previous functions and procedures are assumed to be tested with the conformance test suite.

6.2.4 Test cases for the data part

In conformance testing, the tester has access to the lower interface and is able to send PDUs with valid or erroneous content to the IUT. The tester can also examine the content of the PDUs received from the IUT. In interoperability testing, the data part of the protocol is tested indirectly using the ASPs. At the PCOs (Sender or Receiver), there is one-to-one mapping between the ASPs and the messages. It is assumed that an ASP has the same parameters (IEs) as the corresponding message but has none of the common message fields (Parameter discriminator, message length, and CR). Common message fields (parameters) are considered local to the protocol and should not be accessible by the upper layer protocol. Commercial ATM API (WinSock2 [WINS2]), which we use for realization of the test execution environment, justifies our assumption since it does not provide read or write access to these parameters in its function calls.

We can set the parameters of the messages sent by the Sender or Receiver by setting parameters of the corresponding ASPs. The intention is to reuse the sets of representative PDUs constructed for parameter variation in the conformance testing, and to use them to test the data part with the interoperability test suite. Test cases from the interoperability test suite are instantiated using the corresponding PDU values developed for the data part in the conformance testing.

6.3 Validation of the interoperability test suite

A test suite may be validated in the SDL environment using the system shown in Figure 6.5. SDL specifications are directly connected without any network model in between. The tester is realized as a single block and the test suite is implemented as a sequence of SDL procedures. Since there is no module representing the network, a small number of changes to the SDL specification are necessary to make the SDL configuration executable. When a SETUP message is sent by the protocol to the network, Connection Identifier IE (ConnID) is not mandatory, while in another direction this IE is mandatory. Since the Receiver is expecting SETUP with the ConnID present, we have to change the Sender specification to generate a SETUP message with ConnID. Similar change to the Receiver is required for CONN and CONN_ACK message. They have mandatory ConnID IE in network to user direction, while in user to network ConnID is optional. In this case, it is a responsibility of Receiver to play the role of a network and to generate the ConnID in corresponding message.

In building the SDL system, we wanted to use the same SDL specification for the two blocks, Sender and Receiver. However, instead of initializing these block instances with the same definitions, we were forced by the commercial tool to copy the whole definitions and to treat the block as a different one. Even then, we could not share common procedures, for example, procedures for checking parameters, and it was necessary to rename procedures in order to avoid the name conflict. However, this would be impractical when the specification consists of 10 000 lines. We were still unable to solve the problem even after contacts with tool specialists. Specialists from the tool supplier were unable to offer any better solution. For this reason, validation of the interoperability test suite was not performed as planned. It might be possible, however, to use other SDL-based tools to validate interoperability tests.

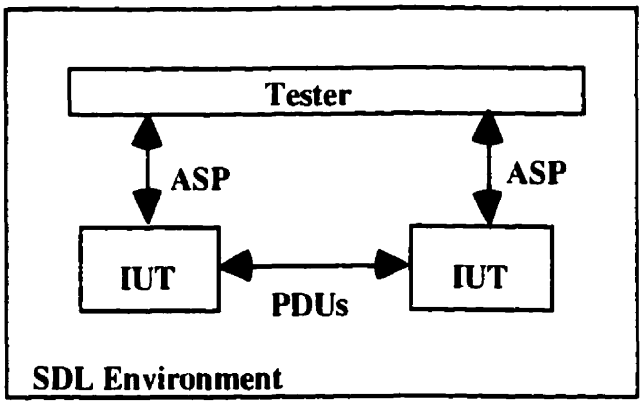

Figure 6.5: Validation of the interoperability test suite

6.4 Conclusion

An interoperability test suite is needed to test the interoperability of the implementations in a real system configuration. Instead of testing each implementation in isolation, the functional system consisting of properly interconnected implementations is configured, and its services to the user are tested. In this chapter, system under test (SUT) is defined as consisting of two Signaling protocol entities connected via a network. We assumed that there are no errors introduced by the network, and there is no loss of the PDUs. To develop tests for interoperability of the Signaling protocols, the SUT is modeled by a global FSM representing the behavior of all components. The global FSM was obtained by combining the two FSM models of the Signaling protocol and its behavior was defined in terms of ASPs.

Using the global FSM to model the system, test cases for the control part were developed using the TAG tool. After completing the FSM with looping null transitions for unspecified primitives, a test suite with state identification sequences was obtained. The resulting test suite consists of 44 test cases. Assuming that an ASP has the same parameters (IEs) as the corresponding message, test cases for the data part were constructed using the PDU values developed in the conformance testing. An SDL system for validating the test suite was proposed.

Chapter 7

Conclusion

As stated in the Introduction, the objective of this work was twofold. The first goal was to develop conformance and interoperability test suites for the ATM Signaling protocol using the tool chain. Part of the conformance test suite was developed for the control part of the ATM Signaling protocol using the tools. Additional test cases for parameter variation and certain timers were derived manually. The interoperability test suite was developed by considering a system of two signaling protocol entities connected through a dummy network model. The obtained tests could be used in practice during the development process of ATM cards.

The second goal was to evaluate the experimental tool chain developed at UofM. Our impressions of the UdeM tool chain are:

(1) The majority of test cases for the control part of the protocol can be developed automatically;

(2) Test suites developed using the FSM test generation method have complete fault coverage in terms of the FSM model of the protocol (provided that the FSM is reduced);

(3) Test cases for the data part could be validated using the SDL specification as a reference. Since PDUs for parameter variations were developed manually and independently from the SDL

specification, validation against the SDL specification provides an increased confidence in the quality of the test cases.

(4) The automated tools give a possibility for multiple iterations through the development process. The changes (if needed) in the SDL specification or the intermediate FSM model could be made more easily because the test developer could regenerate the test case in a few steps. Consequently, the test developer could gradually refine the SDL specification, FSM model and test suite, starting from a less elaborated SDL specification and gradually extending it with additional features.

As a result this work a number of possible improvements to the UofM tools can be suggested:

(1) The FEX tool could be enhanced to make use of the TAG options to test timers. Currently, FEX does not translate the timer transitions from the SDL specification to the FSM timer transitions;

(2) The TAG tool could be improved to produce test cases for transitions with two timers. Also, the restriction on the starting transition not being a looping transition could be removed since the tool cannot handle protocols like Signaling protocol;

(3) Managing a test suite of more than one hundred test cases manually is a difficult and error prone task. If tools include some facility that helps group test cases, manipulate them, and search for a particular test case, the test manipulation process could proceed more efficiently;

(4) Usually, SDL specifications are composed of several processes. Therefore, to obtain the global behavior of the specification, The FEX tool should be able to extract an FSM from each process and combine them into a single global FSM.

These improvements can be areas of future work.

References

[ATMF94] ATM Forum, " Introduction to ATM Forum Test Specifications," af-test-0022.000, December, 1994

[CCITT83] CCITT Draft Recommendation X.200, "Reference model of open systems interconnection for CCITT applications," June 1983

[CCITT92] CCITT, COM X-R, 17-E, Geneva, March 1992: Recommendation Z.100 - CCITT Specification and Description Language (SDL) and Annex A to the Recommendation

[CCITT88] CCITT Recommendations Z.101-Z.104 (Blue Book Series), SDL, CCITT, 1988

[GEOD] Geod is product of VERILOG SA, France, http:\\www.verilog.fr\

[ISO7498] ISO International Standard 7498, "Information processing systems - Open Systems Interconnection - Basic Reference Model," Oct. 1983

[IS9074] "Estelle: A Formal Description Technique Based on an Extended State Transition Model," Int. Organization for Standardization, IS 9074, 1988

[IS8807] "Information processing systems-Open System Interconnection-LOTOS-A Formal Description Technique Based on Temporal Ordering of Observed Behavior," Int. Organization for Standardization, IS 8807, 1988

[ISO9646] Information Processing Systems - OSI conformance testing Methodology and framework, ISO/IEC JTC 1, IS9646, 1991

[I.311] ITU-T: Recomendation I.311, "B-ISDN Service Aspects, " Rev. 1, Geneva, 1993

[MSC94] Z.120 (1993), Message Sequence Chart (MSC), IUT-T, September 1994

[Q2931] ITU-T: Draft Recommendation Q.2931, "B-ISDN User-Network Interface Laer 3 Protocol," Geneva, 1993

[SDT] SDT is product of Telelogic AB, Sweden, http:\\www.telelogic.com\

[TTCN] ISO/TC97/SC21, "The tree and tabular combined notation, " Annex E of Part 2 [ISO9646], edited by A. Wiles, December 1987

[UNI31] ATM Forum, "ATM User-Network Interface Specification Version 3.1," September, 1994

[WINS2] Windows Sockets 2 Application Programming Interface, An Interface for Transparent Network Programming Under Microsoft Windows, revision 2.1.0, January 22, 1996,

[ArPh92] N. Arakawa, M. Phalippou, N. Risser, T. Soneoka, "Combination of conformance and interoperability testing," FORTE, 1992

[Boch78] G. v. Bochmann, "Finite state description of communication protocols," Computer Networks, North Holland, 1978

[Boch87] G. v. Bochmann, "Semiautomatic implementation of communication protocols" IEEE Transactions on Software Engineering. SE-13(9), Sept. 1987

[Boch87-1] G. v. Bochmann, "Usage of protocol development tools: the result of a survey," Protocol specification, Testing, and Verification, VII, H. Rudin, C.H. West, Elsevier Science Publishers B.V. IFIP 1987

[BoPe97] G. v. Bochmann, A. Petrenko, O. Bellal, S. Maguiraga, "Automating the Process of Test Derivation from SDL Specification," SDL - Forum , 1997

[BeHo89] F. Belina and D. Hogrefe, "The CCITT specification and description language SDL." Networks and ISDN Systems, 16, North-Holland,1988/89

[Bern94] P. J. Bernhard, "A Reduced Test Suite for Protocol Conformance Testing," ACM Transaction on Software Engineering and Methodology, Vol 3 No3, July 1994

[Chow78] T. S. Chow, "Testing software design modeled by finite state machines," IEEE Trans. on Softw. Eng. SE-4, 3 (May), 1978

[DaKl91] H. van Dam, H. Kloosterman, E. Kwast, "Test derivation for standardized test methods," 4 th International Workshop on Protocol Test Systems, 1991

[Gone70] G. Gonec, "A method for the design of fault detection experiments," IEEE Trans. Comput. C-19,6(June), 1970

[Henn64] F. C. Hennie, "Fault Detecting Experiments for Sequential Circuits, " Proc. Of 5th Annual Symposium on Switching Circuit Theory and Logical Design, Princeton, NJ 1964

[Higa94] T. Higashino and G. v. Bochmann, "Automatic Analysis and Test Derivation for a Restricted Class of LOTOS Expressions with Data Parameters," IEEE Trans., SE-20, No. 1, 1994.

[Knig87] K. G. Knighston, Terry Knowles, John Larmouth, "Standards for Open System Interconnection," McGraw Hill, 1987

[Koh78] Z. Kohavi, "Switching and Finite Automata theory," McGraw Hill, New York, N.Y., 1978

[LeYa96] D. Lee, M. Yannakakis, "Principles and Methods of Testing FSM - A Survey," IEEE Proceedings, vol.84, No.8, August 1996

[Linn90] R. J. Linn, "Conformance Testing for OSI Protocols, "Computer Networks & ISDN Systems, Vol. 18, 1989/90

[Main96] W. Mainvis, "Intégration de nouvelles fonctionnalités dans un outil de dérivation de tests pour les protocoles", DEA Thesis, Université de Montréal (in collaboration with CRIN, Nancy, France), August 1996.

[Marc97] R. Marcocci, " Implementation of the ATM Signaling Protocol ," Master Thesis in progress, University of Montreal, 1997

[More90] L. J. Morell, "A Theory of Fault - Based Testing, " ," IEEE Trans. on Softw. Eng. SE-16, 8 (August), 1990

[Myer79] G. J. Myers, "The Art of Software Testing, " Wiley-Interscience publication, John Wiley & Sons, 1979, 177p.

[Nash83] S. C. Nash, "Automated implementation of SNA communication protocols," In Proc. IEEE International Conference on Communications," June 9-22, 1983

[NaTs81] S. Nito, M. Tsunoyma, "Fault detection for sequential machines by transition tours," IEEE Fault Tolerant Computing Conference, IEE , New York, 1981

[Rayn87] D. Rayner, "OSI Conformance Testing," Computer Networks and ISDN Systems, Vol. 14, 1987

[Petr91] A. Petrenko, "Checking experiments with protocol machines, " in Proceedings of the IFIP 4th International Workshop on Protocol Test Systems (IWPTS 91), the Netherlands, 1991, pp. 83-94.

[PoSm82] D. P. Pozefsky, F. D. Smith, "A meta-implementation for system network architecture," IEE transactions on Communications, COM - 30:1348-1355, June 1982

[SaBo87] B. Sarikaya, G. v. Bochmann, Eduard Cerny, "A Test Design Methodology for Protocol Testing," IEEE Trans. Software Eng., Vol. SE - 13, No. 5, May 1987

[SaDa88] K. K. Sabnani, A.T. Dahbura, "A Protocol Test Generation Procedure," Computer Networks and ISDN Systems, Vol. 15, No. 4, 1988

[SiCh87] D. Sidhu, A. Chung, "Experience with Formal Methods in Protocol Development," FORTE'89, 1989

[SiBl90] D. Sidhu, T. P. Blumer, "Semi-automatic implementation of OSI protocols," Computer networks and ISDN systems, January 1990

[Tan96] Q. M. Tan, A. Petrenko, and G. v. Bochmann, "A test generation tool for specifications in the form of state machines," in Proceedings of the International Communications Conference (ICC) 96, Texas, June 1996, pp.225-229.

[Vasi73] M. P. Vasilevski, "Failure Diagnosis of Automata, " translated from Kibernetika, No.4, July-August, 1973

[Vuon89] S. T. Vuong , "The UIOv-method for Protocol Test Sequence Generation, " Proc. Of the 2nd Int. Workshop on Protocol Test Systems, Berlin, Germany, October 3- 6, 1989

[Yao96] M. Yao, "On the Development of Conformance Test Suites in View of their Fault Coverage," PhD thesis, University of Montreal, 1996

Appendix A: The FSM model

The FSM model is represented using the TAG format. The Proc FSM does not include the coordinator transitions. The FSM is organized in three sections: variables, states, inputs, outputs and transitions. Variables are used to condition and, therefore, unfold the inputs. Meaning of the variables and their values are explained in comments in the same line. For example, the value of the variable `Mie` indicates which type of error the input message should have. When the relation "!=" is used, it means that the message or primitive may be initialized with any value or the parameter not equal to one indicated. Consequently, the test case may be completed in several different ways. When the value of incomp variables is 1 (used only with the STATUS), it indicates that the current state in STATUS message must be incompatible with the state where message is received.

Messages in the input and output sections are marked with `PDU`. Transitions are grouped by the starting state. Comments #CR, #CA, #CC, #SE, #MS, #MIEM, #MIECE, #UIE, #NMIECE, #RUIE next to transitions determine their function. For example, Transitions that realize the Call Request function of the protocol have comment #CR. Later, these comments are used to group the corresponding test cases in test groups.

```
 * Q2931 Fsm specification: Proc FSM (no coordinator functions) */
 *------------------------------------*/

variables :
rs integer; /* Reason for rejection */
cs integer; /* Current state */
ca integer; /* Cause value */
incomp integer;  /* 0=compatible states 1= incompatible states */
Mie integer; /* 1=MIEM;   2=MIECE    */
NMie integer;/* 1=UIE;   2=NMIECE ;    3= RUIE*/

/*----------------------------------------------------------*/
States:
U0: initial;
U1;U3;        /* Call Request */
U6;U8;U9;    /* Call Answer */
U10;          /* Call Active */
U11;U12;     /* Call Clearing */
/*----------------------------------------------------------*/
Inputs:

      Setup :PDU;
      Setup(Mie=1):PDU; /* mandatory and non-mandatory*/
      Setup(Mie=2):PDU;
```

```
        Setup(NMie=1):PDU;
        Setup(NMie=2):PDU;
        Setup(NMie=3):PDU;
        Conn   :PDU; /* same */
        Conn(Mie=1) :PDU;
        Conn(Mie=2) :PDU;
        Conn(NMie=1):PDU;
        Conn(NMie=2):PDU;
        Conn(NMie=3):PDU;
        Conn_Ack     :PDU;
        Conn_Ack(NMie=1)   :PDU; /* no IE */
        Conn_Ack(NMie=3)   :PDU;
        Call_Pro     :PDU;
        Call_Pro(Mie=1) :PDU;
        Call_Pro(Mie=2)    :PDU;
        Call_Pro(NMie=1)   :PDU; /* only non-mandatory */
        Call_Pro(NMie=2)   :PDU;
        Call_Pro(NMie=3)   :PDU;
        Rel          :PDU;
        Rel(Mie=1)   :PDU; /* only mandatory */
        Rel(Mie=2)   :PDU;
        Rel(NMie=1)  :PDU;
        Rel(NMie=3)  :PDU;
        Rel_Com            :PDU;
        Rel_Com(Mie=1)     :PDU; /* mandatory and non-mandatory */
        Rel_Com(Mie=2)     :PDU;
        Rel_Com(NMie=1)    :PDU;
        Rel_Com(NMie=2)    :PDU;
        Rel_Com(NMie=3)    :PDU;
        Status(cs=0)       :PDU;
        Status(cs!=0,incomp=1)  :PDU;
        Status(cs!=0,ca=96,incomp=0)  :PDU;
        Status(cs!=0,ca=97,incomp=0)  :PDU;
        Status(cs!=0,ca=99,incomp=0)  :PDU;
        Status(cs!=0,ca=100,incomp=0) :PDU;
        Status(cs!=0,ca=101,incomp=0) :PDU;
        Status(cs!=0,ca=30,incomp=0)  :PDU;
        Status(cs!=0)      :PDU;
        Status(Mie=1)      :PDU;
        Status(Mie=2)      :PDU;
        Status_enq  :PDU;
        Status_enq(NMie=1) :PDU;
        Init_stat_enq      :PDU;
        Setup_res;
        Setup_req;
        Rel_req;
        Rel_resp;
        Rel_resp(ca=88);
        Rel_resp(ca=17);
        Rel_resp(ca=21);
        Rel_resp(ca=23);
        Proceeding_req;
        Unrecognized;

/* timers */
        T313 : timer;
        T308 : timer;
        T310 : timer;
        T303 : timer;
        T322 : timer;
/*-------------------------------------------------------------*/

Outputs:
        Conn         :PDU;
        Conn_Ack     :PDU;
        Call_Pro     :PDU;
        Setup        :PDU;
        Status(ca=101)     :PDU;
        Status(ca=96)      :PDU;
        Status(ca=97)      :PDU;
        Status(ca=100)     :PDU;

        Status(cs=0,ca=30)        :PDU;
        Status(cs=1,ca=30)        :PDU;
        Status(cs=3,ca=30)        :PDU;
        Status(cs=6,ca=30)        :PDU;
        Status(cs=8,ca=30)        :PDU;
        Status(cs=9,ca=30)        :PDU;
        Status(cs=10,ca=30)       :PDU;
        Status(cs=11,ca=30)       :PDU;
        Status(cs=12,ca=30)       :PDU;
        Status_enq         :PDU;
        Rel(ca=16)         :PDU;
        Rel(ca=102)        :PDU;
        Rel(ca=96)   :PDU;
        Rel(ca=97)   :PDU;
        Rel(ca=99)   :PDU;
        Rel(ca=100) :PDU;
        Rel(ca=101) :PDU;
        Rel_Com            :PDU;
        Rel_Com(ca=96)     :PDU;
        Rel_Com(ca=100)    :PDU;
        Rel_Com(ca=101)  :PDU;
        Rel_Com(ca=88)     :PDU;
        Rel_Com(ca=17)     :PDU;
        Rel_Com(ca=21)     :PDU;
        Rel_Com(ca=23)     :PDU;

        comb1:Rel_Com:PDU,Rel_conf;
        comb2:Conn_Ack:PDU,Setup_conf;
        comb3:Rel_Com(ca=96):PDU,Rel_conf;
        comb4:Rel_Com(ca=100):PDU,Rel_conf;
        comb5:Rel_Com(ca=41):PDU,Rel_conf;
        Setup_ind;
        Setup_conf;
        Setup_comp_ind;
        Rel_conf;
        Rel_conf(ca=31);
        Rel_ind;
        Rel_ind(ca=31);
        Proceeding_ind;
/*-------------------------------------------------------------*/
Transitions:

/*U0: Null state*/
      U0    ?Setup            !Setup_ind   >U6;/*#CR*/
      U0    ?Setup_req  !Setup             >U1, start T303;/*#CR*/

/* unexpected messages are not received in U0,
   coordinator is filtering them */
/* Unexpected Msg */
/* IE Miss/Err */
      U0    ?Setup(Mie=1)     !Rel_Com(ca=96)   >U0;   /* MIEM */
      U0    ?Setup(Mie=2)     !Rel_Com(ca=100)  >U0;   /* MIECE */
      U0    ?Setup(NMie=1)    !Setup_ind  >U6;  /* UIE */
      U0    ?Setup(NMie=2)    !Setup_ind  >U6;  /* NMIECE */
      U0    ?Setup(NMie=3)    !Setup_ind  >U6;  /* RUIE */
```

```
/*U1: Call Initiated*/
     U1    ?Call_Pro   !Proceeding_ind   >U3, stop T303, start T310;
/*#CA*/
     U1    ?Conn       !comb2            >U10, stop T303; /*#CA*/
     U1    ?Rel_Com    !Rel_conf    >U0, stop T303;
                                          /*#CC first clearing mess. */
     U1    ?Status_enq !Status(cs=1,ca=30)     >U1; /*#SE*/
     U1    ?Status(cs=0)      !Rel_conf    >U0, stop T303; /*#SE*/
     U1    ?Status(cs!=0, incomp=1) !Rel(ca=101)
                >U11, stop T303, start T308;         /*#SE*/
     U1    ?Status(cs!=0,ca=96,incomp=0) !Rel(ca=96)
                >U11, stop T303, start T308;         /*#SE*/
     U1    ?Status(cs!=0,ca=97,incomp=0) !Rel(ca=97)
                >U11, stop T303, start T308;         /*#SE*/
     U1    ?Status(cs!=0,ca=99,incomp=0) !Rel(ca=99)
                >U11, stop T303, start T308;         /*#SE*/
     U1    ?Status(cs!=0,ca=100,incomp=0)       !Rel(ca=100)
                >U11, stop T303, start T308;         /*#SE*/
     U1    ?Status(cs!=0,ca=101,incomp=0)       !Rel(ca=101)
                >U11, stop T303, start T308;          /*#SE*/
     U1    ?Init_stat_enq     !Status_enq >U1,start T322;
     /*#SE*/
     U1    ?Status(cs!=0,ca=30,incomp=0)!Null  > U1, stop T322;
     U1    ?Rel_req    !Rel(ca=16) >U11, stop T303, start T308;
     /*#CC*/
     U1    ?T303/[#<1] !Setup             >U1, start T303;
     U1    ?T303/[#=1] !Rel_conf    >U0;
     U1    ?T322/[#=0] !comb5       >U0;
/* Unexpected Msg */
     U1    ?Conn_Ack   !Status(ca=101)    >U1;  /*#MS*/
     U1    ?Rel        !comb1             >U0, stop T303;   /*#MS*/
/* IE Miss/Err */
     U1    ?Call_Pro(Mie=1)!Status(ca=96)   >U1;
                                    /* MIEM, response to SETUP */
     U1    ?Call_Pro(Mie=2)!Status(ca=100) >U1;
                                    /* MIECE, response to SETUP */
     U1    ?Call_Pro(NMie=1)!Proceeding_ind
                >U3, stop T303, start T310;    /*#UIE */
     U1    ?Call_Pro(NMie=3)!Proceeding_ind
                >U3, stop T303, start T310;    /*#RUIE*/
     U1    ?Conn(Mie=1)!Status(ca=96)     >U1;
                                    /* MIEM, response to SETUP */
     U1    ?Conn(Mie=2)!Status(ca=100) >U1;
                                    /* MIECE, response to SETUP */
     U1    ?Conn(NMie=1)!comb2      >U10, stop T303;  /*#UIE*/
     U1    ?Conn(NMie=2)!comb2      >U10, stop T303;  /*#NMIECE */
     U1    ?Conn(NMie=3)!comb2      >U10, stop T303;  /*#RUIE*/
     U1    ?Rel_Com(Mie=1)!Rel_conf(ca=31)      >U0, stop T303;
/*#MIEM*/
     U1    ?Rel_Com(Mie=2)!Rel_conf(ca=31)      >U0, stop T303;
/*#MIECE*/
     U1    ?Rel_Com(NMie=1)  !Rel_conf    >U0, stop T303; /*#UIE*/
     U1    ?Rel_Com(NMie=3)  !Rel_conf    >U0, stop T303; /*RUIE*/
     U1    ?Status_Enq(NMie=1)      !Status(cs=1,ca=30)  >U1; /*#UIE*/
     U1    ?Status(Mie=1)           !Status(ca=96)  >U1; /* MIEM */
     U1    ?Status(Mie=2)           !Status(ca=100) >U1; /* MIECE */

/*U3: Outgoing Call Proceding*/
     U3    ?Conn !comb2             >U10, stop T310; /*#CA*/
     U3    ?Rel  !Rel_ind    >U12, stop T310; /*#CC*/
     U3    ?Rel_Com!Rel_conf >U0, stop T310; /*#CC first clearing mess.
*/
     U3    ?Status_enq !Status(cs=3,ca=30)     >U3;         /*#SE*/
     U3    ?Status(cs=0)      !Rel_conf    >U0, stop T310;   /*#SE*/
```

```
      U3     ?Status(cs!=0,incomp=1) !Rel(ca=101)
                       >U11, stop T310, start T308;   /*#SE*/
      U3     ?Status(cs!=0,ca=96,incomp=0) !Rel(ca=96)
                       >U11, stop T310, start T308;  /*#SE*/
      U3     ?Status(cs!=0,ca=97,incomp=0) !Rel(ca=97)
                       >U11, stop T310, start T308;   /*#SE*/
      U3     ?Status(cs!=0,ca=99,incomp=0) !Rel(ca=99)
                       >U11, stop T310, start T308;   /*#SE*/
      U3     ?Status(cs!=0,ca=100,incomp=0)!Rel(ca=100)
                       >U11, stop T310, start T308;   /*#SE*/
      U3     ?Status(cs!=0,ca=101,incomp=0)!Rel(ca=101)
                       >U11, stop T310, start T308;   /*#SE*/
      U3     ?Init_stat_enq    !Status_enq >U3,start T322;
 /*#SE*/
      U3     ?Status(cs!=0,ca=30,incomp=0)!Null  > U3, stop T322;
      U3     ?Rel_req    !Rel(ca=16) >U11, stop T310, start T308;
      /*#CC*/
      U3     ?T310/[#=0] !Rel(ca=102)       >U11, stop T310, start T308;
      U3     ?T322/[#=0] !comb5       >U0;
/* Unexpected Msg */
      U3     ?Conn_Ack    !Status(ca=101)    >U3;  /*#MS*/
      U3     ?Call_Pro    !Status(ca=101)    >U3;  /*#MS*/
/* IE Miss/Err */
      U3     ?Conn(NMie=1)!comb2      >U10, stop T310; /*#UIE*/
      U3     ?Conn(NMie=2)!comb2      >U10, stop T310; /*#NMIECE*/
      U3     ?Conn(NMie=3)!comb2      >U10, stop T310; /*#RUIE*/
      U3     ?Rel(Mie=1) !Rel_ind(ca=31)    >U12, stop T310; /*#MIEM*/
      U3     ?Rel(Mie=2) !Rel_ind(ca=31)    >U12, stop T310; /*#MIECE*/
      U3     ?Rel(NMie=1)!Rel_ind     >U12, stop T310; /*#UIE*/
      U3     ?Rel(NMie=3)!Rel_ind     >U12, stop T310; /*#RUIE*/
      U3     ?Rel_Com(Mie=1)!Rel_conf(ca=31)      >U0, stop T310;
 /*#MIEM*/
      U3     ?Rel_Com(Mie=2)!Rel_conf(ca=31)      >U0, stop T310;
 /*#MIECE*/
      U3     ?Rel_Com(NMie=1)  !Rel_conf    >U0, stop T310; /*#UIE*/
      U3     ?Rel_Com(NMie=3)  !Rel_conf    >U0, stop T310; /*#RUIE*/
      U3     ?Status_Enq(NMie=1)      !Status(cs=3,ca=30)   >U3; /*#UIE*/
      U3     ?Status(Mie=1)           !Status(ca=96)   >U3; /*# MIEM */
      U3     ?Status(Mie=2)           !Status(ca=100)  >U3; /*# MIECE */

/*U6: Call Present */
      U6     ?Rel         !Rel_ind     >U12; /*#CC*/
      U6     ?Status_enq !Status(cs=6,ca=30)      >U6;  /*#SE*/
      U6     ?Status(cs=0)!Rel_conf  >U0;  /*#SE*/
      U6     ?Status(cs!=0,incomp=1) !Rel(ca=101)
                   >U11, start T308; /*#SE*/
      U6     ?Status(cs!=0,ca=96,incomp=0) !Rel(ca=96)
                   >U11, start T308; /*#SE*/
      U6     ?Status(cs!=0,ca=97,incomp=0) !Rel(ca=97)
                   >U11, start T308; /*#SE*/
      U6     ?Status(cs!=0,ca=99,incomp=0) !Rel(ca=99)
                   >U11, start T308; /*#SE*/
      U6     ?Status(cs!=0,ca=100,incomp=0)!Rel(ca=100)
                   >U11, start T308; /*#SE*/
      U6     ?Status(cs!=0,ca=101,incomp=0)!Rel(ca=101)
                   >U11, start T308; /*#SE*/
      U6     ?Init_stat_enq!Status_enq      >U6;              /*#SE*/
      U6     ?Status(cs!=0,ca=30,incomp=0)!Null  > U6, stop T322;
      U6     ?Setup_res!Conn          >U8, start T313;  /*#CA*/
      U6     ?Proceeding_req    !Call_Pro    >U9;  /*#CA*/
      U6     ?Rel_req    !Rel(ca=16) >U11, start T308; /*#CC*/
      U6     ?Rel_resp(ca=88)!Rel_Com(ca=88)      >U0;  /*#CC*/
      U6     ?Rel_resp(ca=17)!Rel_Com(ca=17)      >U0;  /*#CC*/
      U6     ?Rel_resp(ca=21)!Rel_Com(ca=21)      >U0;  /*#CC*/
```

```
        U6      ?Rel_resp(ca=23)!Rel_Com(ca=23)       >U0;   /*#CC*/
        U6      ?T322/[#=0] !comb5       >U0;
/* Unexpected Msg */
        U6      ?Conn           !Status(ca=101)    >U6;   /*#MS*/
        U6      ?Conn_Ack       !Status(ca=101)    >U6;   /*#MS*/
        U6      ?Rel_Com        !Rel_conf    >U0;   /*#MS*/
        U6      ?Call_Pro       !Status(ca=101)    >U6;   /*#MS*/
/* IE Miss/Err */
        U6      ?Rel(Mie=1) !Rel_ind(ca=31)    >U12; /*#MIEM*/
        U6      ?Rel(Mie=2) !Rel_ind(ca=31)    >U12; /*#MIECE*/
        U6      ?Rel(NMie=1)       !Rel_ind     >U12; /*#UIE*/
        U6      ?Rel(NMie=3)       !Rel_ind     >U12; /*#RUIE*/
        U6      ?Status_Enq(NMie=1)       !Status(cs=6,ca=30)   >U6; /*#UIE*/
        U6      ?Status(Mie=1)            !Status(ca=96)    >U6; /* MIEM */
        U6      ?Status(Mie=2)            !Status(ca=100)  >U6; /* MIECE */

/*U8: Connect Request*/
        U8      ?Conn_Ack      !Setup_comp_ind          >U10, stop T313; /*#CA*/
        U8      ?Rel           !Rel_ind     >U12, stop T313;          /*#CC*/
        U8      ?Status_enq !Status(cs=8,ca=30)      >U8;   /*#SE*/
        U8      ?Status(cs=0)       !Rel_conf           >U0, stop T313;
        /*#SE*/
        U8      ?Status(cs!=0,incomp=1) !Rel(ca=101)
                        >U11, stop T313, start T308;  /*#SE*/
        U8      ?Status(cs!=0,ca=96,incomp=0) !Rel(ca=96)
                        >U11, stop T313, start T308; /*#SE*/
        U8      ?Status(cs!=0,ca=97,incomp=0) !Rel(ca=97)
                        >U11, stop T313, start T308; /*#SE*/
        U8      ?Status(cs!=0,ca=99,incomp=0) !Rel(ca=99)
                        >U11, stop T313, start T308; /*#SE*/
        U8      ?Status(cs!=0,ca=100,incomp=0)!Rel(ca=100)
                        >U11, stop T313, start T308; /*#SE*/
        U8      ?Status(cs!=0,ca=101,incomp=0)!Rel(ca=101)
                        >U11, stop T313, start T308; /*#SE*/
        U8      ?Init_stat_enq      !Status_enq,start T322   >U8;   /*#SE*/
        U8      ?Status(cs!=0,ca=30,incomp=0)!Null  > U8, stop T322;
        U8      ?Rel_req      !Rel(ca=16) >U11, stop T313, start T308;
        /*#CC*/
        U8      ?T313/[#=0] !Rel(ca=102)        >U11, stop T313, start T308;
        U8      ?T322/[#=0] !comb5       >U0;
/* Unexpected Msg */
        U8      ?Conn           !Status(ca=101)    >U8; /*#MS*/
        U8      ?Call_Pro       !Status(ca=101)    >U8; /*#MS*/
        U8      ?Rel_Com        !Rel_conf     >U0, stop T313; /*#MS*/
/* IE Miss/Err */
        U8      ?Conn_Ack(NMie=1) !Setup_comp_ind    >U10, stop T313;
/*#UIE*/
        U8      ?Conn_Ack(NMie=3) !Setup_comp_ind    >U10, stop T313;
/*#RUIE*/
        U8      ?Rel(Mie=1) !Rel_ind(ca=31)    >U12, stop T313; /*#MIEM*/
        U8      ?Rel(Mie=2) !Rel_ind(ca=31)    >U12, stop T313; /*#MIECE*/
        U8      ?Rel(NMie=1)       !Rel_ind     >U12, stop T313; /*#UIE*/
        U8      ?Rel(NMie=3)       !Rel_ind     >U12, stop T313; /*#RUIE*/
        U8      ?Status_Enq(NMie=1)       !Status(cs=8,ca=30)   >U8; /*#UIE*/
        U8      ?Status(Mie=1)            !Status(ca=96)    >U8; /* MIEM */
        U8      ?Status(Mie=2)            !Status(ca=100)  >U8; /* MIECE */

/*U9: Incoming Call Proceeding */
        U9      ?Rel           !Rel_ind      >U12; /*#CC*/
        U9      ?Status_enq !Status(cs=9,ca=30)      >U9; /*#SE*/
        U9      ?Status(cs=0)       !Rel_conf    >U0;   /*#SE*/
        U9      ?Status(cs!=0,incomp=1) !Rel(ca=101)
                        >U11, start T308; /*#SE*/
        U9      ?Status(cs!=0,ca=96,incomp=0) !Rel(ca=96)
```

```
                >U11, start T308; /*#SE*/
     U9     ?Status(cs!=0,ca=97,incomp=0) !Rel(ca=97)
                >U11, start T308; /*#SE*/
     U9     ?Status(cs!=0,ca=99,incomp=0) !Rel(ca=99)
                >U11, start T308; /*#SE*/
     U9     ?Status(cs!=0,ca=100,incomp=0)!Rel(ca=100)
                >U11, start T308; /*#SE*/
     U9     ?Status(cs!=0,ca=101,incomp=0)!Rel(ca=101)
                >U11, start T308; /*#SE*/
     U9     ?Init_stat_enq    !Status_enq >U9,start T322;   /*#SE*/
     U9     ?Status(cs!=0,ca=30,incomp=0)!Null  > U9, stop T322;
     U9     ?Setup_res  !Conn       >U8, start T313; /*#CA*/
     U9     ?Rel_req    !Rel(ca=16) >U11, start T308; /*#CC*/
     U9     ?T322/[#=0] !comb5      >U0;
/* Unexpected Msg */
     U9     ?Conn       !Status(ca=101)    >U9; /*#MS*/
     U9     ?Conn_Ack   !Status(ca=101)    >U9; /*#MS*/
     U9     ?Call_Pro   !Status(ca=101)    >U9; /*#MS*/
     U9     ?Rel_Com    !Rel_conf    >U0; /*#MS*/
/* IE Miss/Err */
     U9     ?Rel(Mie=1) !Rel_ind(ca=31)    >U12; /*#MIEM*/
     U9     ?Rel(Mie=2) !Rel_ind(ca=31)    >U12; /*#MIECE*/
     U9     ?Rel(NMie=1)        !Rel_ind     >U12; /*#UIE*/
     U9     ?Rel(NMie=3)        !Rel_ind     >U12; /*#RUIE*/
     U9     ?Status_Enq(NMie=1)       !Status(cs=9,ca=30)   >U9; /*#UIE*/
     U9     ?Status(Mie=1)            !Status(ca=96)    >U9; /* MIEM */
     U9     ?Status(Mie=2)            !Status(ca=100)  >U9; /* MIECE */

/*U10:      Active State*/
     U10    ?Rel   !Rel_ind     >U12; /*#CC*/
     U10    ?Status_enq !Status(cs=10,ca=30)     >U10; /*#SE*/
     U10    ?Status(cs=0)       !Rel_conf    >U0;  /*#SE*/
     U10    ?Status(cs!=0,incomp=1) !Rel(ca=101)
                >U11, start T308; /*#SE*/
     U10    ?Status(cs!=0,ca=96,incomp=0) !Rel(ca=96)
                >U11, start T308; /*#SE*/
     U10    ?Status(cs!=0,ca=97,incomp=0) !Rel(ca=97)
                >U11, start T308; /*#SE*/
     U10    ?Status(cs!=0,ca=99,incomp=0) !Rel(ca=99)
                >U11, start T308; /*#SE*/
     U10    ?Status(cs!=0,ca=100,incomp=0)!Rel(ca=100)
                >U11, start T308; /*#SE*/
     U10    ?Status(cs!=0,ca=101,incomp=0)!Rel(ca=101)
                >U11, start T308; /*#SE*/
     U10    ?Init_stat_enq    !Status_enq >U10,start T322;   /*#SE*/
     U10    ?Status(cs!=0,ca=30,incomp=0)!Null  > U10, stop T322;
     U10    ?Rel_req    !Rel(ca=16) >U11, start T308; /*#CC*/
     U10    ?T322/[#=0] !comb5      >U0;
/* Unexpected Msg */
     U10    ?Conn       !Status(ca=101)    >U10; /*#MS*/
     U10    ?Conn_Ack   !Status(ca=101)    >U10; /*#MS*/
     U10    ?Call_Pro   !Status(ca=101)    >U10; /*#MS*/
     U10    ?Rel_Com    !Rel_conf    >U0;   /*#MS*/
/* IE Miss/Err */
     U10    ?Rel(Mie=1) !Rel_ind(ca=31)    >U12; /*#MIEM*/
     U10    ?Rel(Mie=2) !Rel_ind(ca=31)    >U12; /*#MIECE*/
     U10    ?Rel(NMie=1)        !Rel_ind     >U12; /*#UIE*/
     U10    ?Rel(NMie=3)        !Rel_ind     >U12; /*#RUIE*/
     U10    ?Status_Enq(NMie=1)       !Status(cs=10,ca=30)   >U10; /*#UIE*/
     U10    ?Status(Mie=1)            !Status(ca=96)    >U10; /* MIEM */
     U10    ?Status(Mie=2)            !Status(ca=100)  >U10; /* MIECE */

/*U11:      Release Request*/
     U11    ?Rel   !Rel_conf    >U0, stop T308; /*#CC*/
```

```
        U11    ?Rel_Com     !Rel_conf    >U0, stop T308; /*#CC*/
        U11    ?Status_enq !Status(cs=11,ca=30)     >U11; /*#SE*/
        U11    ?Status(cs=0)      !Rel_conf    >U0, stop T308;   /*#SE*/
        U11    ?Status(cs!=0)     !Null >U11; /*#SE*/
        U11    ?Init_stat_enq     !Status_enq,start T322  >U11; /*#SE*/
        U11    ?Status(cs!=0,ca=30,incomp=0)!Null > U11, stop T322;
        U11    ?T308/[#<1] !Rel(ca=102)       >U11, start T308;
        U11    ?T308/[#=1] !Rel_conf    >U0, stop T308;
        U11    ?T322/[#=0] !comb5       >U0;
/* Unexpected Msg */
        U11    ?Conn         !Status(ca=101)    >U11; /*#MS*/
        U11    ?Conn_Ack     !Status(ca=101)    >U11; /*#MS*/
        U11    ?Call_Pro     !Status(ca=101)    >U11; /*#MS*/
/* IE Miss/Err */
        U11    ?Rel(Mie=1) !Rel_conf(ca=31)   >U0, stop T308; /*#MIEM*/
        U11    ?Rel(Mie=2) !Rel_conf(ca=31)   >U0, stop T308; /*#MIECE*/
        U11    ?Rel(NMie=1)       !Rel_conf    >U0, stop T308; /*#UIE*/
        U11    ?Rel(NMie=3)       !Rel_conf    >U0, stop T308; /*#RUIE*/
        U11    ?Rel_Com(NMie=1)   !Rel_conf    >U0, stop T308; /*#UIE*/
        U11    ?Rel_Com(NMie=2)   !Rel_conf    >U0, stop T308; /*#NMIECE*/
        U11    ?Rel_Com(NMie=3)   !Rel_conf    >U0, stop T308; /*#RUIE*/
        U11    ?Status_Enq(NMie=1)      !Status(cs=11,ca=30)   >U11; /*#UIE*/
        U11    ?Status(Mie=1)           !Status(ca=96)   >U11; /* MIEM */
        U11    ?Status(Mie=2)           !Status(ca=100)  >U11; /* MIECE */

/*U12:       Release Indicator*/
        U12    ?Status_enq !Status(cs=12,ca=30)     >U12; /*#SE*/
        U12    ?Status(cs=0)      !Rel_conf    >U0;    /*#SE*/
        U12    ?Status(cs!=0)     !Null >U12;          /*#SE*/
        U12    ?Init_stat_enq     !Status_enq >U12,start T322;  /*#SE*/
        U12    ?Status(cs!=0,ca=30,incomp=0)!Null > U12, stop T322;
        U12    ?Rel_resp    !Rel_Com     >U0;  /*#CC*/
        U12    ?T322/[#=0] !comb5       >U0;
/* Unexpected Msg */
        U12    ?Conn         !Status(ca=101)    >U12; /*#MS*/
        U12    ?Conn_Ack     !Status(ca=101)    >U12; /*#MS*/
        U12    ?Call_Pro     !Status(ca=101)    >U12; /*#MS*/
        U12    ?Rel          !Null        >U12;  /*#MS*/
        U12    ?Rel_Com      !Null        >U12;  /*#MS*/
/* IE Miss/Err */
        U12    ?Status_Enq(NMie=1)      !Status(cs=12,ca=30)   >U12;  /*#UIE*/
        U12    ?Status(Mie=1)           !Status(ca=96)   >U12; /* MIEM */
        U12    ?Status(Mie=2)           !Status(ca=100)  >U12; /* MIECE */

 /*-------------------------------------------------------------*/

end;
```

Appendix B: Coordinator transitions and tests for coherent transitions

In this appendix we show variables, inputs, and coherent transitions that are added to the Proc FSM after combining it with the Coordinator FSM. Coordinator functions are explained in the comments next to the coherent transitions. All coherent transitions belong to a function group #GE (General Error). In the coherent transitions, when starting state is written as *, it means for all states. When the ending state is written as _ , it means the same as the starting state. The list of the inputs that lead to same output and ending state are given in brackets.

```
variables
ge integer; /* error condition :
            1= Protocol discriminator error
            2= Message length too short
            3= CR format error
            4= CR not related to an active call
            5= global CR */
inputs:
 /* inputs processed by the coordinator */
      Setup(ge=1) :PDU;
      Setup(ge=2) :PDU;
      Setup(ge=3) :PDU;
      Setup(ge=4, flag=1):PDU; /* CR is OK but flag is for outgoing call
*/
      Setup(ge!=4):PDU;       /* CR relates to active call or call in
progress */
      Setup(ge=5) :PDU;
      Setup(ge=6) :PDU;

      Conn(ge=1)  :PDU;
      Conn(ge=2)  :PDU;
      Conn(ge=3)  :PDU;
      Conn(ge=4)  :PDU;
      Conn(ge=5)  :PDU;
      Conn(ge=6)  :PDU;

      Call_Pro(ge=1):PDU;
      Call_Pro(ge=2):PDU;
      Call_Pro(ge=3):PDU;
      Call_Pro(ge=4):PDU;
      Call_Pro(ge=5):PDU;
      Call_Pro(ge=6):PDU;

      Conn_Ack(ge=1):PDU;
      Conn_Ack(ge=2):PDU;
      Conn_Ack(ge=3):PDU;
      Conn_Ack(ge=4):PDU;
      Conn_Ack(ge=5):PDU;
      Conn_Ack(ge=6):PDU;

      Rel(ge=1):PDU;
      Rel(ge=2):PDU;
```

```
        Rel(ge=3):PDU;
        Rel(ge=4):PDU;
        Rel(ge=5):PDU;
        Rel(ge=6):PDU;

        Rel_Com(ge=1):PDU;
        Rel_Com(ge=2):PDU;
        Rel_Com(ge=3):PDU;
        Rel_Com(ge=4):PDU;
        Rel_Com(ge=5):PDU;
        Rel_Com(ge=6):PDU;

        Status_enq(ge=1):PDU;
        Status_enq(ge=2):PDU;
        Status_enq(ge=3):PDU;
        Status_enq(ge=4):PDU;
        Status_enq(ge=5):PDU;
        Status_enq(ge=6):PDU;

        Status(ge=1):PDU;
        Status(ge=2):PDU;
        Status(ge=3):PDU;
        Status(ge=4,cs=0):PDU;
        Status(ge=4,cs!=0):PDU;
        Status(ge=5):PDU;
        Status(ge=6):PDU;
/* end of inputs processed by the coordinator */

transitions:
 /* coherent Coordinator transitions */

*       ?(Setup(ge=1),Conn(ge=1),Conn_Ack(ge=1), Call_Pro(ge=1),Rel(ge=1),
        Rel_Com(ge=1),Status_enq(ge=1), Status(ge=1))
        !Null > _;  /*#GE*/ /* protocol discriminator error */

*       ?(Setup(ge=2),Conn(ge=2),Conn_Ack(ge=2),Call_Pro(ge=2),Rel(ge=2),
        Rel_Com(ge=2),Status_enq(ge=2), Status(ge=2))
        !Null > _; /*#GE*/  /* message too short */

*       ?(Setup(ge=3),Conn(ge=3),Conn_Ack(ge=3),Call_Pro(ge=3),Rel(ge=3),
        Rel_Com(ge=3),Status_enq(ge=3), Status(ge=3))
        !Null > _; /*#GE*/  /* call reference format error */

*       ?(Conn(ge=4),Conn_Ack(ge=4),Call_Pro(ge=4),Rel(ge=4))
        !Rel_Com(ca=81) >_; /*#GE*/ /* CR not related to an active call */

*       ?Rel_Com(ge=4) !Null >_;/*#GE*//* CR not related to active call */

*       ?Setup(ge=4,flag=1) !Null > U0; /*#GE*/
        /* CR not related to an active call, but flag has a
        value for outgoing call */

*       ?(Setup(ge=5),Conn(ge=5),Conn_Ack(ge=5),Call_Pro(ge=5),Rel(ge=5),
        Rel_Com(ge=5),Status_enq(ge=5))
        !Null > _; /*#GE*/ /* global CR */

{U1,U3,U6,U8,U9,U10,U11,U12} ?Setup !Null > _; /*#GE*/
  /* Setup is unexpected, related to active call*/

{U1,U3,U6,U8,U9,U10,U11,U12} ?Status(ge=4,cs=0) !Null >_;
        /*#GE*//* Status not related to active call */

*       ?Unrecognized     !Status(ca=97)     >_;  /*#GE*/
```

```
            /* Message not recognized */

{U1,U3,U6,U8,U9,U10,U11,U12}  ?Status(ge=4,cs!=0) !Rel_Com(ca=101)>_;
/*#GE*/
/* Status not related to an active call and with incompatible current
state */

{U1,U3,U6,U8,U9,U10,U11,U12}  ?Status_enq(ge=4)  !Status(cs=0,ca=30)
      >_;         /*#GE*//* Status Enq not related to active call */
```

Two out of the 24 test cases that are developed for testing the coherent transitions are shown below. For each coherent transitions two test cases are generated. If a more refined testing is necessary, the test developer may reduce the input lists in the coherent transitions.

```
/* Test cases derived from q2931.fsm */
DCL   id  INTEGER;
DCL   rs  INTEGER;
DCL   cs  INTEGER;
DCL   ca  INTEGER;
DCL   incomp  INTEGER;
DCL   CR  INTEGER;
DCL   Mie  INTEGER;
DCL   NMie  INTEGER;
DCL   PD  INTEGER;

#Testcase 1:
/* Test purpose: verify the transition in state U0 in input Conn(ge=2)
*/
/* This test case tests a coherent transition*/
/* transitions in set are : */
/*
{ U0, U1, U3, U6, U8, U9, U10, U11, U12 } ?( Setup(ge=2), Conn(ge=2),
Conn_Ack(ge=2), Call_Pro(ge=2), Rel(ge=2), Rel_Com(ge=2),
Status_enq(ge=2), Status(ge=2) } !NULL >_;*/
/* Transition Under Test in state U0 on input Conn(ge=2) : */
      !Conn(ge=2):PDU;
/* Identifying U0 state: */
      !Status_enq:PDU;
      ?Status(ca=30,cs=0):PDU;
/* No postamble needed */

#Testcase 2:
/* Test purpose: verify the transition in state U1 in input
Conn_Ack(ge=4) */
/* This test case tests a coherent transition*/
/* transitions in set are : */
/*
{ U0, U1, U3, U6, U8, U9, U10, U11, U12 } ?( Conn(ge=4), Conn_Ack(ge=4),
Call_Pro(ge=4), Rel(ge=4) } !Rel_Com >_;*/
/* Preamble to U1 state: */
      !Setup_req;
      ?Setup:PDU;
/* Transition Under Test in state U1 on input Conn_Ack(ge=4) : */
      !Conn_Ack(ge=4):PDU;
      ?Rel_Com(ca=81):PDU;
/* Identifying U1 state: */
```

```
        !Status_enq:PDU;
        ?Status(ca=30,cs=1):PDU;
   /* Postamble from U1 state: */
        !Setup(flag=1,ge=4):PDU;
```

Appendix C: Test cases for conformance testing

```
#Testcase 1:
/* Test purpose: verify the transition in state U0 in input Setup */
/* Transition Under Test in state U0 on input Setup : */
      !Setup:PDU;
      ?Setup_ind;
/* Postamble from U6 state: */
      !Rel_Com:PDU;
      ?Rel_conf;

#Testcase 2:
/* Test purpose: verify the transition in state U10 in input Conn */
/* Preamble to U10 state: */
      !Setup_req;
      ?Setup:PDU;
      !Conn(NMie=3):PDU;
      ?Conn_Ack:PDU,Setup_conf;
/* Transition Under Test in state U10 on input Conn : */
      !Conn:PDU;
      ?Status(ca=101):PDU;
/* Postamble from U10 state: */
      !Rel_Com:PDU;
      ?Rel_conf;
```

Testing timers:

```
#TestCase 3:
 /* Test purpose: verify if the timer T303 starts in state U1 on input
Setup_req */
/* No Preamble for U0 state */
/* Part under test for timer T303 */
      !Setup_req;
      ?Setup:PDU, Start T303;
      ?Setup:PDU, Stop T303;
/* Postamble from U1 state: */
      !Rel:PDU;
      ?Rel_Com:PDU,Rel_conf;

#TestCase 4:
 /* Test purpose: verify if the timer T303 fires max times in state U1
*/
/* No Preamble for U0 state */
/* Part under test for timer T303 */
      !Setup_req;
      ?Setup:PDU, Start T303;
      ?Setup:PDU[1];
      ?Rel_conf, Stop T303;
/* No postamble needed */

#TestCase 5:
 /* Test purpose: verify if the timer T303 stops after input Conn in
state U1 */
/* No Preamble for U0 state */
/* Part under test for timer T303 */
      !Setup_req;
      ?Setup:PDU;
      !Conn:PDU;
      ?Conn_Ack:PDU,Setup_conf, Start T303;
      ?Timeout(T303);
/* Postamble from U10 state: */
      !Rel_Com:PDU;
      ?Rel_conf;
```

Transitions with two timers:

```
/* Test purpose: verify if the timer T313 stops after
 input Rel_req in state U8 */
/* Preambule to U8 */
      !Setup:PDU;
      ?Setup_ind;
      !Setup_res;
      ?Conn:PDU, Start T313;
      ?Timeout(max(0,T313-T308));
/* Part under test for timer T313*/
      !Rel_req;
      ?Rel(ca=16):PDU;
      ?Timeout(min(T313,T308));
/* Identify state U11 */
      !Setup_enq:PDU;
      ?Status(ca=30,cs=11);
/* Postamble from U11: */
      !Rel:PDU;
      ?Rel_conf;
```

Testing T322:

```
/* Test purpose: verify if the timer T322 starts after
 input InitiateStatusEnquiry in state U6 */
/* Preambule to U6 */
      !Setup:PDU;
      ?Setup_ind;
/* Part under test for timer T322*/
      !InitStatusEnquiry, start T322
      ?Status_enq:PDU
      ?Rel_Com(ca=41):PDU,Rel_conf;
/* Idnetifying state U0 */
      !Status_Enq:PDU;
      ?Status(cs=0,ca=30);
/* No  postamble needed */

/* Test purpose: verify if the timer T322 stops after
 input Status in state U6 */
/* Preambule to U6 */
      !Setup:PDU;
      ?Setup_ind;
/* Part under test for timer T322 */
      !InitStatusEnquiry, start T322
      ?Status_enq:PDU
      !Status(cs!=0,ca=30,incomp=1):PDU;
      ?Timeout(T322);
/* Idnetifying state U6*/
      !Status_Enq:PDU;
      ?Status(cs=6,ca=30);
/* Postamble from U6 */
      !Rel_Com:PDU;
      ?Rel_conf;
```

Appendix D: Example of the data part testing

In the following, an example of the initialization of the SETUP message for data part testing is given. The `message_content_type` is a type of a general message, while `ie_type` is a type of general IE. The `pdu` is a data of the signal `message` (see Figure 5.4). First, an example of SETUP with valid values is given. SETUP with MIEM and SETUP with MIECE use an SDL procedure `default_setup` that sets the values of the SETUP to valid default values.

```
NEWTYPE message_content_type STRUCT
      pr_disc protocol_discriminator_type;
      CR call_reference_type; /* size 4 bytes */
      message_type octet; /* size 2 bytes */
      ie ie_type;
ENDNEWTYPE;

NEWTYPE ie_type STRUCT
      cause cause_type;
      call_state call_state_type;
      AAL_param AAL_param_type;
      ATM_traffic_desc ATM_traffic_desc_type;
      connection_id connection_id_type;
      QoS Qos_type;
      B_HLI B_HLI_type;
      B_BC B_BC_type;
      B_LLI B_LLI_type;
      B_sending_complete B_sending_complete_type;
      B_repeat_indic B_repeat_indic_type;
      calling_nb calling_nb_type;
      calling_subaddress subaddress_type;
      called_nb called_nb_type;
      called_subaddress subaddress_type;
      transit_network_selection transit_network_selection_type;
  ENDNEWTYPE;
```

SETUP with valid data:

```
 DCL pdu message_content_type;

 TASK pdu!pr_disc := 9;
 TASK pdu!CR!flag := 0; /* incoming call */
 TASK pdu!CR!value := 1;
 TASK pdu!message_type := SETUP;
 /* ATMPara IE values */
 TASK pdu!ie!ATM_param!presence := true;
 TASK pdu!ie!ATM_param!coding_standard := 0;
 TASK pdu!ie!ATM_param!AAL_type := 5;
 TASK pdu!ie!ATM_param!forward_max_size!presence := true;
 TASK pdu!ie!ATM_param!forward_max_size!value := 1024;
 TASK pdu!ie!ATM_param!backward_max_size!presence := true;
 TASK pdu!ie!ATM_param!backward_max_size!value := 1024;
 TASK pdu!ie!ATM_param!SSCS_type!presence := true;
 TASK pdu!ie!ATM_param!SSCS_type!value := ASSURED;
 /* ATMTraf IE values */
 TASK pdu!ie!ATM_traffic_desc!presence := true;
 TASK pdu!ie!ATM_traffic_desc!fpcr_0_1!presence := true;
 TASK pdu!ie!ATM_traffic_desc!fpcr_0_1!value := 0;
 TASK pdu!ie!ATM_traffic_desc!bpcr_0_1!presence := true;
 TASK pdu!ie!ATM_traffic_desc!bpcr_0_1!value := 0;
 /* B_BC IE values */
 TASK pdu!ie!B_BC!presence := true;
 TASK pdu!ie!B_BC!coding_standard := 0;
 TASK pdu!ie!B_BC!class natural := 0;
 TASK pdu!ie!B_BC!clipping := 0;
 TASK pdu!ie!B_BC!user_plane := 0;
 /* B_LLI IE values */
 TASK pdu!ie!B_LLI!nb_occ := 1;
 TASK pdu!ie!B_LLI!occ(1)!presence : = true;
 TASK pdu!ie!B_LLI!occ(1)!coding_standard := 0;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_1!presence := false;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_2!presence : true;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_2!value := 6;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_2!codings!presence := true;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_2!codings!mode := 0;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_2!codings!Q933 := 0;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_2!window_size := 1;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!presence := true;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!value := 6;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!mode!presence := true;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!mode!value := 1;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!default_packet_size!presence :=
true;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!default_packet_size!value := 4;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!packet_window_size!presence :=
true;
 TASK pdu!ie!B_LLI!occ(1)!info_layer_3!packet_window_size!value := 1;
 /* B_HLI IE values */
 TASK pdu!ie!B_HLI!presence := true;
 TASK pdu!ie!B_HLI!hl_type := 0;
 TASK pdu!ie!B_HLI!hl_info!nb_occ := 2;
 TASK pdu!ie!B_HLI!hl_info!occ(1) := 1;
 TASK pdu!ie!B_HLI!hl_info!occ(2) := 2;
 /* QoS IE values */
 TASK pdu!ie!QoS!presence := true;
 TASK pdu!ie!QoS!coding_standard := 0;
 TASK pdu!ie!QoS!class_forward := 0;
 TASK pdu!ie!QoS!class_backward := 0;
 /* CalledNum IE values */
 TASK pdu!ie!called_nb!presence := true;
 TASK pdu!ie!called_nb!coding_standard := 0;
 TASK pdu!ie!called_nb!nb_type := 1;
 TASK pdu!ie!called_nb!plan_id := 1;
 TASK pdu!ie!called_nb!address!nb_occ := 5;
 TASK pdu!ie!called_nb!address!occ(1) := 3;
 TASK pdu!ie!called_nb!address!occ(2) := 2;
 TASK pdu!ie!called_nb!address!occ(3) := 3;
 TASK pdu!ie!called_nb!address!occ(4) := 4;
 TASK pdu!ie!called_nb!address!occ(5) := 9;
 /* ConnId IE values */
 TASK pdu!ie!connection_id!presence := true;
 TASK pdu!ie!connection_id!coding_standard := 0;
 TASK pdu!ie!connection_id!VPAS := 1;
 TASK pdu!ie!connection_id!pref_excl := 0;
 TASK pdu!ie!connection_id!VPCI := 0;
 TASK pdu!ie!connection_id!VCI := 32;
```

SETUP wiht MIEM:

```
 CALL default_setup(pdu);
 TASK pdu!ie!ATM_traffic_desc!presence := false;
```

```
     /* ATMTraff IE is mandatory IE */
```

SETUP wiht MIECE:

```
CALL default_setup(pdu);
TASK pdu!ie!ATM_traffic_desc!fpcr_0_1!presence := false;
     /* fpcr_0_1 must be present in the ATM Traff IE */
```

Appendix E: Example test cases from the interoperability test suite

```
#Testcase 1:
/* Test purpose: verify the transition in state A in input Setup_req_s
*/
/* Transition Under Test in state A on input Setup_req_s : */
      !Setup_req_s;
      ?Setup_ind_r;
/* Identifying B state: */
      !Setup_req_s;
      !Release_resp_r;
      ?Release_conf_s;
/* No postamble needed */

#Testcase 2:
/* Test purpose: verify the transition in state A in input
Setup_req_s(Mie=1) */
/* Transition Under Test in state A on input Setup_req_s(Mie=1) : */
      !Setup_req_s(Mie=1);
      ?Release_conf_s(ca=96);
/* Identifying A state: */
      !Setup_req_s;
      ?Setup_ind_r;
 * Postamble from B state: */
      !Release_resp_r;
      ?Release_conf_s;

#Testcase 3:
/* Test purpose: verify the transition in state A in input
Setup_req_s(Mie=2) */
/* Transition Under Test in state A on input Setup_req_s(Mie=2) : */
      !Setup_req_s(Mie=2);
      ?Release_conf_s(ca=100);
 * Identifying A state: */
      !Setup_req_s;
      ?Setup_ind_r;
 * Postamble from B state: */
      !Release_resp_r;
      ?Release_conf_s;

#Testcase 4:
/* Test purpose: verify the transition in state A in input
Setup_req_s(NMie=2) */
/* Transition Under Test in state A on input Setup_req_s(NMie=2) : */
      !Setup_req_s(NMie=2);
      ?Setup_ind_r,Proceeding_ind_s;
/* Identifying B state: first characterizing sequence*/
      !Setup_req_s;
      !Release_resp_r;
      ?Release_conf_s;
/* No postamble needed */

#TestCase 8:
/* Test purpose: verify the transition in state A in input
Setup_req_s(NMie=2) */
/* Transition Under Test in state A on input Setup_req_s(NMie=2) : */
      !Setup_req_s(NMie=2);
      ?Setup_ind_r,Proceeding_ind_s;
/* Identifying B state: second characterizing sequence*/
      !Proceeding_req_r;
```

```
            ?Proceeding_ind_s;
    /* Postamble from C: */
            !Release_req_s;
            !Release_resp_r;
            ?Release_conf_s;
```