Manuel Alberto Pereira Ricardo

A Methodology for Testing Complex Telecommunications Network Elements

Dissertação apresentada à Faculdade de Engenharia da Universidade do Porto para obtenção do grau de Doutor em Engenharia Electrotécnica e de Computadores realizada sob a orientação do Prof. Doutor Eurico Manuel Elias de Morais Carrapatoso (Prof. Auxiliar) e co-orientação do Prof. Doutor José António Ruela Simões Fernandes (Prof. Associado) da Faculdade de Engenharia da Universidade do Porto

Departamento de Engenharia Electrotécnica e de Computadores
Faculdade de Engenharia da Universidade do Porto
2000

O trabalho de investigação apresentado nesta dissertação foi apoiado pela
JNICT

To Olga, Pedro and Nuno

Abstract

Telecommunication equipments, such as Network Elements, are becoming increasingly complex systems since they have to simultaneously support multiple communication interfaces and several end-user/ management services. Moreover, Network Elements are required to be available (in service) for long periods of time, with service failure to service usage rates in the order of 10^{-5}, to survive overload conditions and to provide services with guaranteed delays for 95% of the usages. These aspects imply that significant project effort has to be placed on the validation of this type of systems.

Traditional Network Element validation practices consist in applying to the equipment two type of tests: conformance and load tests. The former are used to verify the compliance of each communications interface with standard specifications whereas the latter are used to find whether the system can survive for long periods of time and to get simple system measures, such as delays and number of calls or packets. Over the last years, considerable advances were achieved with the application of formal methods to protocol conformance testing. When a protocol is formally specified, it is possible to find a set of tests which can be used to prove a relation between the implementation under test and the specification.

Although these important results are already applicable to real protocols, they are still insufficient to validate complex Network Elements for two main reasons: 1) unlike protocols, these systems are specified using multiple techniques, which are selected by their modeling power and expressiveness rather than by their mathematical basis; 2) these specifications are scattered over tens of documents which, at test time, reveal themselves full of inconsistencies. In addition, the pressure to market a product is usually incompatible with the time required to derive and validate the number of tests which would be required to test these large systems. Thus, there is a gap between formal testing methods and current testing needs.

This thesis attempts to reduce this gap by proposing a testing methodology which improves the current Network Element testing practices by taking advantage of protocol formal testing solutions.

According to the methodology proposed, a Network Element must be first modeled by two sets of components which represent complementary views of

the system: the communications view and the services view.

The communications view concentrates in the communications interfaces of the Network Element, that is, its protocol stacks. A typical communications view component is a protocol state machine. Most of the times these state machines are well known and documented in standards - SDL specifications are often available in ITU and ETSI recommendations. For this reason, the methodology recommends that the communications components should be tested by using directly the most advanced protocol formal testing results.

The service view concentrates on the end-user and management services of the Network Element. A service is usually composed by a set of building blocks (communications components included) possibly shared by other services. A service may also need to use databases, to manipulate complex information and to manage its individual blocks. For these reasons, service specifiers sometimes prefer object oriented modelling techniques. In order to adapt the testing process to this reality, but also to continue taking advantage of formal testing world, the methodology proposes that the services could be modeled by a new set of abstract components - the service view components. These components are specified in an incomplete style, that is, by describing only their user-interfaces interactions. This simplification has, however, two drawbacks: 1) by avoiding the specification of every service building block, that is, the service intra-interfaces behaviour description, numerous invalid behaviours are assumed as valid in the incomplete service specification; 2) simple faults in basic building blocks (e.g. device drivers) may be random in the service abstraction selected and be observable only after long service working times.

The first drawback is overcome by using behaviour tests which are developed based on expected service use cases that are assumed to describe correctly and completly the service utilizations. To overcome the second drawback, the service behaviour tests are made cyclic and allowed to be run for long periods of time so that random faults may be observed. Since very long traces (sequences of events) are available as the result of the execution of cyclic tests, the traditional protocol verdict function is replaced by more general trace evaluation functions which, by incorporating the time dimension, become capable of expressing time-probabilistic-behaviour properties and, by doing that, of providing the means required to express and evaluate the quality of the services.

The described methodology is furthermore enhanced by the proposal of some architectures for service testing. In particular, it is shown how SDL can be used to support them.

The methodology proposed in this thesis is a generalisation of the test method used by the author during the validation of the NEC FA1201 Access Network, which was carried out in INESC Porto during 1995-1999. The main results and contributions of this thesis are, to the best of our knowledge, (1) the testing methodology itself, (2) the service testing method, (3) the method used to describe QoS properties and (4) the service testing architectures.

Resumo

A necessidade crescente dos equipamentos de telecomunicações, entre os quais os Elementos de Rede, suportarem simultaneamente múltiplas interfaces de comunicações e serviços de utilizador/gestão tem feito aumentar significativamente a sua complexidade. Adicionalmente, estes sistemas devem oferecer elevada disponibilidade, taxas de falha de utilização dos serviços na ordem dos 10^{-5}, sobreviver em condições de sobrecarga e fornecer serviços com atrasos máximos garantidos para 95% das utilizações. Como consequência destes requisitos, uma parte significativa dos custos de desenvolvimento destes equipamentos resulta do trabalho de validação.

As práticas tradicionais de validação de Elementos de Rede consistem na aplicação de testes de conformidade e de carga. Os primeiros são usados para verificar a conformidade de cada interface de comunicações com a sua especificação normalizada. Os segundos são usados para avaliar se o sistema sobrevive a longos períodos de utilização, em condições de carga real, e para obtenção de indicadores simples de desempenho, tais como atrasos e número de chamadas ou pacotes. No entanto, durante os últimos anos foram realizados avanços consideráveis na aplicação de métodos formais ao teste de conformidade de protocolos. Quando um protocolo é especificado formalmente é possivel obter automaticamente um conjunto de testes que demonstrem uma relação entre a implementação sob teste e a sua especificação.

Apesar destes resultados poderem ser já aplicáveis a protocolos de complexidade real, são ainda insuficientes para validar Elementos de Rede complexos pelas seguintes razões: 1) ao contrário dos protocolos, estes sistemas são especificados através de múltiplas técnicas que são seleccionadas mais pelo seu poder de modelação e expressividade do que pela suas características matemáticas; 2) estas especificações estão dispersas por dezenas de documentos e, durante o período de testes, revelam normalmente muitas inconsistências. Adicionalmente, a necessidade de se colocar o equipamento no mercado durante a sua *janela de oportunidade* é normalmente incompatível com o tempo necessário para derivar e validar os testes. Pode-se, por este motivo, afirmar que existe um vazio entre os métodos de teste formais e as reais necessidades de teste dos Elementos de Rede.

Esta tese tenta diminuir este vazio propondo uma metodologia de teste de

Elementos de Rede que melhore as práticas de teste actuais tirando partido das soluções de teste formal de protocolos.

Para isso, o Elemento de Rede é representado como dois conjuntos de componentes que traduzem vistas complementares do sistema: a vista das comunicações e a vista dos serviços.

A vista das comunicações representa as interfaces de comunicações do Elemento de Rede, isto é, as suas pilhas protocolares. Um componente típico da vista de comunicações é uma máquina de estados de um protocolo. Normalmente estas máquinas de estados são conhecidas e estão documentadas em normas - existem frequentemente especificações SDL nas recomendações da ITU e da ETSI. Por esta razão, a metodologia proposta recomenda que os componentes de comunicações sejam testados usando os resultados mais avançados do teste formal de protocolos.

A vista dos serviços representa os serviços de utilizador e de gestão fornecidos pelo Elemento de Rede. Um serviço é normalmente composto por um conjunto de blocos constituintes (incluindo os componentes de comunicações), possivelmente partilhados com outros serviços. Um serviço pode ainda aceder a bases de dados, manipular informação complexa e gerir os seus blocos constituintes. Por estes motivos, os especificadores dos serviços usam frequentemente técnicas de modelação orientadas aos objectos. De forma a adaptar o processo de teste a esta realidade, mas também a tirar partido das vantagens dos métodos de teste formal, a metodologia propõe que os serviços sejam modelados por um novo conjunto de componentes abstractos – os componentes da vista de serviços. Estes componentes são descritos de forma incompleta, isto é, através das interacções dos utilizadores com as suas interfaces. Esta simplificação, no entanto, origina dois problemas: 1) não considerando a especificacão de todos os blocos constituintes do serviço, isto é, a descricão dos funcionamentos intra-interfaces, numerosos funcionamentos inválidos são assumidos como válidos na especificação incompleta do serviço; 2) as falhas simples nos blocos constituintes (*device drivers*, por exemplo) podem aparecer de forma aleatória na abstracção de serviço seleccionada e observáveis apenas ao fim de longos tempos de execução do serviço.

Na metodologia proposta o primeiro problema é resolvido através de testes de funcionamento que se baseiam em casos tipo de utilização correcta e completa do serviço. O segundo problema é resolvido transformando os testes de funcionamento em testes cíclicos que devem ser executados durante longos intervalos de tempo de modo a que as falhas aleatórias possam ser observadas. Como resultado destes testes são obtidos traços (sequências de eventos) longos. A função verdicto tradicional usada nos testes de conformidade é, por isso, substituída por funções de avaliação de traços mais genéricas que, incorporando a descrição de tempos, possibilitam a especificação de propriedades de funcionamento probabilístico-temporal e, fazendo-o, fornecem os

mecanismos necessários à descrição e avaliação de propriedades de qualidade de serviço.

A metodologia descrita é ainda melhorada com a proposta de algumas arquitecturas para teste de serviços. Em particular, é demonstrada a utilizaccão de SDL como linguagem de suporte destas arquitecturas.

A metodologia de teste proposta resultou da generalização do método de teste usado pelo autor durante a validação da Rede de Accesso FA1201 da NEC, que decorreu entre 1995 e 1999 no INESC Porto. Os principais resultados e contribuições da tese são (1) a metodologia de teste em si, (2) o método de teste de serviços, (3) o método de especificação de propriedades de qualidade de serviço e (4) a arquitectura de teste de serviços.

Résumé

Les systèmes de télécommunications, parmis lequels les Éléments de Réseau, deviennent systèmes de plus en plus complexes puisqu'ils doivent simultanément supporter multiples interfaces de transmission complexes et services d'usager/ gestion. D'ailleurs, les Éléments de Réseau doivent être disponible (en service) pendant longues périodes de temps, avec un taux de failles de service de 10^{-5}, soufrir surcharges et fournir service avec des délais garantis pour 95% des utilisations. Ces aspects impliquent qu'une part significative des coûts de développement sont mis sur la validation de ce type de systèmes.

Les pratiques traditionnelles de validation d'Éléments Réseau consistent dans l'application de deux type de tests: tests de conformité et de charge. Les premiers sont employés pour vérifier la conformité de chaque interface de communications avec les normes et le deuxième type est employé pour trouver si le système peut résister pendant de longues périodes de temps et obtenir des measures simples sur le système, tels que delais et nombre d'appels ou de trames. Au cours de dernières années, des avances considérables ont été obtenues dans l'application des méthodes formelles aux test de conformité des protocoles. Quand un protocole est décrit dans un langage qui peut être traduit en un certain modèle mathématique, il est possible de développer un ensemble de tests qui peuvent être employés pour montrer une relation entre la réalisation et la spécification.

Bien que ces résultats soient déjà applicables à des protocoles de dimension réel, ils sont insuffisants pour valider les Éléments de Réseau complexes par deux raisons principales: 1) ces systèmes sont habituellement décrits en utilisant multiples techniques de spécification, qui sont choisies plus par leur expressivité que par leur base mathématique; 2) ces spécifications sont dispersées par des dizaines de documents qui, au temps de test, se revèlent pleins d'incohérences. En outre, la pression par rapport au temps de lancer un produit sur le marché est habituellement incompatible avec le temps requis pour dériver et valider le nombre de tests qui seraient exigés pour ces systèmes complexes. Il y a, par ces raisons, une grande distance entre les méthodes de test formelles et les besoins de test actuels.

Cette thèse essaye de contourner ce problème en proposant une méthodologie de test qui améliore les pratiques de test d'Éléments de Reseau actuelles en

profitant des solutions de test formel des protocoles.

Selon la méthodologie proposée, un Élément de Réseau doit être d'abord modelé par deux ensembles de composants qui représentent des vues complémentaires du système: la vue de communications et la vue de services.

La vue de communications représente les interfaces de comunications de l'Éléments de Réseau, c'est à dire, ses piles de protocoles. Un composant typique de la vue de communications est une machine d'états de protocole. Ces machines sont bien connues et documentées dans les normes - les caractéristiques de SDL sont souvent disponibles dans des recommandations d'ITU et d'ETSI. Par cette raison, la méthodologie recommande que les composants de communications doivent être testés en utilisant directement les résultats du test formel des protocoles.

La vue de services se concentre sur les services d'utilisateur et de gestion des Éléments de Réseau. Un service est composé habituellement par un ensemble de modules (composants de communications inclus) probablement partagés avec d'autres services. Un service peut également utiliser des bases de données, manipuler l'information complexe et contrôler ses blocs. Pour ces raisons, les spécificateurs de service préfèrent parfois des techniques orientés aux objects. Afin d'adapter le processus de test à cette réalité, mais de continuer également de profiter du test formel, la méthodologie propose que les services soyent modelés par un nouvel ensemble de composants abstraits - les composants de la vue de service. Ces composants sont indiqués dans un modèle inachevé, c'est à dire, en décrivant seulement les interactions utilisateur-interfaces. Cette simplification a, cependant, deux inconvénients: 1) en évitant la specification de chaque module de service (la description de comportement au delà des interfaces de service), de nombreux comportements incorrects sont assumés comme valides; 2) les défauts simples dans les blocs fonctionnels du composant (par exemple modules de *device drivers*) peuvent être aléatoires dans l'abstraction de service choisie et être observables seulement après de longs temps de service.

Le premier inconvénient est surmonté en utilisant les tests de comportement qui sont développés sur les cas prévus d'utilisation de service qui, on assume, décrivent correctement les utilisations de service. Pour surmonter le deuxième inconvénient, les tests de comportement de service sont rendus cycliques et sont exécutés pendant de longues périodes de sorte qu'on puisse observer des fauts aléatoires. Puisque une trés longue trace (séquence d'events) résulte de l'exécution cyclique du test, la fonction verdict doit être remplacé par une fonction plus générale d'évaluation de traces qui, en incorporant des temps, devient capable d'exprimer des propriétés de comportement dependent du temps et des probabilitées et ainsi, fournir les moyens necessaires pour exprimer et évaluer le qualité du services.

La méthodologie décrite propose aussi quelques architectures pour le test

de services. En particulier, on montre comment SDL peut être employé pour les supporter.

La méthodologie de test proposée est une généralisation de la méthode de test employée par l'auteur pendant la validation du Réseau d'Accès FA1201, fabriqué par NEC, qui a été effectué à INESC Porto pendant 1996-1999. Les résultats et les contributions principaux de ce thèse sont, au meilleur de notre connaissance, (1) la méthodologie de test elle-même, (2) la méthode de test de services, (3) la méthode employée pour exprimer des propriétés de QoS et (4) l'architecture de test de service.

Acknowledgements

I would like to thank my supervisor, Prof. Eurico Carrapatoso, for his friendship, for the freedom and encouragement he gave me to explore new research paths without which this thesis would not have been possible and for the careful verification of the thesis main results. I would like also to thank my co-supervisor, Prof. José Ruela Fernandes, for his friendship and for the stimulating discussions we had about networks. To both of them, also, many thanks for the time they spent in reviewing this thesis and for their useful advice.

In the second place, I would like to thank INESC Porto. There I have always been able to find the resources I needed. In particular, I would like to thank Prof. Mário Jorge Leitão who not only provided me the support required during the NEC project but also some important facilities for writing this thesis.

I would like also to mention and thank JNICT, now FCT, for the three-year PhD. scholarship I was given.

My gratitude goes also to all those who have worked with me during the NEC project. To Jorge Mamede and Fernando Guimarães, who were partners since the first day, to Filipe Pinto, who had to develop an extraordinary complex board, to Luis Anselmo and José António, who were in charge of implementing most of the tasks, to Carlos Silva and Carlos Illa who had to develop the graphical interface and to Sérgio Crisóstomo, who worked on the ISDN part and improved our Web site.

I would like also to thank my colleagues Paula Viana and Raúl Oliveira for their friendship and for having reviewed essential parts of the thesis.

To conclude, I would like to thank my family. To my parents and sister for their encouragement. To my wife, Olga, for her support and understanding and for always reminding me, together with my children Pedro and Nuno, that life is much more than working in a thesis.

Contents

List of Figures

List of Tables

Glossary

AN	Access Network
ASP	Abstract Service Primitive
ATM	Asynchronous Transfer Mode
ATS	Abstract Test Suit
BCC	Bearer Channel Connection
BRI	Basic Rate Interface (ISDN)
ETS	Executable Test Suit
ETSI	European Telecommunications Stadards Institute
FDDI	Fiber Distributed Data Interface
FDT	Formal Description Technique
FSM	Finite State Machine
GDMO	Guidelines for the Definition of Managed Objects
GSM	Global System for Mobile communications
HDLC	Higl-Level Data Link Control
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IAP	Implementation Access Point
IN	Inteligent Networks
ISDN	Integrated Service Digital Network
IOSM	Input/Output State Machine
IP	Internet Protocol
ISO/IEC	International Standards Organization / International Electronical Commission
ITU	International Telecommunications Union
IUT	Implementation Under Test
LE	Local Exchange
LT	Lower Tester
LTS	Labeled Transition System
MSC	Message Sequence Chart
NE	Network Element
OSI	Open Systems Interconnection

PATS	Parametrised Abstract Test Suite
PCO	Point of Control and Observation
PDU	Protocol Data Unit
PETS	Parametrised Executable Test Suite
PICS	Protocol Implementation Conformance Statement
PIXIT	Protocol Implementation eXtra Information for Test
PSTN	Public Switched Telephone Network
V5LE	V5 Local Exchange
SATS	Selected Abstract Test Suite
SDH	Synchronous Digital Hierarchy
SDL	Specification and Description Language
SUT	System Under Test
TCP	Transmission Control Protocol
TMN	Telecommunications Management Network
TTCN	Tree and Tabular Combined Notation
UML	Unified Modelling Language
UT	Upper Tester

Chapter 1

Introduction

Public telecommunications networks have changed dramatically during the last twenty years. First, digital electronics opened the way to new transmission and switching techniques which, gradually with the progress in digital signal prrocessing techniques, led also to the digitalisation of the subscriber access and terminal equipments. During the same period, public data networks started to expand using the concept of packet switching in opposition to circuit switching in the telephony networks. In the last years, digital mobile communications have also exploded. Nowadays, public digital telecommunications networks such as PSTN, ISDN and ATM coexist and interoperate with mobile GSM and packet oriented networks, which are supported, among others, by leased lines, ATM and Frame Relay. At the same time, the logical structuring of these networks also became a reality. Intelligent Networks, for instance, have provided the global telecommunications network with abstraction levels which enable the easy development, deployment and maintenance of new telecommunications services. Global network management techniques, such as those described in the TMN standards, have also achieved some progresses although their success is far from proved.

Meanwhile, IP, which is as old as the first packet networks but not directly controlled by the public operators, is growing at an astonishing rate due to two main factors. First, its unifying concept which enables the integration of multiple subnetwork technologies under the same umbrella. Second, the success of the HTTP/HTML based applications which unlocked the fifteen years old guessing of what would general network services look like. By using TCP/IP, these services are forcing IP to go into small company premises and end customer homes, in parallel with what has happened to the traditional voice service. This fact opens space for a new set of services with real time requirements over IP, such as voice and video, giving to IP the place which ten years ago was thought to be reserved for ATM. The new IP role is supporting the second revolution of the last twenty years in public networks, giving place to the new concept of public telecommunications network.

The high scale deployment of IP, however, brings some problems. First, vital IP services (e.g. Domain Name System) which traditionally run on the hosts will have to migrate to the networks controlled by the operator. On the other hand, new management services will have to be introduced. Global service abstractions similar to those existing in Intelligent Networks will, for that reason, be adopted. Second, some mechanisms to support the contract of services per session, similar to the signalling stacks used to allocate resources in circuit networks, will also be required (e.g MultiProtocol Label Switching for Service Level Agreements). Third, mechanisms for guaranteeing the quality of the services contracted (e.g. delays, packet loss and Bit Error Rate) will also have to be implemented in IP equipment. At last, the high reliability required in typical telecommunications equipment will also be required for IP equipment.

Whatever the type of technology and communications concept used, there are common issues. First, in all cases a network can by described as a set of nodes which are interconnected by links (cables/ optical fibres or wireless). Second, Network Elements are complex systems since they have to simultaneously support multiple complex communications interfaces and several end-user/management services. Moreover, Network Elements are required to be available for long periods of time, to provide services correctly and with guaranteed quality. These factors, helped by the liberalisation of public telecommunications markets, are creating very high demands on the validation of the Network Element prototypes, which is usually achieved by means of testing practices.

1.1 Network Element

1.1.1 Characterisation

This thesis main objective is to propose a methodology for testing a class of telecommunication systems known as Network Elements (NE) [1], which are characterised by:

1. being part of the global public telecommunication infrastructure;
2. having some communication interfaces described in recommendations published by the ITU-T or ETSI;
3. being mainly digital.

Therefore, significant parts of them are either implemented in traditional languages (e.g. C or assembly) used to program microcontrollers/microprocessors or in some hardware language (e.g. VHDL) used to control programmable hardware components. Examples of NEs are digital telephones, digital access networks, digital local exchanges and routers.

The development and validation of a Network Element is a complex issue for which manufacturers may have defined internal rules. The characteristics 2 and 3 mentioned above imply that (1) SDL [2] specifications are available for some NE communication interfaces, since SDL is mandatory for ETSI and ITU-T protocol standards (2) management services can be described as object models [3]. These facts are pushing manufacturers to look for SDL toolsets [4], [5], to use object oriented techniques [6], [7], and to adopt generic software engineering methods and practices. By using SDL toolsets, the SDL specifications can be captured easily from the standards and the number of interpretation problems is reduced. The object oriented methods, when used in SDL or as complement of it, can ease the implementation of the management and end user services. By using recent software engineering techniques [8], [9], [10], concepts such as components, reutilization and model verification can be employed which are known to reduce the development times and improve the final product quality.

The development of a Network Element may be described in terms of the following tasks [11]: 1) Requirements identification and analysis; 2) Specification; 3) Design; 4) Implementation; 5) Testing.

In the *Requirements identification and analysis* task, which is known to be the first task, the services, interfaces and physical NE configurations are identified. When a SDL toolset is used, typical use cases are drawn as Message Sequence Charts (MSC) [12]. This exercise helps to understand the functions required for the NE and provides the project with a set of valuable behaviour requirements.

In *Specification*, the phase that comes next, the NE interfaces (signalling), management and end-user service are jointly considered. The system architecture begins to emerge as a set of predefined interface and management components which are interconnected by new components so that the NE services can be provided. When a SDL toolset is used, a system or a set of systems must be defined in terms of blocks, main processes and signals exchanged among them.

In *Design*, the NE components are characterised in terms of functions and interfaces. In SDL, for instance, the processes must be known along with their signals and lifetimes.

During *Implementation*, every component is finalised. Signalling components, for instance, can be implemented as SDL state machines that, afterwards, can be translated into some programming language. Firmware, device drivers, management and other services must also be implemented in some adequate language.

1.1.2 Testing

Validation is a working principle which can be applied to every project outcome, independently of the development phase, and consists in verifying if it satisfies a set of expectations. A specification document can be read by some external reviewer which checks for completeness, clearness and consistency. An SDL reachability graph can be explored by some program which verifies if the model is free of dead/ livelocks and executes correctly under certain use cases. A final NE prototype can be tested by final end users who, with the help of tools, will verify if the system carries out its functions correctly.

Testing is a particular type of validation which addresses the final project results - the implemented NE components and the NE itself. Two types of testing activities are usually carried out over a NE: development tests and final prototype tests.

Development testing

Development tests, which are a common practice among programmers, are of two types: *component* and *integration* [13].

Component testing consists of evaluating individual components in their development environment with respect to their correctness. Depending on the abstraction model used, a set of use cases are defined so that the good (positive testing) as well as bad (negative testing) behaviour characteristics can be evaluated. Tests are usually short.

Integration testing consists in verifying if a set of components, forming a larger component, still work as expected. These tests are similar to component testing, i.e., a number of short tests will be applied to the new resulting component. The internal component interfaces are usually monitored so that a faulty component can be identified. These tests become very important since they can be used to validate the work of a person, a team or a company.

Component and integration tests are carried out by the engineers involved in the development. Since various types of integration can exist (e.g. hardware/ hardware, hardware/ drivers, drivers/ signalling, signalling/ services, graphical interfaces/ services) a multitude of methods can also be used.

Final prototype testing

Before reaching the market, NEs are required to succeed in some of the following issues: *1)* conformance evaluation process, which is used to verify if the NE interfaces are compliant with the standards; *2)* operator defined acceptance process, which consists in passing a set of tests defined by operators for a variety of situations, which may include quality of service evaluation;

3) interoperability tests which are carried against other vendor equipments, usually in laboratory environment; *4)* pilot field trials, in which they have to show good results, i.e., to have no visible faults and present good performance.

These aspects imply that a significant amount of effort has to be placed on final prototype testing, so that existing problems can be detected and eliminated *at home*. Two types of tests are usually applied to prototypes at this stage: *1)* conformance tests; *2)* load tests.

Conformance tests are applied to the NE interfaces in order to evaluate their compliance with the standards. Test suites are available from the standardisation bodies, described in Tree and Tabular Combined Notation (TTCN) [14], and can run on commercially available protocol analysers.

Load tests are used to evaluate the system performance under simulated load conditions. These tests are becoming increasingly important since market is demanding this type of information which is used to compare equipment from different vendors.

1.2 Testing research

1.2.1 History and state of the art

Research on testing of protocols in telecommunication systems is carried out by a small but active community which has obtained interesting results during the last years. Advances in the area can be followed through two IFIP sponsored conferences: the IWTCS (International Workshop on Testing of Communicating Systems) and the FORTE/PSTV (Formal Description Techniques/ Protocol Specification Testing and Verification), which aggregate contributions on the field from universities, operators and telecom companies from all over the world. The *Computer Networks and ISDN systems* journal can also be a good source of information for newcomers since, regularly, it provides interesting tutorials on the theme. Articles on the testing aspects of telecommunication systems appear rarely in more general conferences or journals, despite the fact that testing represents typically 30% of a project effort.

The main problem addressed by this research community is protocol conformance testing, i.e., which tests to apply to a protocol implementation so that it can be proved equivalent to its specification. The main concerns are automatic test generation (derivation) from the specification and the quantification of the test value (fault coverage or relation existing between the implementation and the specification). The reasons behind the high success of protocol conformance testing in detriment of other types, perhaps more important for manufacturers, are twofold: *1)* the necessity that standardisation bodies have to guarantee that protocol implementations comply with specifications - it is well known that small deviations from specification can disturb seriously the equipments interoperability; *2)* the fact that protocol specifications are described by state machines for which some testing theory was already available from the hardware field.

This research area is often described as having started with work by Moore, in 1956 [15]. There, as well as in [16], the basic Input/Output finite State Machines (IOSM) were defined. The methods for test derivation from these machines have their origin in the checking experiments from automata theory, whose objective was to determine experimentally whether a given state table could describe the behaviour of a finite state machine implementation. Tests for IOSM are usually expressed in the form of input/output sequences obtained from the state machine. The best known methods are the Transition Tours [17], the Distinguishing Sequences [18], the W method, the Unique Input/Output sequences [19] and their recent improvements (e.g. UIOv and Wp). Most of these methods, under certain assumptions, guarantee full fault coverage with respect to certain types of faults.

During the last years, however, formal description techniques, i.e., languages with a supporting mathematical model, became recognised as valuable to describe protocol behaviours. Two main types have emerged: *1)* languages which are based on process algebras models, such as CCS [20], CSP [21] and LOTOS, whose natural semantic model is the Labelled Transition System (LTS); *2)* languages based on Extended Finite State Machines, i.e., IOSM extended with data, such as SDL [2] and Estelle, whose natural model is the IOSM.

Labelled Transition Systems are in general partially specified and non-deterministic where unspecified interactions produce deadlocks. Taking advantage of this characteristic, some testing methods were obtained and concepts such as canonical tester and implementation relation were introduced [22], [23]. Several attempts to transform LTS into IOSM, and vice-versa, [24] were successful and, nowadays, the models start to be used interchangeably.

Due to its success, SDL has emerged as the formalism towards which the most practical research is oriented and which consists in deriving tests from a SDL system and representing them in TTCN. The main SDL tool vendors provide already these facilities in their products.

1.2.2 New areas of research

The following issues are starting to be addressed by the protocol testing community:

1. automatic derivation of distributed tests from concurrent systems described in SDL [25]. A concurrent version of TTCN is already available

[14] and will be improved in the coming years [26];

2. derivation of testing architectures from the architectures of the systems under test, usually described in SDL [27], [28];

3. service testing which is known to have its own characteristics such as object orientation and distributed platforms [29];

4. passive testing, which consists in gathering information from the system under test just by observing it in its natural working environment [29];

5. embedded testing, which consists in testing a components working inside a system.

Two important issues addressed in this thesis are related to recent advances in computer science. The mathematical models described above (FSM and LTS) are being improved with time and probabilities. It means that the language models have also potential to be used as performance models instead of the well-known queue networks or Petri-Nets. Unlike these models, which are used to model system resources, language models have potential to describe, in simple terms, the user observable system aspects. This aspects are promoting two new research areas:

1. derivation of timed tests from timed models [30];

2. derivation of performance, QoS or load tests [10], i.e., time, probabilistic and time-probabilistic tests, from the time-probabilistic behaviour models.

1.3 Thesis work

1.3.1 Motivation

The first research project in which the author participated was the European project RACE BCPN, aimed at developing an ATM network for a business customer environment. By that time, in 1989, the research team was in charge of characterising the traffic that could be expected in the network. For that, a set of teleservices were identified and traffic sources were modeled by semi-Markov chains. Service mix scenarios were identified and, using discrete event techniques, simulations were carried out and results gathered [31].

After that the author was involved in the ESPRIT DAMS project aimed at developing a FDDI-II based network supporting also a variety of connection and connectionless bearer services. In this project, the author participated in the developement of a load test system [32], [33], [34] that could emulate realistic traffic loads so that the network could be evaluated under these conditions. Knowledge from the first project was reused and the load system could let the user define services by configurating load parameters (e.g. call duration, packet rate and packet length) as well as mixed scenarios. The packets generated by a node were time stamped and measurements such as end-to-end delay, delay jitter, packet loss rate and packet insertion rate could be characterised. Results were presented in a statistical style by means of histograms.

In the middle of the project, there was a request to improve the load equipment so that some DAMS signalling procedures could also be tested. Contrary to load testing, in which a small number of protocol data units containing random data are exchanged with the network, in signalling (behaviour testing), a diversity of protocol data units and parameters are required to be interpreted. Moreover, the execution process and results were found quite different - while load tests run for a user defined amount of time and provide probabilistic estimators as results, behaviour tests stop when a verdict is obtained (pass, fail or inconclusive). Based on these complementary testing approaches, a set of questions, not yet addressed by the testing research community but very requested by equipment manufacturers, emerged: What is the real value of each type of test? What type of faults can we detect with which of them? How to improve these type of tests so that effectiveness (fault coverage and execution time) could be improved?

The next project was the RACE project SCORE - Service Creation in an Object Oriented Environment - aimed at defining a methodology and developing a set of tools for the creation of telecom and IN services. Services had to be first described in OMT. Components had next to be identified from the object description and specified in SDL which was translated to C++ for distributed platforms. Components with potential for being reused in other services would have to be classified and made available at several abstraction levels (SDL, C++) and each component would have to be accompanied by a set of tests. The job of the author was to validate the project development process. For that purpose, with the help of the Ashmolean Museum, Oxford, his team developed what was known as the Multimedia Art Directory [35], [36] which consisted in reusing the X.500 stack from ISO-DE package and to develop a new set of multimedia components. The interesting point was that the methodology evaluated was adopted by the two main SDL tool vendors which were also members of the consortium, Verilog (Geode) and Telelogic (Tau) and, nowadays, reminiscences of the SCORE method can still be found in these vendors manuals. This project refreshed an old problem: how to test component QoS using behaviour descriptions techniques? Besides that, SCORE has shown the author a revolutionary form for developing telecommunication software, in which formal methods and object oriented practices were nicely integrated.

In 1995 INESC established a contract with NEC to develop the firmware for a V5 Access Network and to validate it and the author was responsible for the validation part. A test approach was then proposed which was based on the past experience on testing networks, developing object oriented SDL services and on some new ideas. The method consisted in applying (1) service tests, which were used to validate the services provided by the Access Network under realistic load conditions, and (2) interface conformance tests available from ETSI.

Given the author experience and interests, it was decided to focus this thesis on the generalisation of the method used to validate the V5 Access Network and place it in the context of existing testing theories.

1.3.2 Objectives

This thesis proposes a **testing methodology** which satisfies the following requirements:

R1. Shall be applicable to complex telecommunication Network Elements. Complex NEs are the class of real size systems defined in [1].

R2. Shall satisfy the common testing demands, which are assumed to be the following:

- **R2.1.** NEs shall be tested in short times;
- **R2.2.** NEs shall pass the interface conformance tests;
- **R2.3.** NEs shall pass interoperability tests;
- **R2.4.** NEs shall pass operator service acceptances tests;
- **R2.5.** NEs shall be free of user visible faults;
- **R2.6.** NEs shall have their quality evaluated and quantified.

R3. Shall be based on existing testing practices, so that telecom test engineers can use them easily. Conformance and load tests are assumed to be, for that reason, the starting test types.

R4. Shall consider and adapt recent advances in protocol conformance testing. Although the thesis follows an *engineering view point*, the models used in protocol conformance testing for describing problems and solutions shall be followed, whenever available and adequate.

1.3.3 Contribution

The new contributions of the work are the following:

C1. Testing methodology. In simple terms, it consists in combining the behaviour testing of some NE interface components with the behaviour testing of some service components under real load conditions. Among other attributes the methodology claims (1) to enable the simple modelling of the components under test and (2) to be capable of detecting a large spectrum of faults which lead to the satisfaction of the common testing demands described above.

C2. Service testing method. Service testing under real load conditions is one of the two type of tests serving the testing methodology. Based on the modelling of the NE services, on the rigorous definition of each service interface and on the time-probabilistic-behavioural service properties, the service testing method enables the detection of the most complex faults which include those faults randomly visible. It became, by the curiosity it has generated in V5 test engineers and in the protocol test community, one of the main contributions of this thesis.

C3. Specification of QoS properties. Quality of Service requirements are traditionally described as statistical parameters which are loosely associated to the NE interface events. In this thesis, the statistical QoS parameters are incorporated into behaviour descriptions so that the QoS requirements, such as delays and failure probabilities, can be unambiguously interpreted and evaluated.

C4. Service testing architecture. An architecture for carrying out service tests under real service utilization conditions is also proposed. This architecture is a generalisation of the architecture used in the V5 Access Network project which has been worked and validated up to the smallest detail.

1.4 Structure

This thesis consists of seven chapters.

This chapter places the thesis in the context of telecommunications and testing, explains our motivations for this work, introduces the thesis objectives and the results obtained.

The second chapter describes the state of the art in protocol conformance testing that is the field in which testing theories have been developed and put into practice. The current testing practices, the formal testing methods and the basic testing theories are addressed.

The third chapter introduces some techniques used to model the performance and the quality of service required for Network Elements. The common performance models, the mechanisms used by ITU-T for specifying performance and quality and some new models which can represent simultaneously time, probabilities and behaviours are presented.

The fourth chapter presents the methodology proposed for testing telecommunication Network Elements, which is based on current testing practices but improves them by taking advantage from formal testing methods. Test derivation methods and simple test architectures are proposed.

The fifth chapter introduces the ETSI V5 Access Network. As referred above, the NEC FA102 implementation of this network has been used as the testebed for the methodology proposed.

The sixth chapter gives examples on the application of the testing methodology and architectures proposed to the V5 Access Network case study, so that the abstraction models selected to described the methodology can be mapped into real cases.

The last chapter resumes the main results and contributions of this thesis and provides directions for future work.

Chapter 2

Protocol Testing

2.1 Introduction

This chapter describes the state of the art in protocol conformance testing which is the field in which testing theories have been developed and attempts made to put them into practice.

In the second section some basic theory and notations of discrete mathematics are presented, such as sets, logic, relations, functions, input/output state machine and automaton. Notions of automaton traces and automaton accepted traces are also introduced for their relevance in this thesis.

In the third section, the current practices for protocol conformance testing are presented. It consists of a summary of the protocol conformance recommendations that are described from the point of view of the test development process. Test specification, test execution and test result analysis are identified as the main phases. Each of them is then further presented so that the testing process can be made understandable. Testing architectures and test verdicts are also addressed. Relevant in this section are the concepts of test requirement, test purpose, test case and test suite.

In the fourth section, the main test concepts are described mathematically by using the set and relation concepts. Although this approach is not innovative, the concepts introduced in the third section will become clear. The section basically summarises the new recommendation on formal methods for conformance testing, but is oriented towards the work described in this thesis.

The fifth section introduces the basics of testing theories for input/output state machines, which are used directly in conformance testing or as the basis for more efficient methods. Particularly relevant in this section is the understanding of the basic test derivation method.

The last section addresses test derivation from SDL. Firstly, the SDL reachability graph, i.e. the large automaton that can model the SDL overall behaviour is introduced. Then, test derivation is presented by describing a method which reuses the input/output state machine techniques and another method, closer to the approach based on requirement testing. Relevant for this section are the methods used for simulating and representing a system, their understanding in terms of system traces and the derivation of tests by using both concepts.

2.2 Mathematical concepts and notations

The mathematical concepts and notations that will be used in this chapter are introduced in the following paragraphs. [37], among others, provides the background required to fully understand them.

2.2.1 Sets and logic

Sets

Any set is denoted with an upper case character (A, B, C) or a series of upper case characters ($SPECS$, $IMPS$, $TESTS$). Elements of a set are represented with lower case characters (a, b, c).

The usual operations on sets will be used:

$\{a, b, c, \ldots\}$	The set containing elements a, b, c, ... The order in which elements appear in a set is not important.
$\emptyset$	The empty set, i.e., a set with no elements.
$a \in A$	a is an element of the set A.
$\{a \in A \mid P(a)\}$	The set containing all elements of A for which $P(a)$ is true. Sometimes $\{a \mid P(a)\}$ can be used if A can be deduced from the context.
$\#A$	The number of elements of the set A.
$A \subseteq B$	A is a subset of B, all elements of A are also elements of B.
$A = B$	The set A is equal to set B, A is a subset of B and B is a subset of A.
$A \subset B$	A is a proper subset of B, A is a subset of B and A is not equal to B.
$A \cap B$	The intersection of A and B, the set of all elements that are both in A and B.

$A \cup B$	The union of A and B, the set of all elements that are in A, in B or in both.
$A \times B$	The Cartesian product of A and B, denoting the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$.
$A - B$	The set difference between A and B, that is, the set containing all elements of A that are not in B.
$Powerset(A)$	The powerset of A, that is, the set containing all the subsets of A.

Logic

The following logic notations will be used:

$\neg p$	Not p, the negation of p.
$p \wedge q$	p and q, the conjunction of p and q.
$p \vee q$	p or q, the disjunction of p and q.
$p \Rightarrow q$	p implies q, also read $\neg p \vee q$.
$p \Leftrightarrow q$	p is equivalent to q, $(p \Rightarrow q) \wedge (q \Rightarrow p)$.
$\forall a \in A$	For all elements of set A.
$\exists a \in A$	There exists an element a in set A.

2.2.2 Relations and functions

Relations

Relations will be represented as lower case abbreviations (rel). Let A and B be sets, then a binary relation rel between A and B is a subset of their Cartesian product

$$rel \subseteq A \times B.$$

The element $a \in A$ is related to $b \in B$ if $(a, b) \in rel$. The $a\ rel\ b$ notation can be used in alternative. The domain of the relation rel is defined as the set containing all the elements $a \in A$ which are related to some $b \in B$ by rel. That is $\{a \in A \mid \exists b \in B : (a, b) \in rel\}$

A binary relation rel on A is a subset of $A \times A$ and can have the following properties:

$reflexive$	$(a, a) \in rel$ for all $a \in A$
$antireflexive$	$(a, a) \notin rel$ for all $a \in A$
$symmetric$	$(a, b) \in rel \Longrightarrow (b, a) \in rel$ for all $a, b \in A$
$antisymmetric$	$(a, b) \in rel$ and $(b, a) \in rel \Longrightarrow a = b$
$transitive$	$(a, b) \in rel$ and $(b, c) \in rel \Longrightarrow (a, c) \in rel$

A relation that is anti-reflexive, antisymmetric and transitive is called *partial order*. A partial order relation containing every pair (a, b) such that $a, b \in A$ is called a *total order* (every pair of elements can be compared). A relation which is reflexive and transitive is called *pre-order*.

Functions

Functions are denoted by lower case abbreviation ($func$). A partial function $func$ is a relation between two sets A and B with the property that for each $a \in A$ there exists *at most* one $b \in B$ such that $(a, b) \in func$, that is

$$\forall a \in A : \forall b_1, b_2 \in B : ((a, b_1) \in func \wedge (a, b_2) \in func) \Rightarrow b_1 = b_2$$

The signature of a function, will be provided as $func : A \to B$. A total function $func : A \to B$ is a partial function such that the domain of $func$ is A.

2.2.3 Input/output state machine

A state machine is a mathematical model of a system with discrete inputs and outputs. The system can be in any of a finite number of internal configurations or states. The state of the system summarizes the information concerning past inputs that is needed to determine the behaviour of the system on subsequent inputs.

An Input/ Output State Machine, in particular, is a machine that (1) explicitly distinguishes input events from output events and (2) its transitions can have 2 events associated - one input and one output. This machine is also referred to as *Mealy machine*.

An *Input/Output State Machine*, $IOSM$, can be defined as a 5-tuple,

$$IOSM = (S, I, O, T, s_0)$$

where

- S is the set of all possible states;
- I is the set of input events, including the non-controllable input ε;
- O is the set of output events, including the non-observable output ε;
- $T \subseteq S \times I \times O \times S$ is the set of transitions;
- $s_0 \in S$ is the $IOSM$ initial state.

An $IOSM$ is said to be:

- *finite*, if both the number of elements of the sets S and T are finite;
- *deterministic*, if for every state and every input there is at most one transition defined;
- *complete*, if a transition exists for every combination of states and inputs.

$IOSM$s are often used to describe parts of protocols. The first test theories were derived for this model.

2.2.4 Automaton

Sometimes there is no need to distinguish input from output events. In this case, a system can be represented by an automaton containing single event transitions and defined as

$$A = (S, E, T, s_0, F)$$

where

- S is the set of all *states*;
- E is the set of observable *events*;
- $T \subseteq S \times (E \cup \{\tau\}) \times S$ is the set of single event transitions (the *transition relation*), which describe the possible machine transitions, whether τ represents a generic unobservable event. It implies that the machine can change state with no event being observed;
- $s_0 \in S$ is the machine initial state;
- $F \subseteq S$ is the set of acceptance states, whether an acceptance (or final) state is an automaton state where the system is supposed to have reached some objective.

When represented graphically, an automaton is shown by means of circles and uni-directed arcs. A state is represented by a circle whereas a final state is represented by a double circle and the automaton initial state is depicted as a circle pointed out by a dashed arrow. A transition is represented by an uni-directed arc connecting the transition initial state to the transition final state. The arc is labelled with the transition event.

A *trace* is a finite sequence of observable events. The set of all traces over E is denoted by E^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in E^*$, then $\sigma_1.\sigma_2$ is the concatenation of σ_1 and σ_2. The length of a trace, denoted by $\mid \sigma \mid$, gives the number of visible events in σ.

Let θ and σ be traces. θ is said to be a *prefix* of σ if there is some $\alpha \neq \epsilon$ so that $\theta.\alpha = \sigma$. For instance, the trace $\sigma = a.b.c$ has the following prefixes: ϵ, a and $a.b$.

Let $s, s' \in S,\ \ \mu_i \in E \cup \{\tau\},\ \ a_i \in E$ and $\sigma \in E^*$. Then

$$\begin{array}{lll}
s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in T \\
s \xrightarrow{\mu_1 \ldots \mu_n} s' & =_{def} & \exists s_i, s_j \ldots s_n : s = s_i \xrightarrow{\mu_1} s_j \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \ldots \mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1 \ldots \mu_n} s' \\
s \xRightarrow{\epsilon} s' & =_{def} & s = s' \text{ or } s \xrightarrow{\tau \ldots \tau} s' \\
s \xRightarrow{a} s' & =_{def} & \exists s_i, s_j : s \xRightarrow{\epsilon} s_i \xrightarrow{a} s_j \xRightarrow{\epsilon} s' \\
s \xRightarrow{a_1 \ldots a_n} & =_{def} & \exists s_i, s_j, \ldots, s_n : s = s_i \xRightarrow{a_1} s_j \xRightarrow{a_2} \ldots \xRightarrow{a_n} s_n = s' \\
s \xRightarrow{\sigma} & =_{def} & \exists s' : s \xRightarrow{\sigma} s' \\
s \textbf{ after } \sigma & =_{def} & \{s' \mid s \xRightarrow{\sigma} s'\}
\end{array}$$

Based on the above definitions, the following can be also defined:

- the *traces* of an automaton are all the sequences σ of observable events $(\sigma = a_1.a_2 \ldots a_n,\ a_1, a_2 \ldots a_n \in E)$ which can lead the automaton from its initial state s_0 up to any valid automaton state $s \in S$.

$$Traces(A) \ =_{def} \ \{\sigma \in E^* \mid s_0 \xRightarrow{\sigma}\}$$

- the automaton is said to have *finite behaviour* if every trace has a finite length. In this case the number of traces becomes also finite;

- the automaton is said to be *finite-state* if the number of reachable states is finite, that is, the number of states of the set

$$\{s' \mid \exists \sigma \in E^* : s_0 \xRightarrow{\sigma} s'\}$$

  is finite;

- a trace is said to be *accepted* by the automaton if, at least, one of the states reached by the automaton after executing the trace is also an acceptance state

$$Accept(A) \ =_{def} \ \{\sigma \in E^* \mid (s_0 \textbf{ after } \sigma) \cap F \neq \emptyset\}$$

- A is *deterministic* if, for all $\sigma \in E^*$, s **after** σ has at most one element

This automaton is equivalent to the automaton presented in [37] used to describe regular expressions. The set of events E represents, in this case, an input alphabet and an accepted trace, that is a sequence of events leading the state machine from its initial to an acceptance state, represents a word.

In *Labelled Transition Systems* [38], E represents a set of labels of system interactions. In this case, F can be interpreted as $F = S$ and $Traces(A) = Accept(A)$.

2.3 Conformance testing methodology

This section introduces the Open Systems Interconnection (OSI) view on conformance testing, as described in the standard documents [39], [40], [14], [41], [42], [43] and [44]. These documents define the methodology, provide a framework for specifying tests and describe the procedures to be followed during testing. Tutorials are presented on [45] and [46].

The primary objective of conformance testing is to demonstrate by means of testing practices that an implementation of a protocol conforms to its specification. Conformance, in this context, means that the implementation shall not have behaviours which are not allowed by the protocol specification. Only the functional aspects of the protocol specifications are evaluated by conformance tests. The conformance to standardised performance, robustness and reliability aspects are *explicitly* out of the scope of this standard methodology.

Although testing of the complete implementation observable behaviour may be theoretically possible, in practice a large number of tests, some of which very long, would be required. For that reason, the OSI conformance methodology directs the conformance tests towards the detection of implementation faults.

A simplified picture of the OSI conformance methodology is presented in Fig. 2.1. It consists of three phases: test suite specification, test suite application and test result analysis.

2.3.1 Test suite specification

As shown in Fig. 2.1, the test suite specification activity is aimed at the production of a set of test cases, named *Test Suite*, that will be used in the next phases to evaluate the Implementation Under Test (IUT) compliance to the standard protocol.

Test purpose development

The first step of this phase consists in the definition of a set of test purposes. A test purpose is a textual description of a protocol conformance require-

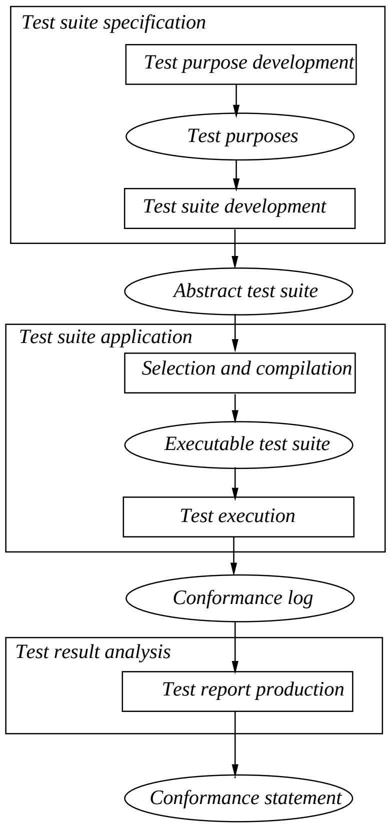


Figure 2.1: OSI conformance testing methodology and framework

ment such as a capability or a behaviour that must be implemented by the implementation under test.

The development of test purposes is done in three steps: *1)* the testable conformance requirements are identified based on the protocol specifications and on the Protocol Implementation Conformance Statements (PICS). Conformance requirements can be described positively (stating what is required to be done) or negatively (stating what is required not to be done); *2)* a set of test groups with clear objectives are identified, so that an adequate coverage of the conformance requirements is obtained; *3)* the test purposes are specified so that they reflect the test group objectives and cover conveniently the set of conformance requirements. Each test purpose can cover one or more conformance requirements.

Assuming that protocols are usually specified as state machines, the test purposes are directed towards the evaluation of the implementation behaviour with respect to the following aspects:

- **valid behaviour.** A test purpose will be defined for each relevant state/input event combination with the purpose of verifying if the transition is correctly implemented;

- **PDUs sent and received by the implementation.** There are test purposes concerned with the verification of individual parameter values and combinations of parameters values. In the first case, for each PDU and for each parameter, a set of test purposes will be defined. If the parameter is an integer, for instance, the test purposes will verify the boundary values and one randomly selected mid-range value. For combination of parameters values, a test purpose for each important combination will be defined;

- **timers.** One test purpose, at least, will be defined for the expiration of each protocol timer;

- **invalid events.** These are expected events but containing syntactically or semantically invalid parameter values. A set of test purposes is recommended for each parameter. For integer parameters, for instance, two test purposes will be defined for the invalid parameter values adjacent to the allowed boundary values defined in the base specification, plus another randomly selected invalid value;

- **inopportune events.** These are correct events but received in states where they were not expected. One test purpose is also generally required for each of these events.

Test suite development

The *test suite* is developed from the protocol specification and the test purposes. The protocol specification describes the behaviour allowed for the implementation. The test purpose describes the part of the implementation that will be tested.

The test suite consists of test cases, where each test case is aimed at verifying one test purpose. The development of a test suite is done in two steps: *1)* a testing architecture is selected; *2)* one test case is specified for each test purpose, taking into account the test architecture selected.

A test case consists of three parts: the preamble, the body and the postamble. The preamble describes the sequence of events that lead the IUT from a stable testing state of the specification into the initial testing state. The test body describes all the sequences of observable events required to evaluate the test purpose and that lead the implementation from the initial testing state to a final testing state. The postamble is the sequence of events that lead the IUT from the final state of the test body back to a stable testing state.

Each test case is representable in a tree like form where, after a service primitive is sent by the tester, a set of alternative primitives can be received from the implementation under test. Every sequence of input/output events of the test case will have a verdict associated, which can be of three types: fail, inconclusive and pass. A *fail* verdict means that the sequence of events observed is forbidden by the protocol specification. An *inconclusive* verdict means that, although not forbidden by the specification, the sequence of events observed is not sufficient to draw a conclusion with respect to the test purpose. A *pass* verdict is associated with the event sequence(s) which are not forbidden by the specification and enable the test purpose verification.

ITU-T and ETSI test suites are described in a standard test language, the *Tree and Tabular Combined Notation (TTCN)* [14]. TTCN is the language used by ETSI and ITU-T to describe protocol conformance tests. Similarly to SDL, it has a graphical and a textual format. The graphical format consists of a set of tables which are used to define *Protocol Data Units* (PDUs), *Abstract Service Primitives* (ASPs), new data types as well as to declare variables and behaviours. Test behaviour tables are organised in a tree like format and a test output event is tipically followed by a set of alternative test input events. The behaviour tables describe also the verdicts of a test. Although very used in protocol testing, the language is often considered very far from modern programming languages. Efforts to overcome this gap are, at the moment, underway in standard bodies.

2.3.2 Test suite application

The test suite application can be decomposed in two parts: *1)* the test case selection and compilation; *2)* the test case execution.

Selection and compilation

In this phase both a test system and information about the system under test are required.

First, the relevant test cases are selected from the test suite based on the manufacturer declaration of the capabilities implemented. Some of the protocol facilities can be optional or depend on optional facilities (conditional).

Then, the selected test suite is parameterised according to some protocol values selected by the manufacturer.

Finally, the parameterised selected test suite, still described in TTCN, is compiled for a protocol analyser which will be used as the test system.

Execution

Before the implementation is physically tested, a static conformance review must be carried out to check if the implementation really meets all the optional and conditional capabilities which were claimed to be implemented by the manufacturer.

After this review, the compiled test cases are applied to the implementation. The sequence of test events (signals) which are observed during the execution of a test case is called the test outcome. It comprises time stamps, data and parameter values for each event. The test outcome, that is stored in a conformance log, leads to a verdict if it matches one of the event sequences defined in the test case.

A test campaign is the execution of a set of test cases.

2.3.3 Test result analysis

Since the methodology addresses mainly the detection of errors, the test result analysis is restricted to a test report which lists the test verdicts and provides a description of the errors.

The results of conformance testing are documented in one or more conformance test reports.

2.3.4 Testing architectures

The conformance test architectures, named by OSI as abstract test methods, are defined by means of two test components - the *Lower Tester* (LT) and the *Upper Tester* (UT). The lower tester controls and observes the lower boundary of the implementation under test, usually via the underlying service provider. The upper tester controls and observes the upper service boundary of the implementation. Coordination of the upper and lower testers is achieved, when necessary, by means of Test Coordination Procedures (TCP). Fig. 2.2 provides an overview of the four main methods, which are briefly described in the following paragraphs.

Local test method

In the local test method there are two *Points of Control and Observation* (PCO): one between the lower tester and the underlying service provider and the other between the upper tester and the implementation under test. The implementation upper service boundary is required to be a standard interface.

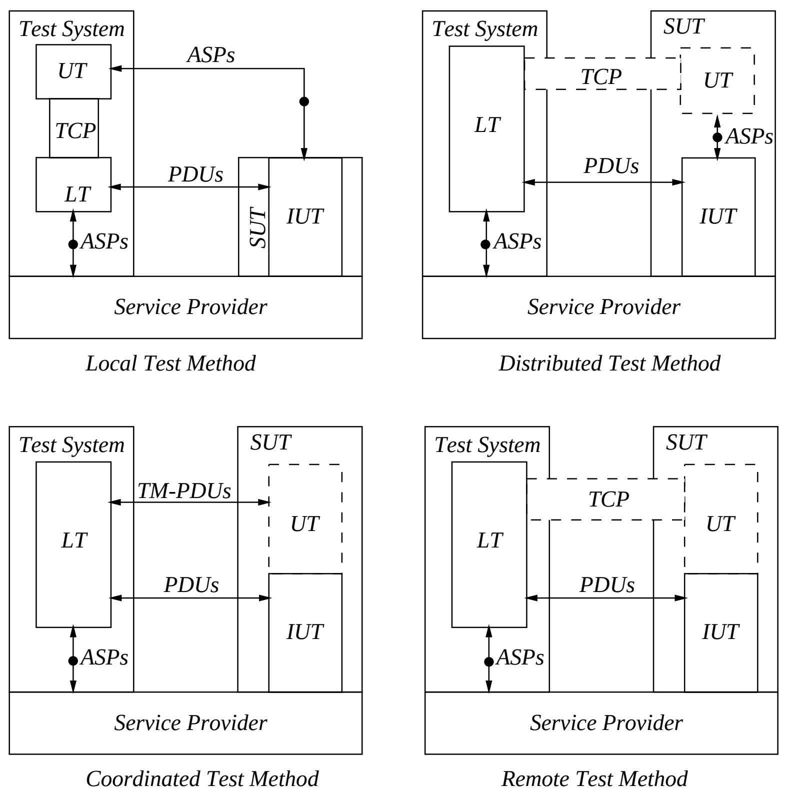


Figure 2.2: Overview of abstract test methods

The lower tester sends and receives Abstract Service Primitives to the service provider and, as usual in the OSI model, communicates with the implementation under test via Protocol Data Units.

The upper tester, which is located within the test system, exchanges service primitives of the service provided with the implementation under test.

Distributed test method

In the distributed test method there is one point of control and observation between the lower tester and the underlying service provider, and another point of control and observation between the upper tester and the implementation.

The upper tester is located in the System Under Test (SUT). Hence, this test method requires the upper service boundary of the implementation to be either a human user interface or a standard programming interface.

Coordinated test method

The coordinated test method requires only one point of control and observation located beneath the lower tester. The test coordination procedures are specified by means of a test management protocol. The upper tester, located above the upper service boundary of the implementation, implements the test management protocol.

Remote test method

The remote test method uses only one point of control and observation located beneath the lower tester and no access to the upper service boundary of the implementation is required. Test coordination procedures are expressed informally in the test suite, but no assumption is made regarding their feasibility or their realization.

Although there is no upper tester, some of its typical functions may be executed by the system under test.

Multi-party methods

A generalisation of the testing methods is also defined in the standards. It allows for multiple lower testers, each representing one of the real systems with which the implementation needs to communicate. In addition, there may be zero or multiple upper testers and a lower tester control function which coordinates the lower testers and the verdict assignment.

2.4 Formal methods in conformance testing

This section presents part of the results of the joint ISO/ITU-T working group of *Formal Methods for Conformance Testing* (FMCT), which are complementary to the concepts presented in the last section. These results are being made available as a new standard [47].

The main objectives of this standard are: *1)* to demonstrate that the formal testing theory developed during the last years can be applied to the conformance evaluation practices; *2)* to evaluate the possibilities of improving the test development process through computer aided generation of test cases from the formal specifications that ITU-T, ISO and ETSI are using to describe protocols.

[48] and [49] provide tutorials on this subject.

2.4.1 The meaning of conformance

A precise description of the concept of conformance implies that implementations must be modeled. In this way, conformance can be defined as relations between models of the specifications and models of implementations or, in alternative, by satisfaction of requirements by the models of the implementations.

Specifications and implementations

Let s be a specification that prescribes the behaviour of a protocol in a formal description technique and $SPECS$ denote the set of all the possible specifications. One example of $SPECS$ is the SDL language and, in this case, s is a particular description in SDL.

An implementation iut consists, in the telecom environments, of a combination of hardware and software having physical connectors or programming interfaces for communication with its environment or end users. Let $IMPS$ denote the set of all possible implementations.

The first main difference between a specification and an implementation is that the former is a mathematical model, whereas the latter is a physical object. In order to describe the concept of conformance, these objects must be related. Implementations, however, cannot be compared with models since they are not models. Therefore it is not possible to define a direct relation between an implementation iut and a specification s.

For that reason, it is *assumed* that an implementation iut must be modeled. Let us assume that m_{iut} is a model of iut and that $MODS$ represents the set of all possible models, that is, the intermediate formalism used to model the implementation. Examples of such formalisms are automata, input/output state machines or sets of traces. This assumption is known as the *Test Assumption*. The model m_{iut} of the implementation is, *a priori*, unknown. Testing will be used to learn about this model and decide about the conformance to the specification model s.

Conformance of an implementation to a specification

The conformance between an implementation iut and a specification s is characterized by a relation between the model of the implementation m_{iut} and the specification s. This relation is called an *implementation relation* and will be denoted as imp with the following signature:

$$imp \subseteq MODS \times SPECS$$

An implementation iut (Fig. 2.3) is said to conform to a specification s with respect to a relation imp if $(m_{iut}, s) \in imp$ or, using the alternative notation, m_{iut} *imp* s. In this case, m_{iut} is a conforming model of s with respect to *imp*.

A specification can have several conforming implementations. For $s \in SPECS$ and an implementation relation *imp*, the set M_s denotes the set of all conforming models in $MODS$, and is given by:

$$M_s = \{m \in MODS \mid m \ imp \ s\}$$

Fig. 2.3 illustrates how a specification $s \in SPECS$ determines a set of conforming implementations I_s whether I_s denotes the set of implementations which can be described by models in M_s. Therefore the set I_s is the set of implementations that implement the specification s correctly.

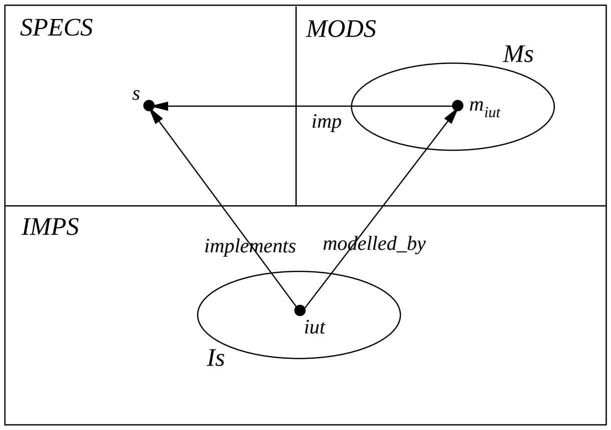


Figure 2.3: Relations between $IMPS$, $MODS$ and $SPECS$

Examples of implementation relations (*imp*) which can be used if both $MODS$ and $SPECS$ are selected to be the SDL language are, for instance, *trace equivalence* or *trace preorder*. In *trace equivalence*, the set of traces of the implementation should be equal to the set of traces of the specification, that is, $Traces(m_{iut}) = Traces(s)$. It means that a conforming implementation must be able to show all the sequences of events previewed by the specification and cannot show any sequences which are not prescribed by the specification. In *trace preorder*, the first set must be included in the second – $Traces(m_{iut}) \subseteq Traces(s)$. This means that only part of the behaviour specified by s has to be implemented and, on the other hand, the implementation is not allowed to show any sequence of events which is not previewed in the specification s. An implementation which does nothing is, in this case, a conforming implementation.

There is another way of defining conformance that is based on the concept of *satisfaction of requirements.* A conformance requirement describes usually a simple behaviour which must, or must not, be observable in implementations. It is a property which has to be satisfied by the model m_{iut} of the implementation so that the implementation can conform to that particular requirement.

Let $REQS$ denote the set of all requirements that can be expressed in a particular requirements language. In the requirements approach, a specification can be alternatively expressed as a set of requirements $R_s \subseteq REQS$. where an element $r \in R_s$ represents a single conformance requirement.

The conformance between an implementation and a specification, in the requirement approach, is characterised by a relation between the model of the implementation m_{iut} and the requirement r. This relation, called *satisfaction relation*, is denoted as sat and has the signature:

$$sat \subseteq MODS \times REQS$$

An implementation IUT conforms to specification R_s if the model m_{iut} can satisfy all the conformance requirements r in R_s. The set M_{R_s} of models of conforming implementations, in the requirements approach, is given by:

$$M_{Rs} = \{m \in MODS \mid \forall r \in R_s : m \ sat \ r\}$$

The two forms of specifications, (s, imp) and (R_s, sat), can be combined as described in Fig 2.4. The resulting set of conforming implementations is described by:

$$M = M_s \cap M_{R_s}$$

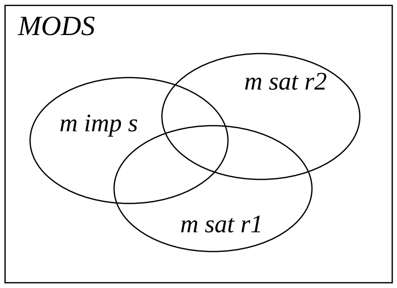


Figure 2.4: Combining the two forms of specifications

2.4.2 Testing execution

Let T represent a test suite consisting of a set of test cases t, such that $t \in T$.

The execution of the test case t consists in executing the tester which implements t in combination with the IUT. During this combined execution an observation $\sigma \in OBS$ is obtained, where OBS is the set of all the observations.

The observation σ (a trace) is then evaluated by a function. The result of such evaluation, in the case of conformance testing, can be *pass*, *fail* or *inconclusive*. The evaluation of the verdict evaluation function depends on the test case t executed:

$$verd_t : OBS \rightarrow \{pass, inconclusive, fail\}$$

An implementation under test IUT passes a test case t if the execution of the IUT with t leads to an observation σ for which the verdict pass is assigned.

$$IUT\ passes\ t \Leftrightarrow\ verd_t(\sigma) = pass$$

The subset of $MODS$ for which $verd_t(\sigma) = pass$ is called the *test purpose* P_t

$$P_t = \{m \in MODS\ \mid\ verd_t(\sigma) = pass\}$$

Thus, the objective of testing an IUT with a test case t is to conclude whether the model m_{iut} of the IUT is a member of its test purpose $P_t \in MODS$, i.e.,

$$IUT\ passes\ t \quad \Longleftrightarrow \quad m_{iut} \in P_t$$

An IUT passes a test suite T if and only if it passes all the test cases in the test suite:

$$IUT\ passes\ T\ \Leftrightarrow\ \forall\ t \in T :\ IUT\ passes\ t$$

If an IUT passes a test suite it follows that the model of IUT is a member of all the test purposes of the test suite:

$$IUT\ passes\ T\ \Leftrightarrow\ m_{IUT} \in P_T,$$

where $P_T = \cap_{t \in T}\ P_t$.

2.4.3 Conformance testing

An IUT conforms to a specification if and only if $m_{iut} \in M_s$ or, equivalently, IUT passes T if $m_{iut} \in P_T$. The ideal test generation method would be one which could derive a test suite T such as $M_s = P_T$.

Depending on the relation between P_T and M_s, a test suite T can be characterised as follows:

- **Exhaustive.** A test suite is exhaustive if the set P_T of all the models passing the test suite T is a subset of the set of conforming models M_s: $P_T \subseteq M_s$. This means that all passing implementations are compliant.
- **Sound.** A test suite T is sound if the set of conforming models M_s is a subset of the set P_T of models that pass implementation IUT: $M_s \subseteq P_T$. This means that all implementations that do not pass are not compliant.
- **Complete.** A test suite T is complete if it is both sound and exhaustive, that is, the set of conforming models equals the set of models that pass the implementation: $P_T = M_s$.

Test Generation

Test generation is the process of deriving a test suite from a formal specification. It may be described as the function *gen* that generates a test suite T from a specification s and an implementation relation *imp*:

$$gen_{imp} : SPECS \rightarrow Powerset(TESTS)$$

The test suites are required to be sound so that no conforming implementation is rejected.

2.5 IOSM test derivation

Telecom software in general and protocols in particular are by nature reactive, where reactive is defined as a computer program whose role is to maintain an ongoing interaction with its environment rather than to execute some task and terminate [50]. This family of programs includes most of the program classes whose correct and reliable construction is considered to be particularly important. Concurrent and real time programs, embedded and process control programs as well as operating systems are other examples of reactive programs.

Some families of models representing this type of programs were developed during the last decades. In this section, the more often used model in protocol theory and, thus, in protocol testing, is presented - *the finite state machine*.

A finite state machine represents a reactive system whose behaviour can be represented by a finite and discrete number of states and in which transitions between states mainly reflect the interactions of the machine with its environment.

A number of interesting operations can be carried out over finite state machines, such as minimization of a machine and combination of two machines [37]. As demonstrated in [51] in the context of protocol specification, verification and testing, finite state machines can also be extended with finite range variables that, in turn, are also modeled as finite state machines which can be combined with other machines.

A particular type of these machines is used in protocol modelisation - the $IOSM$.

In the following sections, the basic methods for test derivation of protocols modeled as $IOSM$ are introduced. Tutorials on this subject can be found in [51] and [52].

2.5.1 Basic conformance testing method

This method is based on the principle that, to be conform, an implementation must have the same *control structure* as its specification, modeled as an $IOSM$. Implementation and specification are said to have the same structure if (1) they model equivalent sets of states and (2) they allow the same state transitions.

The basic method assumes that the specification is modeled as a minimal finite $IOSM$ where a state machine is minimal when it does not have equivalent states. States are said to be equivalent if, for each input, the same output is obtained and the final state is the same or an equivalent state.

In this test derivation method an implementation is assumed to be also an $IOSM$ with the following limiting characteristics:

- **maximum number of states.** The IUT model is a deterministic finite state machine, $IOSM_{IUT}$, with a known maximum number of states, $\sharp S_{IUT}$, where $\sharp S_{IUT} \geq \sharp S_S$. The sets of inputs ($I$) and outputs ($O$) must also be known;

- **finite response time.** The IUT produces a response to an input signal within a known and finite amount of time;

- **strongly connected state machine.** The states and transitions of the IUT form a strongly connected machine in which each state is reachable from all other states;

- **complete state machine.** The IUT is modeled by a complete finite state machine, i.e., the IUT can react in every state to every input.

The conformance test derivation process can be simplified if, in addition to the properties described above, another set of characteristics are part of the IUT:

- **status message.** When a status message is received, the IUT responds with an output message that uniquely identifies its current state. The current IUT state does not change;
- **reset message.** When the IUT receives a reset message, it responds by making a transition to the initial state s_0, independently of its current state. The IUT does not need to produce an output;
- **set message.** When the set message is received by the IUT in the initial state of the machine, the IUT responds by making a transition to the state that is specified in the parameter of the message. The IUT does not need to produce an output.

With these three messages, *set*, *reset* and *status*, the internal structure of the IUT and the specification can be easily compared. A conformance test used to decide about this equivalence can be obtained as follows:

for all possible combination of states $s \in S_S$ and inputs $i \in I_S$ execute the following steps:

1. use the reset message to bring the IUT to the initial state s_0 and then use the set message to transfer the IUT to state s;
2. apply input signal i. Verify that any output received, including the null output ε, matches the output o required by the specification $IOSM_S$;
3. use the status message to interrogate the IUT about its final state. Verify that the final state matches the state described in $IOSM_S$.

A test suite generated according to this algorithm verifies that the IUT is capable of correctly implementing all the transitions of the protocol specification. The inputs tested will include the set, reset and status messages. If the IUT passes these tests, it is capable of reproducing the behavior of $IOSM_S$, but it remains unknown whether the IUT is capable of any other behaviour.

In order to remove the set, reset and status messages from the specification and to reduce the length of the test suite, some techniques were developed. The set message is removed by a *transition tour*, the reset message is removed by *homing sequences* and the status messages are removed by *unique input output sequence.*

2.5.2 Transition tour

A transition tour [53] avoids the use of set messages. A tour through the $IOSM_S$ is planned in which each transition is visited at least once. At best, a transition tour starts with a single reset message and exercises every transition once followed by a status message that verifies the transition final state.

The problem of finding a transition tour is a standard problem of graph theory. The states and the transitions of the state machine form a directed graph. An *Euler tour* in a directed graph is a sequence of transitions that starts and ends at the same state and contains every transition exactly once.

A sufficient condition for the existence of an Euler tour is that the graph is both strongly connected and symmetric, that is, every state must be the destination and the origin of the same number of transitions. An algorithm for deriving a transition tour from a strongly connected and symmetric state machine can be found in [51].

If the state machine is not symmetric, a transition tour can yet be derived. This can be done be augmenting the state machine by duplicating some transitions that, for this reason, will be visited more than once. The problem of finding a transition tour over a non-symmetric graph, in which a transition is exercised at least once and possibly more than once, is known as the Chinese Postman Problem.

2.5.3 Homing sequences

The reset message can be replaced by a sequence of messages called *homing sequence* [51], [18]. A homing sequence brings a system back to its initial state whatever the current state. In general, a homing sequence is defined as an adaptative procedure, in which the responses generated by the machine can be used to determine the next input message. It can be shown that all strongly connected finite state machines have a homing sequence and that it can be derived algorithmically.

2.5.4 Unique input output sequences

The status message can be replaced by a sequence of transitions called *state signature* or *Unique Input/Output sequence (UIO)* [19].

A UIO sequence is the minimum-cost sequence of input events found for a state which generates a sequence of outputs that is unique for that state. Thus, a UIO sequence uniquely determines the state in which the IUT is when the UIO sequence began and has a meaning that is opposite to the homing sequence: it is used to identify the first instead of the last state in the sequence.

Neither all states have a UIO sequence nor all UIO sequences are necessarily different. In some cases, a single sequence of inputs can be found that identifies all the states in a finite state machine. Such sequence is called a *distinguishing sequence*. The simplest method of finding UIO sequences is to enumerate all input output/sequences and to check them for the UIO property. An algorithm computing these sequences is presented in [51].

A UIO sequence leaves the IUT in a state which is different from the state that it verifies. After performing the transition and its corresponding UIO sequence, the transition tour has to be continued from another state than it was planned. Therefore, the transition tour has to be changed. This is done by deriving a pseudo-state machine in which pseudo-transitions are defined that consist of the initial transition itself plus the UIO sequence. A tour over this pseudo-state machine is known as the Rural Chinese Postman Problem.

2.5.5 Other derivation methods

For IOSM that do not have UIO or distinguishing sequences, the *characterizing sequences* method defines partial distinguishing sequences each of which distinguishes a state s_i from a subset of the remaining states, instead of distinguishing s_i from every state in the $IOSM$. The complete set of such input sequences is called the characterising set W of the $IOSM$.

Other methods that improve the basic methods introduced above have been developed recently, such as the T method, the U method, the D method, and the W_p methods.

2.5.6 Implementation relations

The simplest method of comparing an implementation with a specification $IOSM_S$ is to think of the implementation as a probably faulty finite state machine $IOSM_{IUT}$.

One relation often used to compare finite state machines is the *equivalence relation* [37]. In this relation, an IUT $IOSM_{IUT}$ implements $IOSM_S$ if the IUT produces the same outputs as $IOSM_S$ for *all possible* sequences of inputs applied. Equivalence does not mean equality. If $IOSM_S$ is minimal, many equivalent implementations exist, having more states than $IOSM_S$.

Another implementation relation, the *quasi-equivalence* [37], is often used when $IOSM_S$ is partially defined. In this relation, an IUT modeled by $IOSM_{IUT}$ implements $IOSM_S$ if two conditions are satisfied: *1*) for each input sequence of $IOSM_S$ the IUT produces the same output sequence as defined by $IOSM_S$; *2*) for *all other* input sequences any output sequence generated by the IUT is acceptable.

The selection of the appropriate implementation relation is important for testing since it determines the boundary between conforming and faulty implementations.

2.6 SDL test derivation

The $IOSM$ described above has been for several years the main formalism used to describe protocol behaviour. Most of the existing protocols have $IOSM$ descriptions that are usually complemented with textual parts, in natural language, which describe the formats of the messages and their fields. This is true both for the Internet community protocols such as PPP, TCP or FTP, as well as for the public network protocols such as GSM and ISDN.

In the last decade, however, much progress has been achieved in what concerns the description of protocol behaviour. The system interfaces described in standards became more complex and support multiple protocols that, in the traditional approach, would have to be described by one or two $IOSM$ for each protocol. The coordination between the several state machines would be, as well as their data parts, described in natural languages. On the other hand, languages for behavioural specification have been also standardised and robust graphical toolsets supporting them became available.

In order to avoid descriptions that could have multiple possible interpretations and take advantage of the formal description techniques, standards bodies, such as ITU and ETSI, started to complement their system interfaces, the $IOSM$ based descriptions, with SDL specifications [54], [55].

In this section the state of the art in conformance test derivation from SDL specifications is introduced.

2.6.1 Operational model

SDL [2] is a powerful language with complex concepts. In order not to deviate this thesis from its objectives only a simple model of the SDL operation mode is presented. The complete SDL operational model may be found in [56].

A *system*, in SDL, is modeled as a set of state machines that are named *processes*. Processes communicate with other processes by exchanging *signals*. For that purpose, each process has an associated *queue* (mailbox) that is used to store the signals sent by other processes. This communication mechanism of temporarily storing the signals classifies the SDL interprocess communication mode as asynchronous, that is, the output and the input of a signal by the processes involved are not executed at the same instant. A process in SDL can be extended with data, i.e., each process may have its own variables of given types.

A process may be selected for running if it has some transition enabled. Let us assume, for simplicity, that a process has a transition enabled only if it has some signal in its queue. Once selected for running, the transition cannot be neither stopped nor interrupted by any other process. During a transition, a signal is consumed from the process queue, new values can be given to the process data variables and other signals can be sent to the other process queues. The order by which the enabled transitions are executed is, *a priori*, unknown.

There is in SDL one mechanism to tackle time - the *timer*. A timer belongs to a process and each process can have more that one timer. During a transition, a process can set one of its timers to send a signal some time units later. This signal, as all the others, is placed in the queue of the process that owns the timer. Since only one transition can be executed at a time and there are no time bounds defined for its execution, there is no guarantee regarding the instant that the process will receive the timer signal.

2.6.2 Intermediate models

Automatic derivation of conformance tests from SDL specifications use intermediate models that capture only the relevant characteristics of the system.

These models, and the associated theory, come from the field of concurrent programs verification [57] and they are used to detect program problems such as deadlocks, livelocks and prove system invariances [50]. Some of the verification methods are based on the exploitation of the states of a system and try to prove that none of the system states (nor sequences of states) present "bad characteristics".

In testing, namely in black-box or conformance testing, proofs cannot be made based on the states of the system. Instead, *sequences of observable events* must be used.

Reachability graph

An intuitive form of representing the behaviour of a SDL specification is the graphical representation of its states graph.

In Fig. 2.5 a simple SDL system is presented. The system is composed of two processes and is *closed*, that is, no signals from or to the system environment are expected. In general, some assumptions about the environment are required for the automated simulation (graph generation) of SDL systems. The hardest assumptions consider that the environment has maximum behaviour and can output any signal to the system at any time. The easiest ones assume that the environment is cooperative in the sense that the environment will output only expected signals and only when all the internal queues are empty, i.e., the system handles one excitation at a time.

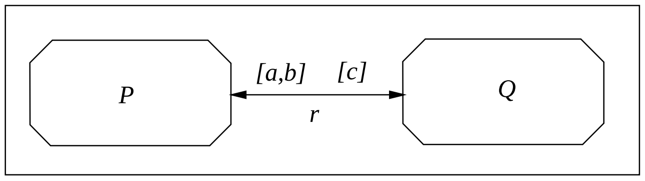


Figure 2.5: Simple SDL System

The behaviour of the two processes composing this system is presented in Fig. 2.6.

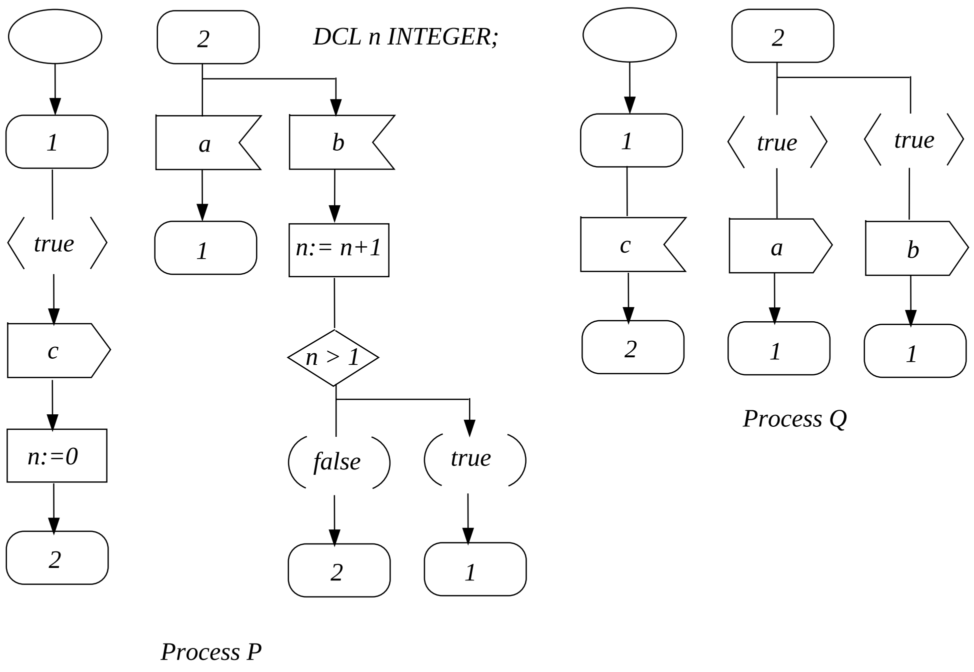


Figure 2.6: Processes P1 and P2 in SDL

From these two processes and assuming the SDL operational model described above, the graph presented in Fig. 2.7, known also as *reachability graph*, can be built. A state in this graph is represented by the internal values of all its variables and internal queues. Transitions between states represent the relevant actions. In Fig. 2.7 the state of the system is represented by four variables for process P and three variables for process Q. $P.state$

and $Q.state$ represent the control states of the two processes. $P.sender$ and $Q.sender$ contain the identification of the process which has output the last signal received, respectively, by process P and Q. $P.queue$ and $Q.queue$ contain the ordered sequence of signals in the queue of processes P and Q. $P.n$ represents the variable n of process P.

Although only reachable states are required to be represented, for real dimension systems millions of states can exist. In order to visit all the states of this graph some methods are required to limit the simulation time spent visiting all the states, as well as the amount of memory used to store the information about the states already visited.

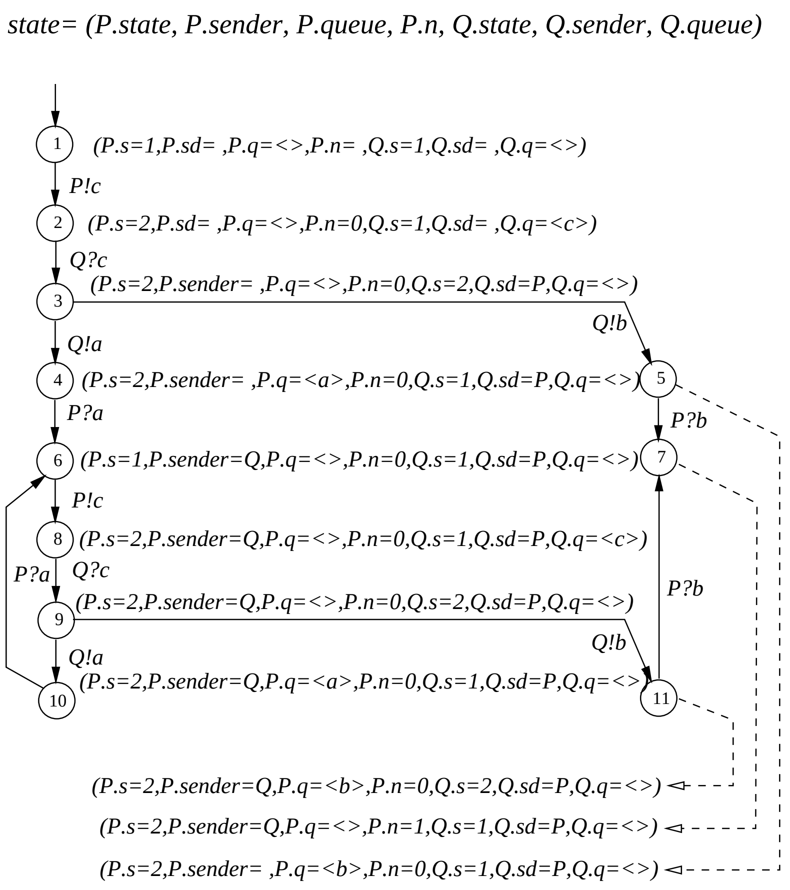

state= (P.state, P.sender, P.queue, P.n, Q.state, Q.sender, Q.queue)
1 (P.s=1,P.sd= ,P.q=<>,P.n= ,Q.s=1,Q.sd= ,Q.q=<>)
P!c
2 (P.s=2,P.sd= ,P.q=<>,P.n=0,Q.s=1,Q.sd= ,Q.q=<c>)
Q?c
3 (P.s=2,P.sender= ,P.q=<>,P.n=0,Q.s=2,Q.sd=P,Q.q=<>)
Q!a
Q!b
4 (P.s=2,P.sender= ,P.q=<a>,P.n=0,Q.s=1,Q.sd=P,Q.q=<>)
5
P?a
P?b
6 (P.s=1,P.sender=Q,P.q=<>,P.n=0,Q.s=1,Q.sd=P,Q.q=<>)
7
P!c
8 (P.s=2,P.sender=Q,P.q=<>,P.n=0,Q.s=1,Q.sd=P,Q.q=<c>)
P?a
Q?c
P?b
9 (P.s=2,P.sender=Q,P.q=<>,P.n=0,Q.s=2,Q.sd=P,Q.q=<>)
Q!b
Q!a
10 (P.s=2,P.sender=Q,P.q=<a>,P.n=0,Q.s=1,Q.sd=P,Q.q=<>)
11
(P.s=2,P.sender=Q,P.q=<b>,P.n=0,Q.s=2,Q.sd=P,Q.q=<>)
(P.s=2,P.sender=Q,P.q=<>,P.n=1,Q.s=1,Q.sd=P,Q.q=<>)
(P.s=2,P.sender= ,P.q=<b>,P.n=0,Q.s=1,Q.sd=P,Q.q=<>)

Figure 2.7: SDL system state graph

Simulating an SDL system

The reachability graph can be obtained by exhaustively simulating a SDL system. The exhaustive exploration of a SDL system is usually performed in one of two modes: *1)* breadth first; *ii)* depth first.

Breadth first. Each node of the graph of the example is numbered. The increasing numbering shows the order in which the states are visited. When explored in this mode all the transitions at a given level are explored before passing to the level below.

Depth first. In this mode, the leftmost part of the graph is explored first. Taking into consideration the number of the states of Fig. 2.7 the following sequence of states is explored in the depth mode: $1, 2, 3, 4, 6, 8, 9, 10, 11, \ldots$.

Labelled Transition System

The SDL reachability graph can be formally described as a *Labelled Transition System* (LTS), which is similar to a large automaton. Initially presented in [58] for SDL and further refined in [59] and [60], this model captures the basic concepts of the state of the system and the elementary events which modify the state in the course of a computation.

In order to model a SDL system as a single automaton, each of its components must be modeled first. As said before, in SDL a system is composed of processes and each process has one queue and some variables.

A variable is modeled by a set of states representing their values, where the variable initial state represents its initial value. The state space of all the variables owned by process i, P_i, is the cross product of the state space of each variable. It is defined as the *data space* of P_i and is denoted by S_i^D.

A signal queue is also modeled as a set of states representing the queue ordered signal containments. The set of states of the queue associated to process i is defined as the *signal space* of P_i and is denoted by S_i^S.

A process, defined in SDL as a state machine, has also a set of states that represent the SDL control states. This set is defined as the *control space* of P_i and is denoted by S_i^C.

Based on that, each SDL process P_i can be modeled as an automaton defined as $P_i = (S_i, E_i, T_i, s_{0i})$, where

- $S_i \subseteq S_i^C \times S_i^D \times S_i^S$ is the set of states of P_i;
- E_i is the set of events of P_i. The events belonging to E_i are the reception of a signal from its queue ($?a$), the output of a signal ($!a$) and the assignement of a value to a variable ($n := 0$);

- $T_i \subseteq S_i \times (E_i \cup \{\tau\}) \times S_i$ is the transition relation of P_i;
- $s_{0i} \in S_i$ is the initial state of P_i.

The model of a SDL system, $A_{SDL} = (S, E, T, s_0)$, can now be constructed by combining the process automata according to the SDL operational semantics.

The state space S of a system A_{SDL} is defined as

$$S \subseteq S_1 \times S_2 \times \ldots \times S_n$$

The system initial state s_0 is the state in which all the processes are in their initial state

$$s_0 = (s_{01}, s_{02}, \ldots, s_{0n})$$

For simplicity, and without losing any modeling power, events are assumed to be disjoint, i.e., two events cannot occur at the same time

$$E = \{ \ (e_1, e_2, \ldots, e_n) \ \mid \ \exists_i : e_i \in E_i \ \wedge \ \forall_{j \neq i} : e_j = \tau \}$$

The transitions which model the valid behaviour of the system can be described as follows. Let s and s' be states of A_{SDL}, i.e., $s = (s_1, s_2, \ldots, s_n),\ s \in S$ and $s' = (s'_1, s'_2, \ldots, s'_n),\ s' \in S$, where $s_i, s'_i \in S_i$. Events e are also required to be events of A_{SDL}, $e = (e_1, e_2, \ldots, e_n),\ e \in E$. The set of transitions T of the system can then be described as

$$T = \{(s, e, s') \mid \exists_i : (s_i, e_i, s'_i) \in T_i \ \wedge \ \forall_{j \neq i} : (\ (s_j, \tau, s'_j) \in T_j \ \vee \ s_j = s'_j \ )\}$$

It means that a system can progress only when one of its processes computes an allowed event – $(s_i, e_i, s'_i) \in T_i$. As a consequence of that, the other processes can change their state – $(s_j, \tau, s'_j) \in T_j$ – (one more signal in the queue, for instance), or remain in the same state – $s_j = s'_j$.

2.6.3 Testing equivalence

A common method for derivating conformance tests from SDL descriptions is to reuse the $IOSM$ test theories [61] [62].

An automaton can be formed from a SDL description by using the techniques presented. However, in order to limit the number of transitions and states some steps are required.

The first one is the transformation of the SDL specification. The SDL constructors adding more states to the automaton, such as the dynamic process creation mechanism or the save construct, are usually eliminated or substituted by simpler ones. Parameters of signals from the environment are limited in range and only selected sets of values are allowed. The environment can be made to have maximum behaviour or be cooperative, as described above. When with maximum behaviour, the environment is simulated by processes having only one control state. In that state, the environment can always send and receive all the signals expected by the system. A cooperative environment, in alternative, is assumed to output a new signal only when all the signal queues of the system are empty. In this case the system reacts to one external signal at time.

The second one is the limitation of the length of the signals queues, during system simulation. Queues are assumed to support a small number of signals (one or two). In this case, the process transitions in which a signal has to be output to a full queue are not considered in the states graph.

The third one is the reduction of the graph. In this phase, transitions describe the SDL relevant actions. For testing, only the externally visible actions, such as the exchange of signals between the system and its environment are relevant. Thus, unobservable events must be renamed as internal and the graph must be reduced in order to decrease the number of states. In this step, the state machine obtained must also be minimised.

A set of tests can now be derived from the reduced and minimised state machine by using, for instance, the UIO method.

2.6.4 Testing requirements

Derivation of conformance tests from SDL specifications can also be achieved based on test requirements or test purposes, as suggested by the ISO methodology. In this approach, a requirement is understood as a property that has to be satisfied by the implementation under test [60], [63], [64]. The specification, naturally, must also satisfy the requirement.

A test purpose, in the SDL world, is very often described in a MSC as a sequence of events which must be observed during system execution. Unlike the automated test methods associated with the $IOSM$ which give a pass or fail verdict, the ITU test methodology defines three type of verdicts. Pass means that the combined execution of the test case and the implementation gives a trace (a log) which meets the requirement and does not violate the specification. Inconclusive means that the trace does not violate the specification but does not provide information required to evaluate the purpose. Fail means that the trace violates the specification.

The correct description of this problem requires that the test purpose/ requirement, the specification and the implementation can be related.

Temporal properties

A temporal property can be expressed as a set of traces and is usually classified in two types [50]: (1) safety and (2) progress. A *safety* property says that a good thing always occurs. A *progress* property says that a good thing will eventually occur. If the good thing is required to occur at least once, then the progress property is refined as a *guarantee* property. Other refinements of progress properties are *response*, *persistence*, *reactivity* and *obligation*. Only *safety* and *guarantee* properties will be considered.

Let Φ be the set of traces defining a temporal property of a system, where a trace is a sequence of observable events. Φ is said to be in E^* ($\Phi \subseteq E^*$) where E is the set representing the system visible events and E^* contains all the possible sequences of events. Let us denote a safety property by Φ_{saf} and a guarantee property by Φ_{gua}.

An implementation can also be represented by the set of traces $Traces(A_{IUT})$, i.e., the traces of the unknown automaton A_{IUT} which models the implementation.

Several relations can be established between the two sets of traces (Φ and $Traces(A_{IUT})$):

- $\Phi = Traces(A_{IUT})$;
- $\Phi \subseteq Traces(A_{IUT})$;
- $Traces(A_{IUT}) \subseteq \Phi$;
- $\Phi \cap Traces(A_{IUT}) = \{ \}$.

The verification of these relations by means of testing is difficult to achieve since it may require the knowledge of $Traces(A_{IUT})$ which may be infinite. Since the execution of one test gives only one implementation trace, $\sigma \in Traces(A_{IUT})$ - the log, it would be preferrable to verify a relation based on the single trace obtained.

Let us, for that reason, give *weak interpretations* of safety and guarantee properties by using only one trace instead of a set of traces to decide about *property satisfaction.*

A *safety property* is said to be *not satisfied* if, during a test, the implementation presents a trace σ which is not defined by the property, i.e., $\sigma \notin \Phi_{saf}$. In this case, $Traces(A_{IUT}) \subseteq \Phi$ is proved to be $FALSE$. Otherwise, the property is assumed to be *satisfied.*

A *guarantee property* is said to be *satisfied* if, during a test, the implementation shows a trace σ which belongs to the property, i.e., $\sigma \in \Phi_{gua}$. In this case $\Phi \cap Traces(A_{IUT}) = \{ \}$ is proved to be $FALSE$. Otherwise, the property is assumed to be *not satisfied.*

Properties as automata

Since a property is represented by a set of event sequences, non-enumerative methods for describing this set are required. Two mathematical models are typically used for that: temporal logic and automata.

Temporal logic is a language used to express formulas over states or events. In testing only sequences of events are of interest. Besides the usual logical operators, such as $\wedge$ and $\vee$, temporal logic has also temporal operators, such as *always* and *eventually.*

Automaton is a well-known method for representing words from a known alphabet. Using this method, the sequence of events defining the property Φ is represented by the traces accepted by the automaton, i.e., the traces which lead the automaton A_Φ from its initial state s_0 until an acceptance state $s \in F$. A property is thus defined by

$$\Phi = Accept(A_\Phi)$$

Specifications and use cases as properties

An SDL specification can be thought as a system property which must be satisfied by the implementation. In this case, a *safety property*, Φ_{saf}, in the sense that the implementation will not be allowed to present a trace which violates the SDL specification.

One way of representing the SDL specification as a safety property is to use the SDL *Labelled Transition System* (automaton) and assume that all the reachable states are also acceptance states,

$$Accept(A_{SDL}) = Traces(A_{SDL})$$

An use case (test purpose) defined based on visible events, so often available in the form of a MSC [12], can also be described as a property which must be satisfied by the implementation. In this case, a *guarantee property*, Φ_{gua}, since at least one implementation trace is required to be the sequence of events represented in the MSC.

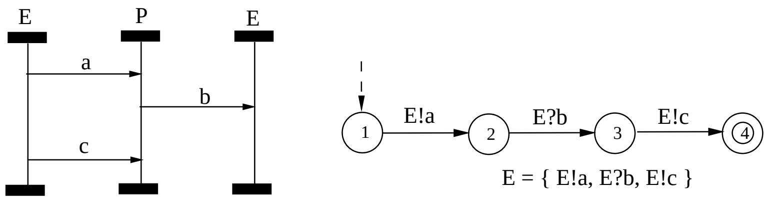


Figure 2.8: A MSC and its equivalent guarantee automaton

Fig. 2.8 shows an example of an automaton representing the guarantee property expressed by a MSC. Let A_{MSC} denote the automaton representing the test purpose, that is, the guarantee property.

Trace evaluation

Let us assume the existence of a test case t. The trace evaluation function, $verd_t()$, i.e., the function which evaluates the trace σ that has been obtained as a result of the joint execution of t and the IUT, can take one of three values *pass*, *inconclusive* or *fail*:

- *pass*, if $\sigma \in Accept(A_{SDL}) \wedge \sigma \in Accept(A_{MSC})$
- *inconclusive*, if $\sigma \in Accept(A_{SDL}) \wedge \sigma \notin Accept(A_{MSC})$
- *fail*, if $\sigma \notin Accept(A_{SDL})$

Test case derivation

The question arising when testing by requirements is: *What test shall be applied to the implementation in order to decide if it satisfies the requirement?* The generation of a test case based on a MSC describing the test purposes and on a SDL specification has been addressed in [63], [64], [60].

First, the MSC is transformed into a guarantee automaton. The SDL, modeled as an LTS, is explored in the breadth first mode. Contrary to the graph exploration method presented above, in which states are never visited twice, this method, which is extended to work on traces, can visit a state more than once. Additional criteria for stopping the LTS exploration are, for that reason, required. The objective of the search is to find sequences of observable events that can simultaneously satisfy two conditions: *c1*) start and stop at the automaton initial state s_0; *c2*) can lead the guarantee automaton into the final success state.

The search of sequences of events satisfying these conditions is carried out by increasing levels of tree exploration. A configurable parameter defines the depth to which the tree must be explored. The traces satisfying *c1* and *c2* are classified as *possible pass observables*.

Possible pass observables are then evaluated for uniqueness. Starting from the shortest possible pass observable, the tree is explored again in order to evaluate if there is no other trace having the same sequence of observable events. If this is the case, the test can be formed. As usual, the test is described in the form of an (observable) events tree, in which the terminal events have the *pass* or *inconclusive* verdicts associated as described above. Sequences of events not previewed by the SDL specifications are marked as *fail*.

2.7 Conclusions

Protocol conformance testing is, surely, the testing field for which more advances were obtained. Although the target of this thesis is the test of systems, much can be gained from existing protocol testing theories.

The first conclusion of the chapter is that a system can be modeled with respect to its behaviour. Independently of its complexity it can be described as an automaton with, perhaps, millions of states and transitions.

The second conclusion is that such models can be obtained by simulation techniques. For that purpose, a system has first to be closed with some environment and, after that, the closed system can be explored taking into account the operational semantics of the specification language, so that all the states and transitions can be found. Some methods exist for that end.

The third and last conclusion is that, although representable in terms of states and transitions, testing concepts such as equivalences, requirements and tests are better understood in terms of traces and set of traces, where a trace is a sequence of observable events from the system initial state to some other state. It is the log that test engineers are used to. Traces, however, can be very large and the complete sets are *a priori* unknown.

Chapter 3

NE Performance Specification

3.1 Introduction

This chapter presents some techniques used to model and describe the performance requirements for Network Elements.

In the next section, some fundaments and notations for probabilities and statistics are reviewed, such as random variables, confidence intervals and Poisson and Markov processes.

In the third section, the models commonly used in the characterisation of the performance of Network Elements are introduced. First, the concept of traffic, the associated models and its measurement are presented. Then, the well known queueing models are revisited. Finally, the traditional loss and delay systems are also reviewed, from the telecommunications point of view. Relevant in this section are the concepts of probabilistic traffic sources and of performance state machine.

In the fourth section, the mechanisms used by ITU to specify performance and quality of service are introduced. Unlike behaviour, whose specifications are operational, performance recommendations are described by means of requirements. Parameters characterising the network element delays, failures and availability are defined along with some boundary values. Two examples are given - ISDN and digital exchanges. These examples are relevant for the case study addressed in this thesis.

In the fifth section, a new concept for specifying performance is introduced. Time and probabilities are associated with existing behaviour models. The concept, coming from computer science, is new and far from being proved. Yet, these specification methods seem very promising since they attempt to fill the gap between the two worlds. SDL improved with time as well as the promising timed automaton are, for that reason, presented. Probabilistic and time probabilistic automata extensions are also addressed.

3.2 Mathematical concepts and notations

In this section some notations and definitions from basic probability theory are reviewed. For a more through treatment refer to [65], [66], [67].

3.2.1 Probabilities and statistics

Random variables

An *experiment* is defined as a process whose outcome is not known with certainty. The set of all possible outcomes of an experiment is called a *sample space* and it is denoted by S. The outcomes themselves are called the *sample points* of the sample space.

A *random variable* is a function which assigns a real number to each *sample point* of S. Random variables will be denoted by capital characteres (X, Y) and their values by lower case ones (a, b). The following notations will be used:

$P(X = x_i)$	The probability that the discrete random variable takes on the value x_i.
$p(x_i)$	$p()$ is the *probability mass function*, defined for discrete random variables. For a particular value x_i, $p(x_i) = P(X = x_i)$ and $\sum_{i=-\infty}^{\infty} p(x_i) = 1$.
$f(x)$	$f()$ is the *probability density function* for continuous random variables. $P(X \in B) = \int_B f(x)dx$ where $\int_{-\infty}^{\infty} f(x)dx = 1$.
$P(X \leq x)$	Probability associated to the event $\{X \leq x\}$.
$F(x)$	The *probability distribution function* of the random variable X. For discrete variables it is defined as $F(x) = P(X \leq x) = \sum_{x_i \leq x} p(x_i)$. For continuous variables, $F(x) = P(X \in [-\infty, x]) = \int_{-\infty}^{x} f(y)dy$.
$E[X]$	*Mean* or expected value of the random variable X. For discrete random variables it is defined by $E[X] = \sum_{i=-\infty}^{\infty} x_i p(x_i)$. For continuous random variables $E[X] = \int_{-\infty}^{\infty} x f(x)dx$. Sometimes $E[X]$ is also represented by μ_X. For the constant c, $E[cX] = cE[X]$.
$Var[X]$	Denotes the *variance* of the random variable X and is defined as $Var[X] = E[(X - \mu_X)^2] = E[X^2] - \mu_X^2$. It is also referred as σ_X^2, where σ_X is the *standard deviation* defined by $\sigma_X = \sqrt{Var[X]}$.

Estimators and confidence interval

Suppose that $O_1, O_2, \ldots, O_n$ are n independent observations of a physical process which can be characterised by the random variable X. For simplicity, the observations O_i can also be considered independent and identically distributed random variables with unknown mean μ and variance σ^2.

The estimation of the mean value of the observations, that is the *sample mean* represented by $\overline{X}(n)$, is given by

$$\overline{X}(n) = \frac{\sum_{i=1}^{n} O_i}{n}$$

$\overline{X}(n)$ is also referred as an estimator of μ_X such that $E[\overline{X}(n)] = \mu_X$. Intuitively this means that, if a very large number of independent experiments is carried out and each one results in a $\overline{X}(n)$, the average of $\overline{X}(n)$ will be μ_X. Similarly, the *sample variance* can be defined by

$$S^2(n) = \frac{\sum_{i=1}^{n} [O_i - \overline{X}(n)]^2}{n-1}$$

where $S^2(n)$ is an estimator of σ_X^2, since $E[S^2(n)] = \sigma_X^2$.

The difficulty of using $\overline{X}(n)$ as an estimator of μ_X without any additional information is that there is no way to assess how close $\overline{X}(n)$ and μ_X are. $\overline{X}(n)$, however, can be considered a random variable with variance

$$Var[\overline{X}(n)] = Var\left[\frac{\sum_{i=1}^{n} O_i}{n}\right] = \frac{\sigma^2}{n} = \frac{S^2(n)}{n}$$

where σ^2 is approximated by $S^2(n)$. If, additionally, $\overline{X}(n)$ is assumed to have a normal distribution with mean μ_X and variance σ^2/n it can be demonstrated that the $100(1-\alpha)$ *percent confidence interval* for μ_X is given by

$$\overline{X}(n) \pm t_{n-1,1-\alpha/2}\sqrt{\frac{S^2(n)}{n}}$$

where $t_{n-1,1-\alpha/2}$, for $0 < \alpha < 1$, is the upper $1 - \alpha/2$ critical point for the *t Student* distribution with $n-1$ degrees of freedom if $n < 50$ or

$$\overline{X}(n) \pm z_{1-\alpha/2}\sqrt{\frac{S^2(n)}{n}}$$

where $z_{1-\alpha/2}$, for $0 < \alpha < 1$, is the upper $1-\alpha/2$ critical point for the standard normal distribution if $n \geq 50$.

3.2.2 Poisson and Markov processes

Poisson process

A Poisson process is a renewal process in which renewal periods (e.g., packet or call interarrival times) are independent and distributed according to the exponential probability density function:

$$f(\tau) = \lambda e^{-\lambda\tau};\ \tau \geq 0$$

where the mean of the interarrival time, τ, is given by

$$E[\tau] = \int_0^\infty \lambda t e^{-\lambda t} dt = \frac{1}{\lambda}$$

It can be demonstrated that the probability $p_t(k)$ of k arrivals occurring within a time interval of length t, is given by

$$P(ARR_t = k) = p_t(k) = \frac{(\lambda t)^k}{k!}\, e^{-\lambda t};\quad k = 0, 1, \ldots$$

and the average number of arrivals during the same time interval t is given by

$$E[ARR_t] = \sum_{k=1}^{\infty} k\ p_t(k) = \ \lambda t$$

Hence, the arrival rate λ can also be interpreted as the average number of arrivals per time unit. Poisson processes have, among others, two properties deserving reference:

- **superposition property.** If $A_1, A_2, \ldots, A_n$ are independent Poisson processes (or traffic sources) with rates $\lambda_1, \lambda_2, \ldots, \lambda_n$, respectively, then their superposition is also a Poisson process, with rate $\lambda_T \ = \ \lambda_1 + \lambda_2 + \ldots + \lambda_n$. This property, together with the limit central theorem, implies that the Poisson distribution approaches the normal distribution when the parameter λt increases. More precisely, if the random variable K has a Poisson distribution with mean λt, then the distribution of $(K - \lambda t)/\sqrt{\lambda t}$ approaches the standard normal distribution $N(0, 1)$ when $\lambda t \to \infty$. In practice, this approach is used for $\lambda t > 20$;

- **decomposition property.** If a Poisson process A, with rate λ, is decomposed into processes $B_1, B_2, \ldots, B_n$, by assigning each arrival in A to B_i with a probability q_i such that $q_1 + q_2 + \ldots + q_n \ = \ 1$, and independently of all previous assignments, then $B_1, B_2, \ldots, B_n$ are Poisson processes with rates $q_1\lambda,\ q_2\lambda,\ \ldots,\ q_n\lambda$, respectively and are independent of each other.

Markov process

Let us consider a stochastic process $X = \{X_t \mid t \geq 0\}$, where X_t are random variables whose space state S is discrete and finite and whose time parameter is assumed to take arbitrary non-negative real values.

The process X is called a Markov process if it has the Markov property, that is, the path followed by X after a given moment t depends only on the state at that moment, X_t, and not on its past history:

$$P(X_{t+s} = j \mid X_u; u \leq t) \; = \; P(X_{t+s} = j \mid X_t); \quad j = 0, 1, \ldots; \; s, t \geq 0$$

A Markov process has state transition probabilities that depend only on s, that is, the time interval between the instants when the process enters state i and state j, respectively

$$P(X_{t+s} = j \mid X_t = i) \; = \; q_{ij}(s); \quad i, j = 0, 1, \ldots; \; s \geq 0$$

The evolution of a Markov process can be described as follows: the process enters a state i ($i = 0, 1, \ldots$) and remains there for a random period of time exponentially distributed with parameter μ_i. At the end of that period, the process moves to another state, say j ($j = 0, 1, \ldots;\ j \neq i$), with some constant probability q_{ij}. Then it remains in state j for a period of time distributed exponentially with parameter μ_j and moves to state k with probability q_{jk}.

A more usual form of representing this probabilistic behaviour is to use the instantaneous transition rates of the Markov process, characterised by the products:

$$a_{ij} = \mu_i q_{ij}; \quad i, j = 0, 1, \ldots; \; i \neq j$$

The instantaneous transition rate a_{ij} can be interpreted as the average number of transitions from state i to state j, per unit of time spent in state i. This view leads to another interpretation of the Markov process operation which is more in line with telecommunications: the process enters a state i; at that moment, a competition between the state i outgoing transitions starts; based on the transition rate a_{ij}, every state i outgoing transition evaluates a time from its exponential distribution; the smallest time wins the competition; after this time, the system changes state using the winner transition.

The last concept to review are the Process Markov balance equations:

$$\sum_{i \in S;\ i \neq j} a_{ij} p_j \; = \; \sum_{i \in S;\ i \neq j} a_{ij} p_i; \; j = 0, 1, \ldots$$

where p_i is the fraction of time or the probability of the process being in state i. Hence, $p_i a_{ij}$ is the average number of transitions that the process makes from state i to state j, per unit of time and in the steady state. It is also known as the flow from state i to state j. The equation states that, for all the states, the flow entering state j equals the flow exiting state j.

3.3 Traditional performance models

From the performance point of view, a Network Element can be modelled by a state machine, composed of states and transitions. Its interpretation, however, differs from the automaton introduced in the last chapter, which was used to represent the system observable behaviours. A state, in the performance context, describes the number of calls or packets in the system. A transition between two states is characterised in terms of (1) the probability of the transition taking place and (2) the waiting time before the transition occurs. A particular class of these machines, the Markov Process, is often used to describe the performance of Network Elements performance, given its exceptional mathematical properties.

Network Elements are usually classified in two types from the engineering point of view [68], [69], [70]: *loss systems* and *delay systems*. The classification of a particular system depends on the network treatment of overload traffic. In a loss system, the overload traffic is rejected without being served. In a delay system, overload traffic is held in queues until the required facilities become available to serve it. Conventional circuit switched Network Elements operate as loss systems, since the excess traffic is usually blocked and not served without a retry on the part of the user. Packet oriented networks have the characteristics of a delay system since packets transversing a network are usually placed in queues waiting for their time to be served. The two worlds, however, touch each other. Traditional loss systems, such as telephony systems, are providing mechanisms for making incoming calls wait. Moreover, with the digitalisation and packetisation of information streams, such as ATM cells or IP packets, for the transport of voice, traditional loss systems may become delay systems during certain phases. Delay systems, on the other hand, become lossy when buffers are full.

3.3.1 Traffic

An arrival from a Network Element user is generally assumed to be purely random and be independent of the arrivals of other users. In general, the traffic offered to a network is fundamentally dependent on both the frequency of the arrivals and the average holding time for each arrival.

Arrivals

Arrivals to a network can be (1) *calls*, where the best example is a phone call, and (2) *packets*, as for instance an IP packet or an ATM cell. Both types of arrivals may need to be combined so that any traffic source may be characterised.

The call concept is strongly associated with telecommunication Network Elements. Firstly, because first telecommunication Network Elements were modelled based on that concept - the phone call - and, secondly, because even for the most recent telecommunication services the call or session concept still continues to make sense from the user and management point of view. In the simplest approach, the call is decomposed into three phases - *establishment*, *data transmission* and *termination*. During establishment, network resources such as timeslots and bandwidth or quality, are negotiated for the duration of the call. During data transmission, the resources allocated are used. In the termination phase, the resources allocated for the call are released. Traditional performance models, however, simplify the call by considering that establishment and termination phases have durations which are irrelevant when compared with the duration of the data transmission phase.

Interestingly, and from the testing point of view, call establishment and termination are the phases which provide more information on the faults of complex systems. During these phases, a number of Network Element components, such as signalling, services and management, are required to inter-work in order to allocate and deallocate the system resources.

The packet model is being increasingly used in performance models. First, they were used to study data networks, such as LANs, X.25 or IP. Then, for modelling ATM traffic sources. More recently, to describe continuous data stream traffic sources over IP, such as voice over IP. Their main application is the description of traffic sources during the data transmission phase of calls. Although resources may have been reserved for these calls during the establishment phase, they may have been allocated based on some probabilistic allocation scheme. Some residual probability of packet data being lost or excessively delayed may need to be characterised.

Measurement

One measure of the traffic generated by a source is the volume of traffic generated over a period of time. Traffic volume, that is a concept comming from circuit oriented networks, is essentially the sum of all holding times during a measurement time interval. A call holding time represents the duration of the call. A packet holding time is the time required by a source to transmit a packet of a given length which, naturally, depends on the packet length and on the transmission bit rate on the medium.

A more useful measure of traffic is the *traffic intensity*, A, which expresses the ratio between the volume of traffic and the period of time used for the measurement. It can be said that

$$A = \frac{\sum_{i=1}^{n} d_i}{T}$$

where d_i is the holding time of call/packet i and T is the period used for measurements. Although traffic intensity is dimensionless (time divided by time) it is usually expressed in *erlang*.

The maximum traffic intensity produced by a source generating one call or packet at time is 1 *erlang*. In the same way, it can be said that the maximum capacity of a server (or channel) is also 1 *erlang*.

If the call/ packet average arrival rate is denoted by λ and the mean holding time is denoted by d_m, then

$$A = \lambda d_m$$

Traffic intensity is a measure of the average traffic generation during a time period and does not reflect the relationship between arrivals and holding times. Many short calls/ packets can generate the same traffic as few long ones.

3.3.2 Waiting queues

The well-known waiting queue is the framework more often used to study the performance of Network Elements. Under certain conditions, algebraic solutions may be found.

In what concerns this thesis, a queuing system is equivalent to a specification language in the sense that it provides a mathematical framework that can be used to describe the performance of networks and verify if they satisfy certain properties, such as maximum mean delays or packet loss probabilities.

A waiting queue is a system where arrivals from a number of sources wait in one queue for a service that will be provided by a number of servers. The arrival and service processes are characterised statistically. The generic notation to describe a queue system is

$$1/2/3/4/5$$

The nature of the arrival process is defined in 1. Usually, interarrival times can be constant (C), exponentially distributed (M) or generic (G). The service time is specified in 2 and can have the same distributions. 3 gives the number of servers of the system. The population of potential customers is given by 4. The last parameter, 5, indicates the capacity of the queue.

For simplicity, this last characteristic is assumed not to contain the jobs in service. The maximum number of jobs present in queue system is, in this case, the maximum number of jobs supported by the queue plus the number of servers. A queue system of type $M/M/1/\infty/\infty$, for instance, indicates that arrival and service times will be exponentially distributed, only one server will be used and both the number of potential clients and the queue capacity will be infinite. This system is often represented as $M/M/1$.

In the same way as a behavioural specification language can be described by a Labelled Transition System, a queue system can also be represented by a performance state machine.

3.3.3 Loss systems

Traditional loss systems are used to model circuit switched networks, such as telephone networks. As said above, usually in this kind of system new arrivals are served only if there are servers available at that time. If this is not the case, the new arrival abandons the system without being served. Loss systems usually process calls. The call model introduced above is, for that reason, appropriate.

From the performance point of view, the events which are considered relevant are (1) $e_{s_i}(t_{s_i})$, which models the set up of call i at time t_{s_i} and (2) $e_{r_i}(t_{r_i})$ which models the release of call i at time t_{r_i}.

Intercall arrivals are usually modeled as Poisson processes. It means that the random variable τ_{arr} describing the time interval between the arrival of two consecutive calls $j-1$ and j, $(\tau_{arr_j} = t_{s_j} - t_{s_{j-1}})$, is considered exponentially distributed according to $f_{arr}(\tau_{arr}) = \lambda e^{-\lambda \tau_{arr}}$, where λ describes the call arrival rate (call/hour). The adoption of Poisson arrivals, although claimed to model traffic sources realistically, is convenient mainly by its superposition and decomposition properties which makes the combination of two Poisson traffic sources yet a Poisson source.

Call holding times of individual sources, $\tau_{hold_j} = t_{r_j} - t_{s_j}$ are assumed to be exponentially distributed or constant. In the first case, $f_{hold}(\tau_{hold}) = \mu e^{-\mu \tau_{hold}}$, where $1/\mu$ models the mean call duration. In the second case, $p_{hold}(T_{hold}) = 1$, that is, all the calls have the same constant duration T_{hold}.

The aim of the study of a loss system is to characterise its blocking probability, that is, the probability of a new call arrival being rejected by the network. To some extent, an waiting queue system of length 0 (no waiting places), $M/M/N/S/0$, can be used to describe a loss system. In this case, the system has S call sources generating call set up events e_s. These events are characterised by interarrival times τ_{arr} which are distributed exponentially with mean λ. The N servers are used to model the call duration, τ_{hold}, also assumed to be exponentially distributed with mean (μ).

Call sources - the telephones - are assumed to generate one call at time. As an example and assuming that the number of sources (S) equals the number of servers (N), then the (instantaneous) generation rate at the moment where all the servers are busy serving a call will be 0, instead of $S\lambda$ described by the model. In order to consider this issue, the model is altered so that each source refrains itself from generating new calls when it is already engaged in a call. In order to maintain the generation rate λ as specified, the source uses an higher generation rate λ_{idle} during idle periods. The new generation rate λ_{idle} is used to describe statistically the time between the end of a call and the beginning of the next call, that is, $\tau_{idle} = t_{s_{j+1}} - t_{r_j}$, where $f_{idle}(\tau_{idle}) = \lambda_{idle} e^{-\lambda_{idle}\tau_{idle}}$ and

$$\lambda_{idle} = \frac{\lambda}{1-A}$$

In other terms,

$$\frac{1}{\lambda_{idle}} = E[\tau_{idle}] = E[\tau_{arr}] - E[\tau_{hold}]$$

This model, which is very common in telecommunications, will also be used in the case study of this thesis. It can be represented by the performance state machine of Fig. 3.1, which depicts a Markov process. States describe the number of calls in the system. Transitions are labelled with a rate which implicitly describes the probability of a transition being undertaken and the time the system will remain in a state before it passes to a neighbour state. Both the generation and termination rates depend on the number of active sources.

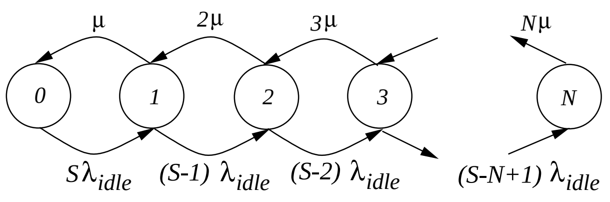


Figure 3.1: Markov process representing a loss system

3.3.4 Delay systems

Delay systems are known to have some capacity to delay service requests until they can be served. Most of the data networks work based on this concept. Depending on the part of the network being modelled, an arrival may represent the request for a packet transmission and the service can, for instance, include the time to process the request and transmit the packet.

Simple delay systems can be modelled by one waiting queue. In this case, three events are relevant: (1) $e_{a_i}(t_{a_i})$, which models the arrival of the service request i to the queue tail at time t_{a_i}; (2) $e_{s_i}(t_{s_i})$, which models the start of service of the arrival i at time t_{s_i}, and represents the transfer of an arrival from the first position of the queue to a server; (3) $e_{e_i}(t_{e_i})$, which represents the end of service for arrival i, at time t_{e_i}. In this case, arrival i is said to leave the system.

Let us assume a generic queue $M/M/N/S/L$ where each source generates Poisson arrivals with rate λ, and each server provides service whose duration is exponential and characterised by μ. There are S sources, N servers and the queue has length L.

The state machine representing this simple performance model is shown in Fig. 3.2. Again, the states represent the number of arrivals in the system (queue plus servers) and the transitions represent the flow rates of the system moves between states.

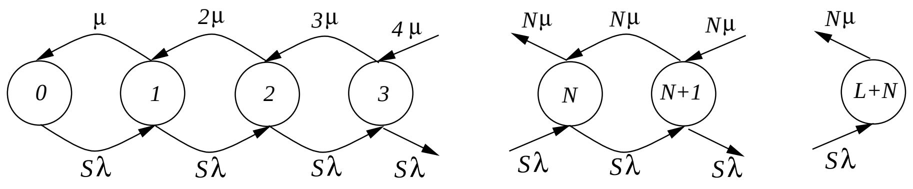


Figure 3.2: Markov process representing a single queue delay system

More complex delay systems can be modelled by queues networks, in which a service request (arrival) after being served is placed (with some probability) at the end of another queue. Complex systems usually loose the mathematical characteristics of simple Poisson processes and their study can only be achieved by discrete event simulation techniques. Delay systems are aimed at characterising the delay and losses of each type of arrivals.

3.4 Performance requirements

3.4.1 ITU-T approach

General characterisation

The ITU-T specification of quality is restricted to the boundary of Network Elements and describes the quality, or performance, required for the communications functions that they provide. Quality aspects are addressed from (1) the users point of view, referred to as *Quality of Service* (QoS) and (2) the network providers point of view, named as *Network Performance* (NP).

The Quality of Service provided by a network and felt by its users is detailed in [71] as being influenced by the following aspects: 1) service support performance; 2) service operability performance; 3) serveability; 4) service security performance. The serveability aspect, which is the main concern of this thesis, is defined as *the ability of a service to be obtained within specified tolerances when requested by the user and continue to be provided without excessive impairment for the request duration.* QoS is expressed by means of parameters that, in turn, are defined based on events observable at the service access points [72].

Network performance, as felt by the network provider, is defined in [71] as *the ability of a network or network portion to provide the functions related to communications between users.* The performance of a network and its components contributes, among other things, to the serveability aspects of the services and is characterised in [72] by a set of measurable and calculable parameters which provide information for system development, network planning, operation and maintenance. The definition of network performance parameters is also defined based on events observable at connection element boundaries such as protocol signals.

The service access points and the connection element boundaries are the points used for measurement. A measurement point is defined in [73] as a point that is located at an interface that separates either customer equipment/customer network or a switching/signalling node from an attached transmission system at which ITU recommended protocols can be observed.

At these measurement points reference events are introduced and used to define the parameters. A reference event is described in [73] as the transfer of a discrete unit of control or user information encoded in accordance with recommended protocols across a measurement point.

Reference events are classified as *entry events* if they entry the Network Element under characterisation or *exit events* on the other case. The time of occurrence of an exit event is specified as the instant at which the first bit of the unit is observed at the measurement point. In case of event retransmissions, only the first occurrence counts. The occurrence time of an entry event is the time when the last bit of the unit enters the network. In case of retransmissions, only the last transmission event must be taken into account.

A QoS/NP parameter is classified, according to its type, as *primary* or *availability*.

Function	Speed	Accuracy	Dependability
access	access delay	incorrect access prob	access denial prob
information transfer	transfer delay transfer rate	error prob extra delivery prob misdelivery prob	loss prob
disengagement	disengagement delay	incorr diseng prob	

Table 3.1: Primary parameters for quality of service

A primary parameter describes a quality factor during the *normal operation* of the system. A primary parameter can also be classified according to the service usage phase and performance criterion. Service usage phases are access, user information transfer and disengagement. Performance criteria are speed, accuracy and dependability. Speed describes the time interval required to perform the function or the rate at which the function is performed. Accuracy describes the degree of correctness with which the function is performed. Dependability describes the degree of certainty (or surety) with which the function is performed regardless the speed or the accuracy. Primary parameters are presented in Table 3.1.

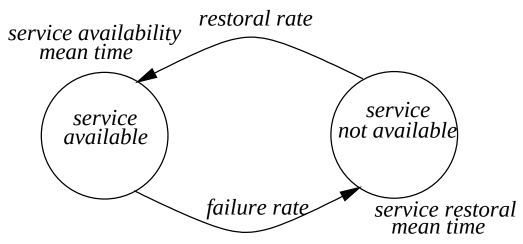


Figure 3.3: Availability parameters

An availability parameter describes the frequency and the duration of service interruptions. Availability parameters are presented in Fig. 3.3.

Delays

Quality and performance primary parameters usually express statistically maximum time intervals between reference events or the maximum number of *bad* situations.

For delays, two maximum values are usually specified: the maximum mean value ($maxmean$) and the maximum value for the 95 percentile ($max95$) [74], [75], [76], [77].

Being D a random variable referring to the time interval between two reference events, the first requirement on delay says that the average delay between these reference events shall be less than $maxmean$

$$E[D] \leq maxmean$$

The second requirement on delay says that for 95% of the cases, at least, the delay D shall be less than $max95$:

$$Prob(D \leq max95) \geq 0.95$$

The $maxmean$ and the $max95$ values tend to be defined as a function of the traffic offered to the network. Traffic intensity (A) or packet rate (λ) are used to characterise this traffic.

Failures

For bad situations, such as *connection set-up failure*, primary parameters are specified as the limiting occurrence probability of some unwanted behaviour [75], [78], [79], [77]:

$$Prob(bad\_situation) \;\; \leq \;\; maxprob$$

where $bad\_situation$ is a free textual description of the failure.

Availability

Based on the primary parameters defined for failures, certain outage criteria can also be defined. For instance, for an ISDN 64 kbit/s circuit switched connection type, if the *connection set-up error* plus the *connection set-up failure* are occurring for more the 90% of the call attempts, the network element is considered to be *out-of-service*.

A Network Element outage may be modelled by a Markov process (as shown in Fig. 3.3) where the following parameters are considered: *1)* the average outage duration; *2)* the failure rate λ; *3)* the restoral rate μ.

3.4.2 N-ISDN

Narrowband ISDN has ten QoS/NP performance parameters defined in international recommendations. Four of them (connection set-up delay, disconnect delay, call set-up error probability and call set-up failure probability) are presented below.

- **connection set-up delay.** This parameter is based on the observations at two measurement points, MP_i and MP_j (see Fig. 3.4) which corresponds to two ISDN S/T terminal interfaces.

  It is defined in [74] as the length of time that starts when a $SETUP$ message message creates a performance-significant reference event at the measurement point MP_i (transference of the last bit of the $I(SET)$ message) and ends when the corresponding $CONNECT$ message returns and creates its performance significant event at the same measurement point (reception of the first bit of the $I(CON)$ message. Called user response times, as d_1 observed in MP_j, are excluded, so connection set-up delay is defined as $d_2 - d_1$.

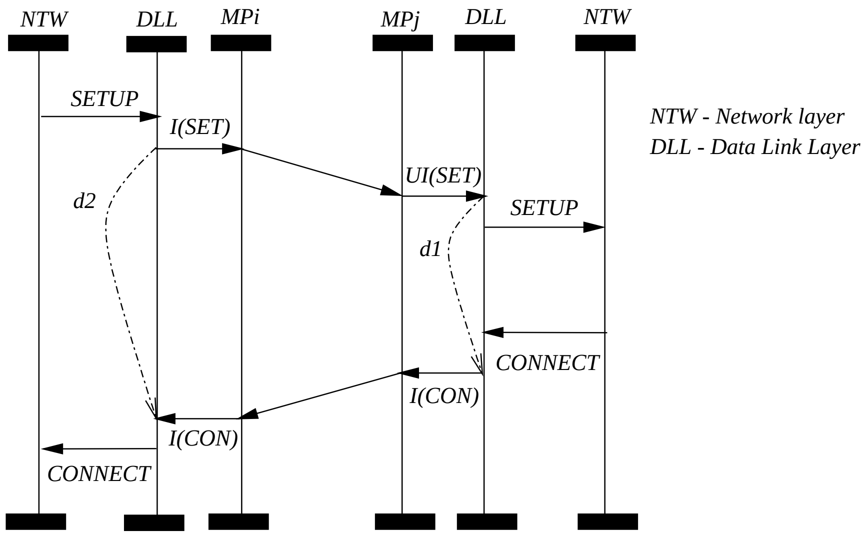


Figure 3.4: Connection set-up delay

For a national network Tab. 3.2 presents the maximum values recommended:

- **disconnect delay.** It is defined [74] as the length of time that starts when a $DISC$ message creates a performance significant reference event

	Connection setup delay
$maxmean$	2900 ms
$max95$	3600 ms

Table 3.2: Connection set-up delay for a national network

at the measurement point MP_i and ends when that $DISC$ message creates a performance significant event at measurement point MP_j, farther from the clearing part, as presented in Fig. 3.5.

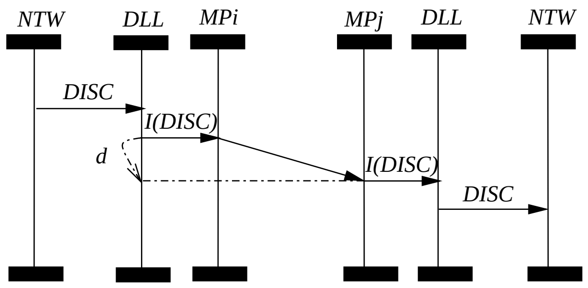


Figure 3.5: Disconnect delay

For a national network, Tab. 3.3 presents the maximum values recommended.

	Disconnect delay
$maxmean$	1250 ms
$max95$	1750 ms

Table 3.3: Disconnect delay for a national network

- **call set-up error probability.** It is defined in [76] as the ratio of total call attempts that result in call set-up error to the total number of call attempts. With reference to Fig. 3.6, a call set-up error is defined to occur in any call attempt in which event b occurs, but event c does not occur within a 200 s period. The call set-up error probability is essentially a wrong number.
- **call set-up failure probability.** It is defined in [76] as the ratio of

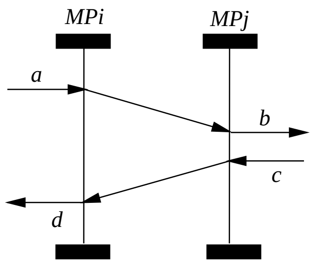


Figure 3.6: Generic reference events for a successful cal set-up

total call attempts that result in call set-up failure to the total number of call attempts. With reference to Fig. 3.6, call set-up failure is defined to occur in any call attempt in which either one of the following is observed within a 200 s timeout time interval: *i*) both events b and d do not occur; *ii*) events b and c occur, but event d does not.

A connection portion is said [78] to be out of service if the measured

$$call\ setup\ error\ probability\ +\ call\ setup\ faillure\ probability\ >\ 0.9$$

3.4.3 Digital exchange

Performance parameters in digital exchanges are defined as functions of reference loads, that is, the traffic mix expected for the Network Element.

Reference loads

Three types of accesses are defined for a digital exchange [77]: *1)* basic analogue line; *2)* ISDN, Basic Rate Interface; *3)* interexchange 64 kbit/s circuits.

The traffic offered by each access to the local exchange, also named *originating traffic*, is modelled by means of two parameters: traffic intensity - A ($erlang$) - and the average *Busy Hour Call Attempts* - $BHCA$ ($call/hour$). They are related by

$$BHCA = \frac{A}{CHT}$$

where CHT is the average *Call Holding Time*, in $hours$.

Two reference loads are typically considered in the definition of the performance parameters: the *Upper level* and the *Increased level.* The Upper level represents the normal maximum busy hour traffic for which the switching resources and interexchange circuits are generally provisioned, typically the mean for the 30 highest busy hours of the year, excluding exceptional days such as Christmas. Tab. 3.4 shows the typical values for the traffic originated by the exchange accesses in the *Upper level* reference load.

Access type	A $(erlang)$	BHCA $(call/h)$	λ, D chan (pac/s)
Analogue line	0.17	6.8	-
ISDN-BRI (2B+D)	2×0.1	2×4	0.05
Interexchange circuit	0.7	$0.7/CHT$	-

Table 3.4: Upper level originating traffic parameters

The *Increased level* reference load represents the reasonably frequent occurring overload conditions, for which the network is expected to provide an acceptable level of degraded performance, typically the mean of the 5 highest busy hours of the year. The values for the traffic originated by the exchange accesses in the *Increased level* reference load are presented in Tab. 3.5.

Access type	A $(Erlang)$	BHCA $(call/h)$	λ, D chan (pac/s)
Analogue line	1.25×0.17	1.35×6.8	-
ISDN-BRI (2B+D)	$1.25 \times 2 \times 0.1$	$1.35 \times 2 \times 4$	> 0.05
Interexchange circuit	0.8	$1.2 \times 0.7/CHT$	-

Table 3.5: Increased level originating traffic parameters

Another relevant aspect when defining the traffic offered to a network element is the consideration of its mixing. For digital exchanges providing both analogue and ISDN services, some mixes are presented in Tab. 3.6.

Access type	Originating (%)	Terminating (%)
Basic analogue line	28	26
Analogue line with suppl. services	32	30
ISDN line, circuit switched	5	5
ISDN line, packet switched	2	2
Interexchange circuit	33	37
Total	100	100

Table 3.6: Digital exchange typical traffic mix

Performance parameters

About 40 performance parameters are defined for digital exchanges [77]. Five of them were selected (local exchange call request delay, incomming call indication sending delay, exchange call release delay, release failure and tone failure) and are presented below.

- **local exchange call request delay.** For *analogue subscriber lines*, call request delay is defined as the interval from when the off-hook condition is recognised at the subscriber line interface of the exchange until the exchange begins to apply the dial tone to the line. For *digital subscriber lines* call request delay is defined as the interval from the instant at which the $SETUP$ message has been received from the subscriber signalling system until the $SETUP\ ACKNOWLEDGE$ message is passed back to the subscriber signalling system. The values recommended are in Table 3.7.

Call request delay		
Statistic	Upper level	Increased level
$maxmean$	400 ms	800 ms
$max95$	600 ms	800 ms

Table 3.7: Call request delay for subscriber lines

- **incoming call indication sending delay.** For calls terminating on *analogue subscriber lines*, this delay is defined as the interval from the instant when the last digit of the called number is available for processing in the exchange until the instant that the ringing signal is applied by the exchange to the called subscriber line. For calls terminating in *digital subscriber lines*, the delay is the interval from the instant the necessary signalling information is received form the signalling system to the instant when the $SETUP$ message is passed to the signalling system of the called digital subscriber line. The values recommended are in Tab. 3.8;

- **exchange call release delay.** This delay is the interval from the instant at which the last information required for releasing a connection is available for processing in the exchange to the instant when the switching network through-connection in the exchange is no longer available for carrying traffic. The values recommended are in Tab. 3.9.

Incoming call indication sending delay				
	Analogue line		Digital line	
Statistic	Upper	Increased	Upper	Increased
$maxmean$	650 ms	1000 ms	400 ms	600 ms
$max95$	900 ms	1600 ms	600 ms	1000 ms

Table 3.8: Incoming call indication sending delay

Call release delay		
Statistic	Upper level	Increased level
$maxmean$	250 ms	400 ms
$max95$	300 ms	700 ms

Table 3.9: Exchange call release delay

- **release failure.** The probability that an exchange malfunction prevents the required release of a connection shall be

$$P(release\_failure) \leq 2 \times 10^{-5}$$

- **no tone.** The probability of a call attempt encountering no tone following the receipt of a valid address by the exchange shall be

$$P(no\_tone) \leq 10^{-4}$$

3.5 Behaviour performance models

As shown above, the performance of Network Elements can be described and studied in terms of performance models such as waiting queues or queues networks. For some classes of these systems algebraic solutions can be found. However, for more complex systems, discrete event simulation techniques are needed. The main objectives of performance models are the study of system delays and losses.

The mechanisms used by international bodies to describe quality were also presented. Contrary to behaviour specifications, which describe in detail the operations of some system components, the performance specifications are provided in a *requirement style.* Random variables for delays, failures and availability are defined and limiting values are ascribed to them. These parameters, however, use information from behaviour descriptions. The two worlds seem, for that reason, to be related.

The results which will be presented in the next section come from computer science and they are recent. Although much work can be found on the subject, only the contributions considered relevant for this thesis will be presented.

Examples come mainly from process algebra [80], [81], event structures [82] and automata. For a matter of consistency, only the basic automata will be addressed.

3.5.1 Timed behaviours

The behaviour automaton does not have mechanisms to express time, time constraints or delays. For instance, the time that an automaton spends in a state or the time consumed in executing a transition cannot be defined.

Protocol specification languages, such as SDL, partially overcome this problem, vital for the specification of timeouts, by defining special processes named *timers*. A timer, which serves only one process, is a concurrent process that can be set to some time in the future and warns its owner about the occurrence of the timeout. The process receiving the timer signal would, in this way, be able to evaluate if the timer signal was received *before* or *after* some other signal. This mechanism, however, cannot be used to describe sequences of precise time events.

Recently some proposals for overcoming this problem have been presented. Two of them will be described next.

SDL timing mechanisms

SDL processes specify amounts of time by means of timers. There are two problems associated with this mechanism:

- **execution times.** Neither a task execution time in a transition nor the transition execution time can be explicitly described;

- **signal queues.** The time at which the process will receive the timer signal cannot be explicitly specified, since it depends on the amount and type of signals existing in the process queue.

In order to overcome these limitations and let SDL be used for the definition of real time systems, some solutions have been proposed. The Queuing SDL [83], proposes three solutions: *general zero time executions*, *time request actions* and *timed states*. The first assumes that, unless explicitly stated by the other two solutions, SDL actions are instantaneous, that is, they take no time to execute. Time request actions are special timed actions that are invoked during an SDL transition and will block the process in that action for the amount of time requested. Since in SDL transitions are atomic, no other transition from other process can be executed during this blocking time. Timed states are states that awake the process δ time units after it has entered the state. This mechanism enables spontaneous transitions to occur at a pre-determined time.

Similar constructs are proposed in [84]. Actions can be specified with respect to time, that is, the process can block after the execution of an action for a specified time interval. On the other hand, the timer is required to have its value available for the owner process. In this way, a SDL continuous signal using a timer value may be used to enable a transition. This second solution solves the timer queue problem, because the synchronous signal will no more depend on the process signal queue.

Timed automata

Alur-Dill [85], [86] developed a simple model for time which addresses behaviours as trace semantics, with the purpose of studying the correctness of real time systems.

In (untimed) trace semantics, the behaviour of an automaton A can be given by the set of its traces, where a trace is a sequence of observable events from the initial state to some other state. If time is introduced, the automaton traces become also timed in the sense that a time value must now be associated with each observable trace event.

A *timed trace* σ_t is then defined as a sequence of pairs (e, t), where e represents an event and t its ocurrence time (e.g. $\sigma_t = (a, 1).(b, 2).(a, 4).(b, 5)\ldots)$. $timTraces(A)$ is the set of timed traces defined by the automaton A.

Alur-Dill, in [85], augment the definition of the automaton so that they can produced timed traces. In traditional automata, the transition selection depends on the event. In the case of a timed automaton the objective is that this choice depends also upon the time of the event relative to the times of previous events. For that, a set of *clocks* are associated with the automaton. A clock can be set to zero simultaneously with any transition. At any instant, the reading of the clock equals the time elapsed since the last time it was reset. With each transition there is also associated a clock constraint and it is required that the transition may take place only if the current values of the clocks satisfy this constraint.

Let us consider the automaton of Fig. 3.7, which uses two clocks, x and y. The clock x gets set to 0 each time the automaton moves from s_0 to s_1 on event a. The check $(x < 1)$? associated with the c-transition from s_2 to s_3 ensures that c happens within time 1 of the preceding a. A similar mechanism of resetting another independent clock y on event b and checking its value on event d ensures that the delay between the following d is always

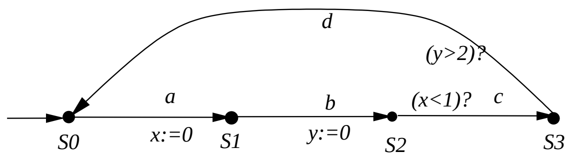


Figure 3.7: Timed automaton with 2 clocks

greater than 2. The trace $\sigma_{t_1} = (a, 2).(b, 2.7).(c, 2.8)$, for instance, is a timed trace of the automata represented in Fig. 3.7, where events a, b and c have been observed in sequence at, respectively, the times 2, 2.7 and 2.8. With this sequence, the automaton has been able to move from state s_0 to state s_1.

A timed automaton is defined by the tuple $A = (S, E, TT, s_0, C)$ where

- S, E and s_0 are defined as usual;
- C is a finite set of clocks;
- TT gives the set of timed transitions represented by the generic transition $s \overset{<\delta,a,\lambda>}{\longrightarrow} s'$, from s to s' on event a, where δ is a clock restriction (e.g. $(x < 1)?$) and $\lambda \subseteq C$ gives the clocks to be reset with this transition (e.g. $\{y\}$).

Automata composition can also be defined easily. Let

$$A_1 = (S_1, E_1, TT_1, s_{0_1}, C_1),\ A_2 = (S_2, E_2, TT_2, s_{0_2}, C_2)$$

be two timed automata. Assuming that the clock sets C_1 and C_2 are *disjoint*, then the composition of the two automata, denoted a $A_1 \times A_2$, is given by the timed automata

$$A = (S_1 \times S_2, E_1 \cup E_2, TT, (s_{0_1}, s_{0_2}), C_1 \cup C_2)$$

with TT defined as:

- for $a \in E_1 \cap E_2$,
  for every transition $s_1 \overset{<\delta_1,a,\lambda_1>}{\longrightarrow} s_1'$ in A_1 and $s_2 \overset{<\delta_2,a,\lambda_2>}{\longrightarrow} s_2'$ in A_2,
  A contains a transition $(s_1, s_2) \overset{<(\delta_1 \wedge \delta_2),a,(\lambda_1 \cup \lambda_2)>}{\longrightarrow} (s_1', s_2')$;
- for $a \in E_1 - E_2$,
  for every transition $s_1 \overset{<\delta_1,a,\lambda_1>}{\longrightarrow} s_1'$ in A_1 and every $s_k \in S_2$,
  A contains the transition $(s_1, s_k) \overset{<\delta_1,a,\lambda_1>}{\longrightarrow} (s_1', s_k)$;
- for $a \in E_2 - E_1$,
  for every transition $s_2 \overset{<\delta_2,a,\lambda_2>}{\longrightarrow} s_2'$ in A_2 and every $s_k \in S_1$,
  A contains the transition $(s_k, s_2) \overset{<\delta_2,a,\lambda_2>}{\longrightarrow} (s_k, s_2')$.

Composition is as simple as for untimed automata. Just the union of the reset of the clocks and the interception of the transition constraints must be taken into account.

3.5.2 Probabilistic behaviours

Probabilistic behaviours have been used to characterise aspects of a system when they are better described using probabilities. Most of the work on modelling probabilistic concurrent behaviours comes from process algebra. The results, however, will be presented from an automaton point of view.

A widely accepted classification of models of probabilistic processes is given in [87]. Three types of models are introduced: *1)* reactive; *2)* generative; *3)* stratified.

In the *reactive model*, the automaton consists of states and labeled transitions associated with probabilities, where labeled means that the transition has an event associated. The restriction imposed by this model is that, for each state, the sum of the probabilities associated with the transitions labeled with the same event is one. It means that, after an event is observed, the next state is determined probabilistically.

In the *generative model*, the automaton consists of states and labeled transitions associated with probabilities. The restriction imposed is that, for each state, either there are no outgoing transitions or the sum of all the outgoing transitions probabilities is one. This is equivalent to saying that both the event and the next state are to be selected probabilistically.

In the *stratified model*, an automaton consists of states, unlabelled transitions associated with probabilities and labeled transitions. The restriction imposed on a stratified process is that, for each state, either there is exactly one outgoing labeled transition, or all the outgoing transitions are unlabelled and the sum of their probabilities is one. Examples of the three models can be observed in Fig. 3.8

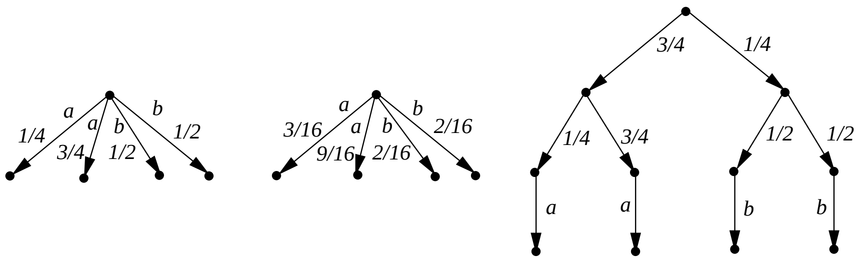


Figure 3.8: Reactive, generative and stratified models

In the remaining of this section some variations of the three models are presented, with respect to the automata specification and composition aspects.

Probabilistic reactive automata

The work of several researchers, such as [87], [88] and [89], has led to a common probabilistic reactive automata definition. A reactive probabilistic automata A is defined as $A = (S, E, RT, s_0)$ where:

- S, E, and s_0 are defined as usual;
- $RT \subseteq S \times (E \cup \tau) \times P(S)$, represented by $s \overset{<a,P>}{\longrightarrow}$, is a set of transitions where P is defined as $P = \{(s_j, p_i) \mid \sum_i p_i = 1\}$ and represents the set of next states annotated with the probabilities.

Once the automaton has engaged in a transition, the final state is selected probabilistically from a given set.

The model does not distinguish *input* from *output* events so, for simplicity, all of them can be interpreted as read (input) events.

The composition of two reactive automata can also be defined. Let $A_1 = (S_1, E_1, RT_1, s_{0_1})$ and $A_2 = (S_2, E_2, RT_2, s_{0_2})$ be two automata. The composition of these two automata, denoted by $A_1 \times A_2$, is the reactive probabilistic automaton A, given by $A = (S_1 \times S_2, E_1 \cup E_2, RT, (s_{0_1}, s_{0_2}))$ being RT defined by
$(s_1, s_2) \overset{<a,P>}{\longrightarrow} \in RT$ iff $P = P_1 \times P_2$, where:

1. if $a \in E_1$ then $s_1 \overset{<a,P_1>}{\longrightarrow} \in RT_1$ else $P_1 = (s_1, 1)$;
2. if $a \in E_2$ then $s_2 \overset{<a,P_2>}{\longrightarrow} \in RT_2$ else $P_2 = (s_2, 1)$.

The product of the two sets P_1 and P_2 is defined as

$$P_1 \times P_2 = \{((s_{j_1}, s_{j_2}), p_{i_1} \times p_{i_2}) \mid (s_{j_1}, p_{i_1}) \in P_1 \wedge (s_{j_2}, p_{i_2}) \in P_2\}$$

and represents the combination of all end states having their probabilities multiplied. The sum of all the probabilities for the same event still continues to be 1, for all the states and events. An example of a composition of two probablistic automata is presented in Fig. 3.9.

Probabilistic generative automata

In [88] a process algebra is defined in which the probabilistic generative model is used. In what concerns the probabilistic view of that theory, events are

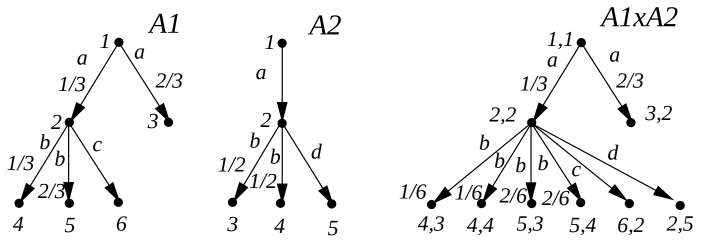


Figure 3.9: Parallel composition of probablistic reactive automata

classified as internal (τ) and external. External events can be *passive (input)* or *active (output)*.

A passive event, such as the reception of a message, is one that requires external *driving*, i.e., the probability of the transition containing this event being executed depends only on the automaton environment. An active event, such as the transmission of a message, is an event controlled by the automaton and to which some statistical attributes can be associated.

Transposing the problem to the automata framework, the generative probabilistic automata $A = (S, E, GT, s_0)$ is used such that

- S and s_0 are defined as usual;
- a transition can be denoted as passive or active and is defined as follows:
  - **active.** $s_i \overset{<a,f_{ij}>}{\longrightarrow} s_j$, where $a \in ACTIVE \cup \{\tau\}$ is an event and $f \in N^+$ denotes its frequency. f_{ij} is a positive integer and means that, for state s_i, this transition will be executed f_{ij} out of the total number of active transitions leaving state s_i;
  - **passive.** $s_i \overset{a}{\longrightarrow} s_j$, where $a \in PASSIVE \cup \{\tau\}$ and denotes the traditional automata transition. Its frequency is undefined and therefore depends on the automaton environment and, for that reason, is not referenced.
- $E \subseteq ACTIVE \cup PASSIVE \cup \{\tau\}$;

If a state has active transitions leaving it, all of them must have frequencies associated. The probability of an active transition t_j leaving the state s_i is given by $\frac{f_{ij}}{f_{i1}+f_{i2}+\ldots+f_{in}}$. A state machine of this type can be thought of as a discrete time Markov Chain, in which each state is a state of the chain and the probabilities of the arcs of the chain are proportional to the frequencies of the automata transitions.

The composition of two generative probabilistic automata is straight forward. Almost everything is done as usual and the following applies for transitions:

- composition $PASSIVE \times PASSIVE$.

$$s_{1_i} \xrightarrow{a} s_{1_j} \times s_{2_i} \xrightarrow{a} s_{2_j} = (s_{1_i}, s_{2_i}) \xrightarrow{a} (s_{1_j}, s_{2_j})$$

- composition $ACTIVE \times PASSIVE$.

$$s_{1_i} \xrightarrow{<a,f>} s_{1_j} \times s_{2_i} \xrightarrow{a} s_{2_j} = (s_{1_i}, s_{2_i}) \xrightarrow{<a,f>} (s_{1_j}, s_{2_j})$$

An example of the parallel composition of two generative probablistic automata is presented in Fig. 3.10.

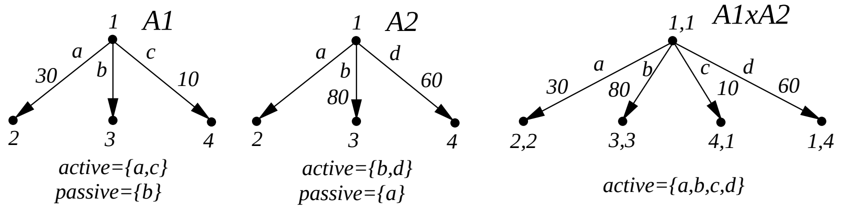


Figure 3.10: Parallel composition of probablistic generative automata

Probable stratified automata

This mode of representing probabilistic behaviours does not allow the existence of transitions expressing simultaneously events and probabilities, therefore the behaviour alternates between probabilistic and deterministic choices. An example of automata composition using this method is shown in Fig. 3.11.

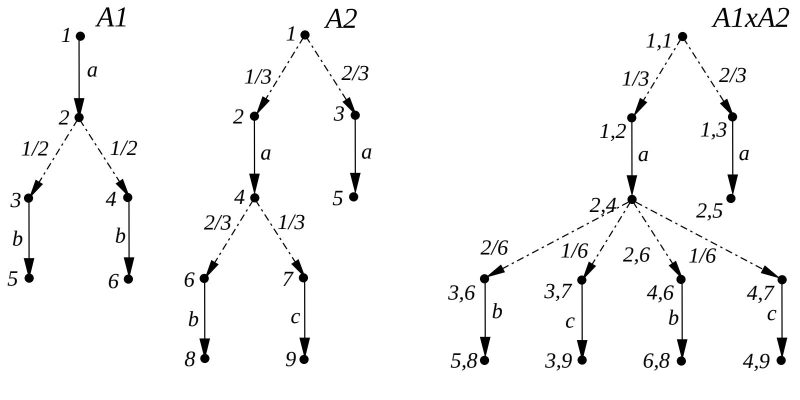


Figure 3.11: Parallel composition of probablistic stratified automata

3.5.3 Time probabilistic behaviours

The development of performance models based on functional models requires that they can describe behaviour, time and probabilities simultaneously. Stochastic Petri-Nets, firstly, and then stochastic process algebra [80], in the last decade, have made important advances with respect to this aspect. The latter has improved the processes description by labelling events with rates λ, thus enabling the transformation of composite processes into *Continuous Time Markov Chains.*

The transformation of these theories into automata, as addressed in [90], seems however particularly appropriate to our test purposes. A particular type of automaton, the *time probablistic automaton*, which classifies events as inputs or outputs, and possesses asynchronous or non-blocking behaviour, in the sense that every input event will be accepted in every state, is rather interesting.

The usual way of starting the definition of a mixed behaviour/performance description is by associating discrete probabilities distributions to transitions from a state. If, however, instead of only one probability distribution for all transitions from state s, several probability distributions are introduced – one over all the output transitions from state s, and separate distributions for each input event a – a kind of hybrid automata between the reactive and generative models described above is obtained.

The introduction of multiple probability distributions still does not solve all the problems. To this end, the concept of delay parameter $\delta(s)$ associated to each state s is required. The idea is as follows: when an automaton, in a composite system, arrives at state s, it draws a random delay time from an exponential distribution with parameter $\delta(s)$. This time ascribes the length of time the automaton will remain in state s before executing its next locally controlled event (output or internal). The competition between several automata trying to gain the control of the next transition is won by the automaton having the least amount of delay left. If independent delay distributions of the competing automata are assumed, a probability can be assigned to the event that will win the competition in any system state. On the other hand, the memoryless property of the exponential distribution makes it irrelevant whether the automata draw the delay times. This last feature makes it possible to give a simple definition of composition for time probabilistic automata.

A time probabilistic automaton can then be described by

$$A = (S, E, T, s_0, \mu, \delta)$$

where

- E is a set of events, partitioned into disjoint sets of input, output and internal events, where $E = E^{in} \cup E^{out} \cup E^{int}$. The set $E^{loc} = E^{out} \cup E^{int}$ is called the set of locally controlled events and the set $E^{ext} = E^{in} \cup E^{out}$ is called the set of external events;

- $T \subseteq S \times E \times S$ is the transition relation, such that, for any state $s \in S$ and input event $e \in E^{in}$, there exists a state $r \in S$ such that $(s, e, r) \in T$. $(s \xrightarrow{e} r)$
- $\mu : (S \times E \times S) \to [0, 1]$ is the transition probability function which is required to satisfy the following conditions:
  1. $\mu(s, e, r) > 0$ iff $(s, e, r) \in T$;
  2. $\sum_{r \in S} \mu(s, e, r) = 1$, for all $s \in S$ and all $e \in E^{in}$. This expression says that for each state s and input event e, the function μ determines a probability distribution such that $s \xrightarrow{e} r$. For each input event there will be one distribution associated, as in the reactive model;
  3. for all $s \in S$, if there exist $e \in E^{loc}$ and $r \in S$ such that $(s, e, r) \in T$, then $\sum_{r \in S} \sum_{e \in E^{loc}} \mu(s, e, r) = 1$. This means that if there is some locally controlled event enabled in state s, then μ determines the probability distribution on the sets of all pairs (e, r) such that e is controlled locally and $s \xrightarrow{e} r$;
- $\delta : S \to [0, \infty[$ is the state delay function, which is required to satisfy the following condition: for all $s \in S$, $\delta(s) > 0$ if and only if there exist $e \in E^{loc}$ and $r \in S$, such that $(s, e, r) \in T$. It means that the state delay δ assigns to each state s a non negative real number $\delta(s)$, which is to be interpreted as the parameter of an exponential distribution describing the length of a random delay period measured from the time state s is entered by the automata until the time it executes its next locally controlled event.

This automaton is particularly suited for composition. Let

$$A_1 = (S_1, E_1, T_1, s_{0_1}, \mu_1, \delta_1),\ A_2 = (S_2, E_2, T_2, s_{0_2}, \mu_2, \delta_2)$$

be two time probabilistic automata such that $E_1^{out} \cap E_2^{out} = 0$ and $E_1^{int} \cap E_2^{int} = 0$. The composite automata $A = A_1 \times A_2 = (S, E, T, s_0, \mu, \delta)$ is defined as follows:

- $E = E_1 \cup E_2$, where $E^{out} = E_1^{out} \cup E_2^{out}$, $E^{int} = E_1^{int} \cup E_2^{int}$ and $E^{in} = (E_1^{in} \cup E_2^{in}) - E^{out}$.

  In this model the combination of an input event with an output event gives an output event which can be used for other compositions (only the input event vanishes);

- T is the set of all $((s_1, s_2), e, (r_1, r_2))$ such that:
  - if $e \in E_1$, then $s_1 \xrightarrow{e} r_1 \in T_1$, otherwise $r_1 = s_1$;
  - if $e \in E_2$, then $s_2 \xrightarrow{e} r_2 \in T_2$, otherwise $r_2 = s_2$;
- $\delta((s_1, s_2)) = \delta(s_1) + \delta(s_2)$;
- μ is defined as follows:
  - if $e \in E^{in}$, then

$$\mu((s_1, s_2), e, (r_1, r_2)) = \mu_1(s_1, e, r_1) \times \mu_2(s_2, e, r_2)$$

  - if $e \in E_1^{loc}$, then

$$\mu((s_1, s_2), e, (r_1, r_2)) = \frac{\delta_1(s_1)}{\delta_1(s_1) + \delta_2(s_2)} \times \mu_1(s_1, e, r_1) \times \mu_2(s_2, e, r_2)$$

  - if $e \in E_2^{loc}$, then

$$\mu((s_1, s_2), e, (r_1, r_2)) = \frac{\delta_2(s_2)}{\delta_1(s_1) + \delta_2(s_2)} \times \mu_1(s_1, e, r_1) \times \mu_2(s_2, e, r_2)$$

An example of the composition of two time probabilistic automata can be observed in Fig. 3.12.

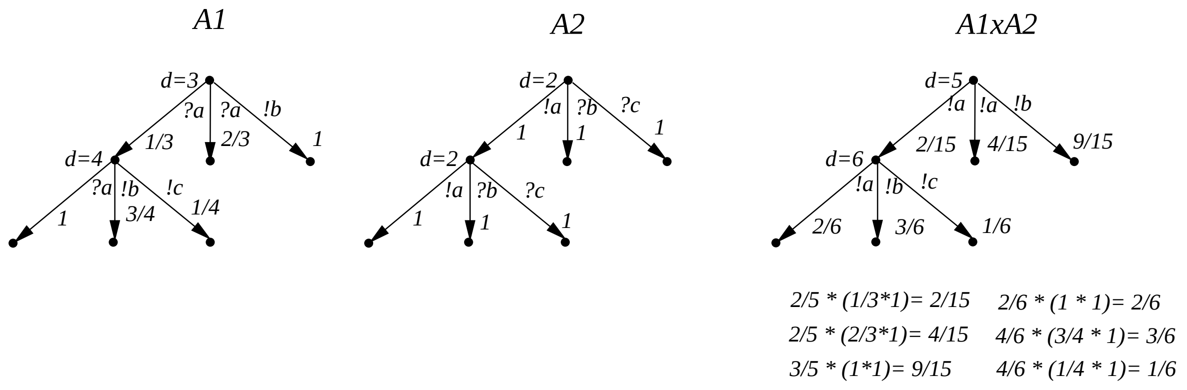


Figure 3.12: Composition of time probabilistic automata

The corresponding continuous time Markov chain, useful for performance evaluation, can be obtained as shown in Fig. 3.13.

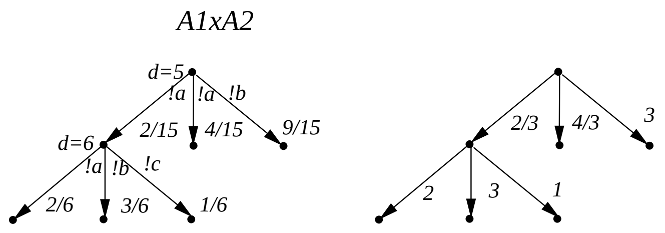


Figure 3.13: The automata as a Markov process

3.6 Conclusions

Performance requirements are, from the testing point of view, very interesting since they raise a new set of requirements which the system has to satisfy. Moreover, they focus on those aspects which are essential both for users and network operators. Although these requirements can be proved satisfied only in the long term operation of the network by means of management techniques, they can be used as a basis for a class of test purposes which, as proved by experience, lead to the detection of complex faults.

The first conclusion of the chapter is that there are, for the class of networks addressed in this thesis, a set of performance and quality requirements available which can give guidance for the development of system test purposes. These requirements are based on parameters for delays, failures and availability and have maximum values defined. The Network Element, in the end, has to satisfy them.

The second conclusion is that the specification of quality does not follow the style of behaviour specifications. While the former are requirement specifications, the latter are operational specifications.

The third conclusion is that behaviour and performance state machine models can be available and represent the same comunication system. Each model, however, concentrates only on the relevant parts.

This leads to the fourth conclusion. There is room, and computer scientists are showing it, to describe and reason simultaneously about behaviour and performance. These models, if available, could be used to generate timed tests which would also evaluate system probabilistic properties.

Chapter 4

NE Testing Methodology

4.1 Introduction

The aim of this chapter is to present a new methodology for testing complex telecommunications Network Elements. As referred in Chap. 1, the methodology is based on existing testing practices but it profits from recent advances in protocol testing theories described in Chap. 2.

The two worlds of practical and formal testing methods are separated by an enormous gap which, from our point of view, is caused by two main reasons:

- telecommunications test engineers, which usually do not have background in computer science, find the formal testing approaches extraordinary complex so they are reluctant to apply them;
- unlike protocols, the Network Elements are specified using multiple techniques, which are selected by their modeling power and expressiveness rather than by their mathematical basis. The formal testing techniques are, for this reason, not directly applicable.

We believe that we are well positioned to contribute to the shortening of this gap since we had the opportunity to test some real networks/ Network Elements and, on the other hand, we have been given the time required to study the formal testing methods and to reflect about their usability in the context of large telecommunication systems.

The approach selected to reach our goal was the most direct. We tried to understand each practical test step from a formal testing point of view. By doing that, we expected to identify opportunities for improving the testing practices by applying formal nearly solutions. This led to a problem: at which level should the test solutions be described: English? Programming like languages? Process algebra/ temporal logic? We would like to be rigorous but, on the other hand, we would like to be readable by test engineers. The decision was to describe our test methodology using the simplest framework which could cope with the *essential* testing problems and solutions and, at the same time, would be familiar to test engineers – an automata based framework.

In the second section of this chapter the problem of testing telecommunication Network Elements is characterised. Their inherent complexity, specification diversity and management expectations are presented and justified. A comparative study between formal testing methods and current testing practices is also carried out. Following the approach that the testing methodology proposed should depart from current testing practices, the two main methodology assumptions are also introduced: tests should be oriented towards components and should follow the use case approach. This section, which is original, was developed based on our practical experience of testing networks and on the study of the formal testing methods.

The third section introduces the framework used to describe the methodology by detailing and making clear some of the aspects presented in Chap. 2. The description of a specification by a safety automaton, where the latter is obtained by simulating the former, and the representation of use cases as guarantee automata are discussed. The derivation of test and verdict functions based on these automata is also presented. It must be recalled that this form of representing specifications and use cases is often used in formal testing. The description given in this chapter, however, is oriented towards the characteristics of the methodology proposed. In particular, the clear division between tests and verdict functions and the assumption of complete use cases is, to the best of our knowledge, presented in an original form.

The fourth section characterises the test derivation method for components which are not completely specified. Two types of incomplete components are addressed: those for which only use cases are known (requirement components) and those which can have also their communication interfaces known (interface components). The main innovation of the section consists basically on the identification of a problem - tests should be provided for incomplete components. The solution presented, which is also new, is an extension of the testing method proposed in section 2.

The fifth section presents the concept of cyclic testing. Our practice shows that most of the faults become visible only after long components working times and, very often, they are random. The basic test derivation method is then improved in order to detect these faults. The description of the components specifications and implementations as a set of random variables and the meaning of faults in this context is also given. This section, which is original, constitutes one of the most important contributions of this thesis.

The sixth section addresses time, which is vital in practical testing. The test method will, for that reason, incorporate the time aspects. The problem, which from the formal point of view is complex, is solved with a hybrid but workable solution: the component specification is described as an untimed automaton while use cases are represented by timed guarantee automata. The basic and cyclic test algorithms are also reviewed to incorporate time. The originality of this section consists mainly on the combination of untimed specifications and timed used cases to derive timed tests. Its strongest point consists in addressing exactly the problems which test engineers have to solve when testing a Network Element with respect to its timings.

The seventh section presents our methodology as a *recipe* by combining the concepts presented in the previous sections and showing their complementary value. The Network Element is modeled as two sets of non-interacting components: the *communication* set and the *service* set. Communication components are tested individually using short tests. Service components are tested both individually and jointly. This section, which is original, is one of the most important of the thesis since it shows to formal testing methods a direction for solving their main problem, i.e. the system size problem. By creating two complementary views of the systems (communication and service), the complex Network Elements are reduced to simple testable components.

The last section presents reference test architectures for the three types of tests (individual communication components, individual service components and joint service components). It is also original since it points out some service based testing architectures which are unusual in traditional load testing.

Partial presentations of the methodology can also be found in [91], [92], [93].

4.2 NE testing problem

4.2.1 Testing complexity

NE subsystems

The Network Elements addressed in this thesis can be complex systems since they may have, simultaneously:

- to support several communication interfaces;
- to provide end-user services;
- to provide management services.

The main purpose of communication interfaces is to ease the interworking of communicating equipment. Standard or *de facto* interfaces, in addition, are targeted at multivendor equipment. For this reason, these interfaces are defined independently of the functions provided by each equipment. NEs, being traditional public voice/data telecommunication equipment, need to implement one or more communication interfaces.

In addition to the interfaces, the NE has to provide a set of functions or services to its users, which can be other equipments or humans. From the implementation point of view, these services have to be harmoniously integrated with the communication interfaces since, at the end, the services are provided through the system interfaces.

An NE has also to implement management services which provide information mainly on accounting and alarm detection. In spite of the efforts carried out by the standard bodies to develop global management systems, in practice NE management subsystems still continue to be proprietary. Independently of that, and from the testing point of view, management can be thought as another set of services, which are usually provided to humans or to some programs.

Specification diversity

The NE testing complexity, however, does not come only from the diversity of the subsystems implemented, which is characteristic of most of the reactive systems, but from their *specification characteristics*. The success of the testing activities strongly depends on the quality of the specifications available, since complete and efficient test suites cannot be developed without complete and rigorous system specifications. A system is said to be in these test-optimal conditions when, for instance, it is described in SDL. Reality, however, is different from this ideal scenario.

The first problem faced by testers is the multitude of techniques used in the description of NE subsystems. An interface, for instance, can be physical or protocol oriented. In the first case, characteristics such as voltage, current, time, frequency, impedance and baud-rate will be presented. In the second case a variety of approachs can also be found. With low probability, an interface can be completely described in SDL and have the message and data types described in ASN.1. The normal situation in ETSI and ITU is to find English descriptions complemented with tables and, in annexes, a rough SDL description. IETF interfaces are described in text and complemented by some illustrating state machine. Proprietary interfaces are, most of the times, described as requirements such as a set message sequence charts describing the main use cases. The requirement style is also found in the American standards recommendations. The management subsystem, if standardised, can be described by an object model where the relations between the objects as well as their main methods are presented. ASN.1 descriptions may in some cases be also available. End-user services, which may be specified as internal documents, can also be described in a variety of techniques. The most frequently found are textual descriptions complemented with figures and some message sequence diagrams. Besides the subsystems descriptions, there are also performance and quality specifications which have to be taken into account and are described as shown in Chap. 3.

The second problem is the number of documents available. NEs are complex mainly because they are large. Large systems are usually split into smaller subsystems whose requirement identification and specification are worked on by multiple persons/ teams. Very easily, tens of documents may be available describing the behaviour of a NE, among requirement/ specification documents and international recommendations. As a consequence of that, a number of inconsistencies are usually found.

The third problem is specification updating. In environments where the time to market drives the project, the time left for updating specifications during the project is usually short. Very often, the specification document considered as the basis for the test derivation no longer describes the latest specification view and, in this situation, the test becomes useless.

Project management requirements

Besides the problems described above, there are also a set of expectations from the project management which, at the end, have to be satisfied. Moreover, the degree of satisfaction of these expectations are, in fact, the final measure of the tests value. They can be stated as follows:

- *The NE shall be tested in short times.* Even when realistic testing policies are adopted, test activities start after the other specification activities. On the other hand, the NE delivery/ production, at the end, should not be delayed by the testing activities. It means that the test related tasks start after the NE specification tasks and end at the same time as the development tasks. In cases where the company leads the business, the situation becomes even more complicated. Some test equipment may have to be developed and tested during the testing activities because commercial test equipment may not be available.

- *The NE shall pass interface conformance tests.* The telecommunication operators are increasingly demanding that the conformance tests made available for the standard interfaces are passed by the implementations and they may pose serious problems. The ETSI or ITU conformance tests are nowadays developed by hand. For that reason, first test releases are faulty and delayed with respect to the specification documents. Passing the conformance tests may require, for these reasons, that the test suites are themselves evaluated and that some effort is placed on their correction and reporting.

- *The NE shall pass interoperability tests.* In order to test complex communication interfaces, wise and skilled project managers try to define small interoperability sessions as soon as possible. These trials may involve peer equipment from the same or multiple companies. Anyway, the kind of interface functions which are tested are usually known and agreed in advance. For strategic reasons, the equipment should not exhibit failures during these small experiences. Bad news propagate very fast.

- *The NE shall have its performance and QoS requirements characterised.* In an increasingly competitive world, selling network equipment is becoming similar to, for instance, selling cars. At the end, all the equipments provide similar functions which must be free of problems. The performance characterisation of the equipment becomes very important since, more and more, equipments tend to be compared based on performance issues. This implies that relevant parameters must be known in advance so that the project management may decide about improvements. For this reason tests and test equipment should be available in time.

- *The NE shall pass the operators acceptance tests.* Before acquiring equipment, telecom operators usually require them for internal evaluation. A variety of interface and conformance tests are usually applied to the equipment. Naturally, the equipment should be able to pass all these tests.

- *The NE shall be free of end-user visible faults.* A good working equipment will be the best argument for selling its next generation substitute. For that reason, the system is supposed to be free of faults which affect the final user activity, that is, the operators customers. Faults must be found in advance.

4.2.2 Formal versus practice

Practical methods

A Network Element is seen by development and test engineers as a set of components, such as boards, drivers, libraries, classes or processes. In this context, a component implementation fault is usually known as a *bug* which can be caused by (1) an incorrect specification interpretation or (2) bad coding of a correctly interpreted specification. When detected, the bug can be located at the component under test or at the resources used by the component.

Bugs are generally detected by tests which can be of three types: (1) component tests, (2) integration tests and (3) load tests. *Component tests* excite the component implementation with short tests and detect problems such as incorrect IOSM transitions implementation and incorrect PDU (de)coding. *Integration tests* are used to evaluate new composite components which are built based on previously tested components. These tests are also short and problems such as components erroneous assumptions on neighbour components or bad resource sharing can be identified. *Load tests* consist of using the NE with real loads. These tests are usually long and problems such as wrong behaviours caused by performance degradation or long term visible faults (variable/ queue underdimensioning, dynamic memory (de)allocation) can be detected.

Formal methods

In formal testing methods, specifications are described by languages having formal operational semantics. Thus, a NE can be mathematically described by, for instance, a directed graph representing the system reachable states which are interconnected by transitions annotated with observable events. When the specification graph $SPEC$ is known, the NE implementation can be also assumed to be modeled by another (unknown) graph IMP. The concept of fault is, then, described at the model level and depends on the relation required between the implementation and the specification models.

If, for instance, instead of graphs the set of traces they can generate are used - $Traces(SPEC)$ and $Traces(IMP)$, two type of equivalences, among many others, can be defined: (a) trace equivalence and (b) trace pre-order.

Trace equivalence requires that $Traces(SPEC) = Traces(IMP)$. In this case a fault is said to exist when a trace σ is found that

$$\begin{gathered}(\ \sigma \in Traces(IMP)\ \wedge\ \sigma \notin Traces(SPECS)\ )\\ \vee\\ (\ \sigma \notin Traces(IMP)\ \wedge\ \sigma \in Traces(SPECS)\ )\end{gathered}$$

In the first case the implementation is said to contain a behaviour which was not specified. In the second case, the implementation is said to have not implemented part of the specification.

Trace pre-order, on the other hand, requires that $Traces(SPEC) \subseteq Traces(IMP)$. This relation requires that all the behaviours specified are implemented but the implementation is not forbidden to have other behaviours. In this case, a fault is said to exist when there is a trace σ which is unimplemented.

$$\sigma \notin Traces(IMP) \;\wedge\; \sigma \in Traces(SPECS)$$

A fault is detected by the observation of a faulty trace. The characterisation of the fault depends on the underlying models. When automata are used, a fault can be a transition not implemented or incorrectly implemented. When the automaton distinguishes input from output events and accepts input events in all the states, i.e. like the $IOSM$ described in Chap. 2, the faults are further classified as output or transition faults.

Formal methods tend to generate tests which depend on the relation expected between the specification and the implementation, such as the methods presented in Chap. 2 for $IOSM$, which can prove trace equivalence.

Concepts such a load tests are difficult to explain in this context. To find a fault under load condition means, in the mathematical view, to place the system in a state where queues are almost full and variables close to their limit values. The test would consist then in verifying transitions departing from these potentially dangerous states.

Comparison

Tab. 4.1 compares current NE testing practices with formal testing approaches with respect to a number of issues. It summarises and compares the results of the previous discussion.

Issues	Formal Methods	Current Practices
Specification	formal semantics	not always available multiple techniques
Faults	deterministic	random
Functional tests	from specs equivalence proof system oriented	from interfaces use case oriented component oriented
Load test	functional problem	important
QoS evaluation	property evaluation	simple accounting
Fault coverage	characterisable	uncharacterisable
Test architecture	specification oriented	implementation oriented

Table 4.1: Comparison between formal and practical testing methods

A specification, in formal testing methods, is supposed to be available in a language capable of representing the system relevant aspects and having rigorous semantics. Languages supporting concurrent systems and data are the preferred. In practice, as mentioned above, specifications are not perfect. They are not always complete nor expressed in rigorous languages. In fact, a specification language is selected more by its modeling power and less by its operational characteristics.

In formal methods, the faults are assumed to be deterministic and the tests are executed only once. Reality shows the opposite. Complex faults, i.e. those requiring the help of a testing team, manifest themselves randomly at the model level. Their detection requires that tests are executed many times or that long tests are used. When clear specifications are available, the deterministic faults are mostly identified by the development engineer during components tests since a good implementor usually interprets them correctly.

Tests, in formal testing methods, are derived from the specifications so that some equivalence between the specification and the implementation can be proved. The system, described in some formal behaviour description language, in which a set of components can also exist (perhaps not the real ones), is simulated so that a minimal observable specification can be obtained. In practical methods, the tests are oriented towards components and they are based on partial specifications, such as component interfaces.

Load tests can, at the model level, be reduced to simple and long functional tests. In practice, load tests are recognised as very important because they address the complete system and are understood as the simplest form of placing the system near its limits in a laboratory environment.

The NE performance evaluation is, in practical testing methods, evaluated under controllable load conditions by monitoring simple system parameters, such as CPU load or delays. In formal testing methods a performance characteristic is foressen as a system property which can be verified by means of a functional like test. Research on these problems is starting since the behaviour models may require also the description of time and propabilistic aspects.

In formal methods the number of faults which can be found by a set of tests out of a maximum number of faults can be determined. Test coverage can then be characterised. There are, however, no means to assess how close the faults modeled are from the real faults. In practice, test coverage cannot be rigorously evaluated and the decision to stop testing is usually taken based on human experience.

In practical methods, the definition of a test architecture is NE dependent and the tests are used in place of the real NE/ components communicating partners. Formal methods researchers are starting to address the problem, trying to infer test architectures from the implicit specification architectures.

The question is then how to improve existing practical methods so that

- the advantages of formal testing methods can be gained;

- the method still applies to real systems;
- the common testing demands are satisfied.

4.2.3 Components

The methodology proposed in this thesis follows the practical approach of considering a NE as a set of components. For this reason, testing is oriented towards components, since it seems to be the human preferred process of testing - it provides compositionally secure components.

A *component* is defined as an entity capable of participating with its environment in a set of common events, such as message reception/ sending or function invocation/ return. NE components are considered reactive in the sense that they are usually non-terminating entities which produce work only when explicitly requested. From the engineering point of view, a component is an object capable of providing to its environment a set of services or functions. Some programming languages have means to explicitly describe component types. C++ has the *Class* construct and SDL has, for instance, the *Block Type* and the *Process Type.*

From the testing point of view, components have to be further characterised. Testing practices show that, according to the way they are described, components can be classified into three categories:

- requirement;
- interface;
- complete.

Requirement components

A component is classified as requirement when its description consists only of a set of requirements. It does not imply that the component knowledge is unavailable, but only that it has not been described in any specification document. Examples of requirement components are computer programs for which initial specifications are vague and the implementer is free to code the program at his will within some usability limits, such as simple descriptions of the main program functions. Let us assume that requirements are expressed in user observable terms.

A requirement is a characteristic that a component implementation has to incorporate. Requirements can be described positively (the component *must* do this) or negatively (the component *must not* do this). Component use cases are requirements described positively since they specify one or more sequence of events which must be executed by the component.

Negatively stated requirements are impossible to evaluate. To prove, by testing, that an implementation does not possess a given trace is almost impossible since the implementation model is, a priori, unknown. While widely used in the verification of known finite models, negatively expressed requirements are incompatible with short test times.

The requirements considered in this section will, for these reasons, be component uses cases.

Interface components

Besides a set of use cases, these components have also their user interfaces defined precisely. This is the specification technique most often used for network components. Instead of specifying the complete component internal behaviour, the specifier describes only the component interactions through each one of its interfaces. This method is implicitly used in object oriented description techniques where a class along with its methods is first identified and then each method is characterised by its signature which is defined in terms of parameter data types and return values. Programs interfacing with humans often follow this approach - the specification is a fast developed prototype application containing no internal functionality but presenting only the windows, the commands and the responses that the component user will be able to find.

Complete components

Additionally to a set of use cases, these components have complete and precise specifications available. This is case when, for instance, a protocol or parts of a system are completely specified in SDL. In this case, the use cases are usually available as Message Sequence Charts and current SDL commercial tools are able to support the automatic generation of behaviour tests.

These components are the preferred in terms of testing but, unfortunately, not always available.

4.2.4 Tests

Based on the current testing practices, the proposed methodology assumes the requirement testing approach, that is, implementations will be tested against known use cases. This test method, due to the human ability to guess about faults and their location, seems to be the most efficient.

Besides that, tests will consider that certain faults become evident only after long working times while other faults manifest themselves randomly. For this reason, load tests will be considered very important since they can excite NEs with realistic environments and, at the same time, provide the framework required for carrying out performance/quality of service related measurements.

4.3 Complete component behaviour testing

This section describes how complete components can be tested with respect to some properties, such as component use cases.

4.3.1 Component safety automaton

A NE component can be represented as a simple reactive system composed of one or more concurrent processes exchanging information among themselves and also their environment. As widely accepted and shown in Chap. 2 for the SDL case, a complete component can be represented by an automaton which results from the simulation of the component processes along with its environment. This automaton is called the *safety automaton* A_S, and was also introduced in Chap. 2.

Each component process is a program containing some internal flow control mechanisms and data variables. From time to time, each process reads/ writes information from/ to other processes or from/ to the component environment. Let us assume that a process is described by an automaton where each state represents a particular combination of internal variable values. The directed transitions between states are labelled with events which correspond to the writing (!) or reading (?) of some combination of values. The process events which are externally invisible, such as a variable update, are considered internal to the process and generically represented by the event τ. Fig. 4.1 presents a simple component containing two concurrent processes.

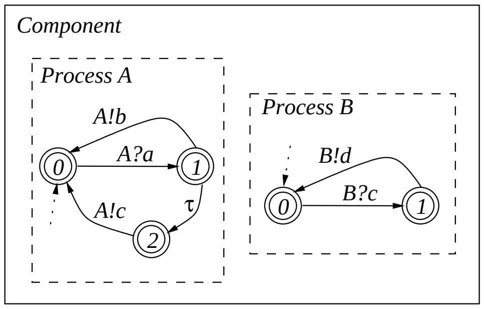


Figure 4.1: Component with concurrent processes A and B

Let Process A be represented by the automaton A:

$$
\begin{aligned}
&A = (S_A, E_A, T_A, s_{0_A}, F_A)\\
&S_A = \{s_0, s_1, s_2\}\\
&E_A = \{A?a, A!b, A!c\}\\
&T_A = \{s_0 \xrightarrow{A?a} s_1, s_1 \xrightarrow{\tau} s_2, s_1 \xrightarrow{A!b} s_0, s_2 \xrightarrow{A!c} s_0\}\\
&s_{0_A} = s_0\\
&F_A = \{s_0, s_1, s_2\}
\end{aligned}
$$

Process A has three states (s_0, s_1, s_2) whose meaning is not relevant for the testing problem. One of the states (s_0) is the automaton initial state. All states are considered final states $(F_A = S_A)$, in the sense that the process can stop in every state. Process A communicates with its environment by means of information elements (a, b and c). For simplicity, these information elements are called *messages*. The receiving/ reading of message a from the component environment is represented by event $A?a$. Process A can also send message b to the component environment, represented by event $A!b$, and send message c, represented by $A!c$, to process B. Process B is represented by

$$
\begin{aligned}
&B = (S_B, E_B, T_B, s_0, F_B)\\
&S_B = \{s_0, s_1\}\\
&E_B = \{B?c, B!d\}\\
&T_B = \{s_0 \xrightarrow{B?c} s_1, s_1 \xrightarrow{B!d} s_0\}\\
&s_{0_B} = s_0\\
&F_B = \{s_0, s_1\}
\end{aligned}
$$

Process B can be interpreted similarly to Process A. In this case, the sending of message d to the component environment is represented by the event $B!d$ while the reception of message c from process A is represented by event $B?c$.

For simplicity, processes A and B are supposed to be *introduced* to each other at compilation time (no dynamic address resolution) and the communication is supposed to be one to one (no multicast facilities). Although some interprocess communication types exist, which are combinations of the sender/ receiver blocking policies and the existence of mailboxes, only two of them will be discussed: synchronous and asynchronous communication. Let us first represent the component containing the concurrent processes A and B by

$$A \parallel B$$

Synchronous communication

This type of communication is usually known by *rendez-vous*. It means that a process can send a message only if the other process is able to receive it, that is, it is in a state containing an outgoing transition which is labelled with the corresponding receiving event. If this is not the case, the sending process

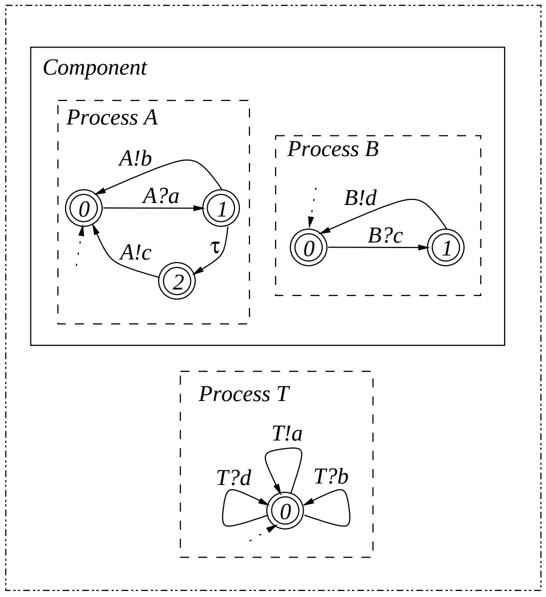


Figure 4.2: Maximum environment for $A \parallel B$

is prevented from sending the message. Since the two automata representing the processes are assumed to have disjoint event sets,

$$\{A?a, A!b, A!c\} \cap \{B?c, B!d\} = \emptyset$$

a message communication will be graphically represented by *two immediate transitions*, which correspond to the sending and the receiving of the message. A similar approach was used in Chap. 2 for building the SDL reachability graph.

The component reachability graph or safety automaton is obtained by simulating the component. In order to close the component $A \parallel B$, some assumptions have to be made on its environment. A maximum behaviour environment is one which is always able to receive all the messages and send any message. The behaviour of such an environment could be modeled as described in Fig. 4.2 and represented by the following automaton

$$\begin{aligned}
&T = (S_T, E_T, T_T, s_0, F_T)\\
&S_T = \{s_0\}\\
&E_T = \{T!a, T?b, T?d\}\\
&T_T = \{s_0 \xrightarrow{T!a} s_0, s_0 \xrightarrow{T?b} s_0, s_0 \xrightarrow{T?d} s_0\}\\
&s_{0_T} = s_0\\
&F_T = \{s_0\}
\end{aligned}$$

T was selected to represent the environment in order to show its similarity with a test program. The simulation of the closed component $(A \parallel B) \parallel$

T can now be carried out according to the communication rules described above. The reachability graph obtained is presented in Fig. 4.3. States are represented as (s_A, s_B, s_T) where s_A, s_B and s_T are the individual process states. The transitions are labelled with one event. Since an event never belongs to more that one automaton, only one process changes state at a time. However, in order to model the *rendez-vous* mechanism, the reception of a message is always required to follow the sending of the same message.

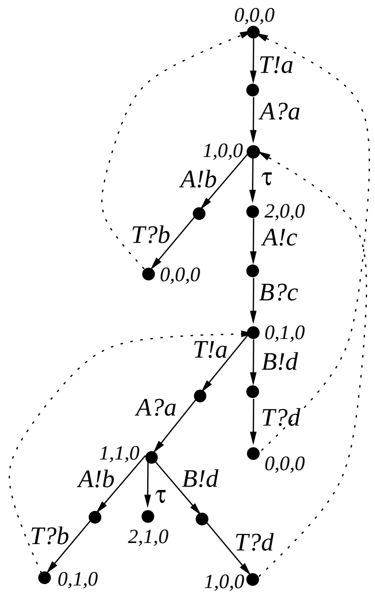


Figure 4.3: $A \parallel B \parallel T$ reachability graph

From the testing point of view, in black-box testing only the events *observed* or *controlled* by the environment are of interest, that is, the event set

$$E_T = \{T!a, T?b, T?d\}$$

For this reason, all the other events, $(A?a, A!b, A!c)$ and $(B?c, B!d)$ will be relabelled as internal events τ. The new graph is presented in Fig. 4.4. This graph is called the *safety automaton* A_S of the component $A \parallel B$. It represents all the possible sequences of events of $A \parallel B$ which, for short, are represented by $Traces(A_S)$. Since all the safety automaton states are also final states, $Traces(A_S) = Accept(A_S)$.

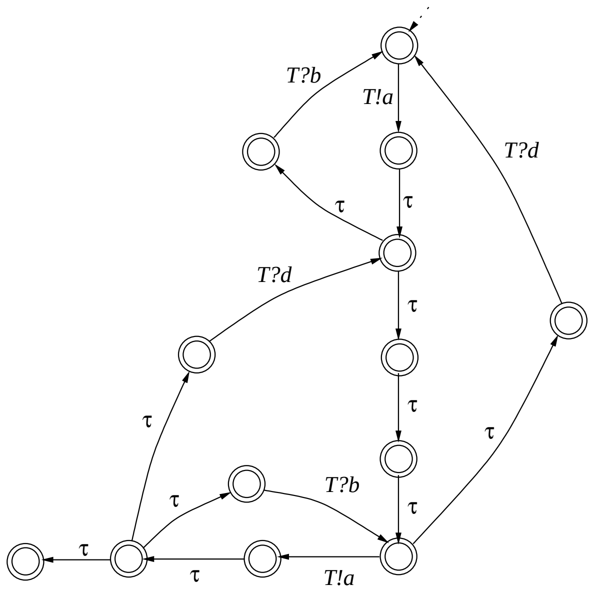


Figure 4.4: $A \parallel B$ safety automaton, A_S

Asynchronous communication

Let us now assume that, instead of synchronous communication, the processes use another communication paradigm. It is assumed in the first place that each process has a mailbox associated which, for simplicity, can contain only one message at a time.

In the synchronous case, both the sending and receiving processes were assumed to be blocking. Now, both the sending and the receiving will be non-blocking. It means that if a process has to send a message, it sends it independently of the receiver process mailbox being full or not. If a receiver process is expecting a message but its mailbox contains another message, it retrieves the message from the mailbox and stays in the same state. This transition is often named an implicit transition.

Let us take the component of Fig. 4.5, where the component is represented by Process C and its environment by Process T.

The reachability graph of $T \parallel C$ as well as its equivalent safety automaton A_S, which is obtained by eliminating the non observable/ controllable events, are presented if Fig. 4.6, where the states are represented by the message contained by each process mailbox. For instance, $< b >< >$ in Fig. 4.6 indicates that the process T has the message b in its mailbox and the mailbox of process C is empty. Processes C and T are always in the control state s_0.

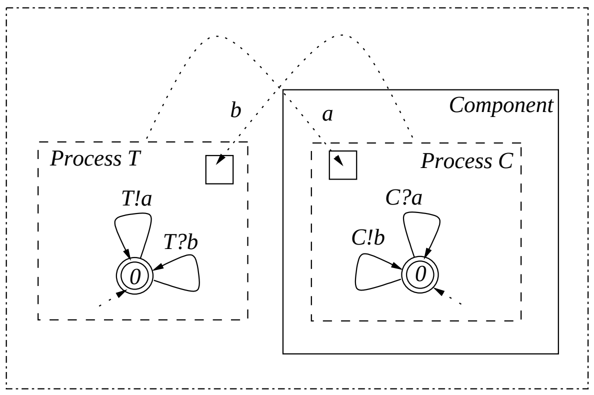


Figure 4.5: Asynchronous communication component

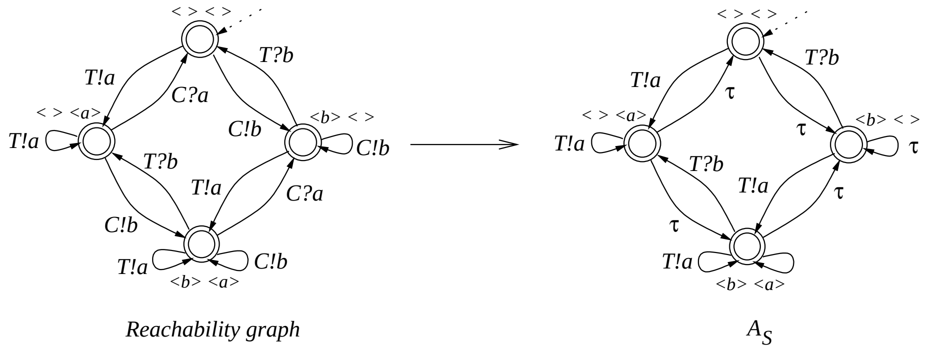


Figure 4.6: Asynchronous component safety automaton

4.3.2 Use case guarantee automaton

Besides the component specification, represented by the safety automaton, another concept is also required - the *use case*. A use case describes one possible component usage and consists of a sequence of messages (send/ receive or read/ write) that a component implementation has to support. Use cases are usually described from the component user point of view.

In this thesis, a use case will be represented by an automaton, denominated the *guarantee automaton* A_G, with the following characteristics:

- $E_G = E_S$. The event set recognised by the guarantee automaton E_G will be equal to E_S, the event set of the corresponding safety automaton. It means that a guarantee automaton reacts to all the component environment observable/ controllable events;

- *complete.* The traces accepted by A_G, *Accept*(A_G), are traces of A_S and capable of driving A_S from its initial state s_{0_S} back to its initial state s_{0_S}, that is, they describe one or more complete loops of the safety automaton. Mathematically it can be defined as:

$$\begin{gathered} Accept(A_G) \subseteq Accept(A_S) \\ \wedge \\ \forall \sigma \in Accept(A_G),\ (s_{0_S}\ \mathbf{after}_{\mathbf{A_S}}\ \sigma)\ =\ \{s_{0_S}\} \end{gathered}$$

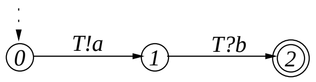


Figure 4.7: Use case defined as a guarantee automaton A_G

Fig. 4.7 shows a use case defined for component $A \parallel B$, whose safety automaton is shown in Fig. 4.4. From the test point of view, the use case defined in Fig. 4.7 says that a tester program, after sending message a to an implementation of $A \parallel B$, should receive message b.

The guarantee automaton A_G describes the automaton of Fig. 4.7:

$$\begin{aligned} & A_G = (S_G, E_G, T_G, s_0, F_G) \\ & S_G = \{s_0, s_1, s_2\} \\ & E_G = \{T!a, T?b, T?d\} \\ & T_G = \{s_0 \xrightarrow{T!a} s_1, s_1 \xrightarrow{T?b} s_2\} \\ & s_{0_G} = s_0 \\ & F_G = \{s_2\} \\ & Accept(A_G) = \{T!a.T?b\} \end{aligned}$$

4.3.3 Test derivation

The question now is how to obtain a test which is capable of evaluating whether an implementation can execute or not a given use case. According to Sec. 2.6.4 (Testing requirements, Behaviour properties) this test can be derived based on the component safety automaton and on an use case guarantee automaton. The former can be obtained from the component specification (e.g. SDL). The latter can obtained from an use case described as a Message Sequence Chart or described directly by the person in charge of specifying the test purposes.

The main objective of a test is to force the implementation to react. Based on the sequence of events observed (the log obtained after the test execution), which consists both on the signals sent and received by the test system, a conclusion about the satisfaction of the safety and guarantee properties can be drawn. Based on that, the test verdict (pass, fail or inconclusive) can be obtained.

During the test of an implementation, this test will be used in substitution of the maximum behaviour component environment (T) assumed during the development of the reachability graph/ safety automaton.

The specification of a test implies the characterisation of two aspects which are closely related but are presented here separately:

- a trace evaluation function $verd(\ )$, which can associate a verdict to every trace obtained as the result of the joint execution of the test and the implementation, that is, $T \parallel IMP$;
- the test T, which will be able to drive the implementation, that is, to send messages as a function of the messages received from the implementation, so that the implementation can be evaluated with respect to the use case in short time (small trace).

Trace evaluation function

The aim of the trace evaluation function $verd()$ is to classify every trace as *pass*, *fail* or *inconclusive.* A trace is classified as *pass* when it does not violate the component specification and enables the observation of the use case. A trace is classified as *fail* when it violates the component specification. A trace is classified as *inconclusive* when it does not violate the component specification but does not enable the use case observation. The $verd()$ function depends on the safety and guarantee automata.

More formally, the trace evaluation function $verd()$ associates the value *pass*, *inc* or *fail* to every possible trace $\sigma' \in E_S^*$, i.e. any combination of events $e \in E_S$. Note that E_S^* is used instead of $Accept(A_S)$ because a faulty implementation modeled by A_I can generate a different set of traces. For simplicity, however, they are assumed to recognise the same events, that is, $E_I = E_S$.

The commonly accepted definition for this function is

$$verd(\sigma') = \left| \begin{array}{ll} pass, & \text{if } \sigma' \in Accept(A_S) \wedge \sigma' \in Accept(A_G) \\ inc, & \text{if } \sigma' \in Accept(A_S) \wedge \sigma' \notin Accept(A_G) \\ fail, & \text{if } \sigma' \notin Accept(A_S) \end{array} \right.$$

The function $verd(\ )$ can be described as an automaton V, with the following characteristics:

- *acceptance condition.* Since the traces that will be observed are a priori unknown, every $\sigma \in E_S^*$ must be accepted by V, that is,

$$Accept(V) = E_S^*$$

- *determinism.* Since each trace is required to be uniquely evaluated, only one final state should be reached by the automaton when interpreting σ, that is

  $$\forall\sigma \in E_S^*,\ \sharp(s_{0_V}\ \mathbf{after_V}\ \sigma\ ) = 1$$

  where $\sharp$ is represents the number of elements in the set.

- *final states.* Every final state of V must be classified as $pass$, inc or $fail$, that is,

  $$F_V = PASS \cup INC \cup FAIL$$

  where $PASS$, INC and $FAIL$ are pairwise disjoint sets.

A trace evaluation function has two problems associated with it. The first concerns the evaluation of very large traces. To verify if a long trace is part or not of (accepted by) the safety automaton would require an evaluation function as complex as the specification. However, even for very long traces, it is possible to evaluate the trace based on a prefix. In fact, as soon as a trace event makes it possible to evaluate that the trace will not be accepted by the *guarantee automaton*, it can be said that the verdict will not be pass. For simplification, the trace will be classified as $fail$ or *inc* based on this *deviating event*. The verdict will be *inc* if the prefix terminating with the deviating event is accepted by the specification, i.e. valid from the specification point of view (A_S), or $fail$ if not. The second problem is the decision about the end of the trace. Since the verdict evaluation function may be required to run on real time, it has to decide if it should wait for a new event or, on the contrary, take the decision based on the information already available. In order to overcome this problem a special event ϕ, representing the last event of a trace, will be added to the end of each trace by the test (T - introduced next), so that the verdict function can interpret the end of the traces and give the verdict.

The automaton V describing the $verd()$ function can be derived from A_G and A_S. It means that the (initially unknown) states, events and transitions associated with V will be created based on the states, events and transitions of A_G and A_S. Below, the steps required for this derivation are presented as an algorithm working over sets of states, events and transitions. See also Fig. 4.8.

1. build an automaton V which is structurally equivalent to A_G, that is, it has the same states and transitions as A_G

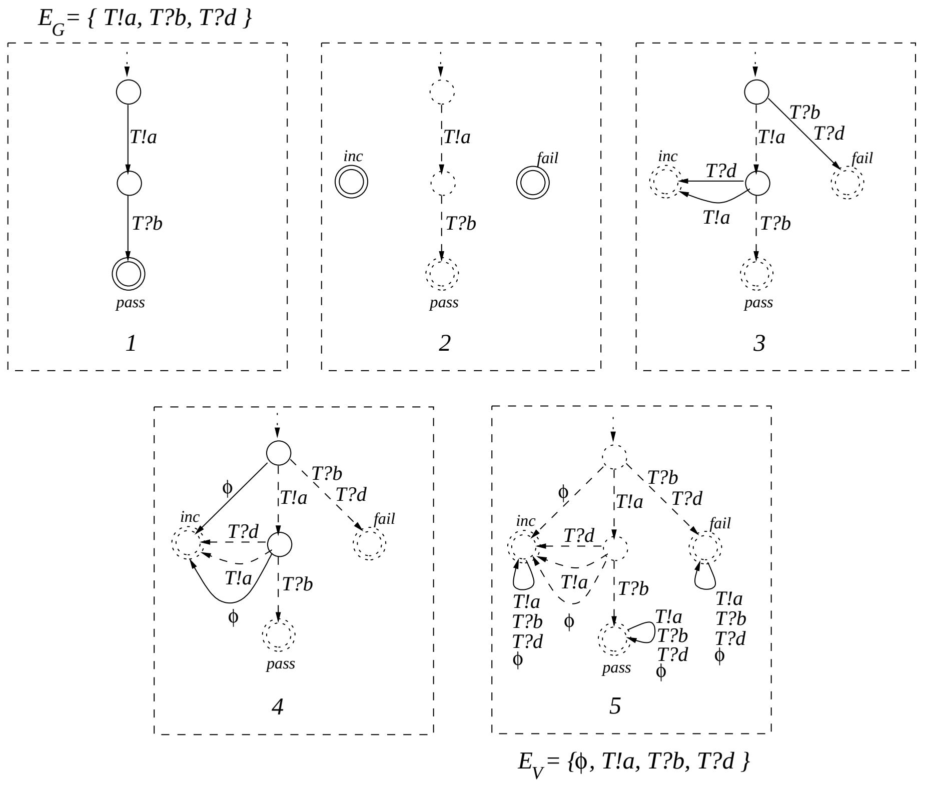


Figure 4.8: $verd(\ )$ for $A \parallel B$

$$\begin{array}{lcl}
V & = & (S_V, E_V, T_V, s_{0_V}, F_V) \\
A_G & = & (S_G, E_G, T_G, s_{0_G}, F_G) \\
S_V & \leftarrow & S_G \\
E_V & \leftarrow & E_G \\
T_V & \leftarrow & T_G \\
s_{0_V} & \leftarrow & s_{0_G} \\
PASS & \leftarrow & F_G \\
INC & \leftarrow & \emptyset \\
FAIL & \leftarrow & \emptyset \\
F_V & = & PASS \cup INC \cup FAIL
\end{array}$$

The $\leftarrow$ symbol is used to represent attribution. For instance,

$$S_V \leftarrow S_G$$

shall be read as: the set of states of V, S_V, becomes equal to the set of states of G, S_G.

2. add two new states s_i and s_f to V and consider them final states. Classify s_i as *inc* and s_f as *fail*, that is,

$$\begin{array}{lcl} S_V & \leftarrow & S_V \cup \{s_i, s_f\} \\ INC & \leftarrow & INC \cup \{s_i\} \\ FAIL & \leftarrow & FAIL \cup \{s_f\} \end{array}$$

3. for each non-final state $s \in S_V - F_V$ do

   (a) define the set M which contains all the events labelling the state s outgoing transitions, that is,

   $$M = \{b \mid s \xrightarrow{b}\}$$

   (b) for each $e \in E_V$ do

      i. if $(e \notin M \ \wedge \ \sigma.e \in \mathit{Accept}(A_S))$, where σ is a trace of the automaton V from its initial state s_{0_V} to state s, $s_{0_V} \overset{\sigma}{\Longrightarrow} s$, then add a new transition $s \xrightarrow{e} s_i$ to T_V,

      $$T_V \leftarrow T_V \cup \{s \xrightarrow{e} s_i\}$$

      ii. if $(e \notin M \ \wedge \ \sigma.e \notin \mathit{Accept}(A_S))$ then add a new transition $s \xrightarrow{e} s_f$ to T_V,

      $$T_V \leftarrow T_V \cup \{s \xrightarrow{e} s_f\}$$

4. for each non-final state $s \in S_V - F_V$ add a new transition $s \xrightarrow{\phi} s_i$ to T_V,

   $$E_V \leftarrow E_V \cup \{\phi\},\ T_V \leftarrow T_V \cup \{s \xrightarrow{\phi} s_i\}.$$

5. for each final state of $s \in F_V$
   for each event $e \in E_V$ add a self transition to s so that

   $$T_V \leftarrow T_V \cup \{s \xrightarrow{e} s\}.$$

The $\mathit{verd}()$ function, where $\sigma \in E_S^*$ $(\phi \notin E_S)$, can now be defined as follows:

$$\mathit{verd}(\sigma.\phi) = \left| \begin{array}{ll} \mathit{pass}, & \text{if } (s_{0_V}\ \mathbf{after_V}\ \sigma.\phi) \subseteq PASS \\ \mathit{inc}, & \text{if } (s_{0_V}\ \mathbf{after_V}\ \sigma.\phi) \subseteq INC \\ \mathit{fail}, & \text{if } (s_{0_V}\ \mathbf{after_V}\ \sigma.\phi) \subseteq FAIL \end{array} \right.$$

Fig. 4.8 exemplifies the construction of the verdict function for the guarantee automaton represented in Fig. 4.7 and the component $A \parallel B$ whose safety automaton is presented in Fig. 4.4. Fig. 4.9 provides a more complete example.

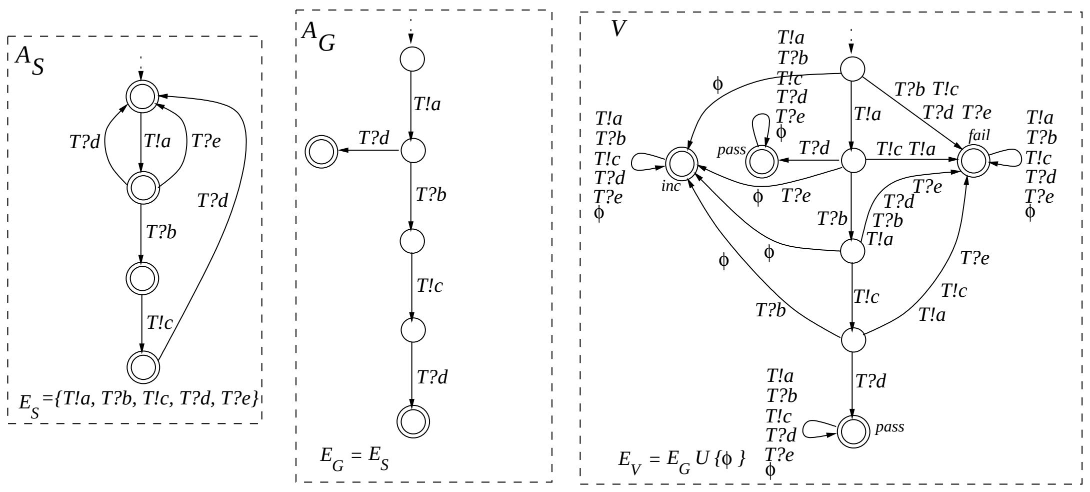


Figure 4.9: Verdict evaluation function

Test

In order to let the *verd*() function, which is represented by the automaton V, decide about the satisfaction of a guarantee (and safety) property by an implementation, a test should be defined so that it can drive the implementation conveniently. While the *passive verd*() function is used to analyse the resulting log, the test, which will be represented by an automaton T, is expected to *actively* communicate with the implementation. Furthermore, the test is also expected to inform the verdict and other trace evaluation functions about the end of a test execution. For that, a new event ϕ (end of test event) will be introduced which will be generated by the test T and interpreted by the trace evaluation functions, so that they can finish the trace (log) evaluation and decide about a verdict for the trace.

Assuming that the environment represented by T can send and receive messages, the test should, in every moment, send the appropriate message and, after that, wait for the implementation reaction. A test T can be represented by a tree, being a tree an automaton where:

- each state can have at most one *incoming* transition;
- all the *outgoing transitions* are labelled either with test controlled or test observed events. In the first case, the tree branching (number of transitions outgoing from a state) has to be exactly one, that is, the test is always deterministic with respect to the events it controls.

In order to make the test short, it must stop as soon as a verdict can be evaluated by *verd*() from the resulting trace. A test T can be derived from A_G according to the following rules (see also Fig. 4.10):

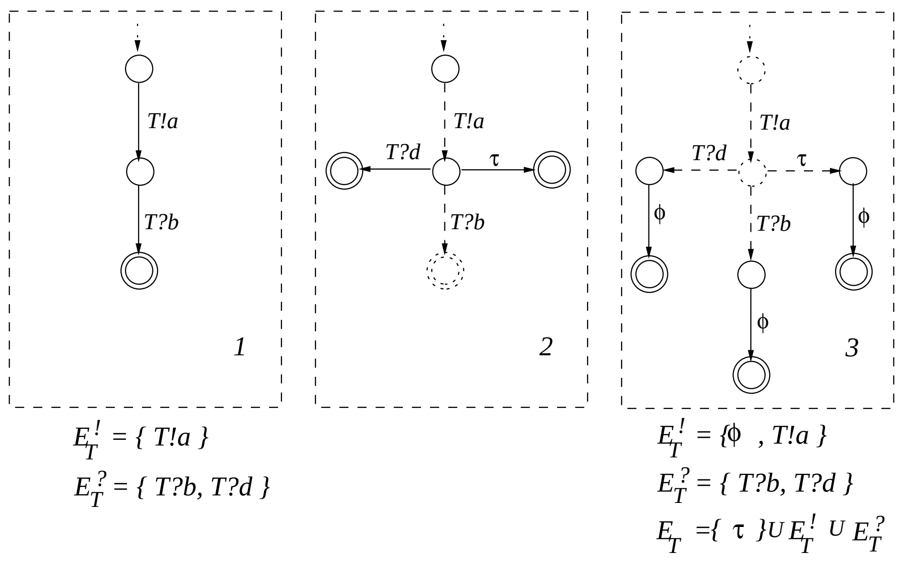


Figure 4.10: Test derivation for $A \parallel B$

1. transform the automaton A_G into a tree, T, capable of defining the same traces,

$$Traces(T) = Traces(A_G) \ \wedge \ Accept(T) = Accept(A_G)$$

   This transformation is usually simple and possible since normal use cases do not contain loops. Let us assume, as described above, that the branching for test controlled events is at most one.

   $T = (S_T, E_T, T_T, s_{0_T}, F_T)$
   $E_T = E_G \cup \{\tau\} = E_T^! \cup E_T^? \cup \{\tau\}$
   $E_T^!$ contains the tester *controlled* events
   $E_T^?$ contains the tester *observed* events
   $E_T^! \cap E_T^? = \emptyset$
   τ is an unobservable event

2. define an auxiliary set A containing all the non-final states of T, that is, $A = S_T - F_T$.
   For each state $s \in A$ do

   (a) define the set M which contains all the events labeling the state s outgoing transitions, that is,

$$M = \{b \mid s \xrightarrow{b}\}$$

(b) if $M \cap E_T^! = \emptyset$ (no test controlled transitions) then for each $e \in E_T^? \cup \{\tau\}$, where τ is an internal and unobservable event which occurs when the implementation does not answer, do if $e \notin M$ then

i. add a new final state s'' to T

$$S_T \leftarrow S_T \cup \{s''\},\ F_T \leftarrow F_T \cup \{s''\}$$

ii. add a new transition $s \xrightarrow{e} s''$ to T_T

$$T_T \leftarrow T_T \cup \{s \xrightarrow{e} s''\}$$

3. define the auxiliary set B which is equal to F_T, that is, $B \leftarrow F_T$. For each final state $s \in B$ do

(a) add a new final state s' to F_T,

$$S_T \leftarrow S_T \cup \{s'\},\ F_T \leftarrow F_T \cup \{s'\}$$

(b) add a new transition $s \xrightarrow{\phi} s'$ to T_T, where ϕ is a test controlled event representing the end of the trace

$$E_T^! \leftarrow E_T^! \cup \{\phi\},\ T_T \leftarrow T_T \cup \{s \xrightarrow{\phi} s'\}$$

(c) remove the state s from the final state set F_T

$$F_T \leftarrow F_T - \{s\}.$$

Fig. 4.10 exemplifies the test derivation process for the guarantee automaton represented in Fig. 4.7 and the component $A \parallel B$ whose safety automaton is presented in Fig. 4.4. Fig. 4.11 provides a more complete example, which is based on the safety and guarantee automata of Fig. 4.9.

Common representation

Common representations of tests do not usually use two separate models, V and T, as described above. Instead, the test tree and the trace evaluation function are represented in only one tree whose final states are labelled with verdicts. $TC = T \times V$, that is, the composition of the two models, is defined as

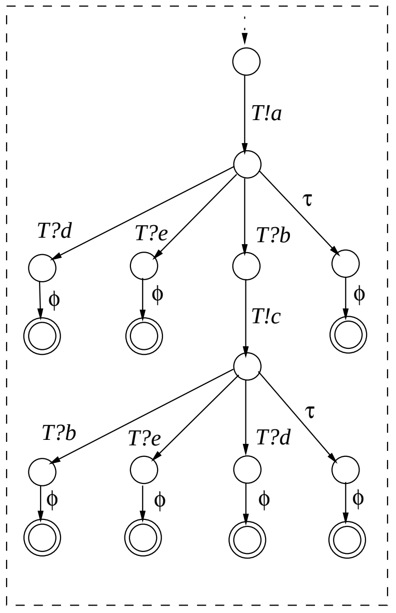


Figure 4.11: Test derivation example

$$
\begin{array}{l}
TC = (S_{TC}, E_{TC}, T_{TC}, s_{0_{TC}}, F_{TC}) \\
T = (S_T, E_T, T_T, s_{0_T}, F_T) \\
V = (S_V, E_V, T_V, s_{0_V}, F_V) \\
S_{TC} = S_T \times S_V \\
E_{TC} = E_T \text{ , where } E_T = E_V \cup \{\tau\} \\
e \in E_V :\ s'_T \xrightarrow{e} s''_T \times s'_V \xrightarrow{e} s''_V \ = \ (s'_T, s'_V) \xrightarrow{e} (s''_T, s''_V) \\
e = \tau :\ s'_T \xrightarrow{\tau} s''_T \ = \ (s'_T, s'_V) \xrightarrow{\tau} (s''_T, s'_V) \\
s_{0_{TC}} = (s_{0_T}, s_{0_V}) \\
F_{TC} = F_T \times F_V
\end{array}
$$

Examples of common test representation are given in Fig. 4.12 (from Fig. 4.8 and Fig. 4.10) and Fig. 4.13 (from Fig. 4.9 and Fig. 4.11), where the test T final states are now labelled with a verdict.

4.4 Incomplete component behaviour testing

In this section, the test derivation method presented is analysed for the other two type of components - the interface and requirement components.

4.4.1 Interface component

Interface components are components for which, by structural reasons, the interface concept is important, being an interface a communication/ inter-

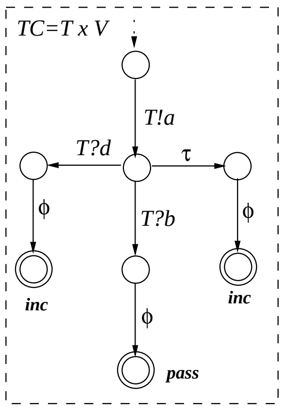


Figure 4.12: Common test representation (I)

action point between the component and its environment. Besides that, interface components are assumed to have specifications which:

1. describe the user-component interactions through every communication interface;

2. provide no clear mappings between the events observed at different interfaces.

If those mappings were known, the component would be said to be completely specified (complete component), since all the behaviours could be foreseen. Similarly, if the component has only one interface, the component is also a complete component.

A large number of network components can be classified as interface components. When describing them, the specifier (1) identifies the interfaces, (2) characterises the user-component interactions for each interface using, for instance, state machines, (3) provides a textual description of the component functions and (4) illustrates the textual description with some component use cases. Similarly to complete components, a test TC is assumed to be generated for a use case, where each use case can be represented by a guarantee automaton, A_G. Then, how is it possible to derive a test for the interface component in these weak specifications?

Safety automaton

Let us assume that:

- an interface component has more than one interface;

- each interface can be described by a process represented by an automaton;

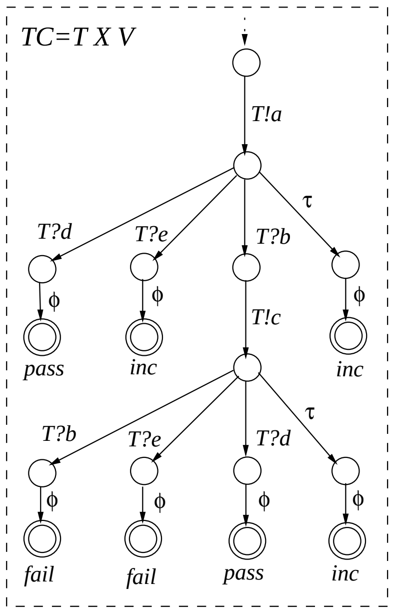


Figure 4.13: Common test representation (II)

- the interface processes do not describe communications between interfaces.

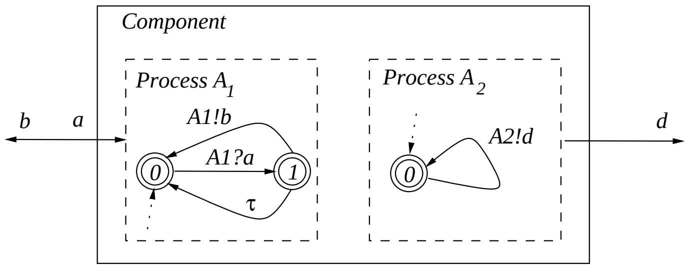


Figure 4.14: Interface component

Fig 4.14 shows a component in this situation. A component safety automaton A^I_S can be obtained when the following steps are executed:

1. assume a maximum behaviour environment (a process T_i) for each component interface i;

2. obtain a safety automaton $A^I_{S_i}$ for each interface i, by combining T_i with the process representing the interface i, A_i, that is, $T_i \parallel A_i$, and by relabelling all the events $e \in E_{A_i}$ with τ;

3. obtain the final interface safety automaton, A^I_S, by composing all the interface safety automaton $A^I_{S_i}$,

$$A^I_S = \prod_i A^I_{S_i}$$

Fig 4.15 exemplifies the process described above. Synchronous communication was assumed between T_i and A_i and some transitions labelled with τ were removed from $A^I_{S_i}$. If the complete component specification A^C_S was known, it could be compared with the interface safety automaton A^I_S by

$$Traces(A^C_S) \subseteq Traces(A^I_S)$$

that is, the interface safety automaton obtained, A^I_S, describes more behaviours than those possibly allowed by the unknown complete component A^C_S.

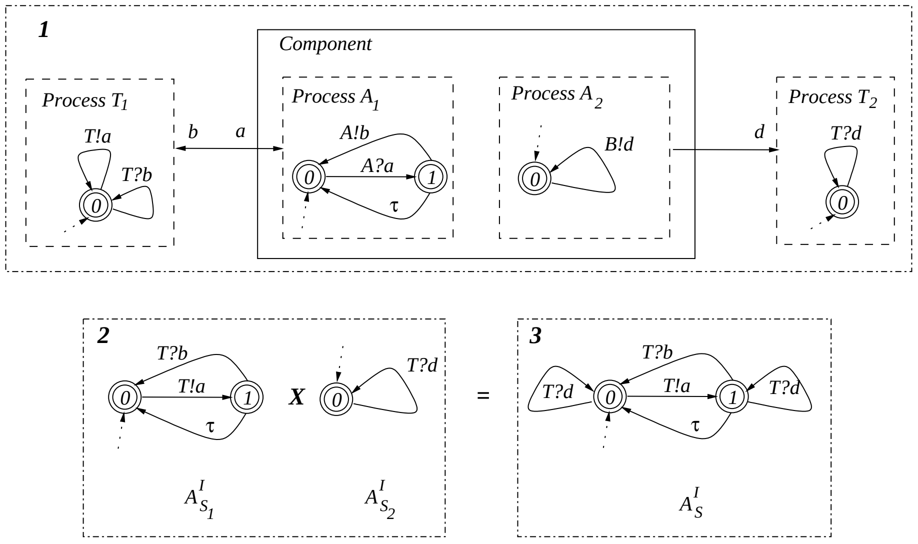


Figure 4.15: Interface component safety automaton

Test derivation

The safety automaton A^I_S is known. Let us also assume that each use case is specified by a guarantee automata A_G, which satisfies also the two conditions presented for the complete specifications:

1. $E_G = E^C_S$. The event set recognised by the guarantee automaton E_G is equal to E^C_S, which is also assumed to be equal to E^I_S. It means

that the guarantee automaton reacts to all the component environment observable/ controllable events;

2. *complete.* The traces accepted by A_G, $Accept(A_G)$, are also traces of A_S^C and capable of driving A_S^C from its initial state s_{0_S} back to its initial state s_{0_S}, that is, they describe one or more complete loops in the unknown safety automaton.

$$\forall \sigma \in Accept(A_G),\ s_{0_S^C}\ \mathbf{after_S^C}\ \sigma\ =\ s_{0_S^C}$$

A use case is an interface automaton which, in fact, describes one or more traces whose events may be visible at different component interfaces.

If the test derivation method presented for complete components is used in this case, the test case TC obtained from A_S^I and A_G would have verdicts which had to be reinterpreted as follows:

- **pass.** In this case, the trace σ observed by the joint execution of the test and the component is said to be accepted by A_G, that is,

  $$\sigma \in Accept(A_G)$$

  Since A_G was assumed to be complete, that is,

  $$Accept(A_G) \subseteq Traces(A_S^C)$$

  σ is a valid specification trace. A *pass* for an interface component has, for this reason, the same meaning as for complete components.

- **fail.** In this case, the trace observed σ is not accepted by A_S^I, that is,

  $$\sigma \notin Accept(A_S^I)$$

  Since $Accept(A_S^C) \subseteq Accept(A_S^I)$, it implies that

  $$\sigma \notin Accept(A_S^C)$$

  A *fail* obtained for interface components has the same meaning as for complete components.

- **inc.** In this case, the trace observed σ is not accepted by A_G,

  $$\sigma \notin Accept(A_G)$$

  and the first event deviating from the guarantee automaton accepted traces is said not to violate A_S^I. Let assume that

  $$\sigma \in Accept(A_S^I)$$

  Two situations should be considered:

1. *σ is a prefix of some $\sigma' \in Accept(A_G)$.* In this case, since σ' belongs also to A_S^C, the prefix σ belongs also to A_S^C, that is

$$\sigma \in Accept(A_S^C)$$

   The *inc* verdict has the same meaning as for complete components.

2. *σ is not a prefix of some $\sigma' \in Accept(A_G)$.* Since $Accept(A_S^C) \subseteq Accept(A_S^I)$ there is no means of deciding if σ is also accepted by A_S^C. An *inc* verdict could be, in fact, an *inc* or *fail* verdict if the complete component was known.

4.4.2 Requirement component

Requirement components are those for which only a set of use cases are known. Test implementations of these components consist in verifying only if they can execute all the use cases, which is the traditional way of testing programs. Let us analyse the test derivation method for this type of component. A test TC is assumed to be derived for each use case.

Safety automaton

A safety automaton for a requirement component has to be an automaton which describes all possible behaviours. Let us denote this automaton A_S^R and assume that $E_S^R = E_S^C$, that is, at least their events are known. In this case

$$Accept(A_S^R) = E_S^{C*}$$

so every event combination is assumed to be a valid trace and

$$Accept(A_S^C) \subseteq Accept(A_S^R)$$

Test derivation

When each use case is defined by a guarantee automaton A_G, which satisfies also the $E_G = E_S^C$ and the *complete* conditions described above, the test case TC obtained from A_S^R and A_G would give verdicts which would had to be interpreted as follows:

- **pass.** The trace σ observed by the joint execution of the test and the component is accepted by A_G, that is

$$\sigma \in Accept(A_G)$$

  Since all the traces accepted by A_G are also traces of A_S^C, i.e.,

$$Accept(A_G) \subseteq Traces(A_S^C)$$

σ is said to be also a valid specification trace. A *pass* verdict in a requirement component has, for this reason, the same meaning as for complete components.

- **fail.** In this case the trace observed σ is not accepted by A_S^R. However, since $Accept(A_S^R) = E_S^{C*}$, all the traces σ are said to be accepted by the component safety automaton. A *fail* verdict will never be obtained.
- **inc.** In this case the trace observed σ is not accepted by A_G,

$$\sigma \notin Accept(A_G) \ \wedge \ \sigma \in Accept(A_S^R)$$

Again, two situations have to be considered:

1. σ is a prefix of some $\sigma' \in Accept(A_G)$. In this case, since σ' belongs also to A_S^C, its prefix σ is said to belong also to A_S^C, that is

$$\sigma \in Accept(A_S^C)$$

Therefore, the *inc* verdict has the same meaning as for complete components.

2. σ is not a prefix of some $\sigma' \in Accept(A_G)$. Since $Accept(A_S^C) \subseteq Accept(A_S^R)$ there is no means of deciding if σ is also accepted by A_S^C. An *inc* verdict can be, if the complete component specification was known, an *inc* or a *fail* verdict.

4.5 Component cyclic testing

Component cyclic testing aims at the identification of faults which typically appear when the system is used for a long period of time. These faults can be located at the component level or at the (shared) resources it uses.

In the first case, faults can be caused by some internal saturation mechanism, such as the use of a variable whose type defines a range smaller than the specified. For instance, the implementation of a *short int* variable instead of the *long* variable specified can cause component observable faulty behaviours. The detection of the problem, in traditional black box testing, may imply that the component is used for a long time.

In the second case, the faults can be of any type. Being located in a component resource which can be also used by other components, these faults can induce faulty behaviours at the component under test; they can be random and their first appearance can take some time.

Although the correct testing procedure would be to test the resources first, (1) theory shows that it is impossible to prove that a resource/ component is free of faults and (2) practice shows that there is neither time nor money to test all the NE components.

A simple solution for detecting this type of faults would be to repeatedly execute a test for a long period of time. Let us assume that a test has the same meaning as described before - a test case $TC = T \times V$ which was obtained from a use case specified by A_G and a specification described by A_S. The usual process of identifying a fault during a test campaign is (1) to run a test until it stops and (2) to inspect the verdict obtained. If the verdict is *inc* or *fail*, then the trace (log) σ is inspected so that the causes originating the deviating trace can be characterised.

Typical behaviour tests (TC) can last from some seconds to a few minutes. On the other hand, cyclic tests are usually executed overnight, when both development and test resources become available and this implies that hundreds or thousands of test results (verdicts and traces) can be obtained. Since, during the first prototype tests, inconclusive/faulty traces can be as high as 30 % of the total traces obtained, a large number of problematic traces are available for analysis. Two problems can then be identified:

- How to repeat the tests without human intervention?
- How to analyse the information obtained from an overnight behaviour test, so that faults can be eliminated, taking into account the project management requirements (most visible faults eliminated first)?

4.5.1 Test repetition

According to the test derivation algorithm presented, a test T is derived from the use case specification A_G. In this case, $E_G = E_S$ and the guarantee automaton is required to define complete loops over the safety automaton, starting and terminating at the component initial state.

When, after a test execution, the verdict obtained is *pass*, the implementation (according to the component specification) is supposed to be again in its initial state s_{0_S} and this implies that the same or another test can be run.

However, if the joint execution of the test with the implementation leads to another verdict, the problem becomes more complex. In the case of an *inc* verdict, the implementation is left in a state which may not be s_{0_S}. In case of *fail*, the implementation is left in a state not previewed in the specification, that is:

$$\begin{array}{lcl} verd_T(\sigma) = pass & \Rightarrow & (s_{0_S}\ \mathbf{after}_{\mathbf{A_S}}\ \sigma) = \{s_{0_S}\} \\ verd_T(\sigma) = inc & \Rightarrow & (s_{0_S}\ \mathbf{after}_{\mathbf{A_S}}\ \sigma) \subseteq S_S \\ verd_T(\sigma) = fail & \Rightarrow & (s_{0_S}\ \mathbf{after}_{\mathbf{A_S}}\ \sigma) \cap S_S = \emptyset \end{array}$$

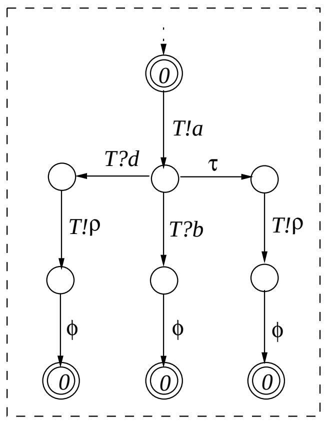


Figure 4.16: Cyclic test (I)

In order to let the test to be repeated, some sequence of events should be added at the end of each trace $\sigma \notin \mathit{Accept}(A_G)$ to force the component back to its initial state. Let us call this sequence a *reset sequence* which, for simplicity, will be represented by one event controlled by the test

$$T!\rho \in E_S$$

Then

$$\forall s \in S_S,\ s \xrightarrow{T!\rho} s_{0_S}$$

Faulty implementations are assumed to correctly implement the reliable reset.

The test derivation algorithm presented should be reformulated as follows (see also Fig. 4.16):

1. transform the automaton A_G into a tree T capable of defining the same traces,

   $$Traces(T) = Traces(A_G) \ \wedge\ \mathit{Accept}(T) = \mathit{Accept}(A_G)$$

   This transformation is usually possible since normal use cases do not contain loops. Let us assume, as described above, that the branching for test controlled events is at most one.

   $T = (S_T, E_T, T_T, s_{0_T}, F_T)$
   $E_T = E_G \cup \{\tau\} = E_T^! \cup E_T^? \cup \{\tau\}$
   $E_T^!$ contains the tester *controlled* events
   $E_T^?$ contains the tester *observed* events
   $E_T^! \cap E_T^? = \emptyset$
   τ is an unobservable event

2. define an auxiliary set A containing all the tree T non-final states, that is, $A = S_T - F_T$.
   For each state $s \in A$ do
   (a) define the set M which contains all the events labelling the state s outgoing transitions, that is,
   $$M = \{b \mid s \xrightarrow{b}\}$$
   (b) if $E_T^! \cap M = \emptyset$ then
   for each $e \in E_T^? \cup \{\tau\}$ do
      i. if $e \notin M$ then
         A. add a new state s' to T
         $$S_T \leftarrow S_T \cup \{s'\}$$
         B. add a new transition $s \xrightarrow{e} s'$ to T_T
         $$T_T \leftarrow T_T \cup \{s \xrightarrow{e} s'\}$$
         C. add a new final state s'' to F_T,
         $$S_T \leftarrow S_T \cup \{s''\},\ F_T \leftarrow F_T \cup \{s''\}$$
         D. add a new transition $s' \xrightarrow{T!\rho} s''$ to T_T
         $$T_T \leftarrow T_T \cup \{s' \xrightarrow{T!\rho} s''\}$$
3. define the auxiliary set B which is equal to F_T, that is, $B \leftarrow F_T$.
   For each final state $s \in B$ do
   (a) add a new transition $s \xrightarrow{\phi} s_{0_T}$ to T_T so that
   $$E_T^! \leftarrow E_T^! \cup \{\phi\},\ T_T \leftarrow T_T \cup \{s \xrightarrow{\phi} s_{0_T}\}$$
   (b) remove the state s from the final state set F_T
   $$F_T \leftarrow F_T - \{s\}$$
4. add the initial state s_{0_T} to the final state set
   $$F_T \leftarrow F_T \cup \{s_{0_T}\}$$

Fig. 4.16 shows the refinement of the test presented in Fig. 4.10 so that the test can be executed cyclicly. Fig. 4.17 shows the same refinement for the test of Fig. 4.11.

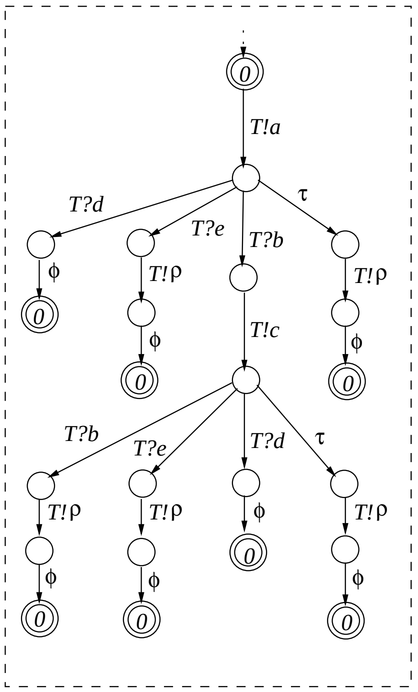


Figure 4.17: Cyclic test (II)

4.5.2 Trace evaluation functions

Assume that the cyclic test T is left executing with a component implementation overnight. When, in the morning, the test is stopped, the person in charge of identifying the faults is provided with, perhaps, some thousands of traces which can have verdicts associated. How to proceed then?

The obvious procedure would be to inspect the log obtained, find the first *inconclusive* or *fail* trace, which could have been obtained after some working hours, and characterise the fault so that it could be eliminated. Project management, however, demands that the component most visible faults are eliminated first, so that interoperability and, sometimes, operator acceptance tests can proceed. In order to support these requirements a classification and quantification of the faults seems appropriated.

Since the test T is known and has control over the component implementation, the set of traces which can be obtained can be known in advance, that is, $Accept(T)$. For the test of Fig. 4.16, that is a refinement of the test presented in Fig. 4.10 which was derived based on the guarantee automaton of Fig. 4.7 and the safety automaton of Fig. 4.4, the following traces can be obtained:

$$\begin{array}{lll} pass & \longrightarrow & \{\ T!a.T?b.\phi\} \\ inc & \longrightarrow & \{\ T!a.T?d.T!\rho.\phi,\ T!a.T!\rho.\phi\ \} \end{array}$$

The test of Fig. 4.17, that is a refinement of the test presented in Fig. 4.11 which was derived based on the guarantee and the safety automata of Fig. 4.9, can generate the following traces:

$$\begin{array}{lll} pass & \longrightarrow & \{\ T!a.T?d.\phi,\ T!a.T?b.T!c.T?d.\phi\ \} \\ inc & \longrightarrow & \{\ T!a.T?e.T!\rho.\phi,\ T!a.T!\rho.\phi,\ T!a.T?b.T!c.T!\rho.\phi\ \} \\ fail & \longrightarrow & \{\ T!a.T?b.T!c.T?b.T!\rho.\phi,\ T!a.T?b.T!c.T?e.T!\rho.\phi\ \} \end{array}$$

where the standard $verd_T()$ function was used to classify the traces and the verdicts have the usual meaning.

The results of the overnight test could, in this way, be classified by the probability of each verdict being obtained or, in alternative, the probability of each type of trace being observed. The first method would be very poor since all the faults observed would be classified in two groups - *fail* and *inc*. The second method, although much more useful, would be, in certain conditions, in the opposite situation since, if both $\sharp E_G$ and $\sharp S_G$ are large, many types of traces could be available. The (faulty) traces of an overnight test could then be distributed in such a way that it would be difficult to select the most representative faults. Experience shows that in first cyclic test campaigns a large number of different faulty traces can be found.

An intermediate classification method would be to evaluate the implementation traces obtained according to the relevance the faults have for users. Suppose a telephony system modeled as a component. A complete use case would consist in (1) establishing a call, (2) transmitting voice tones and (3) terminating the call. Independently of the success of the establishment and the voice transmission phases, the call should always be correctly terminated. A trace function which can evaluate the probability of a phone call being incorrectly terminated seems very important for users, since an error in this phase can bring about serious bad effects. Another trace function could, for instance, evaluate how often a *dial tone* is observed after an *offhook* action, independently of the order in which it has been observed (e.g. before or after a number being dialed).

At the end, we are talking about a simple event sequence filter which will be called the evaluation function $eval_T()$. Like the function $verd_T()$, the function $eval_T()$ can be described by an automaton P, with the following characteristics:

1. $E_P \subseteq E_G \cup \{\phi\}$, that is, it can be blind for some trace events but must observe the end of trace event ϕ generated by the test T;

2. all the sequences σ of visible events terminating with the end of trace event ϕ ($\sigma = e_1.e_2 \ldots e_n.\phi,\ e_i \in E_P - \{\phi\}$) should be accepted by the automaton P, i.e., $\sigma \in Accept(P)$;

3. it is deterministic, that is, whatever the trace $\sigma = e_1.e_2 \ldots e_n.\phi$, only one final state of P must be reached.

Since multiple samples of the component implementation will be analysed (the overnight traces), the automaton P can be represented by a random variable for which the probability $P(P = s_i)$ of its final state being reached and the correspondant probability mass function $p(s_i)$ can be estimated and depend on particular component implementations.

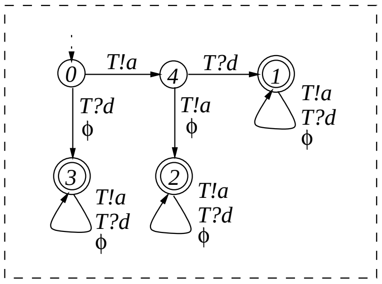


Figure 4.18: Trace evaluation function

Fig. 4.18 shows a possible evaluation function for the test of Fig. 4.17.

The automaton of Fig. 4.18 only recognises events, $T!a$, $T?d$ and ϕ, that is,

$$E_P = \{\ T!a,\ T?d,\ \phi\ \}$$

Non-visible events are deleted from traces.

$T!a.T?d.\phi$	$\longrightarrow$	*pass*	$\longrightarrow$	$T!a.T?d.\phi$	$\longrightarrow$	s_1
$T!a.T?b.T!c.T?e.T?d.\phi$	$\longrightarrow$	*pass*	$\longrightarrow$	$T!a.-.-.-.T?d.\phi$	$\longrightarrow$	s_1
$T!a.T?e.T!\rho.\phi$	$\longrightarrow$	*inc*	$\longrightarrow$	$T!a.-.-.\phi$	$\longrightarrow$	s_2
$T!a.T!\rho.\phi$	$\longrightarrow$	*inc*	$\longrightarrow$	$T!a.-.\phi$	$\longrightarrow$	s_2
$T!a.T?b.T!c.T!\rho.\phi$	$\longrightarrow$	*inc*	$\longrightarrow$	$T!a.-.-.-.\phi$	$\longrightarrow$	s_2
$T!a.T?b.T!c.T?b.T!\rho.\phi$	$\longrightarrow$	*fail*	$\longrightarrow$	$T!a.-.-.-.-.\phi$	$\longrightarrow$	s_2
$T!a.T?b.T!c.T?e.T!\rho.\phi$	$\longrightarrow$	*fail*	$\longrightarrow$	$T!a.-.-.-.-.\phi$	$\longrightarrow$	s_2

According to Fig. 4.18, the traces of Fig. 4.17 would be classified as follows:

$$\begin{array}{lll} s_1 & \longrightarrow & \{\ T!a.T?d.\phi,\ T!a.T?b.T!c.T?e.T?d.\phi\ \} \\ s_2 & \longrightarrow & \{\ T!a.T?e.T!\rho.\phi,\ T!a.T!\rho.\phi,\ T!a.T?b.T!c.T!\rho.\phi, \\ & & T!a.T?b.T!c.T?b.T!\rho.\phi,\ T!a.T?b.T!c.T?e.T!\rho.\phi\ \} \end{array}$$

Let us assume that the overnight test gives as a result the trace ϵ

$$\epsilon = \sigma_1.\sigma_2.\sigma_1.\sigma_3.\sigma_1.\sigma_2.\sigma_3.\sigma_1.\sigma_1.\sigma_1$$

where

$$\begin{array}{rcll} \sigma_1 & = & T!a.T?b.T!c.T?e.T?d.\phi, & \text{is accepted by } s_1 \\ \sigma_2 & = & T!a.T!\rho.\phi, & \text{is accepted by } s_2 \\ \sigma_3 & = & T!a.T?b.T!c.T?b.T!\rho.\phi, & \text{is accepted by } s_2 \end{array}$$

The discrete random variable P characterises the implementation behaviour of the faulty component with respect to the trace evaluation function $eval_T()$. In this case, its probability mass function could be defined as follows

$$p(s) \;=\; \begin{array}{|ll} 0.6, & s = s_1 \\ 0.4, & s = s_2 \\ 0, & s = s_3 \end{array}$$

The component implementation is then specified by a behaviour random variable. The ideal component, one for which all the traces would get a pass verdict, would be represented by the probability mass function $p'()$

$$p'(s) \;=\; \begin{array}{|ll} 1, & s = s_1 \\ 0, & s = s_2 \\ 0, & s = s_3 \end{array}$$

The indication that $p(s_2) \neq 0$ (or $p(s_1) \neq 1$) gives a quantified indication of the type of problems associated with the implementation tested.

4.6 Timed testing

Testing telecommunication equipments with respect to time properties is usually required since there are timeouts which have to be evaluated and, on the other hand, many of the QoS requirements are related to time (e.g. delays). Currently, specifications refer to time vaguely. Neither the formal languages used in telecommunications actually support the precise definition of time nor the state machines used, for instance, by IETF use it. The time characteristics are almost always expressed informally by means of textual descriptions. However, real conformance test suites or operator acceptance tests depend heavily on system time characteristics.

This thesis proposes a hybrid approach for reasoning about timed tests which assumes that:

- *NE specifications are untimed.* It means that the normal safety automaton A_S continues to be developed as presented before;

- *use cases can be timed.* That is, certain use cases may assume the existence of time constraints. The safety automaton A_{G_t} is then represented by the Alur-Dill timed automaton introduced in Chap. 3, where clocks, clock reset and clock constraints are added to the untimed guarantee automaton.

4.6.1 Timed guarantee automaton

Like the untimed version, the timed guarantee automaton A_{G_t} contains the following characteristics:

1. $E_{G_t} = E_S$. The event set recognised by the guarantee automaton, E_{G_t}, is equal to E_S, the event set of the corresponding safety automaton. It means that a guarantee automaton will react to all the events observable/ controllable by the component environment;

2. *complete.* The traces accepted by A_{G_t}, $timAccept(A_{G_t})$, are also traces of A_S and should be capable of driving A_S from its initial state s_{0_S} back to its initial state s_{0_S}, that is, they describe one or more complete loops over the safety automaton. Note that A_S, by not specifying time aspects, does not impose constraints on the timed observation and control of the events. Only the concepts of *after* and *before* concern the untimed model. Let us introduce the function

$$\sigma = Untime(\sigma_t)$$

which transforms the timed trace

$$\sigma_t = (e_1, t_1).(e_2, t_2) \dots (e_n, t_n)$$

into the untimed trace

$$\sigma = e_1.e_2 \dots e_n$$

Let us assume, for simplicity, that $Untime()$ works also for sets of traces, transforming a set of timed traces into a set of equivalent untimed traces. Two timed traces can be transformed into a single untimed trace. An untimed trace can be mapped into an infinite number of timed traces. Mathematically, the completeness assumption introduced above can be described as

$$\begin{gathered} Untime(timAccept(A_{G_t})) \subseteq Accept(A_S) \\ \wedge \\ \forall \sigma \in Untime(timAccept(A_{G_t})),\ (s_{0_S}\ \mathbf{after}_{\mathbf{A_S}}\ \sigma) = \{s_{0_S}\} \end{gathered}$$

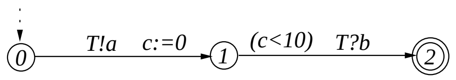


Figure 4.19: Timed guarantee automaton

The timed guarantee automaton can be defined as

$$A_{G_t} = (S_{G_t}, E_{G_t}, TT_{G_t}, s_{0_{G_t}}, F_{G_t}, C_{G_t})$$

where C_{G_t} defines a set of clocks and TT_{G_t} a set of timed transitions. Fig 4.19 introduces graphically a timed version of the guarantee automaton of Fig. 4.7, where

$$\begin{array}{l}
A_{G_t} = (S_{G_t}, E_{G_t}, TT_{G_t}, s_{0_{G_t}}, F_{G_t}, C_{G_t}) \\
S_{G_t} = \{s_0, s_1, s_2\} \\
E_{G_t} = \{T!a, T?b, T?d\} \\
s_{0_{G_t}} = s_0 \\
F_{G_t} = \{s_2\} \\
C_{G_t} = \{c\} \\
TT_{G_t} = \{s_0 \overset{<\ (\ )\ ,T!a,\ (c)\ >}{\longrightarrow} s_1,\ s_1 \overset{<\ (c<10),\ T?b,\ (\ )\ >}{\longrightarrow} s_2\} \\
Untime(\ timAccept(A_{G_t})\ )\ =\ \{T!a.T?b\}
\end{array}$$

For instance, the trace $\sigma_{t_1} = (T!a, 5).(T?b, 8)$ is time-accepted by the A_{G_t} defined in Fig. 4.19, that is, $\sigma_{t_1} \in timAccept(A_{G_t})$ since the time interval between $T!a$ and $T?b$ is 3 time units, which is smaller than the 10 time units specified. On the contrary, the trace $\sigma_{t_2} = (T!a, 5).(T?b, 18)$, for instance, is not time-accepted by A_{G_t} since, in this case, the time interval is 13 time units.

4.6.2 Timed verdict function

The timed trace evaluation function $verd_t()$ should now be able to associate the value *pass*, *inc* or *fail* to every possible timed trace σ'_t so that

$$Untime(\sigma'_t) \in E_S^*$$

The $verd_t()$ can be defined be refining the function $verd()$, as follows:

$$verd_t(\sigma'_t) = \left| \begin{array}{ll} pass, & \text{if } Untime(\sigma'_t) \in Accept(A_S) \wedge \sigma'_t \in timeAccept(A_G) \\ inc, & \text{if } Untime(\sigma'_t) \in Accept(A_S) \wedge \sigma'_t \notin timAccept(A_G) \\ fail, & \text{if } Untime(\sigma'_t) \notin Accept(A_S) \end{array} \right.$$

The function $verd_t(\ )$ can be described as the timed automaton V_t, with the following characteristics:

- *acceptance condition.* All the timed traces σ_t, for which $Untime(\sigma_t) \in E_S^*$, must be accepted by V_t.

- *determinism.* Since each timed trace is required to be uniquely evaluated, only one final state should be reached by V_t when interpreting σ_t.

- *final states.* Every final state of V_t must be classified as *pass*, *inc* or *fail*, that is,
$$F_{V_t} = PASS \cup INC \cup FAIL$$
where $PASS$, INC and $FAIL$ are pairwise disjoint sets.

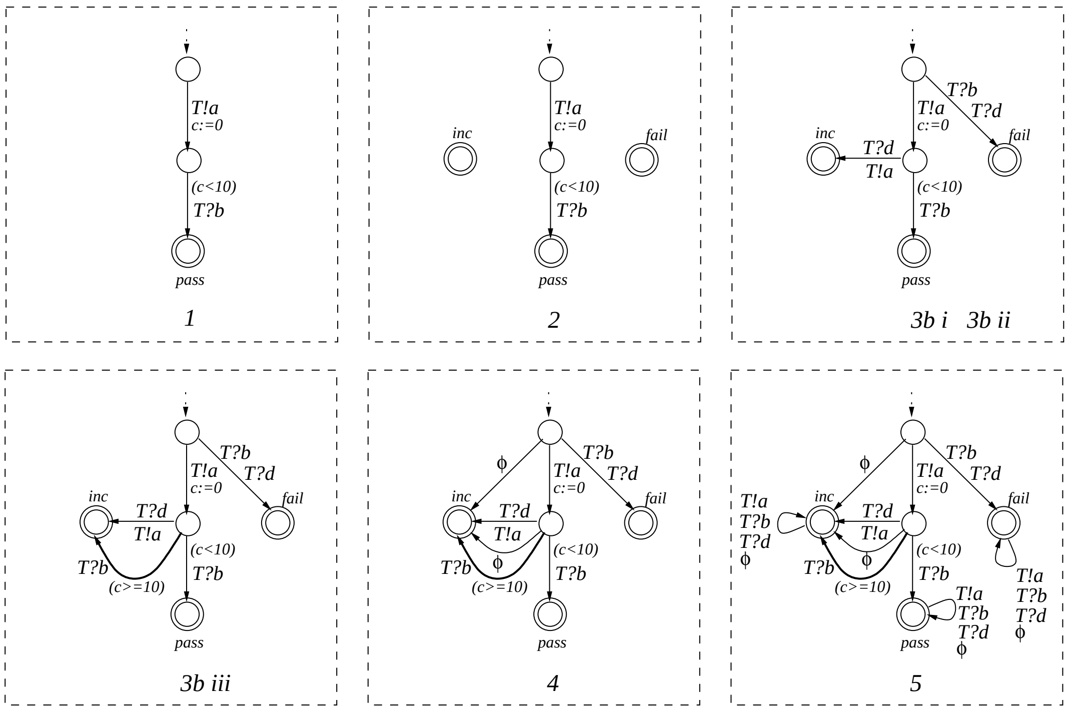


Figure 4.20: Timed verdict automaton

Following the same policy used for obtaining V, the verdict timed automaton V_t can be derivate from A_{G_t} and A_S according to the following steps (see also Fig. 4.20):

1. build an automaton V_t which is structurally equivalent to A_{G_t}, that is, has the same states, clocks and timed transitions as A_{G_t}

$$\begin{array}{lcl}
V_t & = & (S_{V_t}, E_{V_t}, TT_{V_t}, s_{0_{V_t}}, F_{V_t}, C_{V_t}) \\
A_{G_t} & = & (S_{G_t}, E_{G_t}, TT_{G_t}, s_{0_{G_t}}, F_{G_t}, C_{G_t}) \\
S_{V_t} & \leftarrow & S_{G_t} \\
E_{V_t} & \leftarrow & E_{G_t} \\
TT_{V_t} & \leftarrow & TT_{G_t} \\
s_{0_{V_t}} & \leftarrow & s_{0_{G_t}} \\
C_{V_t} & \leftarrow & C_{G_t} \\
PASS & \leftarrow & F_{G_t} \\
INC & \leftarrow & \emptyset \\
FAIL & \leftarrow & \emptyset \\
F_{V_t} & = & PASS \cup INC \cup FAIL
\end{array}$$

In this step each state is required to have only one incoming timed transition, i.e., the automaton V_t should be a tree.

2. add two new states s_i and s_f to V_t and consider them final states. Classify s_i as *inc* and s_f as *fail*, that is,

$$\begin{array}{lcl}
S_{V_t} & \leftarrow & S_{V_t} \cup \{s_i, s_f\} \\
INC & \leftarrow & INC \cup \{s_i\} \\
FAIL & \leftarrow & FAIL \cup \{s_f\}
\end{array}$$

3. for each non-final state $s \in S_{V_t} - F_{V_t}$ do

   (a) define the set M which contains all the events labelling the state s outgoing transitions, that is,

   $$M = \{b \mid s \overset{<(\delta),b,(\lambda)>}{\longrightarrow}\}$$

   (b) for each $e \in E_{V_t}$ do

      i. if ($e \notin M \ \wedge \ Untime(\sigma_t).e \in Accept(A_S)$), where σ_t is a timed-trace of V_t from its initial state $s_{0_{V_t}}$ to state s, $s_{0_{V_t}} \overset{\sigma_t}{\Longrightarrow} s$, then add a new transition $s \overset{<(),e,()>}{\longrightarrow} s_i$ to TT_{V_t},

      $$TT_{V_t} \leftarrow TT_{V_t} \cup \{s \overset{<(),e,()>}{\longrightarrow} s_i\}$$

      ii. if ($e \notin M \ \wedge \ Untime(\sigma_t).e \notin Accept(A_S)$), then add a new transition $s \overset{<(),e,()>}{\longrightarrow} s_f$ to TT_{V_t},

      $$TT_{V_t} \leftarrow TT_{V_t} \cup \{s \overset{<(),e,()>}{\longrightarrow} s_f\}$$

iii. if $e \in M$ then add a new transition $s \overset{<(\neg\delta),e,()>}{\longrightarrow} s_i$ to TT_{V_t},

$$TT_{V_t} \leftarrow TT_{V_t} \cup \{s \overset{<(\neg\delta),e,()>}{\longrightarrow} s_i\}$$

where $\neg\delta$ is the negation of the constraint δ associated with all the state s outgoing transitions labelled with event e. For instance,

$$\delta = c_1 < 10 \wedge 2 < c_2 < 5 \longrightarrow \neg\delta = c_1 \geq 10 \vee (c_2 \leq 2 \vee c_2 \geq 5)$$

4. for each non-final state s of V_t, $s \in S_{V_t} - F_{V_t}$, add a new transition $s \overset{<(),\phi,()>}{\longrightarrow} s_i$ to TT_{V_t},

$$E_{V_t} \leftarrow E_{V_t} \cup \{\phi\},\ TT_{V_t} \leftarrow TT_{V_t} \cup \{s \overset{<(),\phi,()>}{\longrightarrow} s_i\}.$$

5. for each final state of $s \in F_{V_t}$ do
   for each event $e \in E_{V_t}$ add a self transition to s so that

$$TT_{V_t} \leftarrow TT_{V_t} \cup \{s \overset{<(),e,()>}{\longrightarrow} s\}.$$

The $verd_t()$ function, where $Untime(\sigma_t) \in E_S^*$, can now be defined as follows:

$$verd_t(\sigma_t.\phi) = \left| \begin{array}{ll} pass, & \text{if } (s_{0_{V_t}}\ \mathbf{after_{V_t}}\ \sigma_t.\phi) \subseteq PASS \\ inc, & \text{if } (s_{0_{V_t}}\ \mathbf{after_{V_t}}\ \sigma_t.\phi) \subseteq INC \\ fail, & \text{if } (s_{0_{V_t}}\ \mathbf{after_{V_t}}\ \sigma_t.\phi) \subseteq FAIL \end{array} \right.$$

Fig. 4.20 exemplifies the construction of the $verd_t()$ for the guarantee automaton represented in Fig. 4.19 and the safety automaton of Fig. 4.4.

4.6.3 Timed test

Like the untimed version, the timed test T_t will be represented by a timed tree automaton where:

- each state will have at most one *incoming* timed transition;
- every transition is labelled with a test controlled event, test observed event, ϕ or τ. The tester is required to be time-deterministic with respect to its controlled actions.

The test T_t can be derived from A_{G_t} according to the following rules (see also Fig. 4.21):

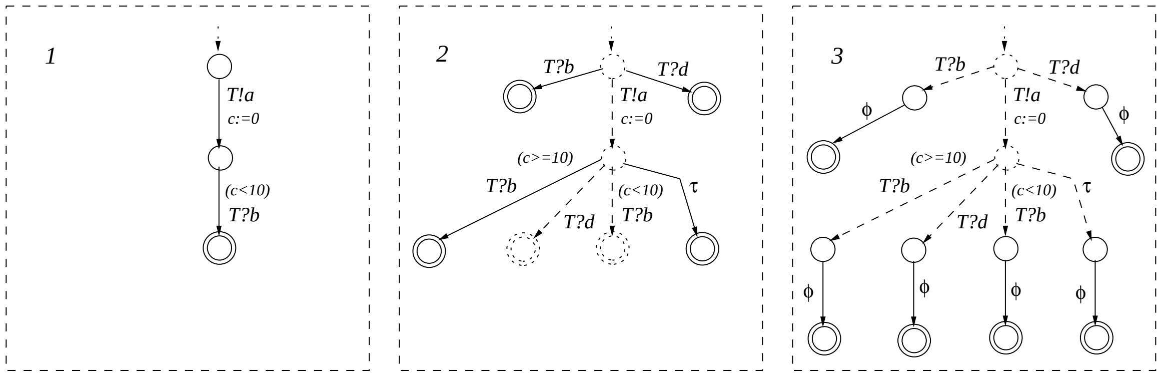


Figure 4.21: Timed test

1. transform the automaton A_{G_t} into a tree, T_t, capable of defining the same timed traces

   $timTraces(T_t) = timTraces(A_{G_t}) \wedge timAccept(T_t) = timAccept(A_{G_t})$

   $T_t = (S_{T_t}, E_{T_t}, TT_{T_t}, s_{0_{T_t}}, F_{T_t}, C_{T_t})$
   $E_{T_t} = E_{G_t} \cup \{\tau\} = E^!_{T_t} \cup E^?_{T_t} \cup \{\tau\}$
   $E^!_{T_t}$ contains the tester *controlled* events
   $E^?_{T_t}$ contains the tester *observed* events
   $E^!_{T_t} \cap E^?_{T_t} = \emptyset$
   τ is an unobservable event

2. define an auxiliary set A containing all non-final states of the tree T_t, that is, $A = S_{T_t} - F_{T_t}$.
   For each state $s \in A$ do

   (a) define the set M which contains all the events labelling the state s outgoing transitions, that is,

   $$M = \{b \mid s \overset{<(\delta),b,(\lambda)>}{\longrightarrow}\}$$

   (b) for each $e \in E^?_{T_t} \cup \{\tau\}$ do

      i. if $e \in M$ then

         A. add a new final state s'' to T_t

         $$S_{T_t} \leftarrow S_{T_t} \cup \{s''\},\ F_{T_t} \leftarrow F_{T_t} \cup \{s''\}$$

         B. add a new transition $s \overset{<(\neg\delta),e,(\lambda)>}{\longrightarrow} s''$ to TT_{T_t}

         $$TT_{T_t} \leftarrow TT_{T_t} \cup \{s \overset{<(\neg\delta),e,(\lambda)>}{\longrightarrow} s''\}$$

         where δ is the constraint associated with the all the state s outgoing transitions which are labelled with event e.

ii. if $e \notin M$ then

A. add a new final state s'' to T_t

$$S_{T_t} \leftarrow S_{T_t} \cup \{s''\},\ F_{T_t} \leftarrow F_{T_t} \cup \{s''\}$$

B. add a new transition $s \overset{<(),e,()>}{\longrightarrow} s''$ to TT_{T_t}

$$TT_{T_t} \leftarrow TT_{T_t} \cup \{s \overset{<(),e,()>}{\longrightarrow} s''\}$$

3. define the auxiliary set B which is equal to F_{T_t}, that is, $B \leftarrow F_{T_t}$. For each final state $s \in B$ do

   (a) add a new final state s' to F_{T_t},

$$S_{T_t} \leftarrow S_{T_t} \cup \{s'\},\ F_{T_t} \leftarrow F_{T_t} \cup \{s'\}$$

   (b) add a new transition $s \overset{<(),\phi,()>}{\longrightarrow} s'$ to T_{T_t}, where ϕ is the test controlled event representing the end of the trace

$$E^!_{T_t} \leftarrow E^!_{T_t} \cup \{\phi\},\ T_{T_t} \leftarrow T_{T_t} \cup \{s \overset{<(),\phi,()>}{\longrightarrow} s'\}$$

   (c) remove the state s from the final state set F_{T_t}

$$F_{T_t} \leftarrow F_{T_t} - \{s\}.$$

Fig. 4.21 exemplifies the test derivation process for the guarantee automaton of Fig. 4.19 and the safety automaton of Fig. 4.4. The main differences to the untimed version are: 1) the timed test input actions have now to be complemented with respect to the time constraints since a message output by the implementation out of the interval defined by the time constraints will lead to an inconclusive verdict and, for that reason, the test will be stopped; 2) the test controlled events may not occur immediately. In this situation the tester, while waiting for the time to send its next message, may receive an implementation message and, in this case, the test must also be stopped.

The common test case representation $TC_t = T_t \times V_t$ is the composition (intersection) of the two timed automata, as defined in Chap. 3. Fig. 4.22 represents the test case for the test T_t of Fig. 4.21 and the verdict function of Fig. 4.20. Note that two clocks were supposed to be represented in TC_t. However, since they were reset and constrain the same transitions, only one of them is represented.

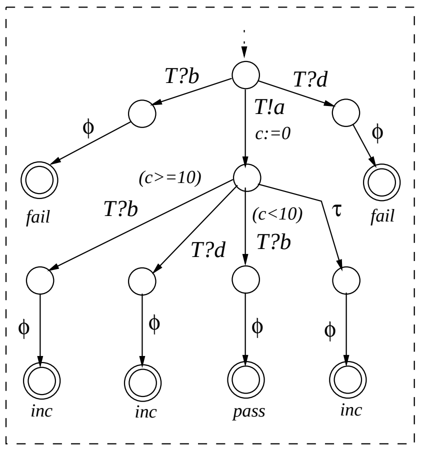


Figure 4.22: Timed test case

4.6.4 Timed test repetition

The algorithm for deriving a cyclic test can also be improved to include time. Assuming the same conditions as for the untimed versions (existence of a reset sequence even for faulty implementations), the following procedure can be used:

1. transform the automaton A_{G_t} into a tree T_t capable of defining the same timed traces,

   $$timTraces(T_t) = timTraces(A_{G_t}) \wedge timAccept(T_t) = timAccept(A_{G_t})$$

   $T_t = (S_{T_t}, E_{T_t}, TT_{T_t}, s_{0_{T_t}}, F_{T_t}, C_{T_t})$
   $E_{T_t} = E_{G_t} \cup \{\tau\} = E^!_{T_t} \cup E^?_{T_t} \cup \{\tau\}$
   $E^!_{T_t}$ contains the tester *controlled* events
   $E^?_{T_t}$ contains the tester *observed* events
   $E^!_{T_t} \cap E^?_{T_t} = \emptyset$
   τ is an unobservable event

2. define an auxiliary set A containing all the tree T_t non-final states, that is, $A = S_{T_t} - F_{T_t}$.
   For each state $s \in A$ do

   (a) define the set M which contains all the events labelling the state s outgoing transitions, that is,

   $$M = \{b \mid s \overset{<(\delta),b,(\lambda)>}{\longrightarrow}\}$$

   (b) for each $e \in E^?_{T_t} \cup \{\tau\}$ do

i. if $e \in M$ then
   A. add a new state s' to T_t
   $$S_{T_t} \leftarrow S_{T_t} \cup \{s'\}$$
   B. add a new transition $s \overset{<(\neg\delta),e,(\lambda)>}{\longrightarrow} s'$ to TT_{T_t}
   $$TT_{T_t} \leftarrow TT_{T_t} \cup \{s \overset{<(\neg\delta),e,(\lambda)>}{\longrightarrow} s'\}$$
   C. add a new final state s'' to F_{T_t},
   $$S_{T_t} \leftarrow S_{T_t} \cup \{s''\},\ F_{T_t} \leftarrow F_{T_t} \cup \{s''\}$$
   D. add a new transition $s' \overset{<(),T!\rho,()>}{\longrightarrow} s''$ to TT_{T_t}
   $$TT_{T_t} \leftarrow TT_{T_t} \cup \{s' \overset{<(),T!\rho,()>}{\longrightarrow} s''\}$$

ii. if $e \notin M$ then
   A. add a new state s' to T_t
   $$S_{T_t} \leftarrow S_{T_t} \cup \{s'\}$$
   B. add a new transition $s \overset{<(),e,()>}{\longrightarrow} s'$ to TT_{T_t}
   $$TT_{T_t} \leftarrow TT_{T_t} \cup \{s \overset{<(),e,()>}{\longrightarrow} s'\}$$
   C. add a new final state s'' to F_{T_t},
   $$S_{T_t} \leftarrow S_{T_t} \cup \{s''\},\ F_{T_t} \leftarrow F_{T_t} \cup \{s''\}$$
   D. add a new transition $s' \overset{<(),T!\rho,()>}{\longrightarrow} s''$ to TT_{T_t}
   $$TT_{T_t} \leftarrow TT_{T_t} \cup \{s' \overset{<(),T!\rho,()>}{\longrightarrow} s''\}$$

3. define the auxiliary set B which is equal to F_{T_t}, that is, $B \leftarrow F_{T_t}$. For each final state $s \in B$ do

   (a) add a new transition $s \overset{<(),\phi,()>}{\longrightarrow} s_{0_{T_t}}$ to T_{T_t}
   $$E^!_{T_t} \leftarrow E^!_{T_t} \cup \{\phi\},\ T_{T_t} \leftarrow T_{T_t} \cup \{s \overset{<(),\phi,()>}{\longrightarrow} s_{0_{T_t}}\}$$
   (b) remove the state s from the final state set F_{T_t}
   $$F_{T_t} \leftarrow F_{T_t} - \{s\}$$

4. add the initial state $s_{0_{T_t}}$ to the final state set.
$$F_{T_t} \leftarrow F_{T_t} \cup \{s_{0_{T_t}}\}.$$

Fig. 4.23 shows the timed version of the test presented in Fig. 4.16, considering the timed guarantee automaton of Fig. 4.19.

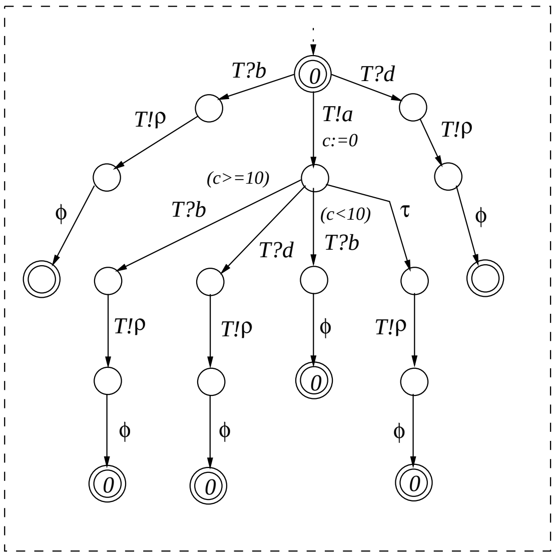


Figure 4.23: Timed cyclic test (I)

4.6.5 Timed trace evaluation functions

Trace evaluation functions can also incorporate time. The function $eval_{T_t}()$ can be described as a timed automaton P_t, with the following characteristics:

1. $E_{P_t} \subseteq E_{G_t} \cup \{\phi\}$, that is, it can be blind for some trace events;
2. all the sequences of visible timed events should be accepted by the automaton P_t;
3. it is deterministic, that is, whatever the trace σ_t, only one final state will be reached.

Since now the traces are timed, there is place to characterise statistically not only the behaviours but also timed behaviours and delays. Let us assume that the timed automaton P_t, which may have clocks, is also equipped with a set of variables D_i, which will be used to store clock values during an automaton transition. Since multiple samples of the component implementation behaviour will be analised (the overnight traces), each variable D_i can be thought as a random variable for which the probability density function $f(\ )$, the mean value $E[D_i]$ or the variance $Var[D_i]$ can be estimated. Fig. 4.24 shows a possible timed evaluation function described as a timed automaton which contains the variable $DELAY$.

Let us assume that the overnight test gives as result the trace ϵ_t

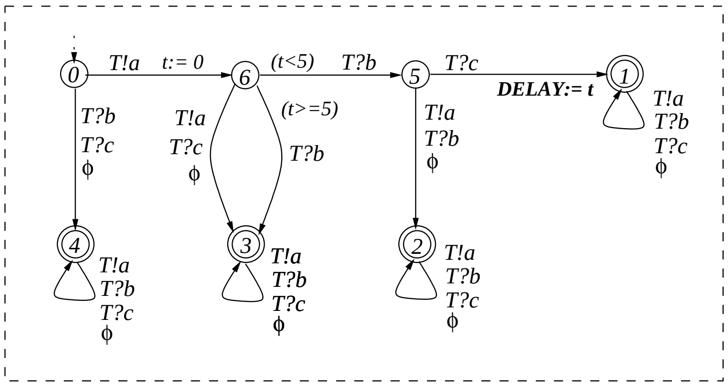


Figure 4.24: Time trace evaluation function

$$\epsilon_t = \sigma_{t_1}.\sigma_{t_2}.\sigma_{t_3}.\sigma_{t_4}, \text{ where}$$

$$\begin{array}{lllll}
\sigma_{t_1} & = & (T!a,3).(T?b,7).(T?c,11).(\phi,11) & DELAY = 8 & P_t = s_1 \\
\sigma_{t_2} & = & (T!a,23).(T?b,30).(T?c,35).(\phi,35) & DELAY = - & P_t = s_3 \\
\sigma_{t_3} & = & (T!a,40).(T?b,43).(T?c,50).(\phi,50) & DELAY = 10 & P_t = s_1 \\
\sigma_{t_4} & = & (T!a,61).(\phi,91) & DELAY = - & P_t = s_3
\end{array}$$

The random variable P_t characterises the implementation with respect to a time-behaviour property, since the final state reached is time constrained. In state 6 of Fig. 4.24, a behaviour decision is taken based on the time associated to the $T?b$ event.

On the other hand, $DELAY$ is a random variable which characterises the delay between $T?c$ and $T!a$ under the constraint that $T?b$ has to be observed between the two events and in less than 5 time units after $T!a$ has been observed. In this case, the sample mean of $DELAY$ would be

$$\overline{DELAY(2)} = \frac{8+10}{2} = 9$$

and its sample variance

$$S^2(2) = \frac{(8-9)^2 + (10-9)^2}{1} = 1$$

It could also be said that the estimated 90% confidence interval for the $DELAY$ would be given by

$$9 \pm t_{2-1,1-0.05}\sqrt{\frac{1}{2}} = \ 9 \pm 4.7$$

The timed trace evaluation function, $eval_{T_t}()$, is said to model the implementation with two random variables - P_t and $DELAY$. Since the ideal component, that is, the specification component which always presents pass traces, can also be modeled by these variables, their comparison (specification and implementation) can give a quantified indication of the type of unwanted behaviours presented in the implementation. The timed trace evaluation function is particularly adequate for describing QoS requirements. As presented in Chap. 3, these requirements are also expressed by means of random variables on delays and failures. For delays, the D_i random variables can be used. For failures P_t random variables can be used.

4.7 Component oriented testing

Network elements are composed of a set of subsystems, such as (1) communications interfaces, (2) end user services and (3) management services, which have specifications that are usually large, sometimes inconsistent and typically described with multiple techniques. How to prove that such a system, containing maybe tens of boards, tens of microprocessors and being developed by tens of engineers is correct, reliable and performant? How to achieve this in short times, so that the system is always *observationally clean* and demonstrable?

In the next sections, a new method will be presented. It is strongly based on our practical experience of testing Network Elements and on the test concepts introduced in the previous sections.

4.7.1 Interface and service models

The first step of the method consists in creating a NE behaviour model, for test purposes, which reuses the available specifications as much as possible. In traditional approaches, modelisation consists in selecting a formal concurrent behaviour language, identify the system main blocks, signals and processes, and in defining each process behaviour. This approach, however, would be time and effort consuming since basically it would duplicate the NE specification activities. It could also reveal itself problematic since the traditional concurrent behaviour languages, such as SDL, are known to have difficulties in representing management and other data processing subsystems.

A more pragmatic approach is used - the NE is modeled from two complementary points of view:

- the *communications view*;
- the *service view*.

Each view consists of a set of simple components.

Communications view

The NE communications view consists of a number of simple components. These components are those which are considered relevant for the NE interoperability with external equipments. Examples are the processes of the NE communications stacks.

These component specifications can usually be found in standard documents where each component is modeled by one or two SDL process/IOSM. Since each component is small and well-known, it can be described as a *complete component*. At the end, the NE communications view will consist of a number of simple components which are described in a *complete component* like style.

Service view

The NE service view consists of a small number of complex components. These components are those found relevant for the NE good service provisioning. Although they become immediately evident when looking at a system, a good advice for identifying them would be to study the NE management specification, usually described using an object oriented approach, where these components are the key system components. Service components do not usually have a direct physical realization. Instead, they are based on several communications and other unmodelled components. Independently of that, they provide a view of the NE which is very close to the view that users have of the system.

Because the service components are complex, they cannot usually be modeled as *complete components* (see Sec. 4.2.3 - Components). However, their user interfaces as well as their use cases can be easily identified, and therefore the interface style of describing components can be used (see Sec. 4.4.1 - Interface component).

4.7.2 Testing communications components

Each communications component should be tested independently. Since each component is small and completely defined, it can be tested using the state machine equivalence testing methods presented in Chap. 2, or the test case approach, presented in detail in this chapter. The first approach provides, obviously, better results. The second, however, is more (time) efficient, hence it seems more appropriate to satisfy the management expectations.

A large number of test cases should then be identified and used to test each communications component. These tests should be small, they need not to be applied repeatedly and no load conditions are required. Test results will be the traditional pass, fail or inconclusive verdict. In this way, each communications component gains a set of tests which should be used every time a new release of the component is introduced in the NE.

With these tests, simple state machine faults will be found, which may have a strong impact on the NE interoperability and service provisioning. A system communicating incorrectly with its environment will never be able to demonstrate its value externally. This is even more true for systems aimed at providing communications facilities.

4.7.3 Testing service components

A service component, as referred above, may be composed of a number of communications and other unmodelled/ untested components. Most of the time, service components include not only real NE components but also some components existing in the test equipments, such as the communications stack emulation components. Multiple instances of each service component exist usually in traditional NEs. Service components should be tested both individually and jointly.

The test of an individual service component is similar to the test of a communications component - a significant number of use cases should first be identified and one test should be derived for each use case. Similarly to the communications components, these tests should be short, not repetitive, applied under no load conditions and give pass, inc or fail verdicts. They should also be reapplied from time to time. Individual service component tests evaluate a variety of service usage conditions.

Joint service component tests are used to evaluate the service components under typical load conditions. They should be applied for a long period of time and use the cyclic and timed test techniques presented in this chapter. Joint service component tests will be referred to as *NE tests*. One *NE test* should be derived for each standard NE load or NE service mix scenario.

A NE test must specify the following aspects:

1. *service mixing.* It consists in deciding the number of instances for each service component. A method to decide this number is to think in terms of the number of expected users for each service, assuming that each user will be served by only one service instance and one service instance serves only one user.

2. *sample of service components instances.* Since a large number of instances can exist for some services (tens, hundreds or even thousands), it is not required to evaluate all of them with respect to their behaviour. For this reason, some service instances must be selected for testing. The criteria for selecting the tested service components must

take into account not only statistical criteria but also the ability to observe potential problems in shared resources.

3. *service components tests.* A test must then be defined for each sample service component instance. For that purpose, a relevant (timed) use case will be selected for each sample instance, which must be transformed into a cyclic timed test.

   As already justified, tests should be *deterministic*, that is, there should be no alternative choices for test controlled events. The same should be considered for the time aspects - if a test controlled event is time conditioned, this constraint should be kept for every test cycle. If this deterministic condition is not observed, the probabilistic trace evaluation function will characterise not only the component implementation probabilistic behaviours but also the tester influence on these probabilistic behaviours. *This condition is particularly important for the results analysis.*

   Another important characteristic of service tests is the specification of the time interval between the end of one cycle and the beginning of the next. The guarantee automaton which specifies the use case represents usually one loop and, for that reason, does not express these time intervals. If the tests (of a component instance) are required to be separated by some time interval, these (deterministic) inter arrival characteristics must now be added to the test. This can be easily achieved by adding a new clock to the test. Fig. 4.25 gives an example of this improvement for the cyclic test of Fig. 4.23.

   The same test type can, naturally, be applied to more than one component instance of the service.

4. *trace evaluation functions.* For each test, one or more trace evaluation functions should be specified, so that the service component instance under test can be evaluated with respect to its most important unwanted/ deviating behaviours. These evaluation functions should also include the parameters required to characterise the service component quality (QoS).

5. *non-sample service components instances.* The service component instances not tested will be loaded with the expected service usage/ traffic profiles. The standard loads referred in Chap. 3 will now be considered. The following technique can be used to define the load for the non sampled service components:

   (a) select a number of relevant use cases from those already known from the individual service components tests;

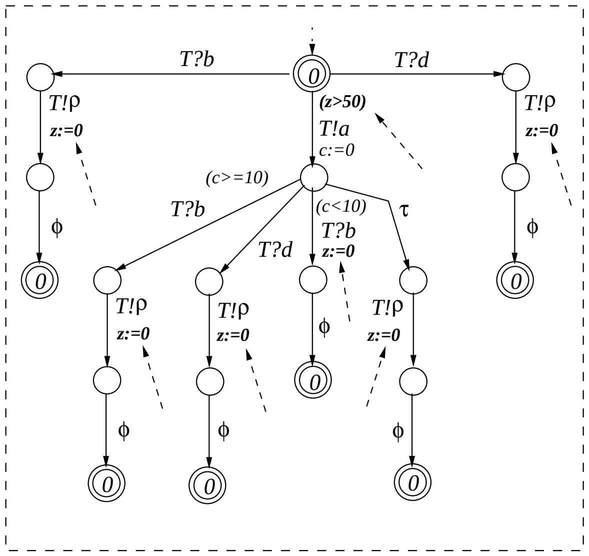


Figure 4.25: Inter test time definition

(b) transform each of these use cases into a cyclic test;

(c) improve each cyclic test with random timed variables, so that the cyclic test is able to describe random aspects, such as call durations and call interarrivals;

(d) represent the cyclic tests in a unique test and let the former cyclic tests be selected randomly.

Randomness is opposed to determinism; to introduce randomness means that the test would be able to select among a set of test controlled events or time intervals associated with test controlled events. Fig. 4.29 (at the end of the chapter) exemplifies the process of defining a random test from a set of two deterministic tests. *Random tests will not be used to reason about components faults but only to load the NE with realistic service usages.* Another interesting aspect is to relate this load test specifications with the delay and loss system events introduced in Chap. 3. Setup, release, arrival and service events can now be precisely associated with the existing test events.

Several NE tests should be specified. Like the component tests, the NE tests should be associated to the NE system - prior to the release of a new NE version, all the NE tests should be re-applied for a long period of time.

NE tests are the most useful tests. They enable the detection of those complex faults which appear only after long working times and which can be random. These faults have direct impact on the service provisioning and, sometimes, also on the NE interoperability. They enable also the precise characterisation of the QoS requirements since trace evaluation functions can capture this type of information.

4.8 Test architectures

4.8.1 Communications component testing

As shown in Chap. 2 (Fig. 2.2), protocol conformance testing proposes several test architectures (local, distributed, coordinated, remote and multiparty).

The communications components are basically the same components addressed by conformance testing. Although in prototype testing the components and the tests could be executed in the same system - the Network Element - their execution in different systems (NE and test system) using a distributed service provider makes the test closer to the real working environment. For that reason, the reference conformance test architectures are preferable for communications component testing.

The *Local* test architecture, in particular, seems to be the architecture more suitable for prototype testing since it makes use of multiple component interfaces and does not require the distribution of the test.

4.8.2 Individual service components testing

Unlike communications components, a service component is a virtual component which is based on a number of real components. Moreover, some of these real components belong to the NE peer communications equipment, such as the peer communications stack components. A possible service test architecture is shown in Fig. 4.26. The *Test* block implements $TC = T \times V$.

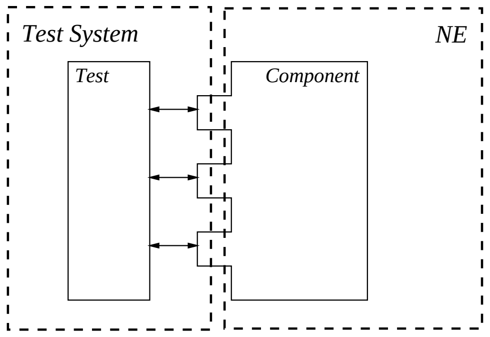


Figure 4.26: Service view component test architecture

Similarly to the communications components, service tests are executed on a system which is different from the NE which, implicitly, forces the NE to interoperate with a different equipment during service tests.

Sometimes the service tests cannot be executed in a single test system. The reasons for that can be performance problems, the impossibility of implementing multiple communications interfaces in a single test system or the physical distance between the NE service interfaces.

The traditional solution for this problem consists in distributing the test by several test equipments. A distributed test architecture brings however some problems, such as:

- the distribution of the test T;
- the location of the verdict function;
- the provisioning of the distributed system with global time.

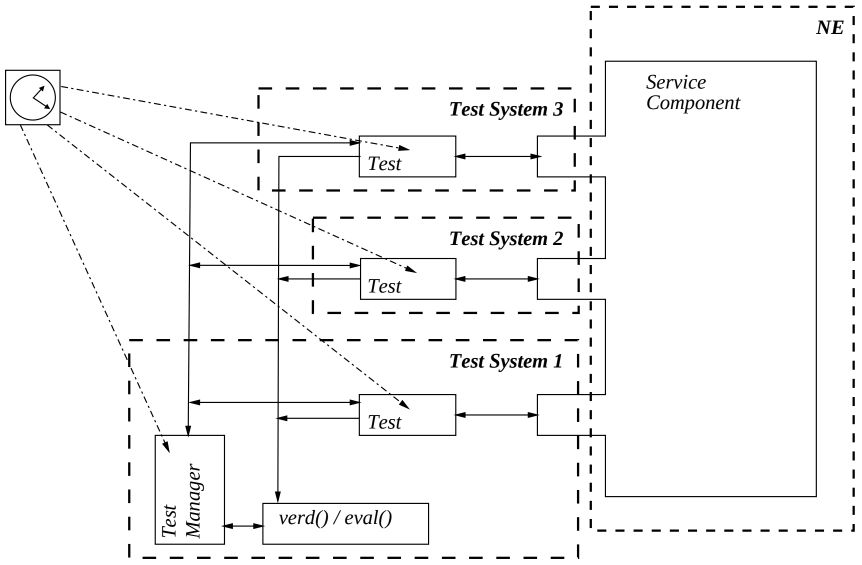


Figure 4.27: Distributed service testing

A possible solution for these problems is shown in Fig. 4.27, where three test systems and the NE under test interoperate.

The test T is split into three parallel testers (*Test*) which are coordinated by a *Test Manager*. The main function of each parallel tester is to send and receive the messages corresponding to its interface. The test manager is in charge of informing each parallel tester about the occurrence of relevant events at other interfaces. Instead of a single tree T, a more complex and distributed tree exists now. The distribution should minimise the number of communications between the Test Manager and (some) parallel testers.

In order to measure service delays and calculate distributed timeouts, a global time mechanism is required. A simple, general and cheap solution for this problem is to take advantage of the timing facilities provided be the GPS system. The available time resolution and precision is sufficient for telecommunications and cheap boards/ device drivers providing digital counters are available. The solution, not new, is more and more being adopted by the manufacturers of protocol testers.

The test distribution becomes simpler if the verdict/ trace evaluation functions are decoupled from the test. A possible solution, as shown in Fig. 4.27, consists in implementing the trace evaluation function(s) in a single block. For this reason, each parallel tester sends its partial timed event sequences to the evaluation block which, after reordering them based on the time information available, evaluates the complete timed traces.

4.8.3 NE testing

Joint service component testing is, from the architectural point of view, an extension of individual service component testing where all the service instances are used simultaneously. Fig. 4.28 exemplifies the concept. Only some service instances are evaluated and the remaining are loaded randomly.

NE testing, in addition, demands (1) probabilistic distribution functions, which may be invoked by multiple service users, and (2) histograms which are used to store evaluation function information, such as final states reached and time intervals.

4.9 Conclusions

In this chapter our methodology for testing telecommunications Network Elements was presented. In order to describe it, a rich framework based on simple (timed) automata was defined. The usage of this framework for reasoning about tests has, in fact, proved very useful. While the essential aspects related to the test of equipments could be reasonably represented, the simplicity of the framework has allowed us to determine the real value of the practical test procedures and, by doing that and abstracting from the real NE complexity, to cover existing gaps and to generalise the method.

The methodology proposed is new by a set of reasons. The first is that it does not rely on complete NE specifications which are unavailable for most of the NEs. Instead, the method relies on the (partial) specifications of some

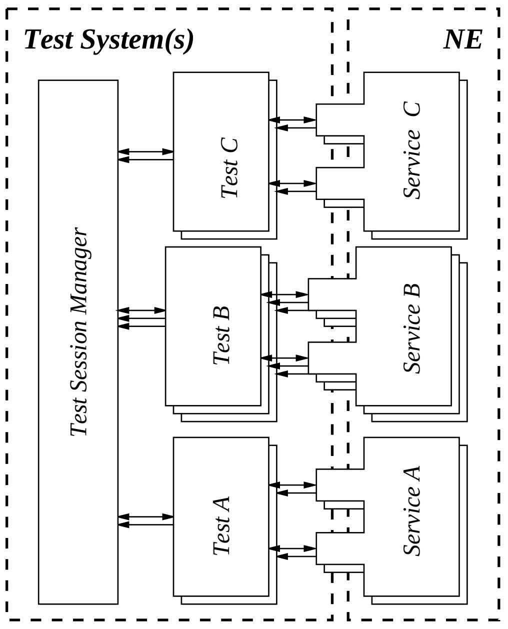


Figure 4.28: NE service testing architecture

components considered relevant for the NE good behaviour. Communications components are typically simple state machines whose description is available in standard documents. Service specifications have usually to be developed but it is assumed that they are described in an interface component style, so they are easy to obtain.

The second innovative aspect is the combination of timed use cases with with untimed specifications which has enabled to derive timed tests and understand them precisely. This approach seems to be the opposite of that generally followed in testing theories where timed tests are derived from timed specifications (safety automata).

The third aspect, and perhaps the most important, is the concept of cyclic tests, which improve the traditional load tests with behaviour information. By doing that, much information about faulty behaviours is gained. The traces obtained during cyclic tests are evaluated by functions different from the traditional verdict function which model the implementation as a set of random variables. By comparing these variables with the corresponding variables from the specification model, the faults become statistically characterised.

The fourth aspect is that probabilistic properties can now be rigorously specified by means of evaluation functions which can be related to the component specifications. By combining behaviour and time, both the failure and delay QoS properties can be rigorously described and evaluated.

The fifth aspect is the service test architecture proposed for NE tests.

By decoupling evaluation functions from tests, a simple architecture was presented, which can capture the diversity of faults provided by this type of tests.

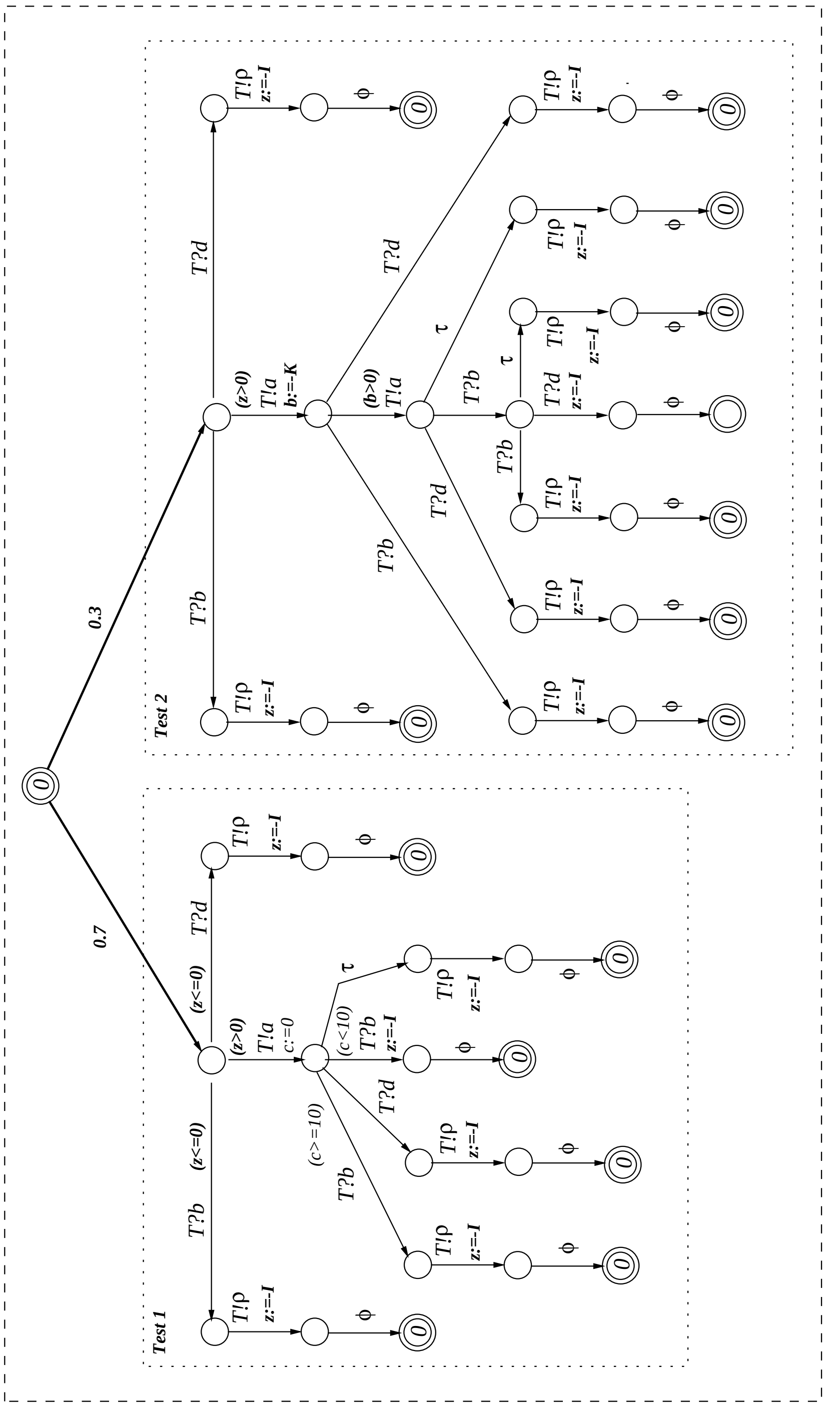


Figure 4.29: Random test

Chapter 5

ETSI V5 Access Network

5.1 Introduction

The Access Network (AN) is the telecommunications infrastructure located between the switching infrastructure and the customer premises equipment. Basically, it consists of transmission equipment terminated by a set of ports to which customers attach their terminal equipment. Its main function is to distribute the telecommunication services and facilities to the customers premises.

With respect to this network, the European Telecommunication Standards Institute (ETSI) has promoted a set of standards aimed at the normalisation of the interface between the Access Network and the switching infrastructure - the V5.1 and V5.2 interfaces.

In the next section, the V5.1 Access Network will be introduced. Firstly, by presenting an overview of the services provided, then by describing its basic architecture and, finally, by describing the architecture of the V5.1 signalling stack.

In the third section, the operation of a PSTN port is described from an operational point of view. A PSTN port can be blocked, unblocked, put in or out of service, have paths (communication channels) associated and be used in telephone calls.

In the fourth section another type of port supported by the V5 access network is introduced - the ISDN Port, and it is also described from an operational point of view. Besides the services provided by the PSTN Port, the access activation and deactivation services are also offered.

The fifth section introduces V5.2 by comparing it with V5.1. Whilst V5.1 works based on a static multiplexer principle, V5.2 is based on a dynamic concentrator principle. This means that V5.2 can have attached terminals requiring more resources than those available at the local exchange side. Some mechanisms for negotiating these resources are, for that reason, required in the access network.

The last section provides an overview on the V5 conformance tests developed by ETSI. The method used to derive the tests, test types and their number are characterised and the testing architecture is also presented.

5.2 Overview

5.2.1 Services

The Access Network shown in Fig. 5.1, and defined in the V5.1 [94] and V5.2 [95] standards, consists of a possibly distributed set of user ports to which conventional terminal equipment, such as the old telephone or a basic rate ISDN terminal, can be connected. Each user port can be controlled by the local Access Network management system and by the the Local Exchange, and each port can be: *1)* put in or out of service; *2)* temporarily blocked, so that no traffic/ calls are accepted by the network.

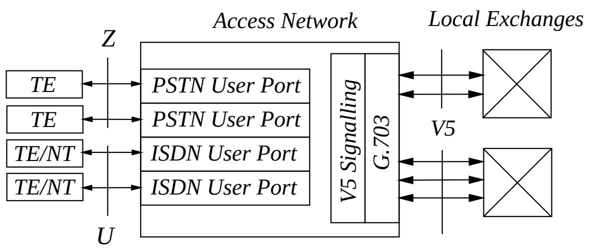


Figure 5.1: V5 Access Network architecture

The PSTN call handling procedures are also converted to digital ones by the V5 Access Network. The current signalling used in the subscriber loop interface for *Offhook*, *Onhook* and *Digits* is sampled by the V5 Access Network and transformed into a three layer protocol interface, similar to the ISDN signalling stack. In this way, V5 Local Exchanges will not receive call signalling by means of current flows but as a set of layer 3 PDUs associated with the V5 PSTN protocol.

For ISDN signalling, the V5 Access Network provides the following facilities: *a)* full layer 2 frame relaying and concentration, so that the peer ISDN signalling stacks can continue to be implemented in the ISDN terminal equipment and in the Local Exchanges; *b)* activation and deactivation of the digital access, which can now be operated by the Local Exchange.

The V5 transmission capabilities, 2.048 Mbit/s links with G.706 framing [96], also enable the digital transmission of 64 kbit/s channels, known as B or bearer channels, which are used for telephone speech transmission and for supporting the B1 and B2 BRI-ISDN data channels.

5.2.2 Functional architecture

The FA1201 is the NEC equipment which implements the V5 Access Network functions. Since the testing methodology proposed in this thesis was developed based on the experience of testing this equipment, a simplified version of its functional block diagram, shown in Fig. 5.2, will be presented. It represents only the main Access Network functionalities and does not include, for instance, management, alarms, protection or E1 link management aspects. It does not also represent the commercial versions of the FA1201 equipment which split the Access Network into Central Office and Remote equipment and also supports high transmission hierarchies requiring optical fibres instead of the electrical cables used in the 2.048 Mbit/s hierarchy.

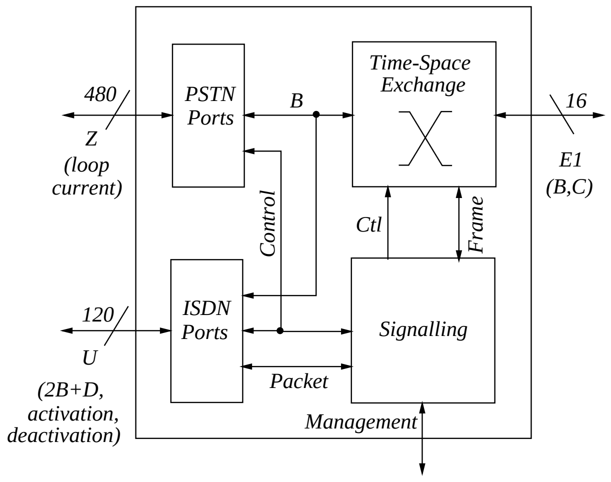


Figure 5.2: AN simple functional architecture

The Access Network presented can support up to 480 PSTN ports providing the standard analogue Z interface and up to 120 ISDN Ports interface at the U reference point [97]. Connections to the Local Exchange are achieved by means of up to 16 E1 links at 2.048 Mbit/s [98].

When a subscriber using a telephone attempts to establish a call the first two actions (events) are to *offhook* the telephone and wait for the *dial tone* so that a sequence of digits can be dialed after that. The telephone, in turn, conveys this information to a PSTN port using loop current signalling as exemplified in Fig. 5.3. The *PSTN Ports* block, in Fig. 5.2, detects the

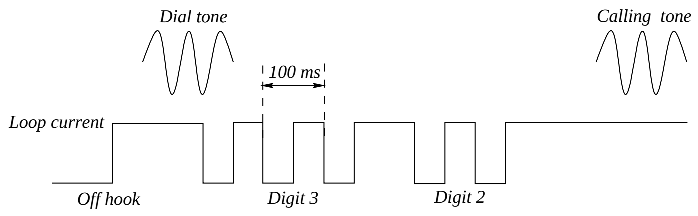


Figure 5.3: Z interface loop current signalling

offhook event and sends a message to the *Signalling* block through the *Control* channel. The *Signalling* block, which implements the V5 stack, creates a HDLC frame referring to the *offhook* event. This frame is then sent octet by octet to the *Time-Space Exchange* through the *Frame* channel, which forwards it to the Local Exchange through an E1 C channel, as shown in Fig. 5.4. The Local Exchange, after interpreting the *offhook* event, starts sending, octet by octet, the *dial tone* digitised according to the law A, through one of the E1 B channels. This channel, in the Access Network, must be routed by the *Time-Space Exchange* to the *PSTN Ports* using the internal *B* channel. The PSTN ports block transforms the digital tone into an analogue signal that is output to the telephone through the Z interface.

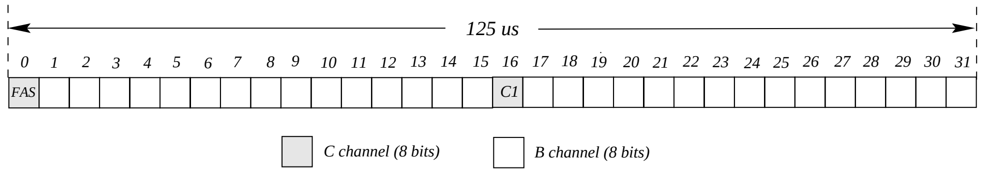


Figure 5.4: E1 interface frame structure

A similar example can be given for ISDN. Assuming that a digital access was already activated and that regular SL signals [97] were exchanged at the *U* interface, the digital telephone starts to establish a call by sending HDLC frames to the Local Exchange, where the peer signalling stack resides. A frame sent by the ISDN telephone through the 16 kbit/s D channel, Fig. 5.5, is transferred by *ISDN Ports* block to the *Signalling* block through the *Packet* channel. The *Signalling* block, without changing the frame contents, relays it to the Local Exchange through a *C* channel. When receiving a HDLC frame from the Local Exchange through the *C* channel, the *Signalling* block must be able to detect that the frame is an ISDN one and route it to the corresponding user port at the *ISDN Ports* block, through the *Packet* channel.

Each block of the simplified model introduced in Fig. 5.2 is responsible

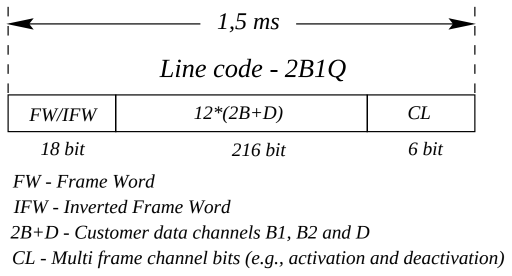

1,5 ms
Line code - 2B1Q
FW/IFW
12*(2B+D)
CL
18 bit
216 bit
6 bit
FW - Frame Word
IFW - Inverted Frame Word
2B+D - Customer data channels B1, B2 and D
CL - Multi frame channel bits (e.g., activation and deactivation)

Figure 5.5: U interface frame structure

for the following functions:

- **PSTN Ports.** Represents the set of user ports provisioned which, in this model, can be up to 480. For each telephone, the block performs the following functions: *a)* converts loop current signalling into semantically equivalent digital messages that are exchanged with the *Signalling* block in both directions; *b)* converts, in both directions, analogue voice speech signals into an octet stream coded according to the law A, which are exchanged with the *Time-Space Exchange*;

- **ISDN Ports.** Represents the set of ISDN user ports provisioned which can be up to 120. For each digital access, the block can: *a)* start activation of the digital access which can be requested by the external NT through the U interface or by the *Signalling* block through the *Control* channel; *b)* deactivate the digital access following a request sent through the *Control* channel by the V5 *Signalling* block; *c)* transfer the D channel bits from the U interface to the *Signalling* block, through the *Packet* channel, in both directions; *d)* transfer in both directions, through the internal *B* channel, the B1 and B2 channel octets between the U interface and the *Time-Space Exchange*;

- **Time-Space Exchange.** Represents a 480×480 time-spatial slot exchange and performs the following functions: *a)* exchange with the Local Exchange digital information that can be distributed among the 16×2.048 $Mbit/s$ E1 links, using a 32 time slot frame structure, HDB3 coded, and (un)frame them appropriately; *b)* provide the Access Network clock, that must be recovered from one of the links driven by the Local Exchange; *c)* route the E_1 C channels, which in V5.1 can be up to 3 (time slots 16, 15 and 31), to the *Signalling* block in both directions, through the internal *Frame* channel; *d)* route the bearer channels to the corresponding user ports, according to a routing table received by the *Signalling* block through the *Ctl* channel which, for V5.1, is static. Routing is carried out in both directions;

- **Signalling.** Represents the block in which the V5 signalling is handled and provides the following functions: *a)* receive and transmit HDLC frames from/to the internal *Frame* channel, interpret them and execute the corresponding orders that, in this model, can be a user port order or an answer to the Local Exchange; *b)* route in both directions the HDLC frames between the *Packet* and the *Frame* channels so that ISDN related messages can be relayed; *c)* program the *Time-Space Exchange*; *d)* execute management commands through the *Management* channel.

5.2.3 V5.1 signalling architecture

The V5.1 interface is used to control one E1 link and the user ports associated with it. From one to three channels of the link (timeslots 16, 15 and 31) can be used to control the interface. The remaining timeslots, up to 30 (see Fig. 5.4), may be used as bearer channels. Since concentration is not supported in V5.1, the number of PSTN ports provisioned must be smaller than the number of bearer channels available. For a network with 16 E1 links, 16 simultaneous V5.1 interfaces are required.

The remote control of the Access Network user ports can be achieved by a set of tasks which, as usual, are represented as a set of layered state machines. Fig. 5.6 shows them for the Access Network side.

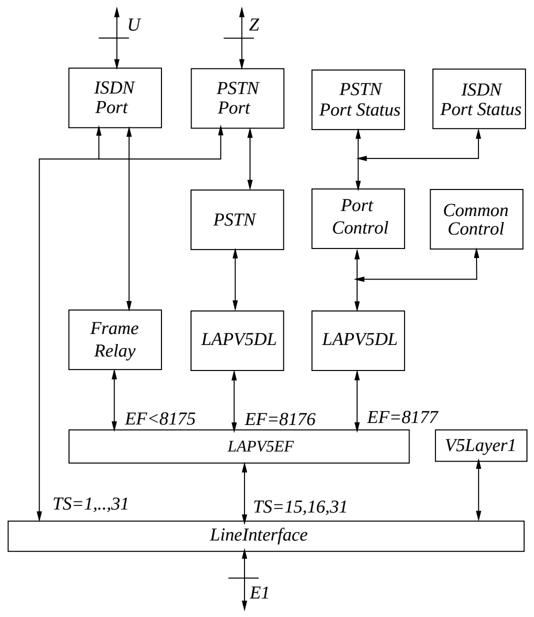


Figure 5.6: V5.1 protocol stack

Each state machine can be shortly characterised as follows:

- **LineInterface.** Terminates E_1 links, that is, decodes the line code, recovers line clock, switches timeslots and provides alarm information;

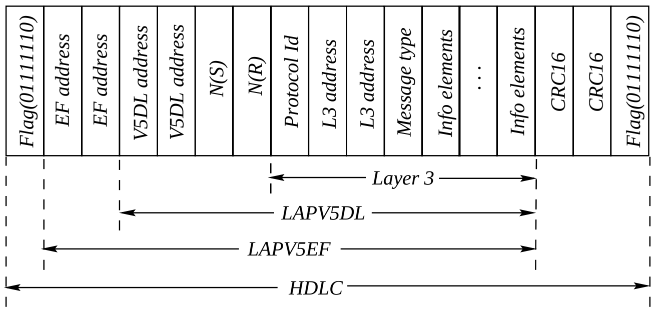


Figure 5.7: V5 message structure

- **LAPV5EF.** All the information exchanged between the Access Network and the Local Exchange is conveyed in Envelope Function frames, shown in Fig. 5.7. The LAPV5EF main functions are: *a)* when receiving the messages from the layer 2 state machines (*LAPV5DL* and *Frame Relay*, in Fig. 5.6), create correct LAPV5EF frames by inserting valid EF addresses and send them to the respective C channel controller (HDLC controller); *b)* when receiving frames from a C channel, verify if they are valid, extract the EF address and route the frames to the addressed layer 2 state machine. The number of LAPVEF state machines must be equal, for each V5.1 interface, to the number of C channels in use (1-3);

- **LAPV5DL.** The LAPV5DL main task is to convey information between layer 3 state machines in the AN and the corresponding peer entities in the LE. LAPV5DL is a simplified version of LAPD where dynamic layer 2 address assignment and layer 2 disconnection was eliminated, since data links are required to be operational for the interface lifetime. The transmission of information frames is possible only after LAPV5 state machines at AN and LE are synchronised. LAPV5DL implements a *Go Back n* [99] protocol (no selective retransmission) where up to 7 messages can be transmitted without acknowledgment (window 7) and multiple acknowledgement is possible. Two LAPV5 state machines exist for each V5.1 interface - one for the PSTN state machines and another for the control state machines. The address of each LAPV5 state machine, the *V5DL address* in Fig. 5.7, is the same as the EF address, that is, 8176 for PSTN and 8177 the for control state machines;

- **Frame Relay.** Packet data, arriving as HDLC frames from the ISDN D channels, are concentrated into C channels and sent to the Local Exchange. Each ISDN port has an EF address associated, which is equal to the layer 3 port address. The Frame Relay main function is to manage the routing between ISDN port internal addresses and EF addresses, which should be less than 8175;

- **PSTN.** PSTN is a stimulus protocol and does not control the call procedures in the AN. It rather transfers, digitally and reliably, information about the analogue line state over the V5.1 interface. Besides that, it controls the path (bearer channel) between the PSTN subscriber port and the Local Exchange virtual port. This path, which is required for call signalling (e.g., transport of tones), will be established before call handling procedures can take place and terminated after the termination of a call. The number of PSTN state machines should be equal to the number of PSTN user ports provisioned and their address, the V5 L3 port address, can be in the range 0 to 32767;

- **Port Control.** The Port Control state machine is used to convey information between the Port Status in the AN and the virtual Port Status in the LE. It conveys one information element at a time whose reception has to be acknowledge by the peer. It uses a symmetrical protocol and the number of state machines required for the Access Network must be equal to the number of ports provisioned (PSTN plus ISDN). The addressing is based on the layer 3 port addresses;

- **PSTN Port Status.** This state machine, in cooperation with its peer at the Local Exchange, provides the coordinated administration of the PSTN ports for the AN and the LE management. Each PSTN port can be blocked or unblocked. When blocked, no calls are accepted. Each PSTN Port Status state machine is associated with a Port Control state machine;

- **ISDN Port Status.** In addition to the blocked and unblocked procedures, the ISDN Port Status state machine is also responsible for maintaining the AN and LE managements synchronised on the activation and deactivation of the ISDN port accesses. Each ISDN Port Status state machine is also served by a Port Control state machine;

- **Common Control.** The Common Control state machine is used to convey information between the management of the AN (not represented in Fig. 5.6) and the LE. It handles one management information element at a time whose reception has to be acknowledged by its peer. It is a symmetrical protocol and only one state machine exists.

5.3 PSTN port operation

In V5.1, the operation of a PSTN port is determined by two main state machines: the PSTN and the PSTN Port Status, as presented in Fig. 5.8.

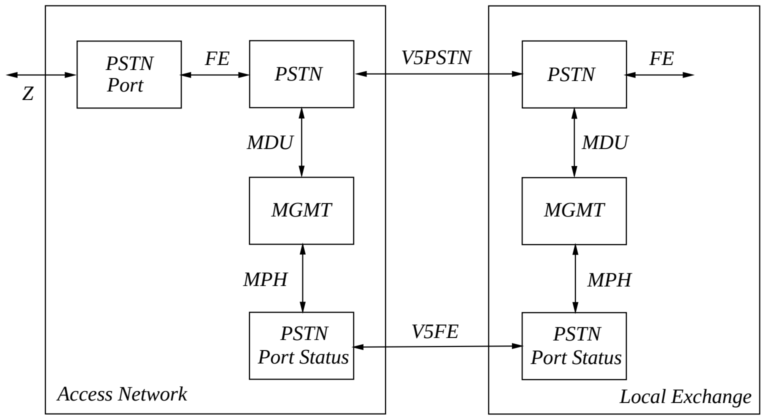


Figure 5.8: PSTN port functional model

In the following sections all those state machines are described by listing and commenting the relevant protocol and service data units and by detailing their main procedures.

5.3.1 PSTN Port Status

Three main functions are provided by the PSTN Port Status state machines: *a)* blocking; *b)* blocking request; *c)* coordinated unblocking.

Blocking a PSTN Port can be initiated by both sides (AN and LE). However, since the Access Network management has no knowledge about the port call state, it will only invoke this procedure under abnormal conditions, such as failures, which affect the service provided by the port.

A blocking request can be invoked by the Access Network to request a non urgent port blocking (e.g., deferrable for maintenance). The Local Exchange, knowing the current state of the call, can block the port or defer the blocking until the end of the current call.

The unblocking of a blocked port requires coordination. The port can be unblocked only when both parts (AN and LE) agree.

V5FE	AN $\leftrightarrow$ LE	Description
FE201	$\leftarrow$	LE requests/accepts coordinated unblocking
FE202	$\rightarrow$	AN accepts/requests coordinated unblocking
FE203	$\leftarrow$	LE requests blocking
FE204	$\rightarrow$	AN requests blocking
FE205	$\rightarrow$	AN requests non-urgent blocking

Table 5.1: PSTN Port Status protocol data units

MPH	MGMT $\leftrightarrow$ Status	Description
MPH-UBR	$\leftarrow$	LE requests/accepts coordinated unblocking
MPH-UBR	$\rightarrow$	MGMT accepts/requests coordinated unblocking
MPH-UBI	$\leftarrow$	Coordinated unblocking completed
MPH-BI	$\leftarrow$	LE requests blocking
MPH-BI	$\rightarrow$	MGMT requests blocking
MPH-BR	$\rightarrow$	MGMT requests non-urgent blocking

Table 5.2: PSTN Port Status management service data units (AN)

Tab. 5.1 lists the protocol data units exchanged by the PSTN Port Status state machines at the AN and LE (*V5FE*), along with their meaning. Tab. 5.2 presents the service data units (primitives) exchanged by the PSTN Port Status state machine at the AN with the AN management (*MPH*). UBR, UBI, BR and BI stand, respectively, for *UnBlock Request*, *UnBlock Indication*, *Block Request* and *Block Indication.*

The PSTN Port Status state machine is described as an $IOSM$ having 4 states and 16 transitions [94].

5.3.2 PSTN

PSTN provides a set of procedures that are mainly related to the establishment and termination of a communication channel and are used to transfer the line signals between the analogue access port and the corresponding virtual port, at the Local Exchange. Particular out of band call handling aspects, such as a digit, are dealt with the primitive *FE-line signal* which is transported transparently to the local exchange which interprets it.

Tab. 5.3 lists the protocol data units exchanged by the PSTN state machines at the AN and the LE (*V5PSTN*), along with their meanings. Tab. 5.4 presents the service data units exchanged by the PSTN state machine with the PSTN port (*FE*), at the AN. Tab. 5.5 presents the service data units exchanged by the PSTN state machine with the AN management (*MDU*).

The PSTN state machine at the AN is described as an $IOSM$ having 8

V5PSTN	AN ↔ LE	Description
ESTABLISH	↔	Initiation of a PSTN path
ESTABLISH ACK	↔	Positive response to PSTN path initiation
SIGNAL	↔	An electrical condition described in a message
SIGNAL_ACK	↔	Acknowledgment of signal message
DISC	↔	Initiation of clearing the path
DISC_ COMPLETE	↔	Positive response to path clearing

Table 5.3: PSTN protocol data units

FE	Port ↔ PSTN	Description
FE-subscriber_seizure	→	Subscriber wishes to originate a PSTN path
FE-subscriber_release	→	Subscriber indicates release during the initiation of the PSTN path
FE-line_signal	↔	Detection or activation of one electrical condition on the subscriber line circuit

Table 5.4: PSTN port service data units (AN)

states and 91 transitions [94].

5.4 ISDN port operation

In V5.1, the operation of an ISDN port is determined by two main blocks: the ISDN Port Status and the Frame Relay, as presented in Fig. 5.9. In the following sections, the state machines used to control the ISDN port are introduced.

5.4.1 ISDN Port Status

Five main mechanisms are supported by the ISDN Port Status state machines: *a)* blocking; *b)* blocking request; *c)* coordinated unblocking; *d)* activation; *e)* deactivation. The blocking/unblocking procedures are similar to the PSTN user port.

In order to reduce power consumption, a basic rate ISDN line not involved in calls is kept deactivated, in a low power state. In this state, there is no signal transmission in both directions of the interface (U) and the clocks are off or running unsynchronised. Before a call is initiated, special patterns are exchanged to wake up the transceivers at both sides of the interface. At the

MDU	MGMT $\leftrightarrow$ PSTN	Description
MDU-CONTROL(block)	$\rightarrow$	AN management indication to block the subscriber port
MDU-CONTROL(unblock)	$\rightarrow$	AN management indication to unblock the subscriber port

Table 5.5: PSTN management service data units (AN)

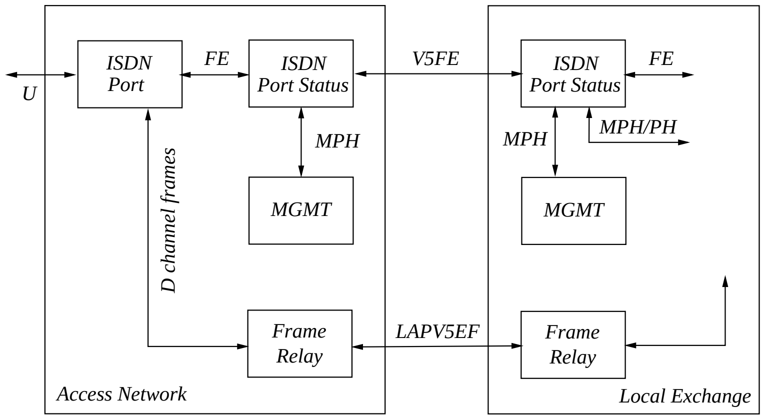


Figure 5.9: ISDN port functional model

U interface a set of SL frames are exchanged so that the port and the NT equipment can again become synchronised. The activation and deactivation of the user digital access will be controlled by the Local Exchange when the port is in an operational state. If the port is non-operational, the control is given to the AN, for maintenance. Activation can be initiated either by the user side or the Local Exchange. The deactivation of the access will always be initiated by the Local Exchange.

In Tab. 5.6 the messages used to carry out the control and activation at the V5 interface (*V5FE*) are listed. Tab. 5.7 shows some primitives used by the management of the Access Network to control and be informed about the ISDN port status (*MPH*). Tab. 5.8 introduces the main primitives exchanged between the ISDN port and the ISDN Port Status (*FE*) at the Access Network side.

The ISDN Port Status state machine at the AN can be described as an $IOSM$ that has 8 states and 87 transitions [94].

V5FE	AN $\leftrightarrow$ LE	Description
FE101	$\leftarrow$	LE requests access activation
FE102	$\rightarrow$	Activation initiated by the user
FE104	$\rightarrow$	Access activated
FE105	$\leftarrow$	LE requests access deactivation
FE106	$\rightarrow$	Access deactivated
FE201	$\leftarrow$	LE requests/accepts coordinated unblocking
FE202	$\rightarrow$	AN accepts/requests coordinated unblocking
FE203	$\leftarrow$	LE requests blocking
FE204	$\rightarrow$	AN requests blocking
FE205	$\rightarrow$	AN requests non-urgent blocking

Table 5.6: ISDN Port Status protocol data units

MPH	MGMT $\leftrightarrow$ Status	Description
MPH-UBR	$\leftarrow$	LE requests/accepts coordinated unblocking
MPH-UBR	$\rightarrow$	MGMT accepts/requests coordinated unblocking
MPH-BI	$\leftarrow$	LE requests blocking
MPH-BI	$\rightarrow$	MGMT requests blocking
MPH-BR	$\rightarrow$	MGMT requests non-urgent blocking
MPH-I1	$\leftarrow$	LE requests access activation
MPH-I2	$\leftarrow$	User requests access activation
MPH-AR	$\rightarrow$	MGMT requests access activation
MPH-AI	$\leftarrow$	Access activated under control of LE
MPH-I5	$\leftarrow$	LE requests access deactivation
MPH-DR	$\rightarrow$	MGMT requests access deactivation
MPH-DI	$\leftarrow$	Access deactivated

Table 5.7: ISDN Port Status management service data units (AN)

5.4.2 Frame Relay

The Frame Relay main function is to correctly transfer valid HDLC frames between the ISDN Ports and the Local Exchange.

Each D channel, at the Access Network, is terminated by an HDLC controller that strips off the CRC and the flags of the LAP-D frames and makes the remaining fields available for relaying (layer 2 address, control and information fields). Frame relay adds to the remaining frame the EF address, that is, the ISDN layer 3 port address, and sends it to another HDLC controller connected to the 64 kbit/s C channel, which will add new CRC and flag fields.

When the frame comes from the Local Exchange, the C channel HDLC controller makes the frame available to the LAPV5EF which, based on the EF address, sends the frame to the Frame Relay block. This unit analyses

FE	ISDN port ↔ Status	Description
FE1	←	Activate access
FE2	→	Access activation initiated by user
FE4	→	Access activated
FE5	←	Deactivate access
FE6	→	Access deactivated

Table 5.8: ISDN status port service data units (AN)

the EF address, removes it and sends the frame to the port HDLC controller which recalculates new CRC and introduces HDLC flags.

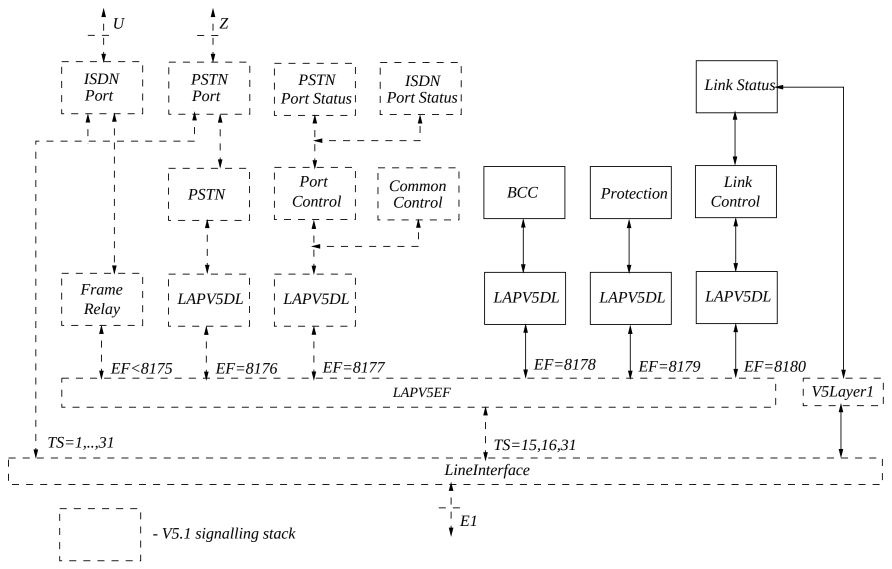


Figure 5.10: V5.2 protocol stack

5.5 V5.2 signalling

While V5.1 works based on a static multiplexer principle, V5.2 is based on a dynamic concentrator principle. The differences of V5.2 with respect to V5.1 are the following:

- **number of links.** A V5.2 interface can have up to 16×2.048 Mbit/s E1 links, while a V5.1 interface supports only one link;

- **concentration.** V5.2 was inherently designed to support concentration of bearer channels, so that the number of channels required by the terminal equipments attached to the Access Network can be larger than the number of bearer channels available at the E1 links. The allocation of bearer channels to user ports is made on a call basis using a new protocol, the *Bearer Channel Connection*. V5.1 does not support concentration and therefore the allocation of bearer channels to ports is static;

- **protection.** When a V5.2 interface is used with more than one E1 link, a particular function exists which protects the control of the interface. This control which, at the startup of the interface, is carried out over timeslot 16 of the primary link and includes messages from the *Port Control* and *Common Control* protocols, is switched to the timeslot 16 of the secondary link in case of failure of the primary link. The protection function is achieved by means of a new protocol - the *Protection* protocol;

- **control.** A new protocol, the *Link Control* protocol, is introduced in V5.2, which is used to the let the V5.2 interface manage the status of multiple links.

The V5.2 signalling interface block diagram is presented in Fig. 5.10. It reuses the V5.1 protocols and addressing formats and introduces four new types of state machines whose main functions may be described as follows:

- **Bearer Channel Connection (BCC).** This protocol provides the means for the Local Exchange to request the Access Network to establish and release connections between specified AN user ports and specified V5.2 interface time slots. The interface bearer channels must be allocated or deallocated on a call basis for both PSTN and ISDN calls under the control of, respectively, the PSTN and the ISDN call control state machines, at the Local Exchange;

- **Link Control.** The Link Control state machine is used to convey information between the Link Status at the AN and the Link Status at the LE. It conveys one information element at a time whose reception has to be acknowledge by the peer. It is a symmetrical protocol and the number of state machines required for the Access Network must equal the number of links provisioned. The addressing is based on 8 bit layer 3 link addresses;

- **Link Status.** This state machine, in cooperation with its peer at the Local Exchange, provides the coordinated administration of links for the AN and the LE management with respect to blocking, unblocking and identification of the links.

The blocking of a single link can be initiated from both sides, although the process is completed under the control of the Local Exchange. In this process, the Local Exchange will disable all non assigned bearer channels in the link from future assignment and, if possible, wait until all bearer channels become unassigned. After that, the LE protects the logical C channels of the link by transferring them to another link.

Unblocking of a single link needs to be co-ordinated at both sides. A link unblock request requires confirmation from the other side before the link is put into operation.

Link identification may be required in some situations, such as after a link failure recovery. In this process, both parts (AN and LE) have to exchange messages in order to verify if they have the same identification for the link;

- **Protection.** The failure of a single C channel or link may affect a large number of subscriber lines or render an interface unoperational. To increase the interface reliability, the Protection protocol reassigns C channels from the failed link to operational ones.

  The Port Control, Common Control, BCC, Link Control and Protection protocols messages are carried on the time slot 16 of the interface primary link. A backup channel is created on the timeslot 16 of another link, which is called secondary link. This pair of C channels constitutes the protection Group 1. When the C channel in the primary link fails, Group 1 ensures the protection of the protocols mentioned above.

  Since the Protection protocol also relies on C channels it cannot protect itself. To overcome single link failures both the primary and secondary channels on timeslot 16 are used simultaneously - the transmitter broadcasts frames into the two C channels and the receiver eliminates the duplicates by analyzing a field of the Protection messages.

  Although the switchover procedure can be requested by both parts, the Local Exchange is in charge of controlling the process.

5.6 ETSI V5 conformance tests

The ETSI V5 recommendations provide a set of conformance test suites whose main objective, as usual, is to verify the correctness of the V5 state machine implementations with respect to their specifications.

5.6.1 Type of tests

The ETSI V5 conformance tests were specified according to the general ISO framework for conformance testing presented in Chap. 2. Tests are, for that reason, oriented towards the state machine transitions.

The protocol state machines are assumed to be input complete. When an input event is received by the state machine and is not expected, the state machine must remain in the same state. This characteristic leads to two types of testable transitions: *a)* opportune (input) event transitions; *b)* inopportune (input) event transitions.

Opportune event transitions are in turn classified in three types: *1)* valid transitions; *2)* invalid transitions; *3)* timer transitions. Valid are the transitions specified by the standard. Invalid are the transitions for which the input events are expected but have syntactically incorrect contents (message fields). Timer transitions are the valid transitions triggered by the expiration of timers which, in this untimed model, are represented as state machine input events.

According to this framework, V5 conformance tests were classified into the following types:

- **Basic interconnection (IT).** Tests aimed at verifying if there is a minimum conformance for the interconnection between the AN and the LE;

- **Capability (CA).** Tests used to verify the capabilities described as supported by the implementation;

- **Valid behaviour (BV).** Tests used to verify if the valid transitions were implemented as specified. Message sequences and message contents are taken into consideration;

- **Invalid behaviour (BI).** Tests used to verify if the IUT is able to react properly when receiving an invalid protocol data unit, where an invalid data unit is defined as a syntactically incorrect message;

- **Timers (TI).** Tests intended to verify if after a timer expiration the IUT behaves as specified;

- **Inopportune behaviour (BO).** Tests used to verify if the IUT reacts properly when an inopportune protocol event occurs. Such event is syntactically correct but it occurs when it is not expected.

In order to test a transition, ETSI also follows the methodology presented in Chap. 2 where each test case is decomposed into 4 parts: *a)* preamble, used to drive the implementation from its initial state to the transition initial state;

b) state transition, where the implementation is excited and the correctness of their answer(s) is evaluated; *c)* state verification, where the transition final state correctness is evaluated; *d)* postamble, that leads the state machine back to its initial state.

Verifying the state machines equivalence by testing all the transitions may, however, be very demanding in terms of number of tests required. In order to overcome this problem two general policies were used by ETSI when defining the V5 conformance test purposes:

- **inopportune event transitions.** Only the inopportune input events visible from the interface point of view, such as protocol data units and timers, are tested. As a consequence, the test of the state machine transitions which are triggered by inopportune internal events is not addressed;

- **procedure oriented testing.** Procedures (normal and exceptional) can be tested in alternative to transitions. Although not rigourously defined, a procedure can be described as a set of transitions which are executed in sequence, so that the state machine can provide a service. Examples of procedures are blocking a port and establishing a path. In this case, the above (in)opportune/invalid event transitions should be read as (in)opportune/invalid procedure *invocation*. These types of test purposes is more in line with the testing by requirements, although there exists a clear relation with the state machine states.

No automatic or formal methods are known to be used during the test purposes definition or test case specifications [100] [101] [102] [103].

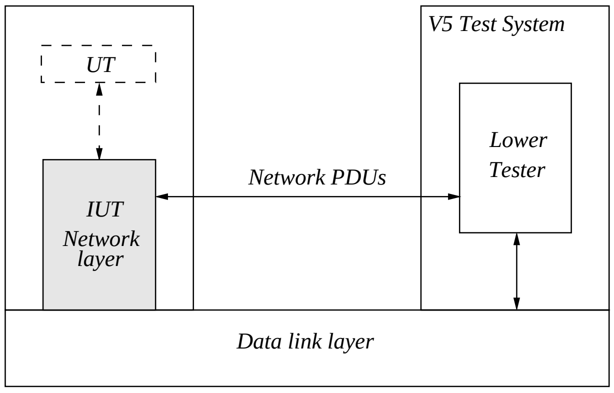


Figure 5.11: Testing method applied to the V5 network layer

5.6.2 V5 test suite

V5 conformance tests are classified as network or data link layer, both for V5.1 and V5.2. Data link layer tests cover the LAPV5DL and LAPV5EF blocks. Network layer tests cover the remaining signalling blocks.

Network layer

The abstract testing method selected by ETSI to test the V5 network layer is the remote test method shown in Fig. 5.11. The Upper Tester of the figure does not, in fact, explicitly exists. However, for some tests, the system under test has to be externally excited and, as a consequence, some interactions at the network service layer may exist.

State Machine	Number of Tests						
	IT	CA	BV	BO	BI	TI	Total
PSTN	1	2	79	41	13	11	145
Common Control	1	2	15	3	12	2	35
Port Control	1	2	3	1	10	2	19
PSTN Status	0	0	7	1	0	0	8
ISDN Status	0	0	11	3	0	0	14
BCC	1	4	21	4	11	2	43
Protection	1	2	23	1	8	5	40
Link Control	1	1	3	1	8	2	16
Link Status	0	0	41	12	0	0	53

IT - Interconnection	CA - Capability	BV - Valid Behav
BO - Inopportune Behav	BI - Invalid Behav	TI - Timers

Table 5.9: V5 network layer conformance tests

Tab. 5.9 shows the number of tests for each network layer state machine [100], [104], [102], [105].

Data link layer

The testing method used for V5 data link layer is the remote test method as well, as shown in Fig. 5.12. Tab. 5.10 presents the number of tests for each data link layer state machine [101], [106], [103], [107].

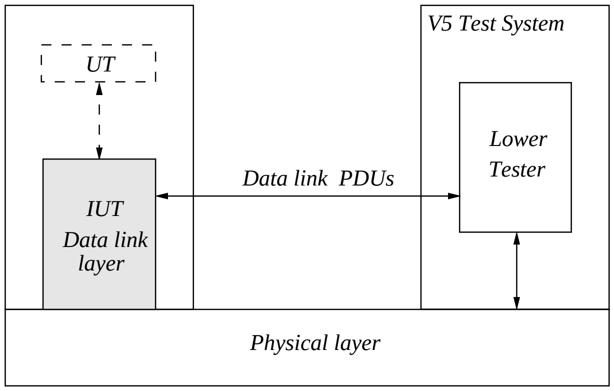


Figure 5.12: Testing method applied to the V5 data link layer

State Machine	Number of Tests						
	IT	CA	BV	BO	BI	TI	Total
LAPV5DL	1	2	82	12	36	8	141
LAPV5EF	1	0	2	0	6	0	9
IT - Interconnection	CA - Capability		BV - Valid Behav				
BO - Inopportune Behav	BI - Invalid Behav		TI - Timers				

Table 5.10: V5 data link layer conformance tests

5.7 Conclusions

The ETSI V5 Access Network presented in this chapter constitutes the case study of this thesis. The prototype implementation tested, the NEC FA1201, was targeted at 480 PSTN user ports and 120 ISDN user ports. It was found, in our perspective, a very complex equipment.

From the hardware point of view, the NEC system consisted of a rack with about 70 boards. More than 60 boards had at least one microcontroller incorporated whereas the most complex could use up to four microprocessors. A large number of buses and cables were required to interconnect these boards.

From the software point of view, it consisted of the firmware for each board and, for the controlling boards, kernels, device drivers, signalling and applications.

For the reasons above, this is a good example for understanding the real testing issues claimed to be solved by the methodology proposed in Chap. 4.

Chapter 6

Test of Access Network Services

6.1 Introduction

This chapter describes the application of the testing methodology presented in Chap. 4 to the validation of the NEC FA1201 Access Network.

The communications components, which were validated by applying the V5 conformance tests presented in Chap. 5, are the V5 signalling components presented in Chap. 5 as well. Since neither the V5 conformance tests nor the V5 protocol analyser required to execute these tests were developed by the author, the results of this activity are not included in this chapter.

In the second section, a method for identifying and modeling the main Access Network services is introduced. These services are identified in the V5 management documents and represented as a hierarchy of objects. A main object - the Access Network - was identified that aggregates three other types of objects: Interface, PSTN Port and ISDN Port. Some of these objects are then further refined so that the Access Network services can be represented with a level of detail adequate for testing.

In the third section, two of these components are specified - the PSTN Port and the Path, which is a component aggregated by the PSTN Port. They are described in terms of basic communications components, some of which reside in the test equipment, and of service-user interactions through the component interfaces.

The fourth section engineers the method. SDL was selected as the test language and its characteristics are described from the test methodology point of view. Two problems associated with the SDL operational model (queues and timed actions) are, in particular, studied and solved.

In the fifth section three tests are introduced and described by their guarantee automata. Using the rules of Chap. 4, tests and trace evaluation functions can be derived and, using the results of the previous section, described as SDL processes.

The sixth section gives two examples of cyclic timed tests for the Path service component. Their refinement from simple service tests, the derivation of reset sequences and the derivation of clock and clock constraints from available traffic models are presented. For each test, a set of trace evaluation functions capable of measuring the quality of the Path component is presented as well.

The seventh section describes the process of testing the NEC FA1201 and gives an overview of the service test equipment.

The eighth section qualitatively evaluates the relevance of the testing method for the validation of the NEC FA1201 Access Network. Some types of faults are presented and pointers to the results of hundreds of tests carried out and to the service test equipment developed are provided.

6.2 Service model

According to the methodology proposed in Chap. 4, NE service components can be identified by studying the NE management models (Sec. 4.7.1, Service view). For the V5 Access Network presented in the last chapter a simple service model was developed which captures the most significant services. This model, presented in Fig. 6.1 using OMT, describes the Access Network

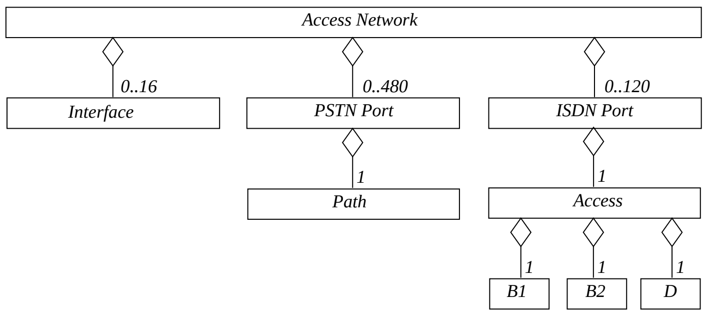


Figure 6.1: V5 AN object oriented view

as a single object which aggregates three types of objects: 1) Interface; 2) PSTN Port; 3) ISDN Port.

6.2.1 Interface

The Interface component (or class, in an object oriented view) models a V5 interface. As shown in Chap. 5, a 16 $E1$ link Access Network can contain up to 16 simultaneous $V5$ interfaces. Independently of the number of ports served and links used, each Interface is said to be, as shown in Fig. 6.2, in one of three states: 1) idle; 2) out of service; 3) in service.

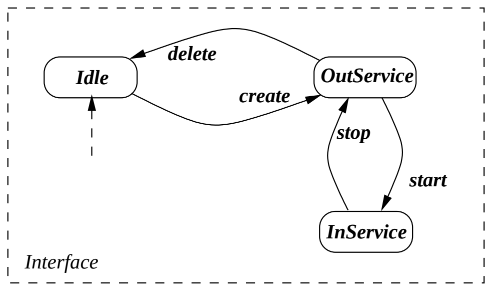


Figure 6.2: V5 Interface dynamic model

Unlike general programs, embedded and real time systems, such as Network Elements, frequently avoid the dynamic creation of objects. For that reason, the components and their resources (e.g. memory space) are created at the system startup. A system user, however, can assume that a component is created only when he orders it. In this case, the idle state describes an undefined state in which an existing component waits for a user creation request.

After it is created, the Interface is said to be out of service. In this state, some layer two and layer three communications components become available and are allocated to the interface, although the V5 paths between the Local Exchange and the Access Network are not yet available for communication.

After successfully started, the Interface comes into service. The V5 Access Network can now communicate with the Local Exchange and the normal operations, such as those involving user ports, can take place.

6.2.2 PSTN Port

The Access Network supports up to 480 PSTN user ports, where an user port represents the port to which an analogue telephone is connected, as well as its resources. Two user port views are required: the management view and the end user view. The PSTN Port object of Fig. 6.1 can be in one of three states: 1) out of service; 2) blocked; 3) unblocked, as shown in Fig. 6.3.

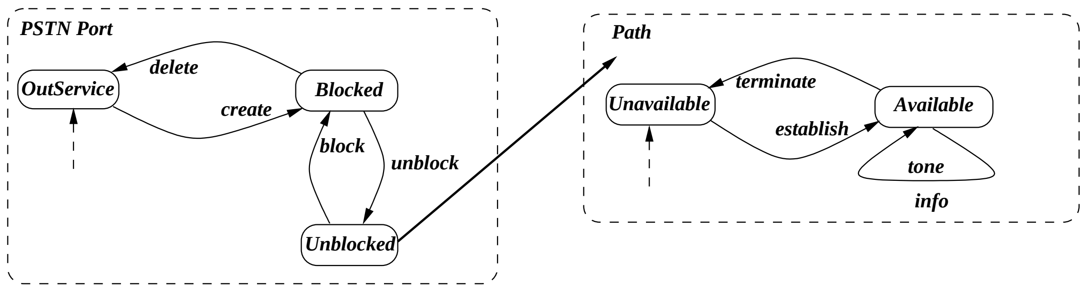


Figure 6.3: PSTN Port dynamic model

A PSTN Port is said to be out of service when, although physically installed and recognised by the Access Network, it has not yet been associated to any V5 interface. When created, which from the V5 point of view means to associate the V5 port resources, such as a PSTN protocol instance, to a V5 interface, the port is said to be in a blocked state. After successfully unblocked, which implies the coordination between the Access Network and the Local Exchange, the PSTN Port moves to the unblocked state, where PSTN paths can be established and terminated by end users.

For this reason, the Path service component is said to have its life conditioned by the permanence of the PSTN Port in the unblocked state. In other words, the Path service component is implicitly created when the PSTN Port *enters* the unblocked state and deleted when the PSTN Port *exits* the unblocked state, independently of the Path state.

A Path is said to be unavailable or available, being available after it has been successfully established. When available, the path can be used to transmit tones and other type of information, such as digits.

6.2.3 ISDN Port

The Access Network under study supports up to 120 ISDN user ports, where an ISDN Port represents the port to which the basic rate interface equipment is connected as well as the resources used. Similarly to the PSTN Port, the management and the end user views will be modeled for testing. The ISDN Port object of Fig. 6.1 can be in three states: 1) out of service; 2) blocked; 3) unblocked, as shown in Fig. 6.4.

The out of service and blocked states have the same meaning as for the PSTN port. After successfully unblocked, the ISDN Port moves to the unblocked state where its digital access becomes available and can be operated. The digital access is modeled by the Access service component.

The Access can then be deactivated or activated. When activated, three independent communications channels become available, which are represented by three other objects, B1, B2 and D, which represent the independent

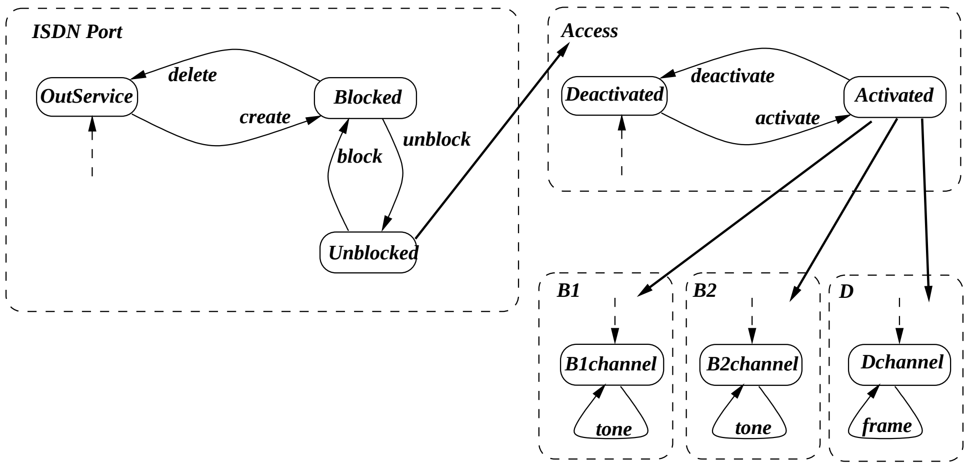


Figure 6.4: ISDN Port dynamic model

bearer channels associated to the ISDN $2B + D$ basic rate interface. Each of them is modeled by a state machine through which general octet streams (B1 and B2) or HDLC frames can be transmitted.

6.3 PSTN service components

In order to be usable for service testing, the service components have to be further specified. In the next sections, the PSTN Port and the Path service components are described in detail. The procedures required to further specify the other components (Interface, ISDN Port, Access, B1, B2 and D) would be similar.

6.3.1 PSTN Port

Composing components

In order to model the distributed procedures associated with the PSTN Port (creation, deletion, blocking and unblocking), some resources in the V5 Local Exchange have to be considered. In communications in general, and in the V5 case in particular, the components available at both sides of the interface must know each other before they start to interoperate. For instance, a PSTN Port, created at the Access Network side, needs an equivalent virtual port at the Local Exchange side, which represents the local exchange view of the real PSTN Port status. The blocking/ unblocking procedures are coordinated by the communication components at both the Local Exchange and the Access Network.

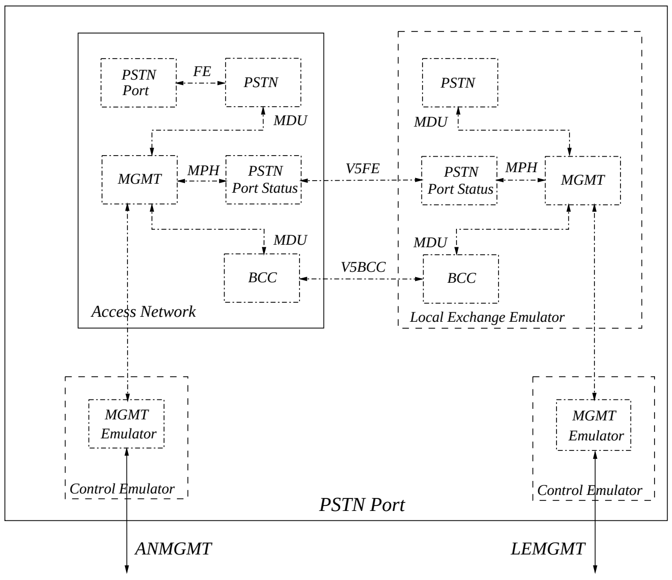


Figure 6.5: *PSTN Port* service component

A testable PSTN Port *service* component must, for that reason, include some emulation components at the Local Exchange side, as shown in Fig. 6.5. In this case, not only the emulation communications components at the Local Exchange are included in the PSTN Port service component, but also some components which support the proprietary management protocols.

The advantage of defining a service component which includes the communications components at both interface sides is that, when used, the most probable interoperation scenarios are exercised. As a disadvantage, faults can be detected not only at the Access Network but also at the Local Exchange and at the Management emulators. This disadvantage can, in some cases, be not real since sometimes, the emulation communications components have to be developed *at home*, during the project lifetime thus, by applying the service tests to these large service components, the emulation communications components are also validated. When a service component passes its test set, we are confident that: 1) the Access Network basic components can interoperate internally to provide the service; 2) the Access Network can interoperate with a peer equipment when providing this service; 3) the emulation tasks are, to some extent, also validated.

Specification

The PSTN Port service component shown in Fig. 6.5 is accessible through two Management interfaces: 1) ANMGMT; 2) LEMGMT. Since a number of basic components are involved, a fast specification can be developed if

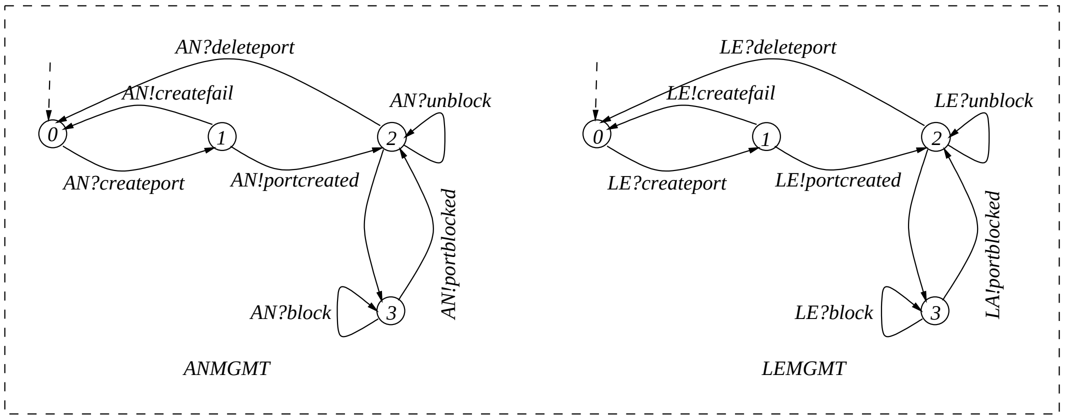


Figure 6.6: ANMGMT and LEMGMT interfaces

the service component is described as an *interface component* (Sec. 4.2.3). Assuming synchronous interactions between the PSTN Port service component and its environment, the two PSTN Port component interfaces can be described by the automata of Fig. 6.6, as it was done in Sec. 4.4.1.

The PSTN port creation, deletion, blocking and unblocking are required to take place both at the Access Network and at the Local Exchange. Fig. 6.7 shows the messages exchanged by the basic PSTN Port service components during the procedure of unblocking of a port, as described in Chap. 5.

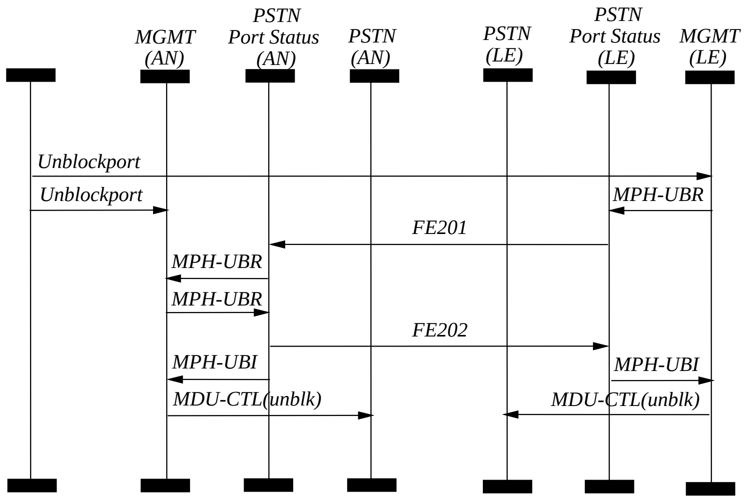


Figure 6.7: Coordinated port unblocking

6.3.2 Path

Composing components

The test of path establishment, termination and information transfer service procedures drives the Path service component to include also some emulation components. The Path service component presented in Fig. 6.8 not only includes communications components which are common to the PSTN Port service component but also some emulation components at the Local Exchange and Terminal Equipment. In fact, Z being an electrical interface, some emulation components which transform electrical signals into software "testable"signals seem to be appropriated.

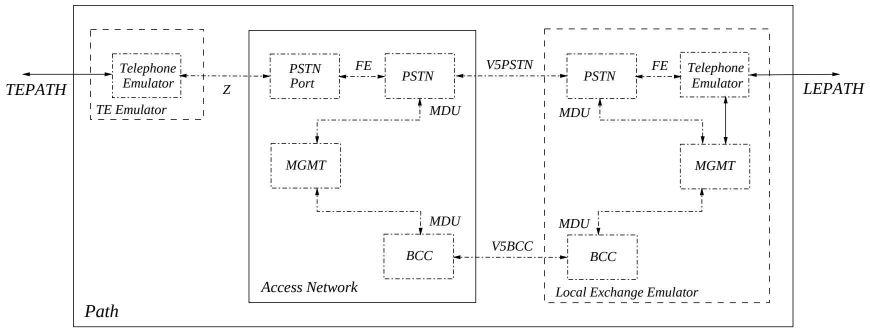


Figure 6.8: *Path* service component

Specification

Similarly to the PSTN Port, the Path service component can be specified as an *interface component* whose automata describe the possible user-component interactions through each interface of the component.

The TEPATH interface of Fig. 6.8, at the terminal emulator side, was defined to emulate as closely as possible the interface used by real end-users, that is, the standard telephone user interface. Relevant to this interface, which is shown in Fig. 6.9, are: *1)* the *Callaborted* signal, which is output by the Telephone Emulator when something goes wrong, such as a dial tone not arriving for 20 s; *2) Callcleared*, which indicates to the user (tester) that the local exchange has decided to terminate the path during the call establishment.

The LEPATH is a hybrid interface which mixes the simple user-telephone interface with facilities usually provided by exchanges, such as the recognition of a valid number. $Remoteoffhook$ and *Remoteonhook*, in Fig. 6.10, are

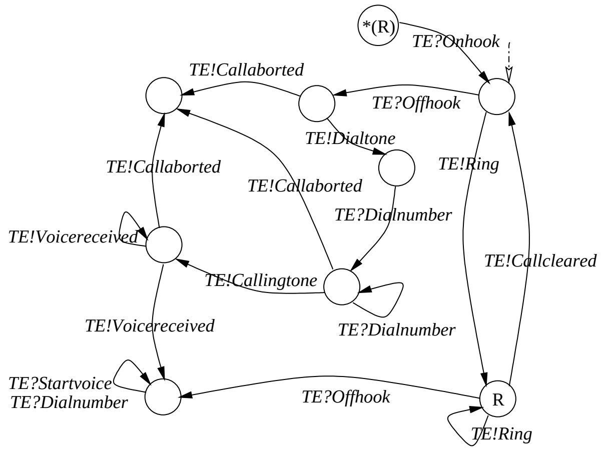


Figure 6.9: TEPATH interface

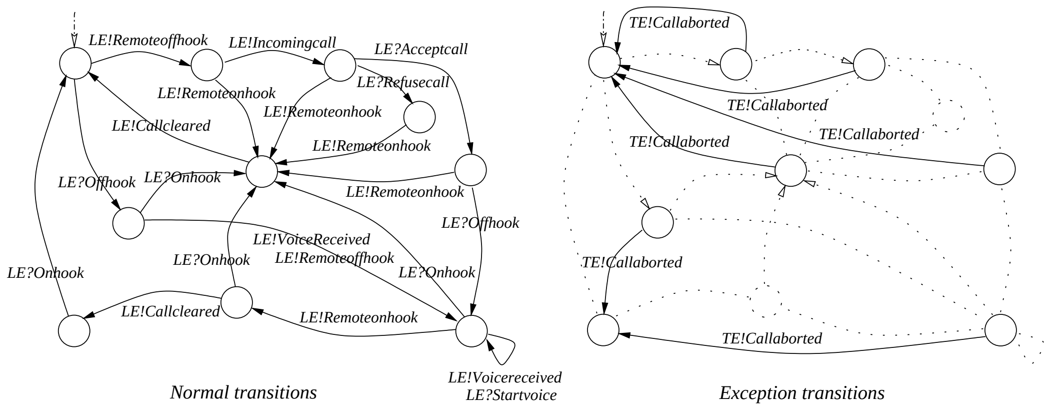


Figure 6.10: LEPATH interface

the Local Exchange view of, respectively, the *Offhook* and *Onhook* signals at the terminal side. *Incomingcall* indicates that a correct number has been received. Calls at the Local Exchange can be accepted or refused by, respectively, the *Acceptcall* and the *Refusecall* events. Fig. 6.10 also shows the abnormal transitions which indicate the existence of internal problems and are signalled by *Callaborted*.

Fig. 6.11, Fig. 6.12 and Fig. 6.13 provide, respectively, an overview of (1) the signals exchanged by the Path components during the establishment of a call initiated by the Local Exchange, (2) the signals exchanged by the Path communication components during the establishment of a call initiated by the Terminal Equipment side and (3) the signals exchanged by the Path service components during the termination of a call by the Terminal Equipment.

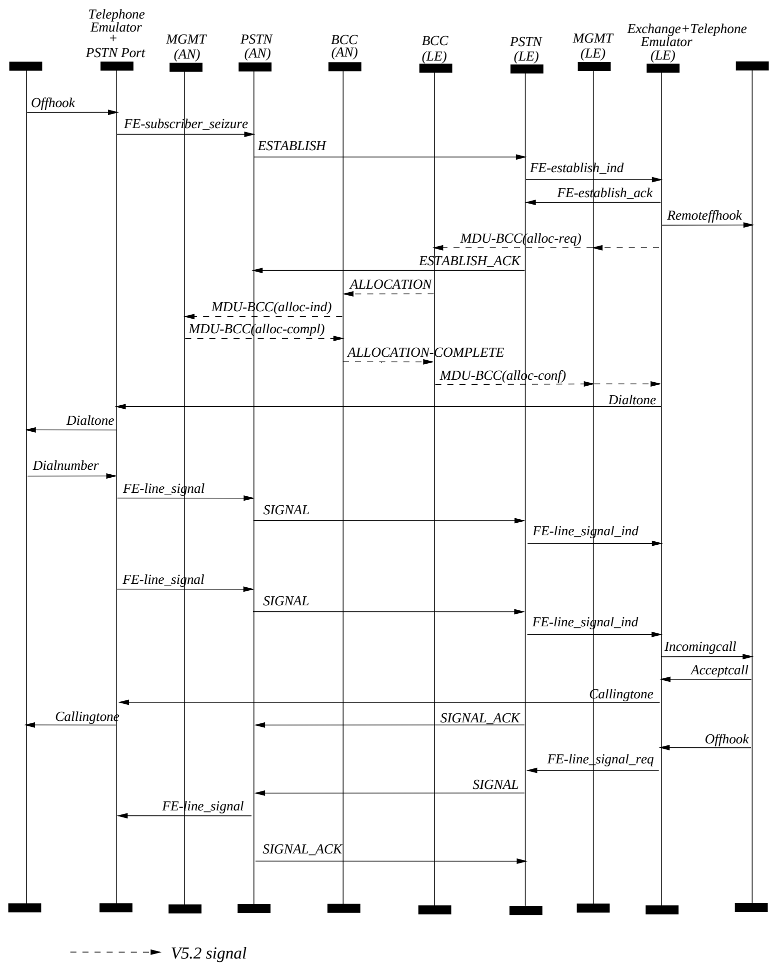


Figure 6.11: Call establishment from TE

6.4 Service testing framework

The implementation of a test system, in which the test components (tests, test managers, trace evaluation functions) and the basic components (emulation components) can coexist concurrently and interoperate, requires a multitasking infrastructure in which the process concept can be easily implemented.

For the case of the FA1201 system a multitasking environment was used which is part of a software creation method developed in the project - the combined method [10]. In this method, a system is described in SDL and combined with C++ classes which are used to implement the parts of the system which are difficult to describe in SDL, such as device drivers, message coding/ decoding procedures and data intensive processing tasks. The SDL part of the system, in turn, is translated into a set of C++ classes which are

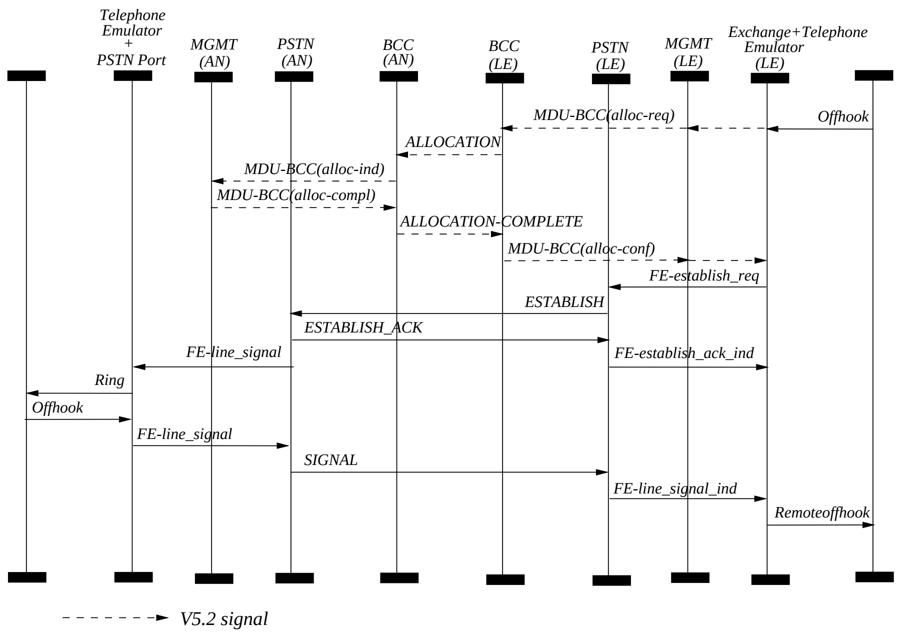


Figure 6.12: Call establishment from LE

compiled and linked with the pure C++ classes. The SDL Process plays here a relevant role: it is translated into a particular C++ class whose instances are selected by a small kernel to run one transition at a time, according to the SDL operational model.

The service test system used to validate the FA1201 system was implemented according to this method. Although very powerful and efficient from the modelisation point of view, the method has the disadvantage of being proprietary and, for that reason, somehow restricts the attempt of this thesis to propose a general testing method. For that reason, a new test environment is presented in simple SDL, so that it can be compiled to run in the more popular real time operating systems for which SDL can be translated by commercial tools.

SDL, as described in Chap. 2, consists of a set of processes that communicate exchanging signals. Each process is described as a state machine where signal input and output events can be defined. Each process holds a queue in which the signals sent by the other processes are stored before being consumed by the owner process. Transitions are atomic. In addition, processes can be extended with data which is manipulated in transitions and a set of predefined data types is also available. SDL contains the two most important characteristics required by a system implementing the testing methodology proposed: (1) it supports the description of concurrent/parallel processes and (2) processes are described as state machines. Besides that, it has ad-

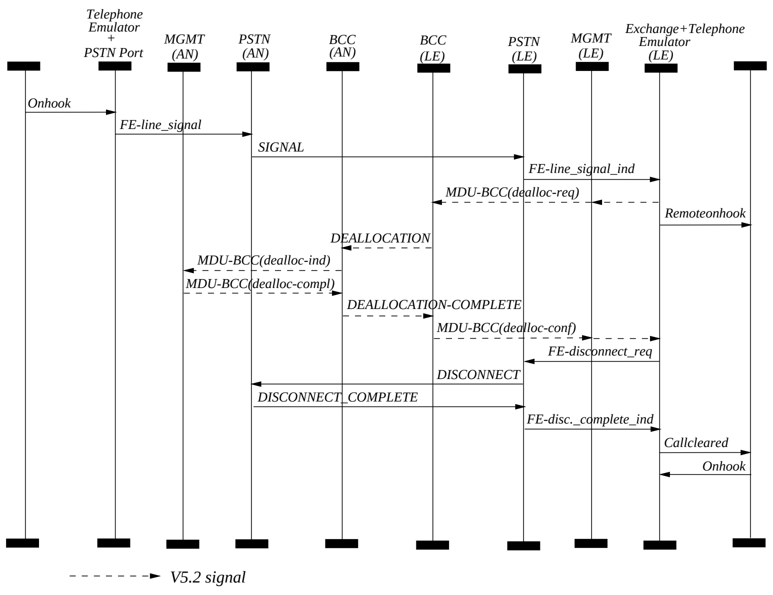


Figure 6.13: Call termination

ditional advantages: it is supported by tools, it has a graphical format and it can be automatically translated into programming languages, such as C or C++. It has, however, two characteristics which are serious drawbacks for the proposed methodology - (1) its asynchronous interprocess communications model and (2) the utilization of the process queue for signalling timeouts.

6.4.1 Communications model

Let us assume that a service component is represented by an SDL Block and each service interface is modeled by an SDL process, as shown in Fig. 6.14 and Fig. 6.15.

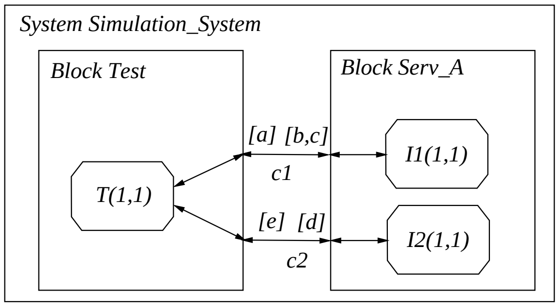


Figure 6.14: Service component represented in SDL

It is also assumed that a Test block containing the test process T, is associated to the service component. This process will, according to Sec. 4.3.3, be used both to derive the $Serv\_A$ safety automaton and to represent a particular test. According to SDL, a signal queue of infinite length is associated whith each SDL process, that is, with interface 1 (I_1), interface 2 (I_2) and test process (T).

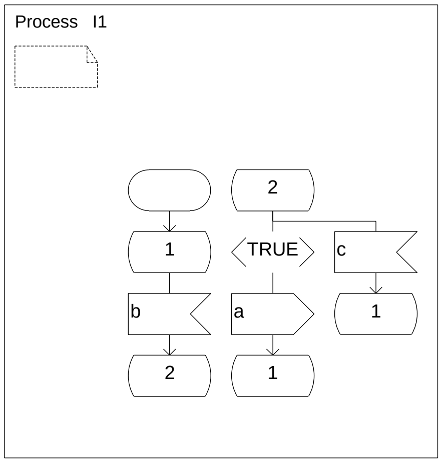


Figure 6.15: Process I_1 state machine

The problem

According to the methodology presented in Chap. 4, the service component safety automaton would be obtained by simulating the three processes and their associated queues. The relevant test events would be, in this case, to get a signal from the test process signal queue and to put a signal into a service interface process queue. This means that the process queues would be considered as part of the component specification. While for certain cases, such as SDL specifications, this situation can be considered acceptable and the tests can be derived, in other cases, such as the V5 service components defined above, queues can be a source of problems.

For the Access Network service components, the interfaces presented above appeared naturally. The TEPATH interface (Fig. 6.8 and Fig. 6.9), for instance, should emulate an analogue telephone. For that reason, it was developed having the user interface of this telephone as a behaviour model. The interactions between a telephone and a human are, by nature, synchronous. The user action of picking up the handset (T!Offhook) and the telephone action of having the handset picked up (TE?Offhook) occur at exactly the same time instant. The artificial inclusion of communications queues in the service test system may, for that reason, lead to wrong test results. Let us suppose that the test T intends to establish a call but, just before it outputs the Offhook signal to the telephone queue (T!Offhook event), the telephone rings, that is, it outputs the Ring signal to the test queue (TE!Ring). From the test point of view, the $T!Offhook.T?Ring$ sequence would be observed. If the test was derived by considering the queue context, this sequence would be considered valid from the safety automaton point of view. The problem arises if, in fact, the telephone implementation is faulty and always rings after the handset is picked up. It would generate the same event sequence, $T!Offhook.T?Ring$ which, due to the process T queue, would be considered valid. The problem would not happen if synchronous user-telephone interactions were assumed, since the event order would always be preserved.

The immediate procedure to test-optimise the SDL communications model is to provide the SDL kernel with synchronous communications. Although this assumption violates the SDL operational model, it is frequently supported by SDL simulation tools. The usual approach for this communication mode is to mark some process communication events as rendez-vous. It means that, for instance, after a process outputs a rendez-vous tagged signal, the kernel assumes that there is an implicit state after this output event and runs immediately after the process transition enabled by the corresponding input signal event, which should also have been marked as rendez-vous. A similar solution was used in the FA1201 validation project. The concept of synchronous process, where all the signal input events are tagged as rendez-vous, was introduced and used.

This type of solution, although optimal from the test point of view, requires the control of the SDL kernel. Since, most of the times, test engineers do not have such control, a pragmatic solution is presented which does not use the rendez-vous SDL "violation". Let us first characterise the process queue and the service components.

Queue influence. When developing tests, engineers rarely think about the queuing effect, although this type of context is present in most of the test environments. The TTCN operational model, for instance, considers the existence of two queues between the tester and the implementation under test. Conformance tests developed by hand, however, rarely consider the queue effect. One of two positions can be taken: 1) these tests may be incorrect because they should but do not consider the queue effect; 2) the queue effect is not considered because there is no real need to consider it.

The position adopted throughout this thesis has been to consider that formal methods should be used to understand and improve practical engineering methods which were assumed as the methodology departure point. Let us come back to the telephone case and suppose that the tests were derived using the *rendez-vous* communications model (no queues) but were executed in a queue based context. The immediate consequence is that, in this case, a $T!Offhook.T?Ring$ sequence would be considered incorrect whereas, in reality and due to the queue, it could correspond either to a good or a faulty implementation. The next question that can now be asked is – in what cases will this sequence correspond to a good implementation (the two events are correct and the sequence problem is due to the queue)? Only when the two events are separated by a few miliseconds, which is the time the kernel makes the $TE?Ring$ signal wait in the process T queue before it is consumed! In all the other situations the test would correctly evaluate the implementation.

This argument could not be used to justify the use of tests derived by synchronous methods in asynchronous environments. But it can be taken into account.

Logging facilities. Another lesson learned from practice is that relevant service components have parts inside the test system. These parts are usually emulation of protocols for which the test system usually provides logging facilities, since this information is important for practical debugging. Considering the Path service component, for instance, let us assume that its *Telephone Emulator* (Fig. 6.8) can log *with time* all the events observed at the $TEPATH$ interface (e.g. $TE?Offhook$ and $TE!Ring$).

Hybrid solution

The proposed solution consists in deriving the service component safety automaton assuming a synchronous communication model, that is, the effect of SDL queues during the simulation of the system is not considered. Tests and verdict functions are then derived based on this safety automaton, according to the procedures presented in Chap. 4. Tests, however, will be implemented as normal SDL processes (with queues). The verdict functions are also implemented as SDL processes but *using the log information generated by the service component instead of the test observed/ generated events.*

Let us assume the situation of Fig. 6.14, where the incompletly specified service component $Serv\_A$ is described by two non-comunicating processes, I_1 and I_2. When *rendez-vous* communication is assumed between these processes and the process T, the interface safety automaton associated with channel $c1$ is, according to Sec. 4.4.1, the automaton (A_{S_1}) shown in Fig. 6.16. For the use case described by the guarantee automaton A_G of Fig. 6.16, the test T and the verdict function V should be those described also in Fig. 6.16. Test T was derived based on the rules presented in Sec. 4.6.3 and the verdict function based on the rules defined in Sec. 4.6.1 which should take into

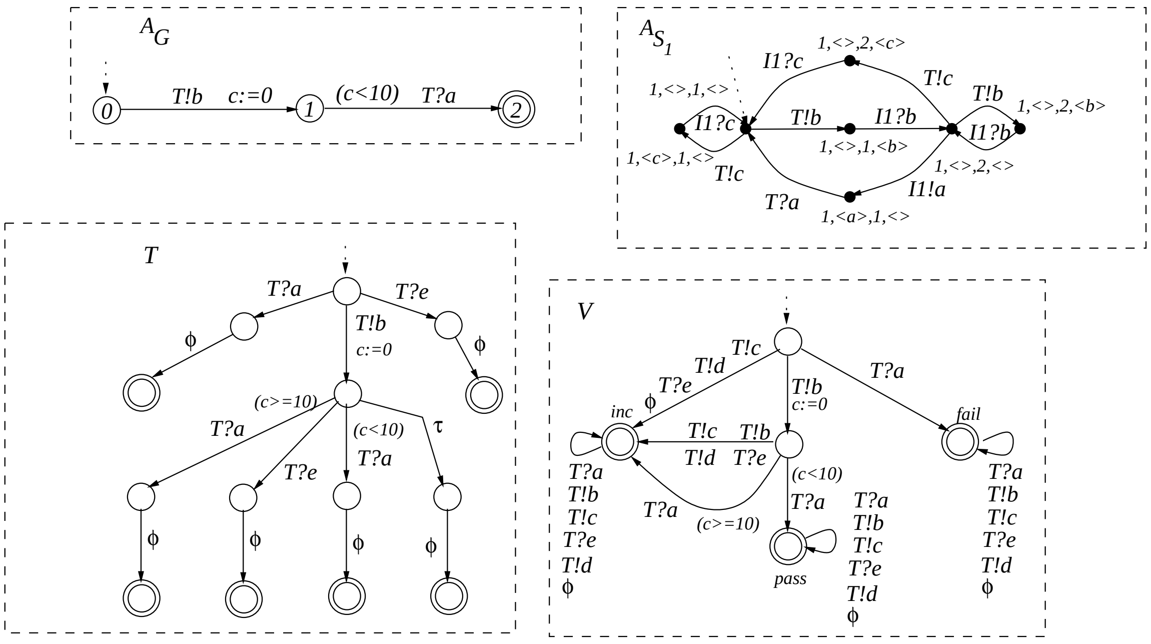


Figure 6.16: Test example - automata

account the verdict interpretation rules presented in Sec. 4.4.1.

Let us now assume that the process I_1 is modified so that it can be able to log all its user visible events, as shown in Fig. 6.17. This logging is represented by the output of some signals. The output of signal $InC(now)$, for instance, represents the timestamped log of $I_1?C$, while the output of $OuA(now)$ represents $I_1!A$. The output of signal $F(now)$ represents the end of trace event ϕ, generated by the test T which, in this case, is directed to process I_1.

In this case, the test T and the verdict function V can be represented in SDL as shown, respectively, in Fig. 6.18 and Fig. 6.19. Now, instead of the test events $T!c$ and $T?a$, for instance, the verdict function uses the events of process I_1, $I_1?c$ and $I_1!a$, which are represented by $InC(now)$ and $OuA(now)$ of Fig. 6.17. This substitution is justified by the fact that, in a synchronous model, the safety automaton generated according to the rules defined in Sec. 4.3.1 (Synchronous communication) will always model these interactions as a sequences of consecutive send (!) and receive (?) events. In this case,

$$s_i \xrightarrow{T!c} s_j \xrightarrow{I_1?c} s_k \xrightarrow{InC} s_l, \quad s_i \xrightarrow{I_1!a} s_j \xrightarrow{T?a} s_k \xrightarrow{OuA} s_l$$

If the 3 transitions are assumed to be instantaneous, we can assume that the 3 events would be timestamped with the same value. Taking this into account, for the verdict and trace evaluation functions the events $T!c$ and $T?a$ can be replaced by their counterparts InC and OuA.

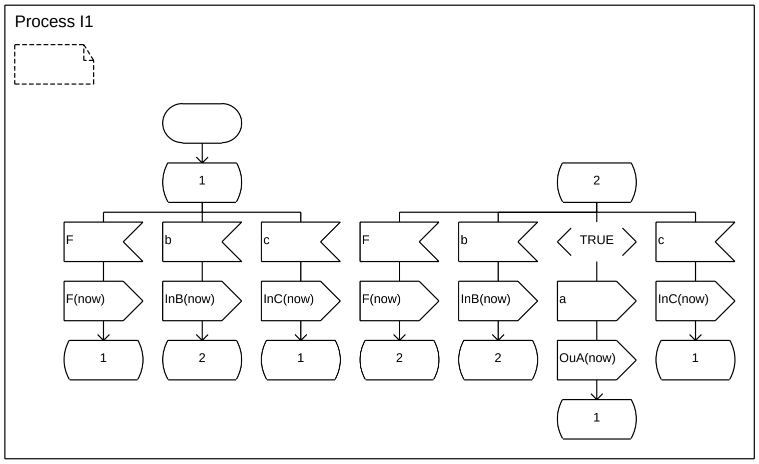


Figure 6.17: I_1 test process modified for log procedures

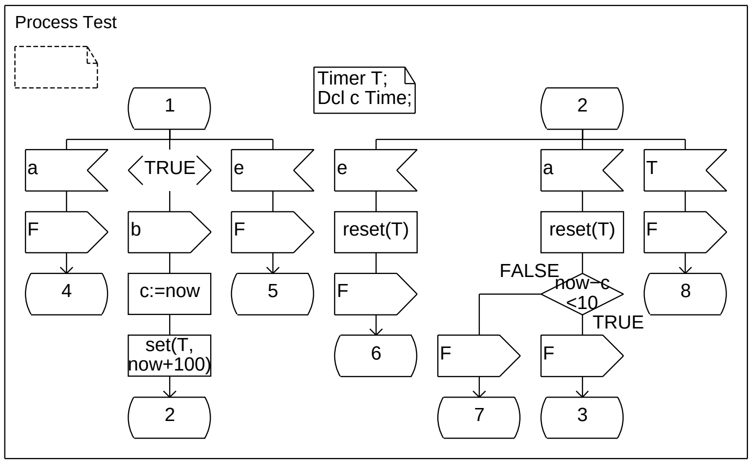


Figure 6.18: Test represented in SDL

The overall test case architecture is presented in Fig. 6.20. In this case, the verdict function is represented ($Verd$) as well as two trace evaluation functions, $Eval_1$ and $Eval_2$, used for cyclic service tests. The $Manager$ process is in charge of controlling the test execution (start, stop, getting results from the trace evaluation functions and provide them externally). The $Demux$ process is in charge of receiving the service component log information and of delivering it to the trace evaluation functions. Note that SDL processes are not heavy entities like, for instance, Unix processes. Typically a set of SDL processes are translated into a single executing process (e.g. Unix process), so this multiplicity of processes is natural in SDL.

With this solution, it is possible to implement simple synchronous service

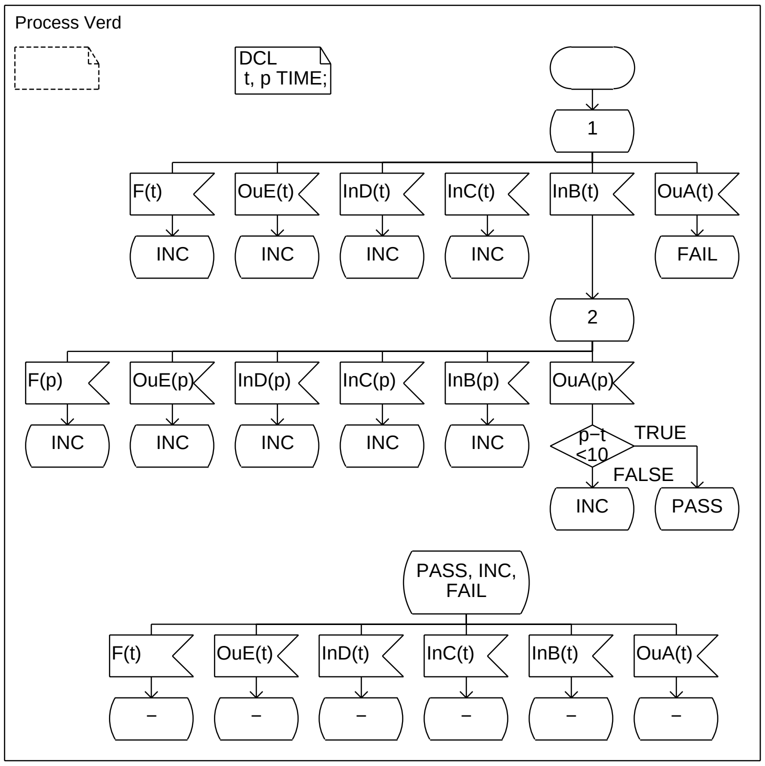


Figure 6.19: Verdict function represented in SDL

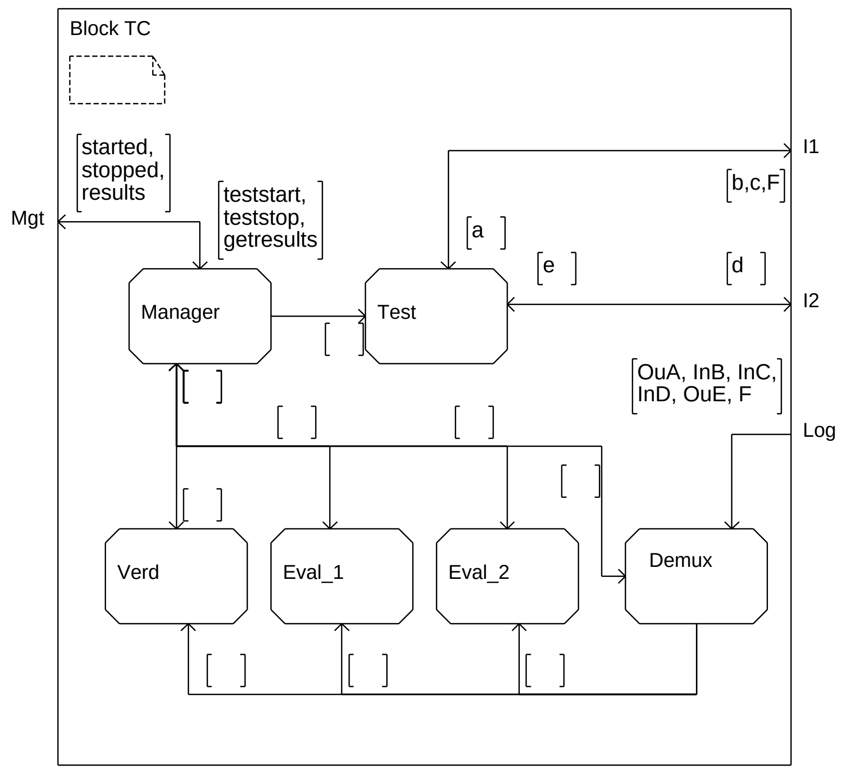

Block TC
started, stopped, results
teststart, teststop, getresults
Mgt
I1
b,c,F
a
e
d
I2
Manager
Test
OuA, InB, InC, InD, OuE, F
Log
Verd
Eval_1
Eval_2
Demux

Figure 6.20: Service test architecture in SDL

tests in standard SDL and, by doing that, to benefit from the SDL advantages. Another advantage of describing tests in pure SDL is that they can be easily integrated with the other test system parts, such as signalling stacks and management components, which were also described in SDL.

Using this approach, tests may drive the service component incorrectly. However, as shown by the telephone example, the probability of such misleading situations occurring can be low and, on the other hand, the implementation traces will always be correctly evaluated. When detected, a problem can be traced in order to verify if the fault has been caused by the service component under evaluation or by the test, due to the queues effects.

6.4.2 Timed actions

The hybrid solution proposed solves the SDL time problems as well. In the methodology proposed tests critically depend on time. It means that (1) the time of occurrence of every test event must be known and (2) some test events may be required to occur at precise time instants or within some time boundaries, e.g. $c = 15$ or $15 < c < 18$. While for the first requirement SDL provides the construct *NOW* which gives the current global time value, for the second, SDL provides no solution. In fact, the SDL solution for accounting time intervals is to set up a timer which, after expiring, sends a signal to a process queue and will be received latter.

In the proposed solution, tests are allowed to be *not time precise*. In fact, although current real time operating systems use time information to schedule processes, they are not capable of guaranteeing actions at precise time instants. In the proposed solution tests will be as good as the operating system. If CPU loads are low, time can be precise. If not, some lack of precision will occur. However, by using the service component log information, the results of the trace evaluation functions will always be correct. In case of problems, the trace can be inspected and the cause assigned to the test or to the implementation.

6.5 Examples of Service tests

Three examples of use cases are provided in this section: 1) PSTN Port operation; 2) Path originating call; 3) Path terminating call.

According to the test method proposed in Chap. 4, each use case must first be defined as a guarantee automaton (A_G) and each service component represented by a safety automaton which, in the Access Network case study, is assumed to be incompletly defined, that is, the *Interface component* defined in Sec. 4.2.3. Also according to the method, every trace accepted by the guarantee automaton should also be a trace of the corresponding safety automaton and define complete loops in it - from the initial state back to the initial state (Sec. 4.2.3, Test derivation).

6.5.1 PSTN Port operation

A possible use case for the PSTN Port service component (Fig. 6.5), which drives the composed safety automaton from its initial state back to the initial state, could be creating, unblocking, blocking and deleting a port where the actions at the AN and LE management interfaces are interleaved. This use case can be represented by the guarantee automaton A_G described in Fig. 6.21 and accepting the following timed traces:

$$
\begin{array}{rll}
timAccept(A_G) = & \{ & AN!createport(t_1).\ AN?portcreated(t_2).\\
 & & LE!createport(t_3).\ LE?portcreated(t_4).\\
 & & AN!unblock(t_5).\ LE!unblock(t_6).\\
 & & LE!block(t_7).\ AN!block(t_8).\\
 & & \underline{LE?portblocked(t_9)}.\ \underline{AN?portblocked(t_{10})}.\\
 & & LE!deleteport(t_{11}).\ AN!deleteport(t_{12}),\\
 & & \\
 & & AN!createport(t_1).\ AN?portcreated(t_2).\\
 & & LE!createport(t_3).\ LE?portcreated(t_4).\\
 & & AN!unblock(t_5).\ LE!unblock(t_6).\\
 & & LE!block(t_7).\ AN!block(t_8).\\
 & & \underline{AN?portblocked(t_9)}.\ \underline{LE?portblocked(t_{10})}.\\
 & & LE!deleteport(t_{11}).\ AN!deleteport(t_{12})\\
 & & \mid\ t_7 - t_6 > 15\ \wedge\ t_{11} - t_{10} > 15\ \}
\end{array}
$$

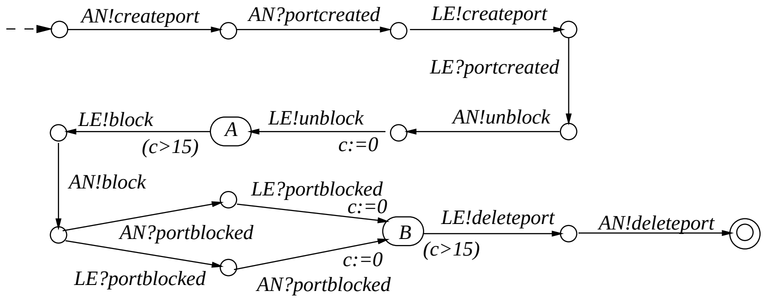


Figure 6.21: PSTN Port use case guarantee automaton

A test is derived for this use case by applying the rules presented in Sec. 4.6. A note must be introduced here: while a test derived from this use case is able to drive the component through a sequence of steps, in fact this may not happen if, for instance, the blocking and unblocking procedures did block/ unblock the path since this can be evaluated only by observing the Path service component. In order to complete the test it is necessary to verify if, for instance, (1) a path cannot be established when the PSTN Port is blocked and (2) a path can be established when the PSTN Port is unblocked. This subject will be addressed in detail after presenting some Path service component tests.

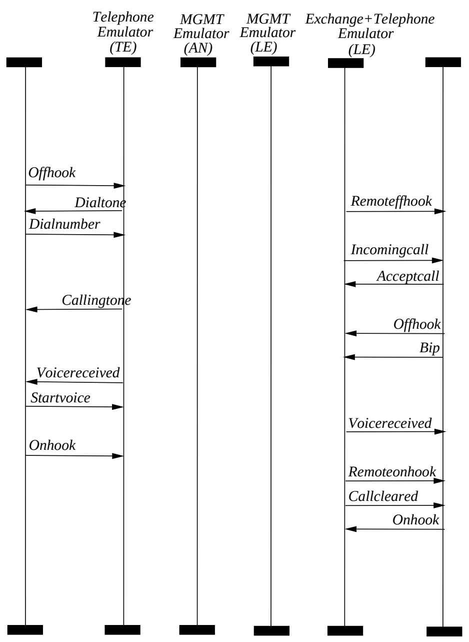


Figure 6.22: Call initiated by terminal equipment

6.5.2 Path originating call

Several use cases can be envisaged for the Path service component (Fig. 6.8). A possible use case could be allowing a call to be established by the terminal equipment side, tones to be transmitted bidirectionally and, at the end, the path to be terminated by the terminal equipment side. These calls are referred as originating calls (the terminal equipment originates the call). Fig. 6.22 gives the main sequence of messages which describes this component usage. This use case can be represented by a guarantee automaton A_G which is shown in Fig. 6.23 and accepts the following traces:

$$\begin{array}{ll} timAccept(A_G) = & \{ \\ & TE!Offhook(t_1).LE?Remoteoffhook(t_2). \\ & TE?Dialtone(t_3).TE!Dialnumber(t_4). \\ & LE?Incomingcall(t_5).LE!Acceptcall(t_6). \\ & TE?Callingtone(t_7).LE!Offhook(t_8). \\ & LE!Bip(t_9).TE?Voicereceived(t_{10}). \\ & TE!Startvoice(t_{11}).LE?Voicereceived(t_{12}). \\ & TE!Onhook(t_{13}).LE?Remoteonhook(t_{14}). \\ & LE?Callcleared(t_{15}).LE!Onhook(t_{16}) \ \mid \\ & t_3 - t_1 < 10 \ \wedge \ t_7 - t_4 < 20 \\ & \} \end{array}$$

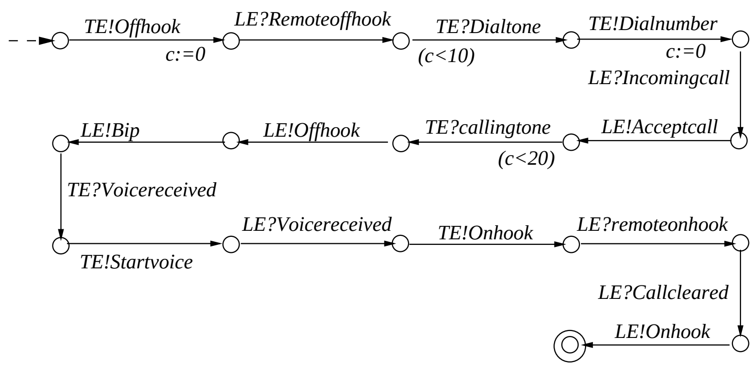


Figure 6.23: Originating call guarantee automaton

This guarantee automaton has been specified to give an example of two time constraints - dial tone should arrive in less than 10 seconds after the $TE!Offhook$ event and the calling tone in less that 20 seconds after the last digit is sent by the terminal equipment. The derivation of the test is simple and follows the rules defined in Chap. 4. The test is presented in Fig. 6.38 (at the end of the chapter).

Note that a path can be established only if the PSTN port is previously created (associated to a V5 interface) and unblocked.

6.5.3 Path terminating call

In the third example, the Path terminating use case, the call is established by the Local Exchange, tones are transmitted bidirectionally and, at the end, the path is terminated by the terminal equipment. Fig. 6.24 gives the sequence of messages exchanged in this case. This use case is represented by the guarantee automaton A_G, accepting the following traces:

$$\begin{array}{rl} timAccept(A_G) = & \{ \\ & LE!Offhook(t_1).TE?Ring(t_2). \\ & TE!Offhook(t_3).LE?Remoteoffhook(t_4). \\ & TE!Bip(t_5).LE?Voicereceived(t_6). \\ & LE!Startvoice(t_7).TE?Voicereceived(t_8). \\ & TE!Onhook(t_9).LE?Remoteonhook(t_{10}). \\ & LE?Callcleared(t_{11}).LE!Onhook(t_{12}) \ \mid \\ & t_2 - t_1 < 5 \\ & \} \end{array}$$

The guarantee automaton has been specified with one time constraint - ring should arrive in less than 5 seconds after the $LE!Offhook$ event.

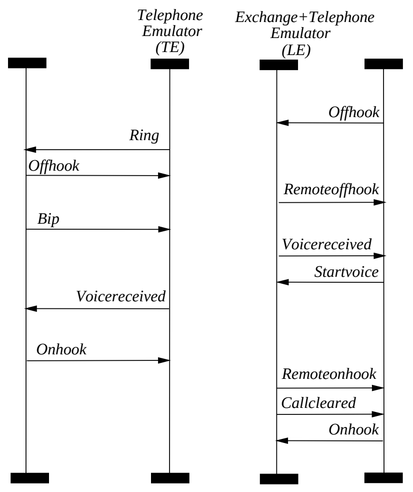


Figure 6.24: Call initiated by the local exchange

6.5.4 Components interactions

Certain operations over some service components may influence the state of neighbour components. In this case, tests should be improved so that the affected component may be also observed. In this section the problem will be addressed for the cases identified.

PSTN Port use case

Two problems were identified in the PSTN Port use cases presented: the unblocking and the blocking of the port under test was not really verified. A solution is to define small use cases over the corresponding Path service component which, coordinated with the main PSTN Port use case, will allow the test to infer about the (un)blocking of the port. Fig. 6.25 presents a possible solution based on two small Path use cases (verify port unblocking and verify port blocking).

The Path unblocking use case verifies if, when unblocked, an offhook, a tone and an onhook can pass through the Access Network and emulation components. The Path blocking use case verifies if, when the port is said to be blocked by the PSTN Port management, the Path component does not exist. In this case, the test tries to establish the Path but expects no dial tone, which, in this case, is signalled by the call aborted message at the terminal equipment side. Note that both Path guarantee automata of Fig. 6.25 (verify port unblocking and verify port blocking) define complete loops over the Path component safety automaton, as required by the complete assumption of Chap. 4. The tests can, is this situation, be also derived according to the rules defined in Chap. 4.

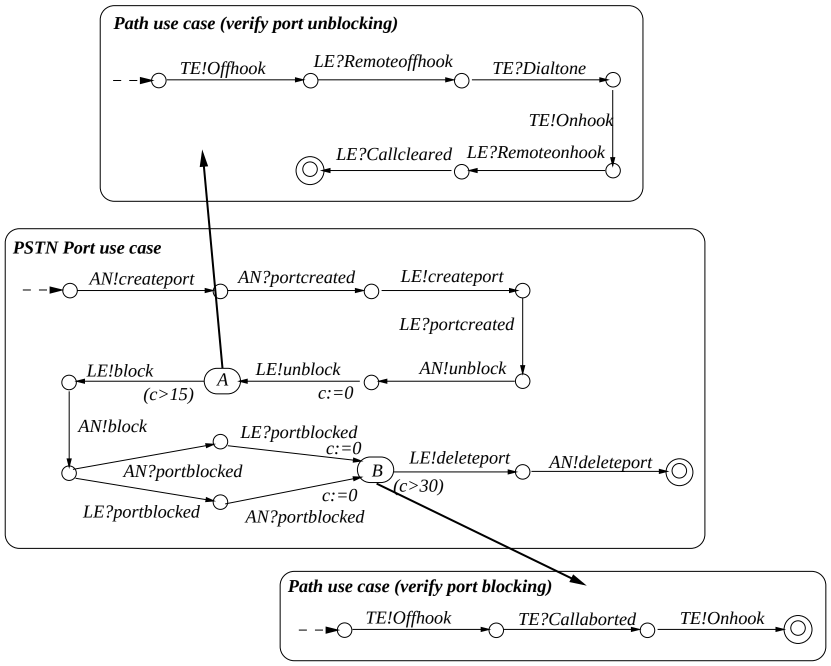


Figure 6.25: Coordinated use cases

The execution of the PSTN Port test implies the coordinated execution of three *parallel* testers: one for the PSTN port, one for the Path unblocking verification procedure and the last for the Path blocking procedure. Fig. 6.26 shows their architecture. Note that while the Path tests are executing, the PSTN Port test will continue to monitor possible answers of the PSTN Port service implementation.

At the end, three verdicts will be obtained. The final verdict will take them all into account.

Originating call use case

In this case, a test is defined for evaluating the establishment of a call from the terminal equipment side. The Path service component, however, does not exist before the PSTN port is created and unblocked, hence the Port must be previously unblocked.

The solution for this problem consists in defining, again, a use case/ test for the PSTN port which creates and unblocks the port, allows the Path test to be executed and, at the end, blocks and deletes the port. Fig. 6.27 gives an example of these tests by presenting the use cases from which they could be derived.

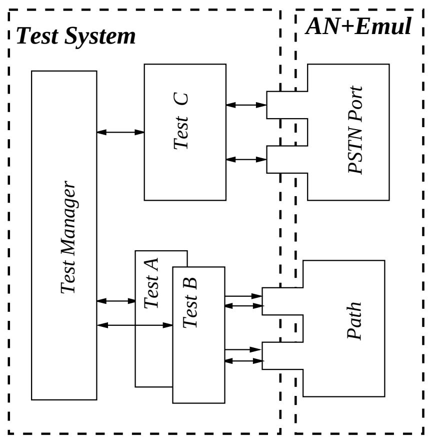


Figure 6.26: Coordinated test architecture

6.6 Path component cyclic tests

According to the method proposed, cyclic tests are used in Network Element testing, that is, in the joint evaluation of the service components (Sec. 4.7.3). The main objectives of this type of tests are to get information on the probabilistic behaviour of the service component implementation and, using that information, to statistically characterise the service component faults and its quality. For this reason, the service component have to be excited with tests that must be repeated a reasonable number of times. In the methodology proposed, the tester controlled behaviour was required to be deterministic so that the statistical trace evaluation could reflect only the service component probabilistic aspects (Sec. 4.7.3, step 3). The components which are not evaluated can have random tests(Sec. 4.7.3, step 5). In order to allow these tests to emulate real Network Element loads, the service mix and service usage must emulate, as closely as possible, expected Network Element traffics.

In the V5 Access Network presented, some service components were identified: the Access Network which aggregates the Interface, PSTN Port and ISDN Port components. The PSTN Port, in turn, contains the Path component while the ISDN port contains one digital access which aggregates the 2B+D communication channels.

When trying to define real Access Network load or usage scenarios some questions arise immediately. How many V5 interfaces will the network typically have? How many PSTN ports? How many ISDN Ports? How often will a PSTN port be blocked? What type of calls will it support and what are their characteristics? How often will an ISDN Port be blocked? How often should access activation/deactivation take place and how should it be modeled? How to characterise statistically the HDLC frame generation process through the ISDN D channel when the access is deactivated?

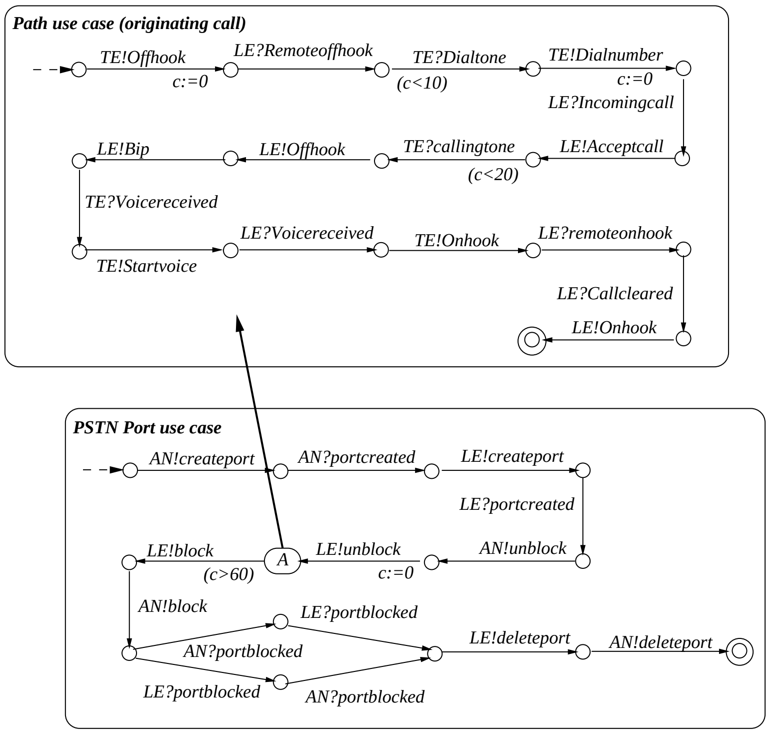


Figure 6.27: Coordinated tests for the originating call

In this section, the Path component will be studied in detail. For the other components the method for defining tests, instance mixing and trace evaluation functions would be similar.

6.6.1 Usage models

The characterisation of a Path service component usage can be made based on standard telephony traffic models. This traffic is characterised in terms of traffic intensity A $(Erlang)$, and call holding time d_m (s), as discussed in Chap. 3. Let

$$\frac{1}{\mu} = d_m \quad \frac{1}{\lambda_{idle}} = \frac{1-A}{A} \times d_m$$

where μ represents the average service rate and λ_{idle} the call arrival rate during idle periods.

The telephony loads considered relevant for the NEC Access Network were the PSTN upper/ increased loads [77] and the V5 upper/ increased loads [108]. These concepts were also introduced in Chap. 3. In addition the maximum telephony load expected by the NEC Access Network was also defined and considered a requirement, as well [109], [110]. Tab. 6.1 sumarises the characteristics of these five models.

Traffic Model	A (Erl)	$1/\mu$ (s)	$1/\lambda_{idle}$ (s)
$PSTN_{upper}$	0.17	90	439
$PSTN_{increased}$	0.21	90	339
$V5_{upper}$	0.7	90	36
$V5_{increased}$	0.84	90	17
$FA1201$	0.83	20	4

Table 6.1: PSTN user port traffic model

6.6.2 Quality of Service

The required quality of service provided by the Path service component can also be inferred from standard QoS specifications. For example, [108] defines *signalling delay* as the time taken by the Access Network to transfer a signal when no other action is required. With respect to PSTN, two test points are defined: the Z and $V5$ interfaces. At the Z interface, entry and exit events occurrences are defined as the time instant when the current falls/ raises below/ above threshold values. At the V5 interface, the occurrence of an entering event is determined by the observation of a relevant message closing flag in a C channel where an exit event is said to occur when the message opening flag is observed at the C-channel. Like the performance specifications presented in Chap. 3, maximum mean ($maxmean$) and 95 percentiles ($max95$) are specified for some reference loads, as shown in Tab. 6.2. Some

Transfer delay		Traffic Model	
		$V5_{upper}$	$V5_{increased}$
Signalling transfer	$maxmean$ (ms)	100	175
	$max95$ (ms)	200	350
Processing non-intensive	$maxmean$ (ms)	200	350
	$max95$ (ms)	400	700

Table 6.2: V5 signalling transfer delay

failure parameters are also defined in [108] as follows:

- **Premature release.** The probability that an Access Network malfunction results in the premature release of an established call in any one minute interval, which should be $P \leq 2 \times 10^{-5}$;

- **Release failure.** The probability that an Access Network malfunction prevents the release of a connection, which should be $P \leq 2 \times 10^{-5}$;
- **Other failure.** The probability of the Access Network causing a call failure for any other reason not identified above, which should be $P \leq 2 \times 10^{-4}$.

According to [77], which defines a set of QoS parameters for digital exchanges, some other parameters can be considered for the definition of the quality required for the Path service component:

- **Call request delay.** The interval from the instant the offhook condition is recognized at the subscriber line interface until the exchange begins to apply the dial tone to the line;
- **Call indication delay.** The interval from the instant the last digit of the called number is available for processing at the exchange, until the instant of the ringing signal is applied to the called subscriber line;
- **Exchange call release delay.** The interval from the instant the last information required for releasing a connection is available for processing the exchange up to the instant that the switch trough-connection becomes unavailable.

6.6.3 Originating call use case

Cyclic test

A cyclic test for the Path component can be defined based on the Originating call test defined in Sec. 6.5.2. This test (shown in Fig. 6.38) and whose use case was presented in Fig. 6.23, must now be made cyclic according to the rules of Sec. 4.6.4. For this reason, the reliable reset sequence $T!\rho$ must now be replaced by meaningful sequences. Assuming that:

1. simple service tests were applied before, which implies that immediate and deterministic component faults were already detected and solved;
2. the service component interfaces are known (code is available for inspection) since they reside inside the test system and particular attention has been payed to its validation,

only the events labelling the alternative transitions of the component interface safety automata need to be considered in the derivation of the test. This simplification, which was not considered in step 2 of the general solution presented in Sec. 4.6.4, is very efficient from the test length point of view (number of transitions). On the other hand, it avoids the difficult task of finding reset sequences for events not predicted in the component safety automaton. This simplification is exemplified in Fig. 6.39, where the reset sequences are shown for states 1 and M of the test in Fig. 6.38. For state 1, for instance, only two alternative transitions were considered in the test: $s_1 \xrightarrow{TE?Ring}$ and $s_1 \xrightarrow{LE?Remoteoffhook}$ which correspond to the alternative transitions in the component safety automaton, shown in Fig. 6.9 and Fig. 6.10. The test presented in Fig. 6.39 must also be timed with respect to the call duration CD and to the intercall interval IC, according to the following values:

$$CD = \frac{1}{\mu} \quad IC = \frac{1}{\lambda_{idle}}$$

CD and IC are constant values as defined by the method in Sec. 4.7.3, step 3, so that the test is made deterministic.

Trace evaluation functions

Five trace evaluation functions (Sec. 4.5.2 and 4.6.5) were defined for this use case: *1)* call establishment; *2)* tone transmission; *3)* call termination; *4)* aborted calls; *5)* refused calls.

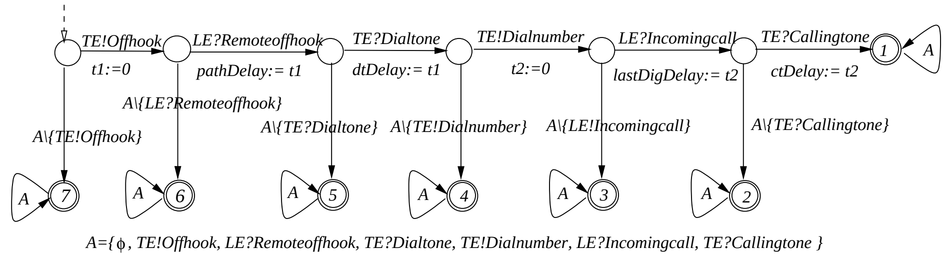


Figure 6.28: Call establishment evaluation function

The *call establishment* function, presented in Fig. 6.28, evaluates how many traces whose first event is a $TE!Offhook$ result in a call correctly established. In addition, relevant delays are also defined. *pathDelay* describes the delay since the subscriber offhooks until the Local Exchange perceives this information. *dtDelay* models the dial tone delay. *lastDigitDelay* models the delay associated with the transference of the last digit of a dialed number through the Access Network (decadic dialing is assumed). *ctDelay* models the calling tone delay, that is, the time interval since the subscriber dials the last digit until the corresponding tone is heard.

The *tone transmission* function, presented in Fig. 6.29, says that every call

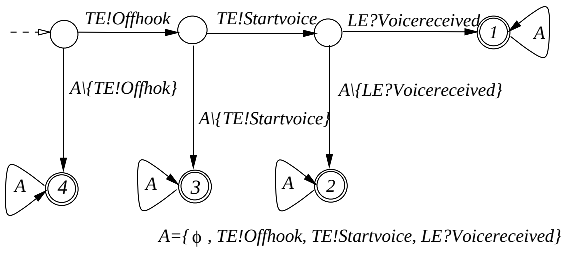


Figure 6.29: Tone transmission evaluation function

establishment, represented by $TE!Offhook$ event, must be followed by the transmission of a tone from the Terminal Equipment to the Local Exchange.

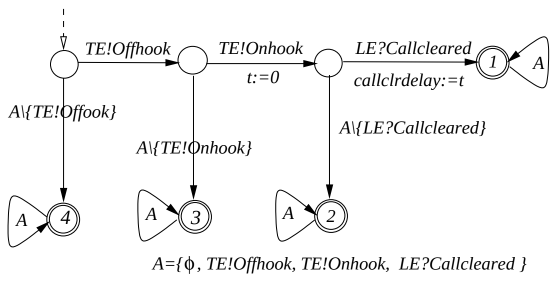


Figure 6.30: Call termination evaluation function

The *call termination* function, presented in Fig. 6.30, states that calls should always be terminated gracefully, that is, with a $LE?Callcleared$ event.

The *call aborted* function, presented in Fig. 6.31, evaluates the number of times a call has had problems. The terminal equipment emulator was designed to send this event if a dial tone, a calling tone or a voice indication were not received when expected, as defined in Fig. 6.9. Since different time intervals reveal different types of failures, a clock t was associated to the automaton of Fig. 6.31.

The *call refused* function, presented in Fig. 6.31, is also an example of a function evaluating faults which are related to the transfer of digits across the Access Network. When the number contained in *LE?Incomingcall(number)* differs from the number dialed in *TE!Dialnumber(number)* the use case is violated and the reset sequence will include a $LE!Refusecall$ event. This evaluation function accounts for this type of situations.

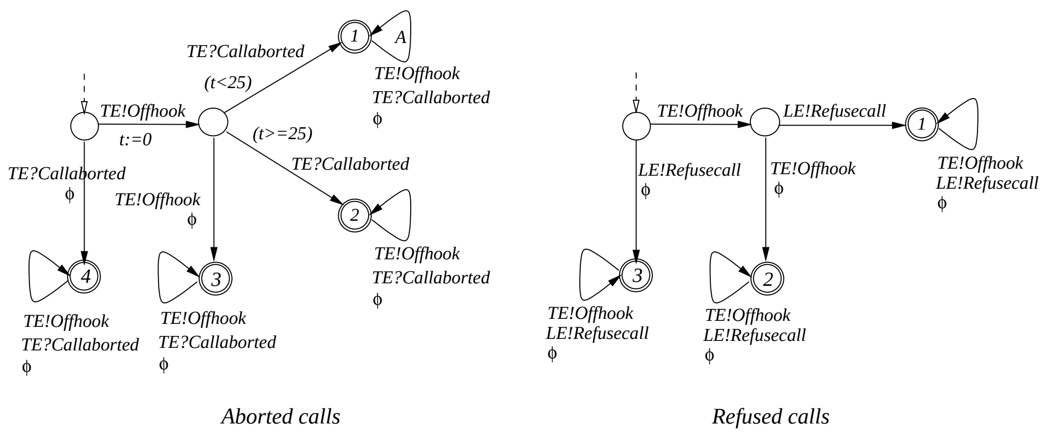


Figure 6.31: Call aborted and call refused functions

6.6.4 Terminating call use case

Cyclic test

Another cyclic test for the Path component can be defined based on the Terminating call test defined above (Sec. 6.5.3). The test must also be timed with respect to the call duration CD and to the intercall interval IC, according to the following values:

$$CD = \frac{1}{\mu} \quad IC = \frac{1}{\lambda}$$

Trace evaluation functions

Four trace evaluation functions were defined for this probabilistic use case, as shown in Fig. 6.32: *1)* call establishment; *2)* tone transmission; *3)* call termination; *4)* aborted calls;

The *call establishment* trace function evaluates the number of $TE?Ring$ events which follow $LE!Offhook$ event, that is, the call establishment phase. *ringDelay* describes the delay between these events, when they are observed in this sequence. The *tone transmission* function evaluates the bidirectional exchange of tones. The *call termination* function evaluates the call termination phase which is said to be terminated gracefully when a $LE?Callcleared$ is observed. The *call aborted* function is used to evaluate failure situations, here represented by the $LE?Callaborted$ event. Since different time intervals may describe different faults, a clock t was associated to the automaton.

6.6.5 Path component random usage

Cyclic tests, as referred above, are supposed to be deterministic with respect to the test controlled events. When carrying out Access Network joint service

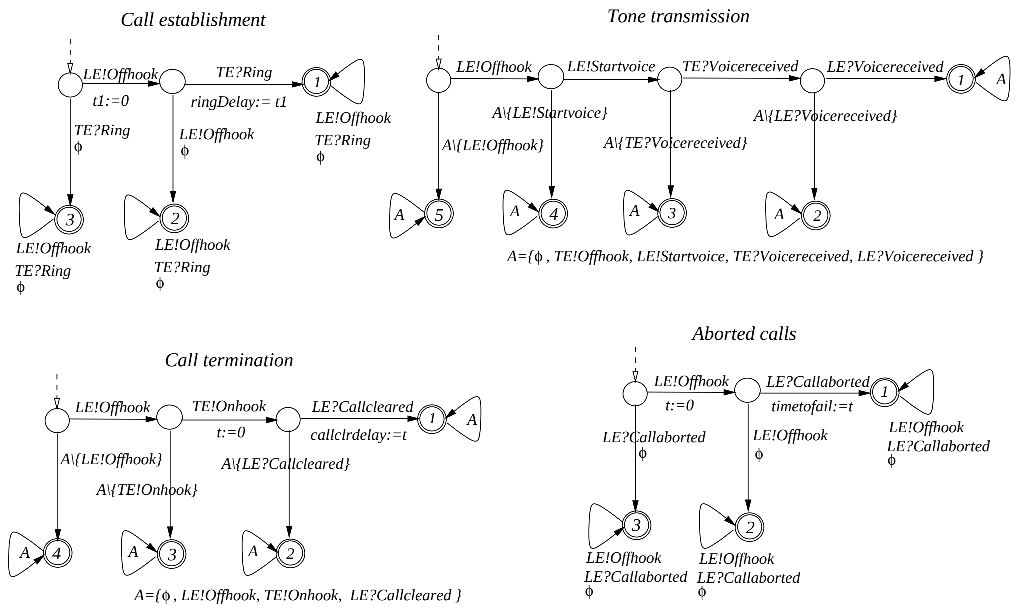


Figure 6.32: Call terminating trace functions

tests, as defined in Sec. 4.7.3, it is not required to evaluate every Path service component instance. The instances not evaluated are used with random tests. A possible random test may combine originating and terminating calls with a 50% probability each. In addition, call duration and intercall interval are now assumed to be exponentially distributed according to the following functions,

$$CD = -\frac{1}{\mu} ln(u) \quad IC = -\frac{1}{\lambda} ln(u); \quad u \in]0,1]$$

where, now, CD and IC take values which are variable and have to be recalculated everytime a constraint is in use (e.g. $icinterval \geq IC$, Fig. 6.39).

6.6.6 V5 Access Network load tests

In the methodology proposed, after being individually evaluated (one instance of each service) these service components must be jointly evaluated with cyclic tests. A joint V5 Access Network test is characterised by the following attributes, as defined in Sec. 4.7.3:

- *User ports.* The number of PSTN and ISDN ports provisioned. A typical example can be an Access Network supporting 480 PSTN ports and no ISDN ports;

- *Sample ports.* The number of ports which will be evaluated. A typical figure can be 10% of the ports;

- *Traffic model.* The call/ packet traffic which will be used to load the Access Network in this configuration. The $V5_{upper}$ traffic model can be used which means that $1/\mu = 90\ s$ and $1/\lambda_{idle} = 36\ s$, according to Tab. 6.1;

- *Test distribution.* The PSTN ports not selected for testing can be loaded with the random PSTN usage assuming also the $V5_{upper}$ traffic model. The sample PSTN ports should be loaded either by an originating or a terminating call test. A 50%/50% distribution for the sample ports is a good example;

- *V5 Interfaces.* The number of V5.1 and V5.2 interfaces supported. A good example can be an Access Network supporting two V5.2 interfaces using the 16 links;

Based on combinations of these parameters several service joint tests can be defined. These will be considered part of the Access Network and applied to the Network Element every time a new equipment version is released.

6.7 Application of the methodology

First, it must be mentioned that the methodology proposed in Chap. 4, which constitutes the core of the thesis, is a generalisation of the methodology used to validate the FA1201 equipment. For this reason, the proposed methodology takes advantage of the experience gained from testing the FA1201 equipment.

In this chapter, the proposed methodology has been applied to the FA1201 case, so that it can be understood. Since, at the time the FA1201 system was tested, the methodology was not so clear, there are minor differences between what has been carried out and what has been described in the previous sections.

Next, these differences will be pointed out. Then, the FA1201 validation process will be described so that each type of test and its real meaning becomes even more clear. At the end, an overview of the service test equipment developed will also be given.

6.7.1 Differences

The differences between the correct application of the methodology and the real testing of the FA1201 system can be described as follows:

- **Service components.** Instead of the 8 service components presented in Fig. 6.1, the NEC FA1201 services have been modeled as only two components - the PSTN Port and the ISDN Port, which were also specified as interface components. The PSTN Port, for instance, instead of the two interfaces mentioned in Fig. 6.8, supported the management interfaces of Fig. 6.5 as well. The definition of 4 interfaces, however, made the concept of cyclic test, which is very important, much more complex. Instead of defining cycles through the component initial state, the cyclic test would have to use another state (e.g. unblocked state). The approach presented in Sec. 6.2 is, for that reason, more clean and enables the use of the generalised test methodology proposed in Chap. 4.

- **ISDN Port service component.** Although the ISDN Port component has been specified along with its tests and trace evaluation functions, there was no opportunity to apply them to an Access Network configuration provisioned with ISDN Ports. In order to avoid the description of unproved solutions, no ISDN examples are included, although they were similar to the PSTN solutions presented.

- **Test distribution.** While the tests presented throughout this chapter are non-distributed, in reality all tests were distributed by the Local Exchange and Terminal Equipment emulators. For that reason some simplifications on the tests were required, some tricks were introduced and complex synchronisation equipment/ functions had also to be developed.

6.7.2 FA1201 testing process

The first tests applied to the NEC FA1201 were the standard ETSI V5 conformance tests. V5.1 layer 2, V5.1 layer 3, V5.2 layer 2 and V5.2 layer 3 tests were performed in this sequence. These tests were used basically because they were available, they are required by the telecom operators and they address important parts of the system.

According to the methodology proposed in Chap. 4, the execution of conformance tests corresponds to the test of the communication components presented in Sec. 4.7.2. If, for instance, the V5 conformance tests were not available, a set of communication components should be identified for testing. These components, called in the proposed methodology the communication components, are those presented in Fig. 5.6 and Fig. 5.10. For most of them there are complete specifications available and therefore, according to the methodology, they could be tested either by use cases or by the state machine methods presented in Chap. 2. This decision would mainly depend on the time available for testing. Sec. 4.7.2 describes the communication component testing approach proposed in the methodology.

FA1201 service tests were addressed at the same time as conformance tests. The initial objective was to develop a simple V5 Local Exchange Emulator which could be used to excite the Access Network. However, the advantages of having control not only over the Local Exchange but also over the Terminal Equipment became evident and it was decided to include also 480 analogue telephone emulators and 120 ISDN emulators [111], [112], [113], [114] in the service test system. In this way, realistic loads could also be simulated. In order to include the service testing methodology in the emulators, services had first to be identified and their interfaces specified as the automata presented in Fig. 6.9 and Fig. 6.10. Based on the specification of the interfaces of the service components, which in the project were known as test interfaces, development of the emulators and the tests started.

The emulator tasks were implemented in SDL and C++ and built so that they rigorously complied with the service interfaces specified. Moreover, these emulators were developed so that the service component interfaces could be made available as an API (Application Programming Interface) [115], [116] which, in fact, was a set of C++ classes. The service interfaces were also made available to the test system graphical interface [117] in order to let humans interact directly with the service components.

Based on the interface specification of the service component, development of the tests started also. According to the methodology proposed, which was followed in the case of service testing, service components must be tested individually and jointly (Sec. 4.7.3). For the PSTN Port service component, for instance, several use cases were informally defined and the corresponding tests were applied using the service test system graphical interface. To be rigorous, the procedures presented in Sec. 4.6.2 and Sec. 4.6.3 would have to be applied.

The joint service tests (or NE tests), described in Sec. 4.7.3, were developed next. These tests require automatic execution since they need to run for hours. Four tests for PSTN Port and four tests for ISDN Port were implemented in SDL, using the service API previously defined. They are large and, for this reason, they are not reproduced in the thesis. Instead, their relevant aspects are presented in Fig. 6.38 and Fig. 6.39. These tests allow the configuration of some parameters (e.g. call duration, intercall interval, number of digits). Each of these parameters could be deterministic or probabilistically distributed with exponential or uniform distribution functions.

In addition, and for each test, a set of trace evaluation functions were defined according to the rules specified in Sec. 4.5.2 and Sec. 4.6.5. Fig. 6.28 to Fig. 6.32 show some of these trace evaluation functions. These properties were specified as annotated MSCs and, using a simple translator (developed by the author), a C++ class was automatically generated to allow the evaluation of individual traces. Fig. 6.20 shows the cyclic test architecture used. The results gathered by these functions were collected in histograms and presented graphically at the user interface.

Several NE tests were defined. Each of them basically reflects one expected configuration and one usage condition of the FA1201 system. After the execution of a NE test, a set of histograms were obtained. By inspecting them, several classes of faults could be identified. Typically, the most likely fault was selected for debugging and the traces obtained overnight were re-inspected with refined trace evaluation functions (PERL scripts) in order to better understand the pattern of the selected fault.

In some cases, the test would have to run again so that it could stop on the first occurrence of the most probable fault. This is the reason why trace evaluation functions need to run in real time - as soon as they reach a particular faulty state they may ask the test manager to stop the test, leaving the service component in the desired faulty state. Using this methodology the most likely service faults could have been eliminated first, satisfying the main project management expectations.

6.7.3 Service test equipment

The service test equipment is composed of two equipment - the V5 Local Exchange Emulator [118] and the Terminal Equipment Emulator [119], as shown in Fig. 6.33.

These emulators are connected to the V5 Access Network as well as to a PC which is in charge of controlling the execution of the tests, of getting test results and providing the graphical interfaces of the service components. This graphical man machine interface can be downloaded from

http://puma.inescn.pt:8080/test/results/load_mmi.html .

The two emulators are also connected by a synchronisation bus. This bus (TTS in Fig. 6.33) enables the Terminal Equipment Emulator to provide a clock and a reset line which is used in the Local Exchange Emulator to feed a timer, [120], [121]. In this way, the two emulators are synchronised with respect to time. In addition, the TTS bus provides a parallel data bus (8 bits) which is used to transfer log events from the Local Exchange Emulator to the Terminal Equipment Emulator, where the trace evaluation functions are located. The TPU (Tones Processing Unit) board is in charge of generating and detecting the tones exchanged by the emulators through the Access Network.

The emulation, test and trace evaluation functions run at the SCU board (Signalling Control Unit) of both systems. For the Local Exchange Emulator (Fig. 6.34), this board contains, among others, the V5 signalling stack (Local Exchange side) and parts of the tests. In the Terminal Equipment emulator

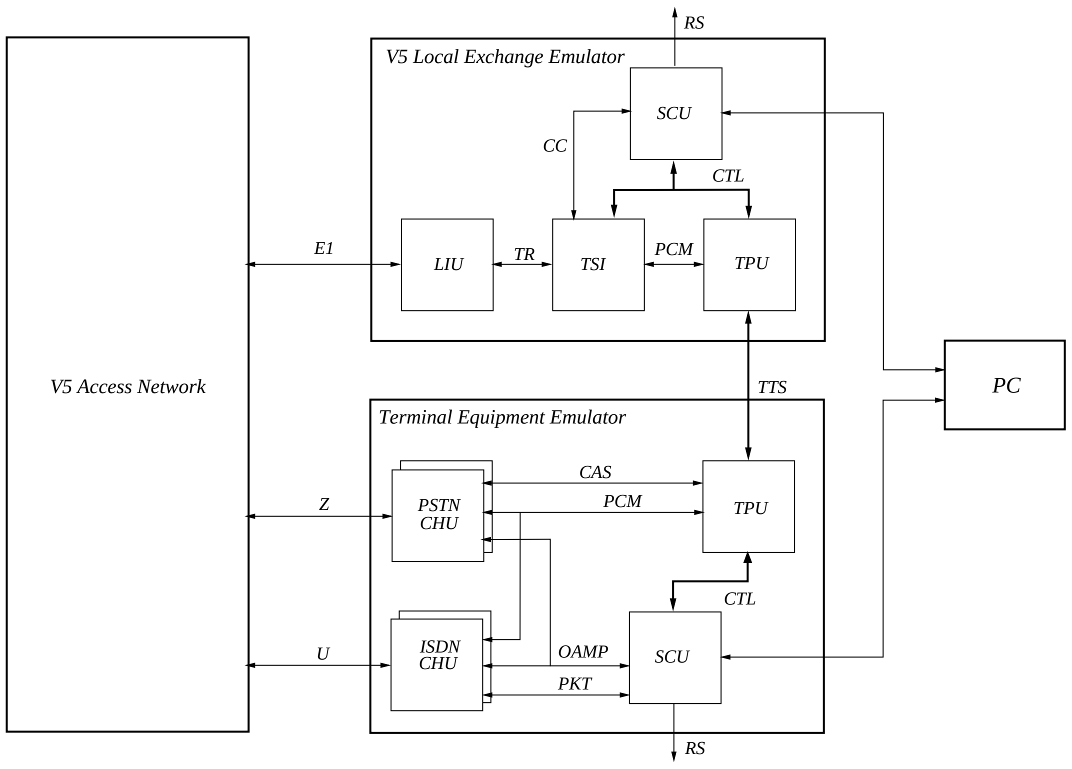


Figure 6.33: Service test equipment

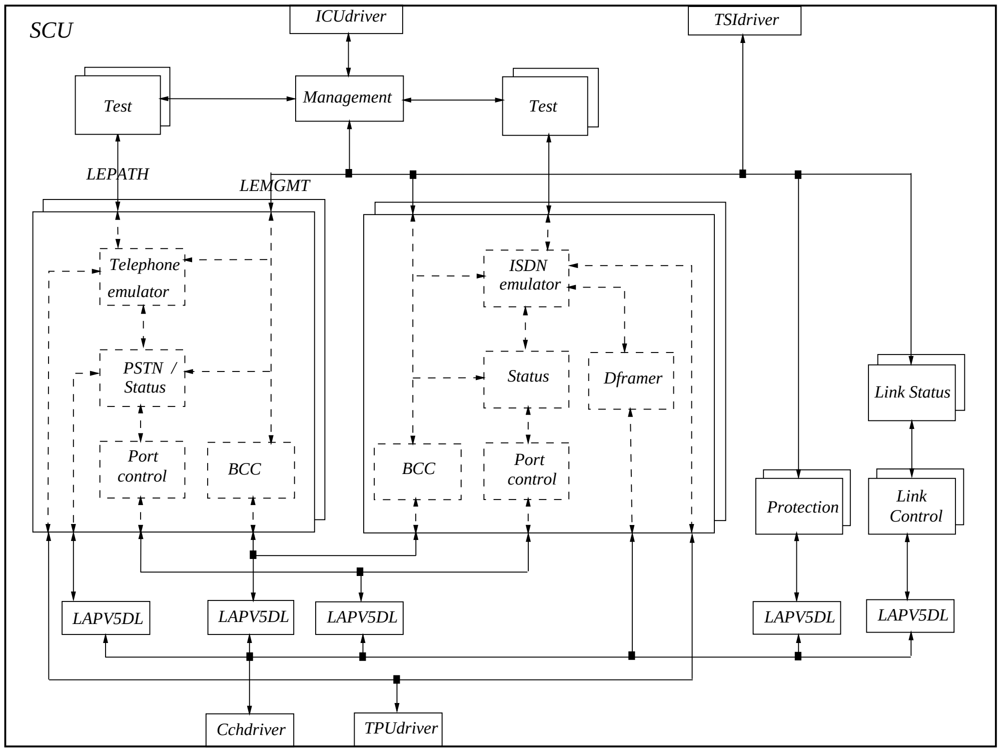


Figure 6.34: V5 Local Exchange Emulator (SCU board)

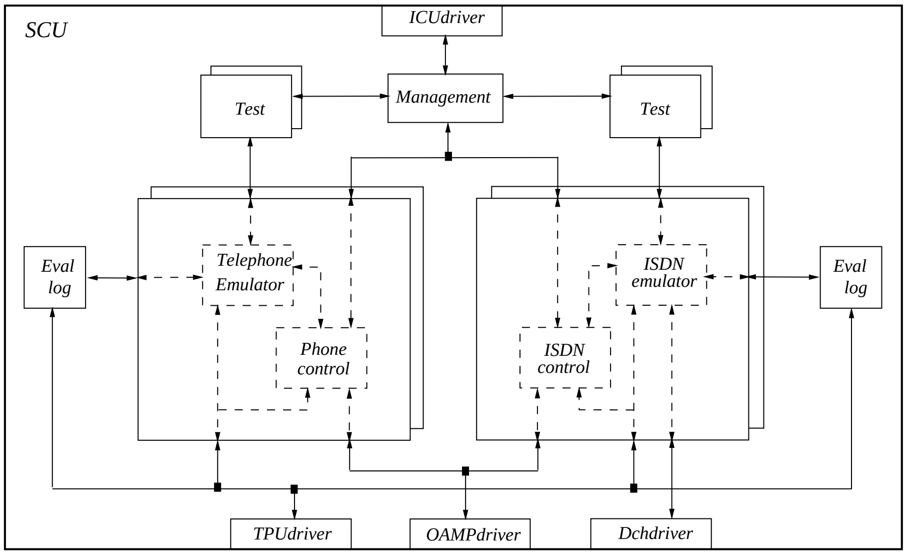


Figure 6.35: Terminal Equipment Emulator (SCU board)

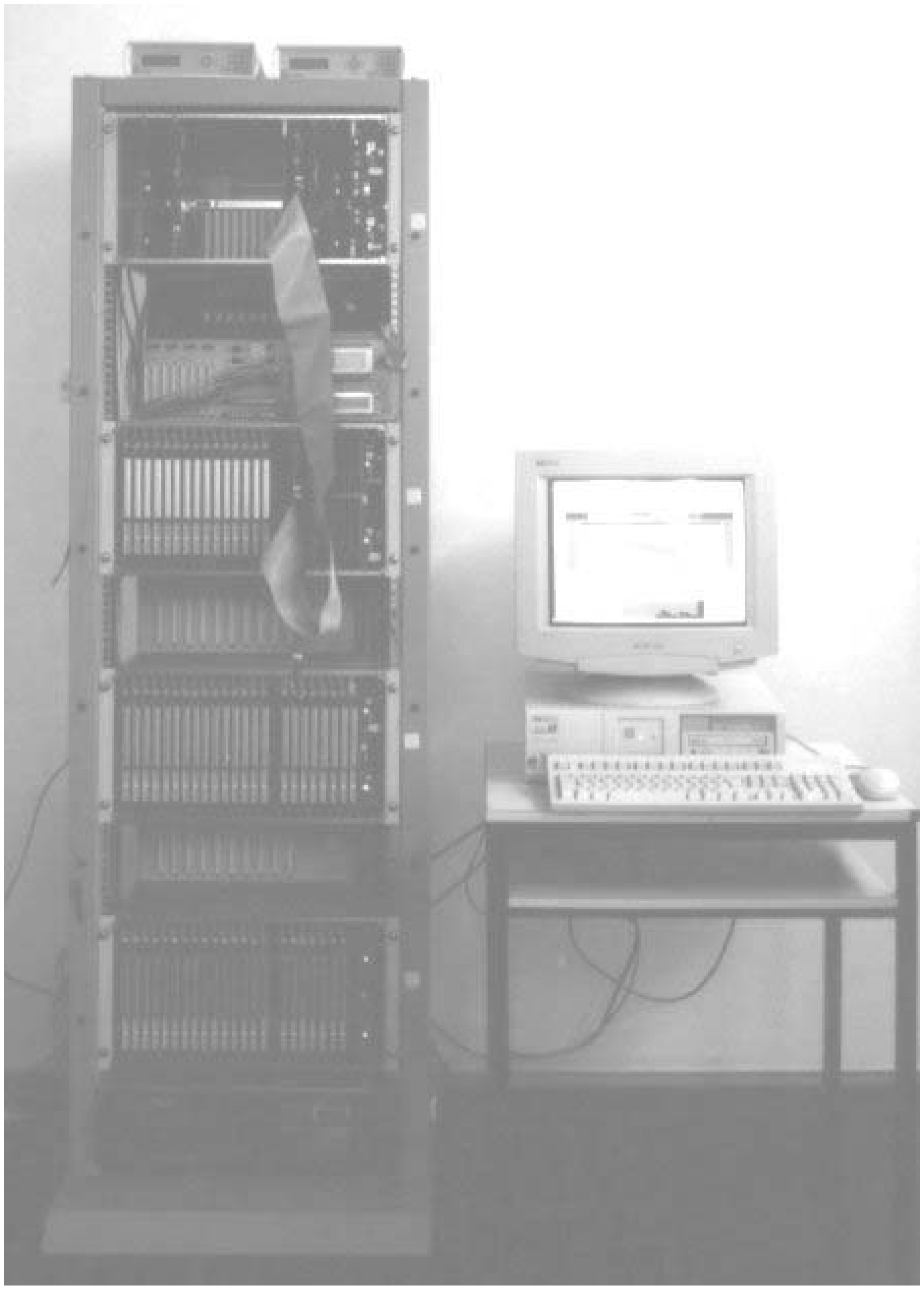


Figure 6.36: V5 service test system

this board contains the basic (layer 1) emulators, parts of the tests and the trace evaluation functions, as shown in Fig. 6.35.

The service test system is shown in Fig. 6.36 and documented in [114].

6.8 The value of the method

The validation of the FA1201 Access Network was carried out in the period 1996-1999 using a preliminary version of the testing method proposed in Chap. 4. About 1000 different tests were applied to the communications components (the V5 protocol conformance tests) and 50 service tests were applied to the service components [122].

6.8.1 Test of communications components

The ETSI V5 conformance tests used to validate the FA1201 communications components enabled the detection of about twenty faults. Due to the fact that these components were implemented in SDL (V5 specifications were available as simple SDL diagrams) faults were generally easy to detect and correct. These faults were mainly caused by the implemented components being more complex than their specifications, that is, containing more states and transitions.

Nevertheless, conformance tests proved useful, since the faults of communications components had impact on the equipment interoperability and, if not found, they could prevent the equipment from interworking with V5 local exchanges and terminal equipments. The results of the final complete V5 conformance testing campaign can be found at

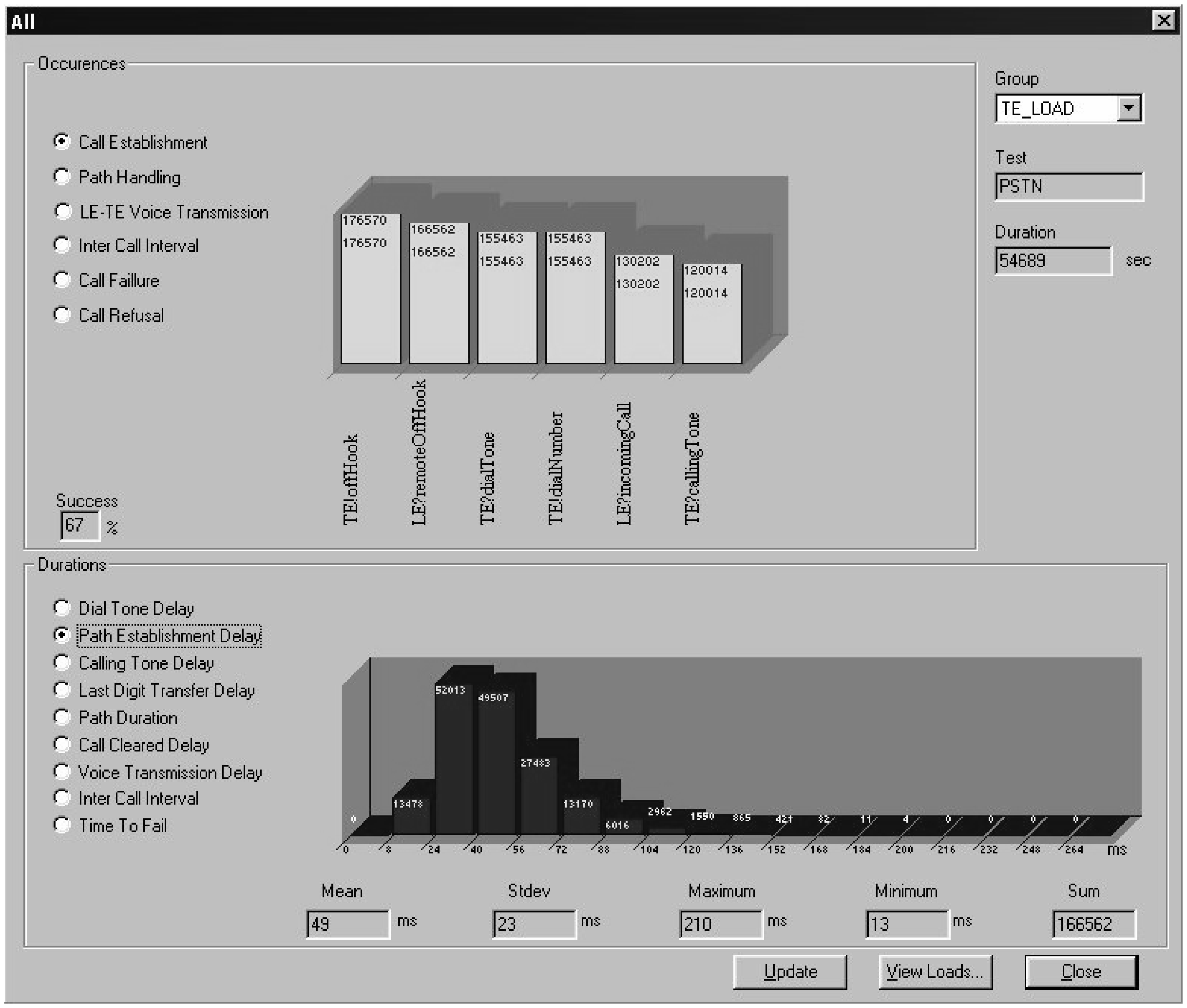
http://puma.inescn.pt:8080/start.htm.

The FA1201 equipment is at present free of communication faults, that is, compliant with the ETSI V5 standards.

6.8.2 Service component tests

Service component tests provided a complementary evaluation view of the system. The simple (non cyclic) service component tests which were applied to single service component instances using the graphical interface did not provide much fault information, since these tests basically replicated the tests previously carried out by the FA1201 developers.

Joint cyclic tests, however, proved very important. As an example, the first results of the Path trace evaluation function presented in Fig. 6.28 gave failure probabilities of about 70%, distributed among all states. On the other hand, *all* the failure states defined in the trace evaluation functions of the



Occurences: [✓ Call Establishment] Path Handling LE-TE Voice Transmission Inter Call Interval Call Faillure Call Refusal

Success: 67 %

Group: TE_LOAD

Test: PSTN

Duration: 54689 sec

Durations: Dial Tone Delay [✓ Path Establishment Delay] Calling Tone Delay Last Digit Transfer Delay Path Duration Call Cleared Delay Voice Transmission Delay Inter Call Interval Time To Fail

Mean: 49 ms

Stdev: 23 ms

Maximum: 210 ms

Minimum: 13 ms

Sum: 166562

Figure 6.37: Results of Path establishment trace function

Path components were reached. Faults were found in most of the basic components. Device drivers and programmable logic devices as well as memory and queue management procedures were the main faulty components.

Due to a set of reasons, the service test equipment was developed by reusing FA1201 components (hardware and communications components). Therefore faults could be found in both systems. The definition of simple testing interfaces (the service component interface automata) has proved very useful to reason about problems independently of the complexity of interactions and fault location.

Another advantage of the method was its ability to statistically classify the faults as a function of equipment configurations. This helped the management of field trials and equipment demonstrations, since the less problematic configurations were known and could be selected in advance.

By presenting the component test results graphically, the concept of *behaviour histogram* was introduced to provide a clear indication of the overnight fault distribution. Fig. 6.37, for instance, presents one window of the graphical interface containing the results of the evaluation of a set of Path service components with respect to the trace evaluation function of Fig. 6.28, where

$$p(s) = \begin{array}{|lrl} 120014/176570 = & 68.0\%, & s = s_1 \\ (130202 - 120014)/176570 = & 5.8\%, & s = s_2 \\ (155463 - 130202)/176570 = & 14.3\%, & s = s_3 \\ (155463 - 155463)/176570 = & 0\%, & s = s_4 \\ (166562 - 155463)/176570 = & 6.3\%, & s = s_5 \\ (176570 - 166562)/176570 = & 5.7\%, & s = s_6 \\ 0/176570 = & 0\%, & s = s_7 \end{array}$$

Depending on the probability of the states being reached, these results can be interpreted as follows, using the automaton defined in Fig. 6.28:

- $\mathbf{s = s_6}$. About 5.7% of the call attempts, represented by the $TE!Offhook$ event which is observed at the $TEPATH$ of Fig. 6.8, have not been followed *in sequence* by the event $LE?Remoteoffhook$ at the LEPATH of Fig. 6.8. It means that this last event was not observed at all or was observed out of sequence. This situation gives and indication of errors which can be interpreted in the simple reference model of Fig. 5.2 as follows:
  - the PSTN Ports block does not correctly sample the loop current at Z interface and, for this reason, it does not understand the offhook event;
  - the PSTN Ports block is aware of the offhook request but it forms a wrong message which is then sent to the Signalling block through the Control channel;

  - the protocol used to transfer messages between the PSTN Ports and Signalling blocks has problems;
  - the Control channel device driver, at the Signalling block, is faulty and, from time to time, looses messages;
  - inside the Signalling block the $FE - subscriber\_seizure$ signal is unable to reach the corresponding PSTN Protocol instance (Fig. 5.6 and Tab. 5.4);

- $\mathbf{s} = \mathbf{s_5}$. For 6.3% of the call attempts the traces obtained indicate that the $TE?Dialtone$ event was not observed after the $TE!Offhook.LE?Remoteoffhook$ sequence. This figure indicates the possible occurence of one of the following problems:
  - difficulties in allocating the E1 bearer channel which is negotiated between the Access Network and the Local Exchange for the transmission of tones;
  - the Time-Space Exchange block incorrectly switches the time slot allocated to the call;
  - connecting problems on the B channel of Fig. 5.2;
  - the PSTN Ports block cannot correctly convert the dial tone digital octets into an analogue signal;
  - the clock is badly recovered from the main E1 link and the drifts introduced originate an analogue signal which cannot be recognised as the dial tone by the tones detection function at the terminal equipment emulator;
  - the local exchange emulator injects the dial tone into a wrong bearer channel;
- $\mathbf{s} = \mathbf{s_4}$. The probability of this state being reached is estimated in 0%, which means that the test has generated a dial number every time a dial tone has followed the $TE!Offhook$ event. The test, in this case, is working correctly;
- $\mathbf{s} = \mathbf{s_3}$. In this case, the $LE?Incommingcall$ event is not observed after the $TE!Offhook.LE?Remoteoffhook.TE?Dialtone.TE!Dialnumber$ sequence, for 14.3% of the call attempts. It is the most important source of problems found by this trace evaluation function and this problem can be caused by:
  - the PSTN Ports block, which is unable to sample correctly the loop current so the number dialed is badly interpreted. In this

case, the emulator at the Local Exchange must have means to detect a wrong number and, in this case, should not generate the $LE?Incommingcall$ signal;

  – the PSTN Ports block, which correctly perceives the digits but, due to overload conditions, is unable to transmit all the digits to the Signalling block through the Control channel;

  – the process in the Signalling block in charge of transmitting the digits received from the PSTN Ports block to the convenient PSTN state machine instance which misroutes, from time to time, the digits to another PSTN instance;

- $\mathbf{s} = \mathbf{s_2}$. For 5.8% of the tests (call attempts) the $TE?Callingtone$ event was not observed or was observed out of sequence. It gives an indication of errors, which can be interpreted as follows:

  – the Time-Space Exchange had been requested by the Signalling to stop exchanging the bearer E1 channel used to transport the tones. In this case, this order must have been given after the dial number since the dial tone has already been received for the same call through the same $E1$ channel;

  – the local exchange emulator may have generated the tone for a $E1$ channel which is different from the timeslot negotiated between the Access Network and the Local Exchange;

  – The PSTN Ports block has stopped converting digital octets into an analogue signal, after the transmission of the digits;

- $\mathbf{s} = \mathbf{s_1}$. 68% of the call attempts succeded in correctly establishing a call, which means that, after eliminating the events non-visible by the call establishment function, 68% of the traces obtained were
  $\sigma = TE!Offhook.LE?Remoteoffhook.TE?Dialtone.TE!Dialnumber.$
  $LE?Incommingcall.TE?Callingtone.\phi$

  This situation is said to be the expected and it indicates the presence of no faults, with respect to the call establishment evaluation function.

Relevant delays were also statistically evaluated. The time histogram of Fig. 6.37, for instance, represents the *pathDelay* of Fig. 6.28. The final FA1201 service test results, which show the equipment free of service faults and having delays evaluated for 90% confidence intervals, are available at
http://puma.inescn.pt:8080/loads.html .

6.9 Conclusions

In this chapter, the application of the service part of the methodology proposed in Chap. 4 to a real Network Element, the NEC FA1201 Access Network, was described.

The first conclusion is that service components can be abstracted quite naturally from existing specifications. In order to access them, they require that parts of the service are located in emulation (test) equipments.

The second conclusion is that a service component can be easily specified by describing only its interfaces. These specifications are simpler and easier to develop than their complex protocol counterparts.

The third conclusion is that SDL is a good language for service testing. It provides the mechanisms required by parallel programs and its drawbacks (queues and timed actions) can be partially overcome.

The fourth and last conclusion is that cyclic service tests are the most valuable tests proposed by the methodology, since they enable the detection and statistical classification of complex problems from the user/ management point of view.

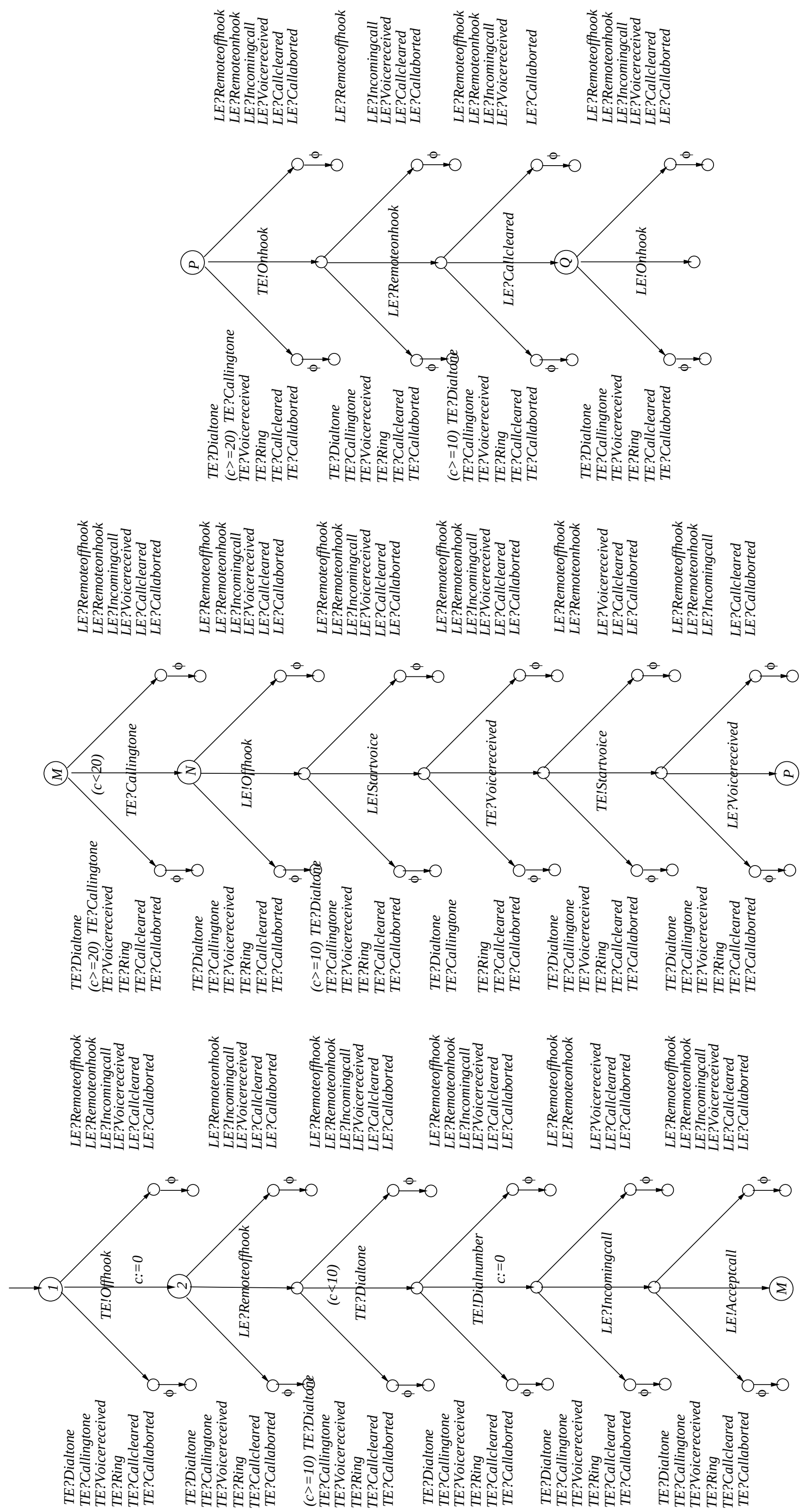


Figure 6.38: Originating call test case

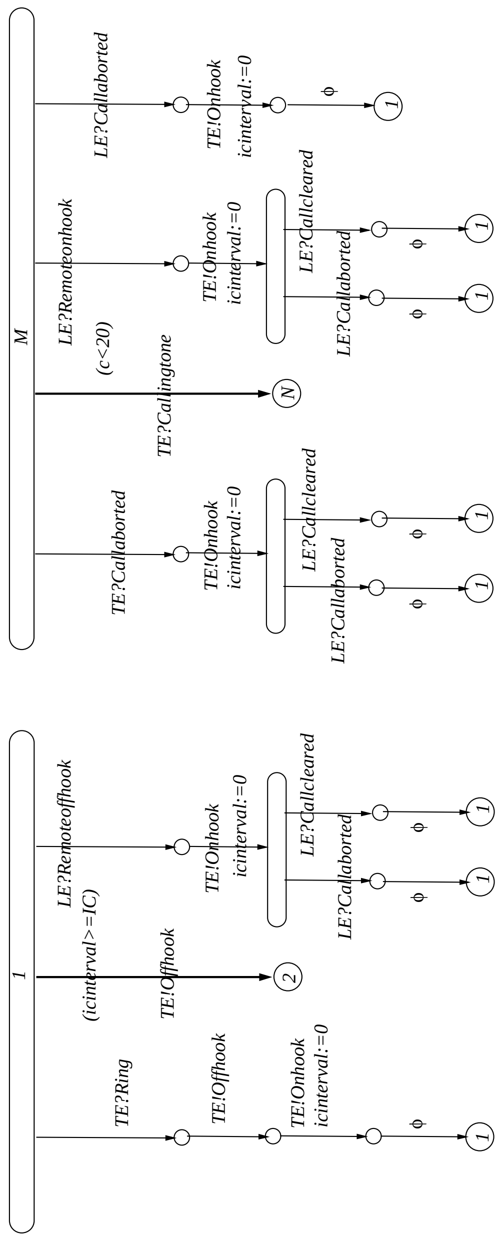


Figure 6.39: Originating call cyclic test case (states 1 and M)

Chapter 7

Conclusions

7.1 Introduction

This thesis has proposed a methodology for testing telecommunications Network Elements which may help implementers to develop high quality equipment and increase the probability of success in trials during which the equipment is externally evaluated.

To achieve efficiency it is required that the testing method can cope with and link complementary views of the system, can be incrementally applied and can provide information about the equipment failures and the quality achieved at any time. The approach adopted was to reuse and improve the traditional testing practices (conformance and load testing), since they are oriented towards the solution of real (most likely) equipment problems but are not aimed at functional Network Element testing. This improvement should carefully take into consideration the recent advances in formal methods applied to conformance testing, since they could provide an adequate framework to precisely understand the meaning and the value of each type of test.

Therefore, protocol conformance testing was selected as the first area of work. Due to the increasing number of protocol based telecommunication systems over the last fifteen years and to the importance of testing for the interoperability of equipment from different vendors, this test area has become quite mature. It is limited in scope since it addresses only the observable behaviour of protocol implementations and explicitly avoids other types of validation, such as system testing or performance evaluation. It has, however, two characteristics that were found appealing for this work. First, the specification and execution testing practices are well documented in *live* standards. Second, a very large effort has been made by the test research community to formalise these tests and to automate the derivation process. The current protocol testing practices and theories were, for this reason, presented in Chap. 2.

The traditional load tests are in a very different situation. They are quite common for the evaluation of equipment but serious research work about their value is just beginning. The ad-hoc methods employed currently are conceptually simple and consist in loading the equipment with expected and overload traffic scenarios. Besides measuring the relevant performance/quality parameters, these tests are used to demonstrate that the equipment is able to survive under real traffic conditions for long periods of time. Our experience in previous projects, however, showed that, if improved with some behaviour information, these tests could be very valuable. For these reasons, instead of presenting the (inexistent) load testing theory and methods, the problem of performance and quality evaluation was addressed from scratch in Chap. 3. First, the common performance models (loss and delay system) were introduced from an (implicit) discrete event simulation perspective. Then, the approach normally used to specify quality and performance parameters was presented by means of examples with the objective of demonstrating the closeness between these specifications and protocol behaviour specifications. The purpose was to suggest the integration of the two worlds which, from the test point of view, would be quite appealing. In order to support this hypothesis well known concepts of computer science, but new in the communications field, such as timed and probabilistic behaviour, were adopted.

In Chap. 4, the proposed testing methodology was presented. First, a mathematical framework was selected - the timed automata. Although it may be argued that it places problems in the representation of real situations since very large automata will be obtained, it has proved very useful for the representation of the essential methodology aspects (behaviour, time and probabilities) while avoiding, at same time, the *noisy* characteristics. Next, it was showed how complex systems could be modelled independently of the availability of rigorous specifications. Then, by reusing the concept of safety and guarantee automata from protocol conformance testing, it was proved how different types of tests could be derived - behaviour, timed behaviour and probabilistic time behaviour. By decoupling trace evaluation functions from tests it was shown how performance measurement tools could be developed and QoS properties evaluated. Taking advantage of these tests, a methodology was proposed which consists in modelling the Network Element as two sets of components (communication and service) and evaluating them with different type of tests.

In Chap. 5, a complex Network Element was presented, to which the testing methodology was applied - the V5 Access Network. Its main building blocks, protocols and services were introduced. The conformance tests developed by ETSI were also presented and characterised.

In Chap. 6, the application of the new part of the methodology (service testing) to a V5 Access Network implementation, the NEC FA1201, was described. Service components were developed based on existing V5 documentation and specified as interface components. Next, a set of relevant simple service tests and time probabilistic tests, along with their trace evaluation functions, were introduced. A solution for implementing short tests (using synchronous communication) and precise times in pure SDL was proposed as well.

7.2 Original contributions

Testing methodology. The methodology proposed consists in modelling the Network Element as two sets of components: communications and service. Communications components are those which implement the communications protocols at every Network Element interface. Service components are abstractions which, in reality, are composed of a number of basic components which can include communications components, management components, operating systems and device drivers. Significant parts of service components, known as emulation components, are required to reside within the test system. Communications components should be tested using well-known protocol conformance testing techniques, which assume that the complete component specifications are known. Service components can, according to the methodology proposed, be partially specified as the so called interface components. The combination and testing of both views of the Network Element enables the detection of a large spectrum of faults, in particular those which have random appearance and manifest only after long working hours and whose consequences are usually catastrophic.

Service testing method. The methodology proposed combines existing testing results (protocol conformance testing) with a new part - service testing. At the time of the author's first international publication [10], and to the best of his knowledge, this service testing approach was new. In fact, and due to the relevance this work (that was presented in invited papers in two conferences and attracted visitors from two companies), similar solutions appeared latter on on commercial V5 test equipments. However, the service testing approach is presented in this thesis with a level of detail and justification which are new. The principle which has made service tests new and so useful is very simple: knowing in advance about the load tests potential to make the complex and problematic situations visible, it was just necessary to improve the expressiveness of the tests so that faults could be described and distinguished. In this way, tests have been treated as behaviour tests using the hypothesis that every test which drives an implementation from its initial state and back to its initial state can be made cyclic. Conformance test suites are full of these test cases. By adding time, using discrete event simulation techniques to generate event inter arrival times, and probabilities for selecting among cyclic tests in random usages, a rich load test concept has emerged. After understanding precisely the meaning of verdicts in behaviour tests it was possible to extend the concept to more generic trace evaluation functions. The probabilistic classification of traces, from which very rich fault information can be gained, as well as the classification of the delays, which is relevant for performance evaluation, were then possible.

Specification of QoS properties. A new, simple and behaviour rich technique for describing QoS propertied was also introduced. In standards, QoS is described by statistical parameters which, as argued in this thesis, are difficult to relate to the corresponding protocol behaviour situations. By using trace evaluation functions it was possible to describe precisely the behaviour situations to which the statistical parameters are related, which can be final states reached or delays.

Service testing architecture. It consists in modelling the services at the test system as a set of components, in exciting component instances with test instances and in separating tests from trace evaluation functions. Particularly relevant is the implementation of this architecture in pure SDL, which can be easily translated into programming languages for the most popular real time operating systems.

7.3 Further research directions

The work described in this thesis can continue in three main directions, which may be seen as promising areas of work: 1) the distribution of the methodology; 2) the improvement of the methodology by applying it to new networks or network elements; 2) the realisation of loosely coupled and distributed test systems.

Distribution of the methodology. Although the service tests in the NEC project were carried out using a distributed test system and using distributed tests, the generalisation of the distribution problems and the proposal (discussion and analysis) of solutions were avoided in this thesis. A new PhD work programme on this theme seems to be appropriated by a set of reasons: 1) this thesis provides the basics for that work; 2) the first attempts to distribute protocol conformance tests are now being attempted by researchers; 3) from the experience in the NEC project, the distribution problems are already known and practical solutions have already been implemented. This work should be done using high level abstraction models, such as timed automata, so that the nature of the problems and the validity of the solutions can be understood and validated. As a complement, a MSc student has already started working on test distribution at the SDL level with the proprietary method used in the NEC project. The problem here consists in providing the proprietary SDL environment with facilities required to support test components in a multi-host environment.

Application of the method to new case studies. As previously mentioned, the proposed method has been generalised based on the V5 case study. It possibly has, for this reason, a number of shortcomings which can only be eliminated by applying the method, or parts of it, to new case studies. Work is already on-going on the following fields:

- *VoIP test trial.* It consists on the develoment of at test environment (tests, test system and network elements) which was motivated by consulting contracts with a U.S. test company (Schlumberger) and is being used as a basis for new projects under negotiation. This trial aims at extending and engineering the methodology towards real time services over IP networks, voice over IP being a good example. The method/test system is addressing not only individual IP equipment (routers, gateways and gatekeepers) but also networks, that is, sets of interconnected systems.

- *UMTS test system.* This is a joint project proposal to the IST programme involving some European companies, namely GMD-Focus, CTS, Nokia and British Telecom. This is an excellent case study because it is similar similar to the V5 case (ITU-T and ESTI specifications, conformance tests being developed) but, due to the UMTS transmission characteristics which accept new calls by lowering the quality of the overall system, requires more service testing. The aim is to develop service tests using the service methodology proposed and to validate a real Network Element using the test system developed by the consortium.

- *Develoment of a TCP/IP test suite.* This project, in cooperation with a small Portuguese company, will consist on the derivation of sets of tests described in TTCN. It is also an excellent case study, since IETF protocols in general and TCP in particular, unlike ITU-T /ETSI protocols, are very much data oriented. Events, in this case, will have to represent combinations of data values.

Loosely coupled testing system. The VoIP and UMTS projects are pushing the extension of the method towards monitoring, that is, passive testing, which consists basically in avoiding tests and working only with trace evaluation functions. Another interesting point is that both types of service tests (passive and active) may have to be supported by test elements which can be separated by long distances and, for that reason, require synchronisation mechanisms without, possibly, any interactions during the test execution. The inclusion of these elements as IP hosts and the representation of test relevant information as MIBs which can be accessed through SNMP is being addressed, so that these test elements can be gracefully incorporated in actual network management schemes.

Bibliography

[1] ITU-T. Vocabulary of switching and signalling terms. Recommendation Q.9, ITU-T, 1988.

[2] ITU-T. Specification and Description Language (SDL). Recommendation Z.100, ITU-T, 1999.

[3] ITU-T. Information Technology - Open Systems Interconnection - Structure of Management Information: Guidelines for the Definition of Managed Objects. Recommendation X.722, ITU-T, 1992.

[4] Telelogic. Tau SDL Suite. http://www.telelogic.se/solution/tools/sdl.asp, Telelogic, 1999.

[5] Verilog. ObjectGEODE. http://www.csverilog.com/products/geode.htm, Verilog, 1999.

[6] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International Editions, 1991.

[7] OMG. Unified Modelling Language. http://www.omg.org/uml, Object Management Group, 1999.

[8] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1995.

[9] SCORE Project. Conclusions and recommendations from SCORE. Deliverable D409, RACE, 1994.

[10] E. Inocencio, M. Ricardo, H. Sato, and T. Kashima. Combined Application of SDL-92, OMT, MSC and TTCN. In Reinhard Gotzhein, editor, *Formal Description Techniques IX - Theory application and tools. Participants Proceedings*, pages 451–466, 1996.

[11] SCORE Project. Process Model, Component Model and Methods/Tool set (version 3). Deliverable D105, RACE, 1994.

[12] ITU-T. Message Sequence Chart (MSC). Recommendation Z.120, ITU-T, 1999.

[13] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering - a Use Case Driven Approach*. Addison-Wesley, 1992.

[14] ITU-T. OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications - The Tree and Tabular Combined Notation (TTCN). Recommendation X.292, ITU-T, 1998.

[15] E. Moore. *Sequential Machines: selected papers* . Addison-Wesley, 1964.

[16] F. Hennie. Fault detecting experiments for sequential circuits. In Princeton University, editor, *Proc. 5th Annual Simp. Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

[17] B. Sarikaya. *Test Design for Computer Network Protocols*. Ph.D. Thesis, McGill University, Montreal, Canada, 1984.

[18] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, Computer Science Series, 1978.

[19] K. Sabnani and A. Dahbura. A protocol testing procedure. *Computer Networks and ISDN Systems*, 4(15):285–297, 1988.

[20] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science, 1989.

[21] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[22] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification, VIII*, pages 63–74, 1988.

[23] J. Tretmans. A Formal Approach to Conformance Testing. In Omar Rafiq, editor, *International Workshop on Protocol Test Systems - IWPTS VI*, pages 261–280, 1993.

[24] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems, Special Issue in Protocol Testing*, 29(1):49–79, 1996.

[25] M. Benattou, L. Cacciari, R. Pasini, and O. Rafiq. Principles and Tools for Testing Open Distributed Systems. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Testing of Communicating Systems - Methods and Applications*, pages 77–92. Kluwer Academic Publishers, 1999.

[26] J. Grabowski and D. Hogrefe. Towards the Third Edition of TTCN. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Testing of Communicating Systems - Methods and Applications*, pages 19–29. Kluwer Academic Publishers, 1999.

[27] A. Ulrich and H. Konig. Architectures for Testing Distributed Systems. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Testing of Communicating Systems - Methods and Applications*, pages 93–108. Kluwer Academic Publishers, 1999.

[28] M. Toro. Decision of Tester Configuration for Multiparty Testing. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Testing of Communicating Systems - Methods and Applications*, pages 109–128. Kluwer Academic Publishers, 1999.

[29] A. Cavalli. Different Approachs to Protocol and Service Testing. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Testing of Communicating Systems - Methods and Applications*, pages 3–18. Kluwer Academic Publishers, 1999.

[30] T. Higashino, A. Nakata, K. Tanigushi, and A. Cavalli. Generating Test Cases For a Timed I/O Automaton Model. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Testing of Communicating Systems - Methods and Applications*, pages 197–214. Kluwer Academic Publishers, 1999.

[31] C. Aguiar, M. Ricardo, and E. Carrapatoso. Traffic mix simulation - an enhanced model. Deliverable A1.2/INE/009, RACE - LACE Project, 1989.

[32] E. Carrapatoso, C. Aguiar, and M. Ricardo. Traffic generator/detector system concept. In *Proceedings of EFOCLAN 90*, pages 197–214. IGI Europe, 1990.

[33] M. Ricardo, E. Carrapatoso, and C. Aguiar. A traffic generator/detector system for high speed networks. In E. Arikan, editor, *Communication Control and Signal Processing*, pages 656–662. Elsevier, 1990.

[34] M. Ricardo, M. Ferreira, F. Guimaraes, J. Mamede, M. Henriques, J. Silva, and E. Carrapatoso. Functional and Non-Functional Testing of

ATM Networks. In *Fiber Optic Networks and Video Communications*. Amsterdam, 1995.

[35] A. Almeida, P. Proenca, and M. Ricardo. Multimedia Art Directory - A Multimedia Application over an ATM Network. In *Fiber Optic Networks and Video Communications*. Amsterdam, 1995.

[36] A. Almeida, P. Proenca, and M. Ricardo. Multimedia Art Directory - A Multimedia Service over an ATM Network. In *ATM Developments 95*. Rennes, 1995.

[37] John E. Hopcroft and Jeffrey D. Ullman. *Automata Theory Languages and Computation*. Addison Wesley, 1979.

[38] Gordon D. Plotkin. *A structural approach to operational semantis*. Computer Science Department, Aarhus University, Denmark, 1981.

[39] ITU-T. OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications – General concepts. Recommendation X.290, ITU-T, 1995.

[40] ITU-T. OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications – Abstract test suite specification. Recommendation X.291, ITU-T, 1995.

[41] ITU-T. OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications – Test realization. Recommendation X.293, ITU-T, 1995.

[42] ITU-T. OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications – Requirements on test laboratories and clients for the conformance assessment process . Recommendation X.294, ITU-T, 1995.

[43] ITU-T. OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications – Protocol profile test specification. Recommendation X.295, ITU-T, 1995.

[44] ITU-T. OSI conformance testing methodology and framework for protocol recommendations for ITU-T applications – implementation conformance statements. Recommendation X.296, ITU-T, 1995.

[45] R. L. Probert and O. Monkwich. TTCN: the international notation for specifying tests of communications systems. *Computer Networks and ISDN Systems*, 23(5):417–438, February 1992.

[46] Pex. A TTCN Tutorial SDL Suite. http://www.etsi.org/pex_tutorial_list.html, Pex, 1999.

[47] ITU. Formal Methods in Conformance Testing. Recommendation Z.500, ITU-T, 1997.

[48] Finn Kristoffersen, Marc Phalippou, and Jan Tretmans. Tutorial pstv'95 – formal methods in conformance testing. In Piotr Dembinski and Marek Sredniawa, editors, *PSTV'95, Fifteenth International Symposium on Protocol Specification, Testing and Verification Systems VI - Tutorial Notes, Varsaw, Poland*, 1995.

[49] Ana R. Cavalli, Jean Philippe Favreau, and Marc Phalippou. Standardization of formal methods in conformance testing of communication protocols. *Computer Networks and ISDN Systems, Special Issue in Protocol Testing*, 29(1):3–14, 1996.

[50] Zhoar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: specification.* Springer-Verlag, 1992.

[51] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions, 1991.

[52] B. Bosik and M. Uyar. Finite state machine based formal methods in protocol conformance testing: from theory to implmentation. *Computer Networks and ISDN Systems*, 22(1):7–33, 1991.

[53] A. Aho, A. Dahbura, D. Lee, and M. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. In S. Aggrawal and K. Sabnani, editors, *Protocol Specification Verification and Testing* , 1988.

[54] ETSI. Methods for Testing and Specification; ETSI Standards Making; Technical quality criteria for telecommunication standards . Draft recommendation eg 201 014, ETSI, 1997.

[55] ETSI. Methods for Testing and Specification; Specifications of Protocols and Services; Validation Methodology for standards using SDL; Handbook . Draft recommendation eg 201 015, ETSI, 1997.

[56] ITU-T. Specification and Description Language (SDL) – Initial Algebra Model. Recommendation z.100, annex c, ITU-T, 1993.

[57] Joseph Sifakis. Tutorial b – the automatic verification of protocols. In A. Dantine, G. Leduc, and P. Wolper, editors, *13th IFIP Symposium on Protocol Specification, Testing and Verification. May 1993, Liege Belgium*, 1993.

[58] Lars Bromstrup and Dieter Hogrefe. TESDL: Experience with Generating Test Case from SDL Specificactions. In Ove Faergemand and Maria Manuela marques, editors, *SDL'89 The Language at Work*, pages 267–279. North-Holland, 1989.

[59] Daniel Toggweiler. Efficient Test Case Generation for distributed Systems specified by Automata. Ph.D. Thesis, Bern University, 1995.

[60] Robert Nahm. Conformance Testing based on Formal Description Techniques and Massage Sequence Charts. Ph.D. Thesis, Bern University, 1995.

[61] Ana Cavalli, Boo-Ho, and Toma Macavei. Test generation for the SSCOP-ATM networks protocol. In Ana Cavalli and Amardeo Sarma, editors, *SDL'97 Time for Testing, MSC and Trends*, pages 277–288. Elsevier, 1997.

[62] G. v. Bochmann, A. Petrenko adn O. Bellal, and S. Maguiraga. Automating the process of tests derivation from SDL specifications. In Ana Cavalli and Amardeo Sarma, editors, *SDL'97 Time for Testing, MSC and Trends*, pages 261–276. Elsevier, 1997.

[63] J. Grabowski, D. Hogrefe, and R. Nham. Test Case Generation with Test Purpose Specification by MSCs. In Ove Faergemand and Amardeo Sarma, editors, *SDL'93 Using Objects*, pages 253–265. North-Holland, 1993.

[64] A. Ek, J. Grabowski, D. Hogrefe, R. Jerome, B. Kock, and M. Schmitt. Towards the Industrial Use of Validation Techniques and Automatic Test Generation Methods for SDL specifications. In Ana Cavalli and Amardeo Sarma, editors, *SDL'97 Time for Testing, MSC and Trends*, pages 245–260. Elsevier, 1997.

[65] I. Mitrani. *Modelling of computer and communication systems.* Cambridge University Press, 1987.

[66] A. Law and W. Kelton. *Simulation Modeling & Analysis.* McGraw-Hill International Editions, 1991.

[67] P. King. *Computer and Communication System Performance Modelling.* Prentice Hall Computer Series in Computer Science, 1990.

[68] J. Bellamy. *Digital Telephony.* John Wiley & Sons, 1991.

[69] L. Kleinrock. *Queuing Systems.* John Wiley & Sons, 1975.

[70] O. Boxma and R. Syski. *Queuing Theory and its Applications - Liber Amicorum for J. Cohen.* North-Holland, 1988.

[71] ITU-T. Terms and definitions related to quality of service and network performance including dependability . Recommendation E.800, ITU, 1994.

[72] ITU-T. General Aspects of Quality of Service and Netwrok Performance in Digital Networks, including ISDNs. Recommendation I.350, ITU, 1993.

[73] ITU-T. Reference Events for Defining ISDN and B-ISDN Performance Parameters. Recommendation I.353, ITU, 1996.

[74] ITU-T. Network Performance Objectives for Connetion Processing Delays in an ISDN. Recommendation I.352, ITU, 1993.

[75] ITU-T. Network Performace Objectives for Packet Mode Communication in an ISDN. Recommendation I.354, ITU, 1993.

[76] ITU-T. Speed of service (delay and throughput) performance values for public data networks when providing international packet-switched services. Recommendation X.135, ITU, 1997.

[77] ITU-T. Digital exchange performance objectives . Recommendation Q.543, ITU, 1993.

[78] ITU-T. ISDN 64 kbit/s Connection Type Availability Performance. Recommendation I.355, ITU, 1995.

[79] ITU-T. Accuracy and dependability performance values for public data networks when providing international packet-switched services . Recommendation X.136, ITU, 1992.

[80] H. Herzog, H. Hermanns, and V. Mertsiotakis. Stochastic Process Algebras - Betwen LOTOS and Markov Chains. In G. Gotzhein and J. Bredereke, editors, *FORTE/PSTV'96, Joint international conference on Formal Description Techniques for Distributed Systems and Communications Protocols and Protocol Specification Testing and Verification – Tutorial notes*, 1996.

[81] I. Schieferdecker. Performance-Oriented Specification of Communication Protocols and Verification of Deterministic Bound and their QoS Characteristics. Phd thesis, Technical University Berlin, 1994.

[82] Joost-Pieter Katoen. Quantitative and Qualitative Extensions of Event Structures . PhD Thesis, Twente University, Netherlands, 1996.

[83] B-Muller-Clostermann and M. Diefenbruch. Queing SDL: A language for the functional and quantitative Specification of distributed Systems. In A. Mitschele-Thiel, B. Muller-Clostermann, and R. Reed, editors, *First Workshop on Performance and Time in SDL and MSC, Erlangen, Germany* , 1997.

[84] Carlos Rodriguez and Joseph Sifakis. Improving Expression of Time Constraints in SDL. In A. Mitschele-Thiel, B. Muller-Clostermann, and R. Reed, editors, *First Workshop on Performance and Time in SDL and MSC, Erlangen, Germany* , 1997.

[85] R. Alur and D. Dill. A theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[86] R. Alur and D. Dill. Automata-theoretic Verification of Real-Time Systems. In *Formal Methods for Real Time Computing*, pages 55–82. Trends in Software Series, John Wiley & Sons, 1996.

[87] R. J. van Blabeek, S. A. Smolka, and B. Steffen. Reactive, Generative and Stratified Models of Probabilistic Processes. In IEEE Computer Society Press, editor, *Proceedings 5th Annual Symposium on Logic in Computer Science, Philadelphia, USA* , 1990.

[88] Marco Bernardo and Roberto Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 1-54, July 1998.

[89] Roberto Segala. Modeling and Verification of Randomized Distributed Real-Time Systems. PhD Thesis, MIT, USA, 1995.

[90] Sue-Hwey Wu, Scott A. Smolka, and Eugene W. Stark. Composition and Behaviors of Probabilistic I/O automata. *Theoretical Computer Science*, 176(1-2):1–38, 1997.

[91] M. Ricardo, J. Mamede, J. Almeida, L. Anselmo, and S. Crisostomo. A method for testing complex Network Elements - the V5 Access Network case study. In *12th IFIP International Workshop on Testing of Communicating Systems, Industrial session, Budapest, Hungary*, 1999.

[92] M. Ricardo, J. Mamede, J. Almeida, L. Anselmo, and S. Crisostomo. Testing V5 Access Networks. In F. Perez Gonzalez, A. Figueiras-Vidal, and N. Gonzalez-Prelcic, editors, *Proceedings of the Fifth Baiona*

Workshop on Emerging Technologies in Telecommunications, Bayona, Spain, 1999.

[93] M. Ricardo, J. Mamede, J. Almeida, L. Anselmo, and S. Crisostomo. Testing V5 AN and LE Network Elements in the Lab. In Schlumberger, editor, *Proceedings of V5 Seminars, Oxford, Munich, Paris, Stockholm, Westford and SanJose,* Invited contribution, 1999.

[94] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.1 interface for the support of Access Network (AN) - Part 1: V5.1 interface specification. ETS 300 324 - 1, ETSI, 1994.

[95] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.2 interface for the support of Access Network (AN) - Part 1: V5.2 interface specification. ETS 300 347 - 1, ETSI, 1994.

[96] ITU-T. Frame alignment and cyclic redundancy check (CRC) procedures relating to basic frame structures defined in recommendation G.704 . Recommendation G.706, ITU, 1991.

[97] ITU-T. Digital transmission system on metallic local lines for ISDN basic rate access. Recommendation G.961, ITU, 1993.

[98] ITU-T. Physical/electrical characteristics of hierachical digital interfaces . Recommendation G.703, ITU, 1998.

[99] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

[100] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.1 interface for the support of Access Network (AN) - Part 3: Test Suite Structure and Test Purposes (TSS & TP)V5.1 specification for the network layer (AN side). ETS 300 324 - 3, ETSI, 1995.

[101] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.1 interface for the support of Access Network (AN) - Part 7: Test Suite Structure and Test Purposes (TSS & TP) specification for the data link layer . ETS 300 324 - 7, ETSI, 1995.

[102] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.2 interface for the support of Access Network (AN) - Part 3: Test Suite Structure and Test Purposes (TSS

& TP)V5.2 specification for the network layer (AN side). ETS 300 347 - 3, ETSI, 1996.

[103] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.2 interface for the support of Access Network (AN) - Part 7: Test Suite Structure and Test Purposes (TSS & TP) specification for the data link layer . ETS 300 347 - 7, ETSI, 1996.

[104] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.1 interface for the support of Access Network (AN) - Part 4: Abstract Test Suite (ATS) and partial Protocol eXtra Information for Testing (PIXIT) proforma specification for the network layer (AN side). ETS 300 324 - 4, ETSI, 1997.

[105] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.2 interface for the support of Access Network (AN) - Part 4: Abstract Test Suite (ATS) and partial Protocol eXtra Information for Testing (PIXIT) proforma specification for the network layer (AN side). ETS 300 347 - 4, ETSI, 1997.

[106] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.1 interface for the support of Access Network (AN) - Part 8: Abstract Test Suite (ATS) and partial Protocol eXtra Information for Testing (PIXIT) proforma specification for the data link layer . ETS 300 324 - 8, ETSI, 1997.

[107] ETSI. Signalling Protocols and Switching (SPS); V interfaces at the digital Local Exchange (LE) V5.2 interface for the support of Access Network (AN) - Part 8: Abstract Test Suite (ATS) and partial Protocol eXtra Information for Testing (PIXIT) proforma specification for the data link layer . ETS 300 347 - 8, ETSI, 1996.

[108] ETSI. Signalling Protocols and Switching, Local Exchange and Access Network Performance Design, Objectives for Call Handling and Bearer Connection Management. ETS DE/SPS-03027, ETSI, 1996.

[109] NEC. FA1201 Product Requirement Specification (PRS). FA1201 Project Report, NEC, 1995.

[110] NEC. FA1201 System Architecture Specification (SAS). FA1201 Project Report, NEC, 1995.

[111] M. Ricardo. Testing the FA1201. FA1201 Project Report, WDT01, NEC, 1995.

[112] M. Ricardo. Development of Testing Equipment for the FA1201 - Preliminary Study. FA1201 Project Report, WDT02, NEC, 1995.

[113] M. Ricardo. Testing of the FA1201 V5.x Configurations. FA1201 Project Report, WDT04, NEC, 1995.

[114] J. Antonio, L. Anselmo, and M. Ricardo. V5 Test Equipment Manual. FA1201 Project Report, WDT14, NEC, 1998.

[115] M. Ricardo. Tests Specification for the V5 Local Exchange and Terminal Equipment Emulators. FA1201 Project Report, WDT05, NEC, 1996.

[116] F. Guimaraes, J. Mamede, and M. Ricardo. TE and V5LE Emulators Object Models. FA1201 Project Report, WDT09, NEC, 1996.

[117] C. Silva and J. Illa. Man Machine Interface Specification . FA1201 Project Report, WDT08, NEC, 1996.

[118] F. Guimaraes and J. Mamede. V5 Local Exchange Emulator Specification. FA1201 Project Report, WDT06, NEC, 1996.

[119] F. Guimaraes and J. Mamede. Terminal Equipment Emulator Specification. FA1201 Project Report, WDT07, NEC, 1996.

[120] F. Guimaraes and J. Mamede. Tones Processing Unit Specification. FA1201 Project Report, WDTPU, NEC, 1996.

[121] M. Ricardo and F. Pinto. Tones Processing Unit. FA1201 Project Report, WDT013, NEC, 1997.

[122] INESC Porto. FA1201 Final Test Results. FA1201 Project Report, WDT16, NEC, 1999.