NOTE TO USERS

This reproduction is the best copy available.

UMI®

Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DE ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Qing LI
AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science
GRADE - DEGREE

Department of Computer Science
FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

A Simple Language for testing SDL Specifications

R. Probert
DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

C.C. Huang

A. William

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

SIMPL-T:

SDL Intended for Management and Planning of Tests

By

Qing Li

A thesis submitted to the school of Graduate Studies and Research in partial fulfillment of the requirement for the degree of

Master of Computer Science

Ottawa – Carleton Institute for Computer Science

School of Information Technology and Engineering

University of Ottawa

Ottawa, Ontario, Canada

May 2004

© Qing Li, Ottawa, Canada, 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-01529-2
Our file Notre référence
ISBN: 0-494-01529-2

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

The correctness and completeness of the requirement specifications are critical in the software lifecycle. Validation of the specifications against the user requirements is essential for ensuring software quality and reducing development cost. SDL (Specification and Description Language) specifications may be validated using TTCN (Testing and Test Control Notation); however TTCN has a much wider scope and broader purpose. For testing SDL specifications, only a small part of TTCN is needed. Therefore, for many projects and organizations, it is not economical or efficient to use TTCN to validate SDL specifications. This thesis introduces a new language SIMPL-T (SDL Intended for Management and Planning of Tests), which is a simple test language for SDL specifications. It is defined as a minimal extension of existing SDL features so as to provide essential test specification functionality. It is understandable to persons familiar with SDL, and therefore immediately useful for SDL users.

ACKNOWLEDGEMENTS

I wish to express my gratitude to the many individuals who were involved in the production of this thesis.

Many thanks to my supervisor, Professor Robert Probert, who provided valuable support throughout this work. This work was made possible by his financial support through NSERC and CITO. He helped me to choose a suitable subject for this thesis and guided me through this research. He also spent hours and hours reviewing my numerous drafts with me and initiating unforgettable discussions. Thank you, Professor Probert!

I also wish to thank my committee, Dr. Alan Williams and Dr. Changcheng Huang for gracefully accepting to review and comment this work.

I especially wish to thank Dr. Alan Williams. I was very fortunate to get to know him during the work on the 1st international *SDL Design Contest*. Since then, he has given me countless guidance, advice and help whenever I asked, regardless how busy he was. As much as I respect him as a knowledgeable professor, I respect him as a very kind person.

I also thank the participants in the bi-weekly meetings of the ASERT group organized by Professor Probert, where I met a group of intelligent and knowledgeable people. Their comments helped improve this work. In particular, Dr. Os Monkewich shared his knowledge and experience with me and collected valuable data for this work; Dr. Bernard Stepien reviewed this thesis and provided valuable comments; and Dr. Daniel Amyot, with his challenging questions, encouraged me to work harder. Thank you all!

This work was made possible by the support of SOLINET. By working with the engineers of the SAFIRE team, especially Mr. William Skelton, the ideas of this work were inspired and refined and the concepts were confirmed. The SAFIRE tool especially facilitated the case study.

I hope my thanks will reach Dr. Tae-Hyong Kim, who is a professor now in Korea. He led me into the SDL world, and he taught me the skills of giving professional presentations. Without his help, I could not have given the successful presentation to win the SDL Design Contest in the SAM'02 Workshop.

I cannot forget Professor Sylvia Boyd, whose class was always interesting and enjoyable. It was in her class where I regained my confidence. She made me believe that I can conquer any difficulties and be at least as good as any other students. Thank you, Dr. Boyd!

Finally, I wish to thank my friends Yinyan and Yiqun for support and encouragement during the whole period of this work.

ABBREVIATIONS

ADT	Abstract Data Type
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
ASP	Abstract Service Primitive
ATS	Abstract Test Suite
BNF	Backus-Nauer Form
CATG	Computer Aided Test Generation
CSR	Common Semantics Representation
EFSM	Extended Finite-State Machine
ETR	ETSI Technical Report
ETS	Executable Test Suite
ETSI	European Telecommunications Standards Institute
FDT	Formal Description Technique
FIFO	First In First Out
FSM	Finite-State Machine
IDL	Interface Description Language
IEEE	Institute of Electrical and Electronics Engineers Inc
ITU	International Telecommunication Union
ITU-T	International Telecommunication Union–Telecommunications Sector
IUT	Implementation under Test
LOTOS	Language of Temporal Ordering Specification

MSC	Message Sequence Chart
MTC	Master Test Component
OMG	Object Management Group
OMT	Object Modelling Technique
OO	Object-Oriented
PCO	Points of Control and Observation
PDU	Protocol Data Unit
PTC	Parallel Test Component
(P)ICS	(Protocol) Implementation Conformance Statement
(P)IXIT	(Protocol) Implementation eXtra Information for Testing
SDL	Specification and Description Language
SIP	Session Initiation Protocol
SUT	System under Test
SUT	Specification under Test (in the context of this thesis)
TSS&TP	Test Suite Structure & Test Purposes
TTCN-3	Testing and Test Control Notation
TTCN-1/TTCN-2	Tree and Tabular Combined Notation
UCM	Use Case Map
UML	Unified Modelling Language
URN	User Requirements Notation

LIST OF FIGURES

Figure 1 Relative Error Correction Cost in a Software Life Cycle 10
Figure 2 Dynamic and Concurrent Test System.. 21
Figure 3 Conceptual View of a TTCN-3 Test System 22
Figure 4 Meaning of alt-statements 24
Figure 5 SDL and TTCN Conceptual Overlap 31
Figure 6 Only Part of TTCN is Needed for Testing SDL-based Specifications 32
Figure 7 Extensions to SDL.. 33
Figure 8 Test Architecture Defined by ITU-T Z.500 35
Figure 9 Test Architecture Defined by ISO 9646.. 36
Figure 10 The Structural View of an SDL System... 42
Figure 11 Signals Travel through Channels and Signal Routes 43
Figure 12 System with Type Declarations.. 45
Figure 13 A SDL-based Test System Architecture 48
Figure 14 Connecting the Tester to the SUT 50
Figure 15 An Example of Using TTCN to Test an SDL Specification 52
Figure 16 Using SDL to Send Stimuli to and Receive Responses from the SUT 53
Figure 17 Example of Output Signal Carrying Data 54
Figure 18 SDL Decision Construct Used in Testing 55
Figure 19 How Checking of Parameters Makes a Difference 57
Figure 20 Using a Timer to Monitor the Response Time of the SUT 59
Figure 21 How Existing SDL Features Used in a Test Case 61
Figure 22 An Example Test Suite Defined in SIMPL-T 71
Figure 23 SDL "OUTPUT VIA" Construct 73

Figure 24 "INPUT VIA" Construct (NEW) 73
Figure 25 Specify the Values of a Parameter in an *INPUT* Construct 78
Figure 26 Overlapped signals: the same signal arriving from different gates/channels 78
Figure 27 Overlapped signals: the values of parameters have overlap 79
Figure 28 An Example Test Case Written in SIMPL-T 87
Figure 29 A Procedure Definition with Preliminary Results 88
Figure 30 SDL System and Its Environment 93
Figure 31 SAFIRE SDL System 94
Figure 32 The Traffic Controller System and Its Environment 98
Figure 33 Traffic Controller System Architecture 99
Figure 34 System Architecture with Block Type Definition 100
Figure 35 Traffic Controller SDL System Implemented in the SAFIRE Environment ... 101
Figure 36 Processor Agent Type, Its Gates and Associated Signal Lists 102
Figure 37 State Chart of the Processor 103
Figure 38 Processor State Diagram – Transition Triggered by Signals from the Sensors 103
Figure 39 Processor State Diagram – Pass Signals from the Controller to the Lights 104
Figure 40 Controller Agent Type, Its Gates and Associated Signal Lists 105
Figure 41 The Ten States of the Controller Process Agent 106
Figure 42 An Example of the Controller Process State Diagram 106
Figure 43 The Traffic Controller SDL System (SUT) and A Temporary User Interface 107
Figure 44 The Test System Configuration in the SAFIRE Environment 108
Figure 45 The Gate and Signal List Declarations in a Test Suite 112
Figure 46 Part of the *Controller_Init* Procedure Diagram 114
Figure 47 Procedure *Open_Y_Direction* 115
Figure 48 Test Case *Max_Timeout_When_Both_Traffic* 116
Figure 49 Test Case *XY_Transition_Defer* 117
Figure 50 Test Harness in SAFIRE 118
Figure 51 SAFIRE CAMPAIGNER – Select Test Cases 119
Figure 52 SAFIRE CAMPAIGNER – Automatic Execution of Selected Test Cases 119
Figure 53 "INPUT VIA" Construct 166

LIST OF TABLES

Table 1 Mapping Key Requirements onto SDL Features 62
Table 2 Key Testing Requirements and Extensions in SIMPL-T 64
Table 3 TTCN Matching Mechanisms 75
Table 4 Calculation of the Result Variable *R* 82
Table 5 Calculation of the Final Verdict 84
Table 6 The Strengths and Limitations of SIMPL-T Comparing to TTCN 131
Table 7 The SIMPL-T Syntactic Meta-notation 169

TABLE OF CONTENTS

ABSTRACT .... **I**

ACKNOWLEDGEMENTS .... **II**

ABBREVIATIONS .... **IV**

LIST OF FIGURES .... **VI**

LIST OF TABLES .... **VIII**

TABLE OF CONTENTS .... **IX**

CHAPTER 1. INTRODUCTION **1**

1.1. BASIC TERMS AND DEFINITIONS 1
1.1.1. Requirements Specification and SDL *1*
1.1.2. Black-Box Testing and White-Box Testing *2*
1.1.3. Verification and Validation *3*
1.1.4. Conformance Testing and Functional Testing *3*
1.1.5. Conformance Testing and TTCN *4*
1.1.6. Validation Testing and Conformance Testing *4*
1.1.7. Computer-Aided Test Case Generation and Test Tools *5*
1.1.8. Other Definitions *6*
1.2. MOTIVATION AND STATEMENT OF RESEARCH PROBLEM 9
1.2.1. Background *9*
1.2.2. Related Work *10*
1.2.3. Problems with Existing Approaches *12*
1.2.4. Motivation *13*
1.2.5. Statement of Research Problem: *14*
1.3. OVERVIEW OF CONTRIBUTIONS AND SCOPE OF THE THESIS 14
1.4. ORGANIZATION OF THE THESIS 16

CHAPTER 2. BACKGROUND: FORMAL LANGUAGES AND TESTING CONCEPTS **17**

2.1. THE INTERNATIONAL STANDARD TEST SPECIFICATION LANGUAGE: TTCN-3 17
2.1.1. History *18*
2.1.2. Basic Concepts *19*
2.1.3. Support Tools *25*
2.2. OTHER RESEARCH 25
2.2.1. MSC and Test Specification *25*
2.2.2. UML and Test Specification *26*
2.2.3. URN/UCM and Test Specification *28*
2.2.4. LOTOS and Test Specification *29*

2.2.5. Summary ... *30*
2.3. SEMANTIC RELATIONSHIP BETWEEN SDL AND TTCN ... 30
2.4. BASIC TESTING CONCEPTS ... 34
2.4.1. Test Architecture Defined by ITU-T Z.500 ... *34*
2.4.2. Test Architecture Defined by ISO 9646 ... *36*
2.4.3. Test Case and Test Suite ... *36*
2.4.4. Test Case Behavior ... *37*
2.4.5. Observation and Reporting ... *37*
2.4.6. Key Requirements of Test Specification Languages ... *38*

CHAPTER 3. SDL: OVERVIEW AND SUITABILITY FOR TEST SPECIFICATIONS ... 40

3.1. OVERVIEW: BASIC CONCEPTS AND HISTORY ... 40
3.1.1. History ... *41*
3.1.2. Basic Concepts ... *41*
3.1.3. SDL-2000 ... *45*
3.1.4. Support Tools ... *46*
3.2. SUITABILITY AND LIMITATIONS OF SDL FOR TEST SPECIFICATIONS ... 47
3.2.1. Test Architecture – Tester(s), SUT and Test Context ... *48*
3.2.2. Test Architecture – Connection between Tester and SUT (PCOs and IAPs) ... *49*
3.2.3. Test Architecture – Communication between Tester and SUT ... *49*
3.2.4. Justification of the Test System Architecture ... *50*
3.2.5. Organization and Management of Tests ... *51*
3.2.6. Test Case Behavior - Send Stimuli to the SUT and Receive Response from the SUT ... *51*
3.2.7. Test Case Behavior – Storing and Transferring Data ... *53*
3.2.8. Test Case Behavior – Flow Control ... *54*
3.2.9. Test Case Behavior – Test Step Repetition ... *55*
3.2.10. Observation – Checking Responses ... *56*
3.2.11. Observation – Measuring the Timing of Responses ... *58*
3.2.12. Assigning and Handling of Verdicts ... *60*
3.2.13. An Example Test Case Written in SDL ... *60*
3.2.14. Summary ... *62*

CHAPTER 4. SIMPL-T: SDL INTENDED FOR MANAGEMENT AND PLANNING OF TESTS ... 63

4.1. OVERVIEW OF APPROACH AND SCOPE OF SIMPL-T ... 63
4.2. ORGANIZATION AND MANAGEMENT OF TESTS ... 65
4.2.1. TTCN Test Suite Structure ... *65*
4.2.2. SIMPL-T Test Suite ... *66*
4.2.3. Syntax of Test Suite in SIMPL-T ... *69*
4.2.4. Examples ... *71*
4.3. CHECKING RESPONSES ... 72
4.3.1. Specifying Expected Gate ... *72*
4.3.2. Specifying Expected Values of Parameters and Matching Mechanism ... *73*
4.3.3. Syntax of SIMPL-T Input ... *80*
4.4. ASSIGNING AND HANDLING OF VERDICTS ... 81
4.4.1. Verdict handling in TTCN ... *81*
4.4.2. Syntax of SIMPL-T Verdict Assigning and Handling ... *85*
4.5. AN EXAMPLE OF A COMPLETE TEST CASE WRITTEN IN SIMPL-T ... 86
4.6. SUMMARY ... 88

CHAPTER 5. CASE STUDY: TESTING A TRAFFIC CONTROLLER SPECIFICATION USING SIMPL-T ... 90

5.1. BACKGROUND: THE REFERENCE SPECIFICATION FOR SDL'03 DESIGN CONTEST ... 91
5.2. TASK FORCE CONCEPTS USED IN SAFIRE ... 93
5.2.1. A System and Its Environment ... *93*

5.2.2. Block, Process and Agent *95*
5.2.3. Class, Agent Type and Agent *95*
5.3. SUMMARY OF SDL'03 DESIGN CONTEST REQUIREMENTS 96
5.4. SPECIFYING THE TRAFFIC CONTROLLER SYSTEM USING SDL (SUT) 98
5.4.1. The System and Its Environment *98*
5.4.2. System Architecture *99*
5.4.3. The Processor Agent Type *101*
5.4.4. The Controller *105*
5.5. TEST SYSTEM CONFIGURATION IN THE SAFIRE ENVIRONMENT 107
5.5.1. A User Interface for Interactive Simulation *107*
5.5.2. The Test System Configuration in the SAFIRE Environment *108*
5.6. TEST SUITE DESIGN IN SIMPL-T 109
5.7. TEST SUITE DEVELOPMENT USING SIMPL-T IN SAFIRE ENVIRONMENT 111
5.7.1. Gates and Signal List Declarations *111*
5.7.2. Timer and Variable Declarations *112*
5.7.3. Procedure Declarations *113*
5.7.4. Test Cases *115*
5.8. TEST HARNESS AND TESTER-SUT CONNECTION 118
5.9. TESTER AND TEST CASE EXECUTION 119
5.10. TEST RESULTS OBSERVATIONS AND MODIFICATIONS MADE 120
5.11. OBSERVATIONS AND ASSESSMENT OF CASE STUDY 121

CHAPTER 6. EVALUATION OF APPROACH 122

6.1. STRENGTHS AND WEAKNESSES OF SIMPL-T FROM CASE STUDY 122
6.2. COMPARISONS WITH TTCN-BASED APPROACHES 123
6.3. INDUSTRIAL APPLICABILITY 126
6.4. ASSESSING SUPPORT FOR IMPORTANT TEST CONCEPTS 127
6.4.1. Ordering Problem in Test Scenarios *127*
6.4.2. Concurrency *128*
6.4.3. Race Condition *129*
6.5. OVERALL ASSESSMENT OF OUR APPROACH AND THE SIMPL-T LANGUAGE 131

CHAPTER 7. CONCLUSIONS AND FUTURE WORK 135

7.1. SUMMARY 135
7.2. CONTRIBUTIONS OF THESIS 136
7.3. FUTURE WORK 138

REFERENCES 139

APPENDIX A: SDL TASK FORCE 145

A.1. INTRODUCTION 145
A.2. TASK FORCE EDITORIAL COMMITTEE 146
A.3. PROPOSAL TO AGM ASSEMBLY SDL '03 146

APPENDIX B: TTCN STANDARD REFERENCE 149

B.1. TTCN TEST SUITE 149
B.2. CONSTRAINTS FOR RECEIVE EVENTS AND MATCHING MECHANISMS 153
B.3. VERDICT 160

APPENDIX C: SEMANTICS OF SIMPL-T 164

C.1. ORGANIZATION AND MANAGEMENT OF TESTS 164
C.2. SPECIFYING EXPECTED GATE A RESPONSE ARRIVES ON 166
C.3. INPUT AND MATCHING MECHANISM 167
C.4. ASSIGNING AND HANDLING OF VERDICTS 168

APPENDIX D: SYNTACTIC DEFINITION OF SIMPL-T 169

D.1. TEST SUITE 170
D.2. INPUT 172
D.3. VERDICT 173

CHAPTER 1. INTRODUCTION

As the earliest stage in the software development lifecycle, the correctness and completeness of the requirements specifications are critical to all subsequent development. Validation of the specifications against the user requirements is an effective means for software quality and development cost reduction.

This chapter briefly introduces the concepts in requirements specification and validation. It addresses the problems in this domain and motivation of our research. It also defines terms used in this thesis. Finally, it describes the scope and contributions of our research, and organization of this thesis.

1.1. Basic Terms and Definitions

This section provides definitions of general concepts used throughout the thesis.

1.1.1. Requirements Specification and SDL

A requirements specification is a set of assertions that describe how the software behaves from the perspective of the user. It is widely accepted that a key to the success of

software development lies in a thorough and rigorous specification. This requires tailored specification languages that fulfill the following requirements[Ells+97]:

- Well-defined concepts
- Unambiguous, clear, precise a concise specifications
- Ability to analyze the correctness and completeness of specifications
- Ability to check conformance of an implementation with respect to its specification
- Ability to check the consistency of specifications
- Suitability for the use of computer-based tools to develop, maintain, verify, validate, analyze and simulate specifications

SDL was developed and widely accepted as a formal specification language. It provides a set of well-defined concepts, which fulfill the requirements listed. These will be discussed in detail in Chapter 3.

1.1.2. Black-Box Testing and White-Box Testing

Black-box testing[Prob82] is a technique, in which testing is conducted without knowledge of the internal workings of the item being tested. For example, when black-box testing is applied to software engineering, the tester would only know the "legal" inputs and what the expected outputs should be, but not know how the program actually arrives at those outputs. It is because of this that black-box testing can be considered testing with respect to the specifications; no other knowledge of the program is necessary. For this reason, the tester and the programmer can be independent of one another, avoiding programmer bias toward his own work. Also, due to the nature of black-box testing, the test planning can begin as soon as the specifications are written.

White-box testing (sometimes called clear-box testing or glass-box testing) is a technique where test data are derived from direct examination of the code to be tested. For white-box testing, the test cases cannot be determined until the code has actually been written.

1.1.3. Verification and Validation

Validation is defined by ETSI (European Telecommunications Standards Institute) as: "process, with associated methods, procedures and tools, by which an evaluation is made that a standard can be fully implemented, conforms to rules for standards, and satisfies the purpose expressed in the record of requirements on which the standard is based; and that an implementation that conforms to the standard has the functionality expressed in the record of requirements on which the standard is based" [ETSEG9902][ETSEG9905].

IEEE (Institute of Electrical and Electronics Engineers Inc) defines verification as: The process of evaluating a system or component to determine whether products of a given development phase satisfy conditions imposed at start of that phase.

1.1.4. Conformance Testing and Functional Testing

Conformance testing[ISO9646][ITUZ500][Wvong98][Knig93] is the process of verifying that an implementation performs in accordance with a particular standard, specification, or environment.

Conformance testing is not intended to be exhaustive, and a successfully passed test suite does not imply a 100-percent guarantee of correctness or consistence. But it does ensure, with a reasonable degree of confidence, that the implementation is consistent with its

specification, and it does increase the probability that implementations will inter-work. The goal of conformance testing is to verify system interoperability by conformance to a standard.

The goal of functional testing is to take a user's view of the system, and check that customer requirements have been met. In particular, the actions and reactions of the system are considered from the user's viewpoint, to ensure that the system behaves as the user expects[Prob+99].

Functional testing is often used interchangeably with validation. In this thesis, we use the term validation.

1.1.5. Conformance Testing and TTCN

ISO 9646[ISO9646] is a framework for conformance testing. ISO 9646 incorporates the TTCN language as ISO 9646-3. One of the basic premises of this conformance standard is that the implementation of the protocol, called an implementation under test (IUT) is a black box.

TTCN is designed for the specification of Abstract Test Suites, which we will discuss in more detail in Chapter 2.

1.1.6. Validation Testing and Conformance Testing

The testing concept we used in this thesis is similar to the concept used in SPEC-VALUE[Amyot01].

The validation testing offered by SPEC-VALUE is different from the conventional conformance testing standardized in [ISO9646]. Conformance testing usually requires a model (e.g. a formal protocol specification) from which a black-box test suite is derived and then used to test an implementation or another model.

In SPEC-VALUE, the test suite is derived from informal requirements and semi-formal scenarios (the UCMs), and the goal is to create and check the first formal model (the LOTOS specification). This test suite, composed of test cases whose purposes can be related to the requirements, is used to validate the model against the requirements, hence the term validation.

Similar to SPEC-VALUE, the test suites we discuss in this thesis are also derived from informal requirements, and the goal is also to use these test suites to validate the SDL specifications against the requirements. Therefore, we use the term validation instead of conformance testing.

1.1.7. Computer-Aided Test Case Generation and Test Tools

Testing software is hard work. The introduction of formal description techniques (FDT) made it possible to develop test tools. With the help of test tools, it is possible to test specifications automatically or semi-automatically, which is faster, more efficient, more accurate and more importantly – repeatable.

However, test tools are not a substitute for software testers. Especially since validation involves user requirements, complete automation is difficult to achieve in practice.

ETSI studied various tools for Computer Aided Test Generation[ETSTR9807] from the point of view of testing methodology. As a result, they determined that none of the tools

studied offered a real alternative to the intellectual processes that are applied when producing test purposes manually. The tools supported used either:

- semi-automated techniques, which rely on user interaction with a simulator to generate MSCs[ITUZ120] which express test purposes; or
- fully-automated techniques, which systematically base test purposes on single state transitions.

In this thesis, we discuss testing specifications rather than implementations, therefore, the test cases have to be independent from the specifications. Thus, when we say automatic testing, we mean "automated test execution" instead of "automated test generation".

1.1.8. Other Definitions

- *Reactive System*

An reactive system, such as an operating system or an elevator control system, is a system which ideally never terminates and is intended to maintain some interaction with its environment. The main features of reactive system are: most behavior is in reaction to internal or external events, as opposed to data or computation.

- *Finite-State Machine*

A finite-state machine (FSM) is an abstract machine that has only a finite, constant amount of memory. It can be conceptualized as a directed graph. There are a finite number of states, and each state has transitions to other states. There is an input string that determines which transition is followed (some transitions may be from a state to itself). A FSM may be represented using a state transition table or a state diagram[Wiki].

- *Extended Finite-State Machine*

An extended finite-state machine (EFSM) introduces variables and guard conditions in addition to the explicit states of a FSM. These variables become implicit states. EFSM is used to solve the problem of state explosion.

- *Test Case*

A test case is a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement[TSWP].

- *Test Suite*

A test suite is a collection of one or more test cases for the software under test[TSWP].

- *Test Automation*

The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions[TSWP].

- *IUT and SUT*

In ITU-T X.290, IUT (Implementation under Test) is defined as "The part of a real Open System which is to be studied by testing, which should be an implementation of one or more OSI Protocols in an adjacent user/provider relationship". SUT (System under Test) is defined as "The real Open System in which the Implementation under Test resides".

In the context of this thesis, we only discuss testing SDL specifications. Therefore, we use SUT representing *Specification under Test.*

- *Test Harness*

A program or a test tool that connects the tester to the SUT.

- *Tester*

A program or a test tool that executes an automated test suite. A tester normally comprises two parts -- a test driver and a test comparator.

- *Test Driver*

A program or test tool used to execute software against a test suite[TSWP].

- *Test Comparator*

A test tool that compares the actual outputs produced by the software under test with the expected outputs for that test case[TSWP].

1.2. Motivation and Statement of Research Problem

1.2.1. Background

As specifications are defined at the earliest stage in the software development lifecycle, their correctness and completeness are critical to all subsequent development. To write clear and unambiguous specifications, the formal language SDL (Specification and Description Language)[ITUZ100] was introduced and has been widely accepted.

Following the standardization of SDL, methodologies and techniques have been developed for automatically generating test cases from SDL specifications[ETSTR9807][Mans+01a], and then using these test cases to validate the implementations. This work can help to gain confidence that the system implemented is consistent with its specification.

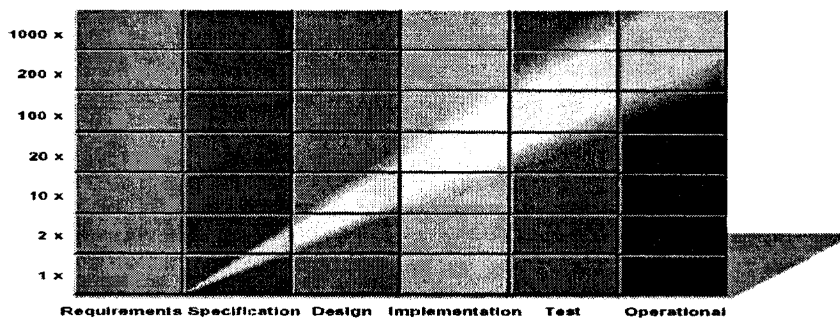
However, before a system is developed, its specification should be validated to ensure that it conforms to the original requirements. This can be a significant means of avoiding the costs of design errors. Experience has shown that validation is capable of finding most errors in executable system specifications. Moreover, once a specification has been validated, the numbers of errors found during product development is significantly reduced[ETSETR95]. This can save development cost dramatically. Figure 1[Hydb01] illustrates relative error correction cost in a software life cycle.

However, test cases generated from a specification cannot be used to validate the specification itself. Test cases have to be designed independently based on user

requirements. Moreover, to write clear and unambiguous test cases, a formal test specification language is needed.

Much research has been done in this area. TTCN and SDL are both used as test specification languages.

Figure 1 Relative Error Correction Cost in a Software Life Cycle



1.2.2. Related Work

The test specification and test implementation language TTCN (Tree and Tabular Combined Notation) is part of the conformance testing standard ISO9646[ISO9646]. TTCN was defined to describe abstract test suites. Its latest version, TTCN-3 (Testing and Test Control Notation) was developed at the European Telecommunications Standards Institute (ETSI)[ETS2003]. It has been expanded to become a truly multi-purpose test language, which can be used for conformance, interoperability, function and regression testing.

TTCN can be used to validate SDL specifications and some commercial tools such as Telelogic's TAU/Tester[Telelogic] include both SDL and TTCN languages in their tool sets.

SDL has also been used to test SDL specifications in some experiments. For example:

Bhaskar and Vimal from Motorola India Electronics Ltd[Bhas+01] conducted SDL-based test automation for a real-time application of SIP (Session Initiation Protocol) in VOIP domain. In this approach, the test cases were written directly in SDL as procedures. These procedures were grouped in Test Groups to organize them as test suites. The master controller test process calls these groups and executes a particular test case based on an input from the tester. The test input was in the form of a script, which consists of a test suite number, a test case number, input parameters and the expected result. After the execution of the test case, the result received from the application will be passed to the verdict handler, which either logs the result or informs the test engineer.

Tretmans[Tret+92] used a similar approach. He modeled the verdict assignments by variables associated to each test case. The verdict assignments that result from test case execution may be passed to the environment via signals.

Lopez applied SDL to formal analysis of security systems[Lopez+03]. He created two additional processes, an intruder and an observer, and used them to test a system's vulnerability.

Tools were also developed to support writing test cases in SDL. For example, the tool TESTGEN-SDL[TGeni] developed by Institut National des Telecommunications (France) allows test sequences written in SDL format.

1.2.3. Problems with Existing Approaches

As TTCN-3 becomes more flexible and powerful, it has also become more complex and more difficult to learn. Tools supporting such a powerful language have become very expensive. Validating SDL specifications only needs a small part of the language. Therefore, for many developers, researchers, and organizations, it is not economical or efficient to use TTCN to validate SDL specifications.

Furthermore, because of the general purpose and wide scope of TTCN, the test cases written in TTCN are not very simple. Many researchers prefer to invent their own approaches rather than to use TTCN.

SPEC-VALUE[Amyot01] used LOTOS processes to write test cases where test events are transformed into LOTOS events. SPEC-VALUE also defined its own notation for test purposes, because, as Amyot, the author, states, "this representation is significantly simpler than a full-fledged standard testing language such as TTCN. TTCN is not used here because it uses a format and a level of detail that would lead to unnecessarily long and verbose descriptions of test purposes" [Amyot01].

With SDL features, one can develop a test system. However, existing versions of SDL do not support all the features needed in a test specification language. In Bhaskar's approach[Bhas+01], test cases were written in procedures and they cannot be organized in groups. This had to be handled manually. Verdicts cannot be handled by SDL; therefore programs have to be written to handle them.

In Lopez's approach[Lopez+03], additional processes were defined to act as the tester. To finish a test, a "stop" statement has to be used to end a process.

1.2.4. Motivation

TTCN, although powerful, is not defined for testing SDL specifications, while SDL has the potential to be a good test specification language. Bhaskar[Bhas+01] identified the SDL concepts which are suitable for test specification:

- It has a strong timer concept, which eases the timing-related tests and aids in load generation at some intervals.
- It has optional and default parameters, which enables data manipulation
- Concurrency is well supported by the language, which enable the simulation of non-existing modules very easily.
- External C code can be integrated. This enable the reuse of data types between the application and test system.
- Predefined libraries such as linked lists, flat files, random number generation are handy.

The benefits of using the same language for requirements specification and test specification can be summarized as:

- Data sharing capability
- Portability
- Components (signals, user-defined data types) reusability

1.2.5. Statement of Research Problem:

To define and investigate the applicability of a simple, useful and efficient language for describing tests of SDL specifications

We call this new language SIMPL-T (SDL Intended for Management and Planning of Tests). It is defined based on existing SDL features with minimal extensions to provide essential test specification functionality, and is shown to address the above Statement of Research Problem.

1.3. Overview of Contributions and Scope of the Thesis

This thesis based on existing SDL features defines a new test specification language SIMPL-T.

It identifies which SDL concepts and elements can be used for testing and suggests how they can be used. It also identifies the limitations SDL has for testing, and analyzes corresponding TTCN concepts which can overcome these limitations. Extensions are defined based on these TTCN concepts with modification and simplification to assure the simplicity of SIMPL-T.

It also presents experience gained during a case study, a realistic, real-time application – a traffic controller- was specified using SDL, and validated using SIMPL-T.

This case study was conducted in SAFIRE environment, which incorporated SIMPL-T into the successful development of the tool set.

The scope of this thesis has been partially influenced by the *SDL Task Force*[SDLTF], which is recognized by the international *SDL Forum[SDLFO]* and is working towards a Focus Group of WP 3 of ITU-T Study Group 17[ITUSG17].

The *SDL Task Force* has the mandate to identify the "simplest, useful subset of SDL" together with extensions for test requirements. It works on the basis of defined questions to be answered, in the same way that the ITU-T works. The task force question (Q 2.5) corresponds to the extensions for test requirements.

The work of this thesis contributed to the Task Force Q 2.5 and was included as part of a draft document for submission to the *SDL Forum* at the *SAM'04 workshop*[SAM04].

As the *SDL Task Force* is dealing with the "simplest, useful subset of SDL", the same criteria have been used for defining the "simplest, useful" extensions for test requirements.

Since SDL was not defined for testing, SDL elements are not all relevant for testing. As TTCN is only used to define the extensions, only relevant SDL and TTCN concepts are discussed.

There are many issues related to the different versions of SDL and TTCN, as well as ongoing activities. There is also a wide-range of details associated with each language. This thesis therefore focuses on the fundamental concepts involved, going into detail only where necessary.

1.4. Organization of the Thesis

The rest of this thesis is divided into six chapters:

- Chapter 2 introduces and compares existing test specification related languages. As an international standard test specification language, TTCN is introduced in detail. This chapter also summarizes key requirements for test specification and gives justifications.
- Chapter 3 discusses the suitability of SDL for test specification.
- Chapter 4 based on the limitations of SDL for test specification identified in Chapter 3, maps the corresponding requirements to TTCN. As a result, possible extension to SDL is address for the new test specification language SIMPL-T.
- Chapter 5 uses a case study to illustrate how the SIMPL-T concepts defined in this thesis are used in developing and testing a Reference Specification for a reactive application – a traffic controller. It also demonstrates a successfully developed commercial test tool SAFIRE based on SIMPL-T concepts.
- Chapter 6 evaluates the case study and the approach.
- Chapter 7 gives conclusions and suggestions for future work.

CHAPTER 2. BACKGROUND: FORMAL LANGUAGES AND TESTING CONCEPTS

There are many languages which may be used to validate specifications. In Chapter 1, we briefly addressed the most accepted one, TTCN[ISO9646][ETS2003]. In this chapter, we give a thorough introduction of TTCN, and analyze its relationship with SDL. We also introduce the other languages and approaches which are relevant to test specification. At the end, we discuss basic testing concepts, which will be used as the foundation for later discussions.

2.1. The International Standard Test Specification Language: TTCN-3

The Testing and Test Control Notation (TTCN-3)[ISO9646] is a language used to write detailed test specifications. TTCN has been used to specify tests for many kinds of applications, including mobile communications (GSM, 3G, TETRA), wireless LANs (Hiperlan/2), cordless phones (DECT), Broadband technologies (B-ISDN, ATM), CORBA-based platforms and Internet protocols such as IPv6, SIGTRAN, SIP and OSP.

The latest version of the language, TTCN version 3 (TTCN-3)[ETS2003][Schu+02], is standardized by ETSI(ES 201 873 series) and ITU-T (Z.140 series). In earlier versions, TTCN stood for Tree and Tabular Combined Notation[Prob+92].

2.1.1. History

Work on Tree and Tabular Combined Notation (TTCN) first started in 1984 in ISO/IEC JTC 1/ SC 21 and in CCITT SG VII as part of the work on OSI conformance testing methodology and framework. It was first standardized as twin texts ISO/IEC 9646-3 and CCITT Rec. X.292 in 1992 as one of the set of seven twin texts (ISO/IEC 9646 and CCITT X.290-series) on OSI conformance testing. Since then TTCN was widely used for describing protocol conformance test suites in standard organizations such as ITU-T, ISO/IEC, ATM Forum, and ETSI, and in industry[ITUSG7].

However, the first edition of TTCN was not suitably designed to describe the concurrent behavior within the tester or within IUT or between them, and the need for an extension to deal with concurrent behavior efficiently was soon recognized. In the process of extension, in addition to a concurrency mechanism, the concepts of module and package were introduced to TTCN to increase reusability and to achieve encapsulation. Also manipulation of ASN.1[ITUX680] encoding was made possible beyond simple syntax declaration. The TTCN extended in this way was adopted as TTCN version 2 (TTCN-2) in ISO/IEC in 1998 and in ITU-T in 1998[ITUSG7].

In spite of the improvements in TTCN-2, TTCN was designed from the beginning with OSI-based protocol conformance testing in mind, and even TTCN-2 was not adequate for various kinds of testing such as: interoperability testing, robustness testing, regression testing, system testing and integration testing or for various emerging testing application areas such as mobile protocols, Internet protocols, service testing, module testing,

CORBA-based platform testing and API testing. Therefore a more flexible and powerful test description language was called for. In order to meet such a need, the Special Task Force (STF) 133 and 156 of ETSI started working on a new version of TTCN in 1998 and completed it in October 2000[ITUSG7].

TTCN-3 retained the proven features of TTCN-2 but was designed to provide the new features listed above. The single most visible difference is that the previous versions adopted as the main description notation the Tabular Notation; and its textual counterpart a machine-processible language for translation to programs. Whereas in TTCN-3, a common programming language-like notation is adopted as the core language with the Tabular Notation and MSC as presentation languages[ITUSG7].

In October 2000, TTCN-3 was approved in ETSI. Then it was submitted to ITU-T as Z.140 (series). Z.140 and Z.141 were approved in July 2001.

2.1.2. Basic Concepts

TTCN-3 has removed many of the peculiarities of the specific application domain of OSI and conformance testing that was present in previous versions of the Tree and Tabular Combined Notation, TTCN. However, TTCN-3 has retained much of the well-proven testing-specific capabilities of earlier versions[ETS2003]:

- The ability to specify dynamic concurrent testing configurations;
- Operations for procedure-based and message-based communication;
- The ability to specify encoding information and other attributes (including user extensibility);
- The ability to specify data and signature templates with powerful matching mechanisms;

- Type and value parameterization;
- The assignment and handling of test verdicts;
- Test suite parameterization and test case selection mechanisms;
- Combined use of TTCN-3 with ASN.1[ITUX680];
- Well-defined syntax, interchange format and static semantics;
- Different presentation formats (e.g. tabular and graphical presentation formats);
- A precise execution algorithm (operational semantics).

TTCN-3 is based on concepts which are independent of any syntax. These concepts are introduced below: modules, test cases, test systems, test verdicts, and components[Grab+03].

Modules and test cases

The principal building blocks of TTCN-3 are modules. For example, a module may define a fully executable test suite or just a library. A module consists of an (optional) definitions part, and an (optional) module control part[ETS2003].

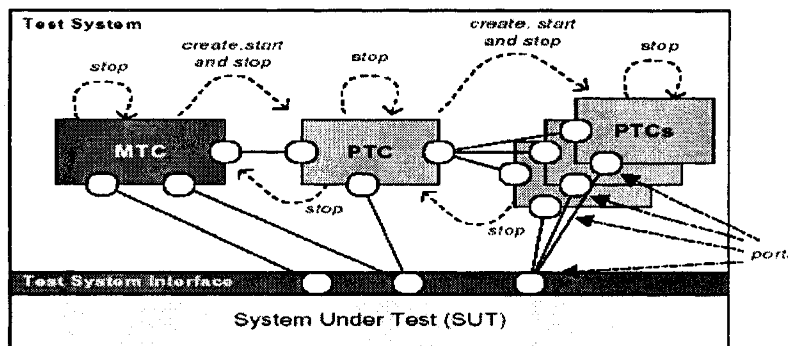
The module definitions part specifies the top-level definitions of the module. These definitions may be used elsewhere in the module, including the module control part. The module control part is the main program of a TTCN-3 module. It describes the execution sequence (possibly repetitious) of the actual test cases. Test cases are defined in the module definitions part and then called in the control part.

The test cases define the behaviors which have to be observed to check whether the system under test (SUT) passes the test or not. Like a module, a test case is considered to be a self-contained and complete specification that checks a test purpose. The result of a test case execution is a test verdict [Grab+03].

Test system

A *test case* is executed by a *test system*. TTCN-3 allows the specification of dynamic and concurrent *test systems*. A *test system* consists of a set of interconnected *test components* with well-defined *communication ports* and an explicit *test system interface*, which defines the boundaries of the *test system* (**Figure 2 [Grab+03]**).

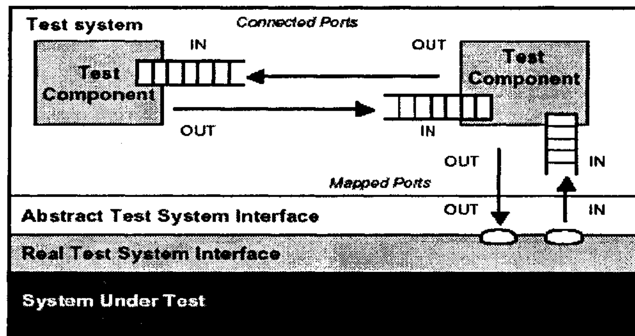
Figure 2 Dynamic and Concurrent Test System



Within every *test system*, there is one *main test component* (MTC). All other test components are called *parallel test components* (PTCs).

For communication purposes, each *test component* owns a set of local *ports*. Each *port* has an in- and an out-direction (Figure 3 [Grab+03]). The in-direction is modeled as an infinite FIFO queue, which stores the incoming information until it is processed by the *test component* owning the *port*. The out-direction is directly linked to the communication partner, i.e., outgoing information is not buffered.

Figure 3 Conceptual View of a TTCN-3 Test System



For the communication among test components and between test components and the SUT, TTCN-3 supports message-based and procedure-based communication. Message-based communication is based on asynchronous message exchange and procedure-based communication is based on calling procedures in remote entities.

Meaning and Calculation of Test Verdicts

TTCN-3 provides a special test *verdict* mechanism for the interpretation of test runs. This mechanism is implemented by a set of *predefined verdicts, local and global test verdicts* and operations for reading and setting *local test verdicts.*

The *predefined verdicts* are *pass, inconc, fail, error and none.* They can be used for the judgment of complete and partial test runs.

During test execution, each test component maintains its own *local test verdict.* When changing the value of a *local test verdict,* special overwriting rules are applied. The overwriting rules only allow that a *test verdict* becomes worse, e.g., a *pass* may change to *inconc* or *fail,* but a *fail* cannot change to a *pass* or *inconc.*

In addition to the *local test verdicts*, the TTCN-3 run-time environment maintains a *global test verdict* for each test case. The *final global test verdict* is returned to the module control part when the test case terminates.

Components and Behavior Definitions

In TTCN-3, behavior is related to a control component and to test components. The control component executes the control part of a module and the test components execute the test cases. Behavior definitions for control and test components are the *module control part, test cases, altsteps* and *functions*.

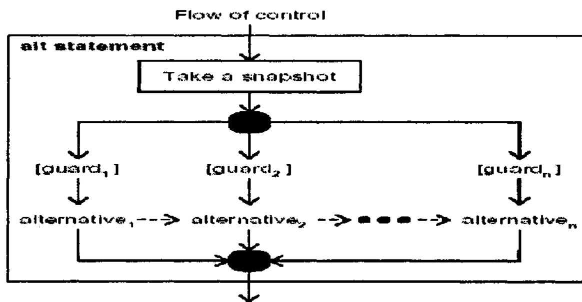
Alternatives and Snapshots

A specific feature of the TTCN-3 (also in previous versions) semantics is *snapshots*. *Snapshots* are related to the behavior of components. They are needed for the branching of behavior due to the occurrence of timeouts, the termination of test components and the reception of messages, procedure calls, procedure replies or exceptions. In TTCN-3, this branching is defined by means of *alt statements*.

An *alt statement* describes an ordered set of alternatives, i.e., an ordered set of alternative branches of behavior (Figure 4 [Grab+03]). Each alternative has a guard. A guard consists of several preconditions, which may refer to the values of variables, the status of timers, the contents of port queues and the identifiers of components, ports and timers. An alternative becomes executable, if the corresponding guard is fulfilled. If several alternatives are executable, the first executable alternative in the list of alternatives will

be executed. If no alternative becomes executable, the alt statement will be executed again.

Figure 4 Meaning of alt-statements



The evaluation of several guards needs some time. During that time, preconditions may change dynamically. This will lead to inconsistent guard evaluations if a precondition is verified several times in different guards. TTCN-3 avoids this problem by using *snapshots*. *Snapshots* are partial module states, which include all information necessary for the evaluation of alt statements. A *snapshot* is taken, i.e., recorded, when entering an alternative. For the verification of preconditions, only the information in the current *snapshot* is used. Thus, dynamic changes of preconditions or messages arriving while alternatives are being evaluated do not influence the evaluation of guards.

Semantics of Timers

TTCN-3 provides a *timer* mechanism. *Timers* are local to components. A component can start and stop a *timer,* check if a *timer* is running, read the elapsed time of a running *timer* and process *timeout* events after *timer* expiration.

2.1.3. Support Tools

As a standardized notation, TTCN-3 is part of a well-defined test framework, which provides the possibilities for tool suppliers to develop practical and efficient tools to support this technology, such as Testing Technologies' tool chain for TTCN-3(TTspec)[TTech], Da Vinci Communications' TERZO[DaV] and Telelogic's Telelogic TAU/Tester[Telelogic].

2.2. Other Research

2.2.1. MSC and Test Specification

Message Sequence Charts(MSC)[ITUZ120] is a formal language used to describe the interaction among a number of independent message-passing instances. MSC-2000[Haugen00] is the latest recommendation from the International Telecommunication Union(ITU) defining MSC. Earlier recommendations of MSC have been issued in 1992 and in 1996.

The purpose of recommending MSC is to provide a trace language for the specification and description of the communication behavior of system components and their environment by means of message interchange. Since in MSCs the communication behavior is presented in a very intuitive and transparent manner, particularly in the graphical representation, the MSC language is easy to learn, use and interpret. In connection with other languages it can be used to support methodologies for system specification, design, simulation, testing, and documentation[ITUZ120].

MSCs are often used to supplement the SDL specifications[Mans+01]. They are useful for giving an overview of specific scenarios and have an important role to play in the understanding and development of the SDL specifications. As stated in the ETSI Guide *Guidelines for facilitating validation and the development of conformance tests*[ETSEG9902], "the MSCs cannot be considered normative in the same sense as the SDL. A system is not required to follow the sequence described in an MSC unless it is the only sequence that is valid according to the SDL. Thus, MSCs should be used to supplement the SDL to give descriptions of some valid sequences of behavior."

MSCs are also used as the basis of test sequences[Prob+01]. ETSI Technique Report *Experiences of the application of SDL and CATG tools for the development of Abstract Test Suites(ATSs)*[ETSTR9807] says "While MSCs can be a useful complement to documenting test purposes they should not be regarded as a complete substitute for textual test purposes, which will often contain additional and necessary information not easily expressible in the MSC format (e.g. verdict assignment). Also, MSCs cannot cope with dynamic alternatives."

In summary, MSC is a useful language to supplement both SDL specifications and test materials, but by itself, MSC is not sufficient to work as a test specification language.

2.2.2. UML and Test Specification

The Unified Modeling Language(UML)[OMG03][KobrUML01][Alhir99] is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as

concrete things such as programming language statements, database schemas, and reusable software components.

Signaling the end of the method wars, the Object Management Group (OMG) first standardized the Unified Modeling Language in 1997. The software industry rapidly accepted it as the standard modeling language for specifying software and system architectures. Although UML is primarily intended for general-purpose modeling, it is receiving extensive use in diverse specialized areas, such as business process modeling and real-time-systems modeling[Bjor03].

One of the harshest criticisms of UML 1.x is that it is too large and complex to implement and use[KobrUML20].

The newest version UML 2.0 made a major revision to UML 1.x. Bugs are fixed and many new language features are added.

UML State Diagrams are used to express the dynamic behavior of a system, to demonstrate how the objects interact dynamically at different times during the execution of the system[Erik+98]. They are very similar to SDL conceptually[ITUZ109]. There is much interest in deployment of SDL systems using UML[Bauer01]. For example, the Telelogic TAU Generation 2[Telelogic] tool set embedded SDL inside UML 2.0.

In summary, whether SDL specifications exist independently or are embedded in UML, they need to be validated. However, UML as a modeling language is not designed for testing, and thus cannot be used to test SDL specifications.

2.2.3. URN/UCM and Test Specification

The Use Case Map notation (UCM)[ITUZ152][UCM] is being standardized as a scenario notation part of the User Requirements Notation (URN)[ITUZ150][UCM][Amyot03] in the Z.150 series.

URN targets the representation of requirements for future telecommunication systems and services. URN contains one notation for the representation of non-functional requirements (NFRs) and a complementary scenario notation for functional requirements. The NFR notation is the Goal-oriented Requirement Language (GRL). The scenario notation is based on Use Case Maps (UCM).

UCM graphical models describe functional requirements and high-level designs with causal scenarios, superimposed on structures of components[He+03][Buhr98].

Since UCMs are non-executable notations, they have to be transformed into scenarios represented in more formal and executable languages to be validated. Some experiments focus on generating MSCs from UCMs, and then converting these MSCs into SDL specifications automatically[He+03]. Since neither the methodology, nor the technique, nor the support tools in this domain is mature, the generated SDL specifications are not reliable. Validation of these SDL specifications is a must.

2.2.4. LOTOS and Test Specification

LOTOS (Language of Temporal Ordering Specification)[ISO8807] is standardized as ISO/IEC 8807. It was initially based on the formal specification language CCS (Calculus of Communicating Systems). Some notation and concepts were later introduced from another similar language CSP (Communicating Sequential Processes). Data typing was considered only later in the development of LOTOS. The abstract data type language ACT ONE was adapted to allow formal specification of data types in LOTOS. An International Standard for LOTOS was completed after 9 years' work. Surprisingly, LOTOS is perhaps the first piece of mathematics to be standardized internationally. Work is ongoing provide an enhanced version of the standard called E-LOTOS (Extensions to LOTOS)[Stir].

LOTOS is a specification language that has been specifically developed for the formal description of the OSI (Open systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general. In LOTOS a system is seen as a set of processes which interact and exchange data with each other and with their environment[Bolo+87].

As a specification language, LOTOS is not developed for testing. Since TTCN provides sufficient support for LOTOS constructs used in most test cases (sequence, choice, interleave, process instantiation, value passing, etc.), TTCN can be used to test LOTOS specifications.

For simplicity, some approaches use LOTOS processes to describe test cases[Amyot01]. However, they only apply to LOTOS specifications, rather than general formal specifications, such as SDL specifications.

2.2.5. Summary

In summary, although all the languages we introduced here are somehow related to testing, none of them, except TTCN, can be effectively and efficiently used as a test language for SDL specifications.

2.3. Semantic Relationship between SDL and TTCN

As we already discussed in the previous section, among all the existing formal (or semi-formal) languages, TTCN is the only test specification language applicable to test SDL specifications. Since TTCN has a much wider application domain, many of the TTCN features and concepts are not needed for testing SDL specifications.

At the same time, TTCN and SDL have overlap in many concepts, e.g. both TTCN and SDL use ASN.1[ITUX680] to describe data types[ETSTR9905]. These overlapped concepts can help to define the SDL features which can be used for testing.

In this section, we investigate the semantic relationship between SDL and TTCN. The output from this study was useful in our SIMPL-T language development.

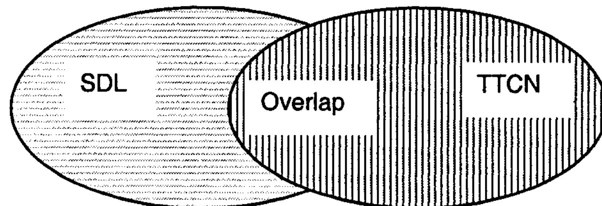
- ***SDL and TTCN have overlap***

A Common Semantics Representation (CSR) for SDL and TTCN was defined by ETSI in [ETS9306]. The term CSR refers to a representation, able to represent the semantics of

objects from different domains in a common model. Such a common model enables the investigation of semantic relations between objects of the different domains. Hence, a common semantics representation for SDL and TTCN enables the definition of formal semantic relations between SDL and TTCN specifications. Such relations could then act as a basis when developing tools for test generation, test validation, etc.

This work demonstrates that there is an overlap in the conceptual models between the languages SDL and TTCN (Figure 5).

Figure 5 SDL and TTCN Conceptual Overlap



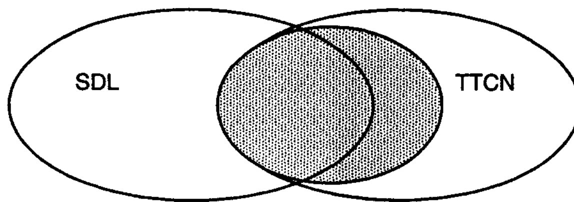
- ***Nothing outside of TTCN is needed for testing SDL specifications***

As an international standard test specification language, TTCN is flexible, powerful and feature-rich. It is applicable to the specifications of all types of reactive system tests over a variety of communication interfaces. Existing TTCN features are enough to test SDL specifications and nothing outside of TTCN is needed.

- ***Only part of TTCN is needed for testing SDL-based specifications***

Since TTCN has a much wider application domain, many of the TTCN features and concepts are not needed for testing SDL specifications. The dotted area in Figure 6 illustrates the part of TTCN needed for testing SDL specifications.

Figure 6 Only Part of TTCN is Needed for Testing SDL-based Specifications

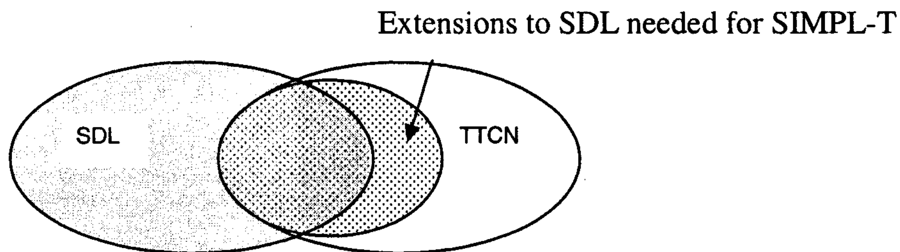


- ***Extensions to SDL can be defined to complete SIMPL-T based on the relationship between SDL and TTCN***

Part of the work of this thesis is to define the minimal extensions to SDL to complete the new test specification language, SIMPL-T.

Base on the SDL-TTCN *overlap* (Figure 5) and the TTCN features needed for testing SDL specification (Figure 6), the extensions needed to SDL can be identified, which are illustrated in the dotted area in Figure 7. We will discussion how these extensions can be done in Chapter 4.

Figure 7 Extensions to SDL



In summary, TTCN is powerful enough to test SDL specifications, but only part of TTCN is needed to test SDL specifications. Since SDL and TTCN overlap in many concepts, what are needed as extensions to SDL can be identified and these extensions can be defined based on TTCN concepts.

2.4. Basic Testing Concepts

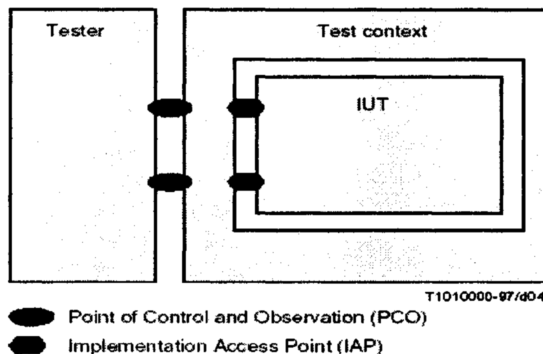
In this section we first discuss the basic testing concepts defined in ITU-T Z.500[ITUZ500] and ISO 9646[ISO9646]. We then derive the key features or requirements for a test specification language.

Testing is a means of extracting knowledge about a system by systematically experimenting with it. The experimentation is carried out by executing test cases. The execution of a test case leads to an observation from which it can be concluded whether a system has a certain property or not[ITUZ500].

2.4.1. Test Architecture Defined by ITU-T Z.500

The test architecture is defined by ITU-T[ITUZ500] as a description of the environment in which the IUT is tested. It describes the relevant aspects of how the IUT is embedded in other systems during the testing process, and how the IUT communicates via these embedding systems and with the tester. It is necessary to model the environment of the IUT because the behavior of the IUT may not be directly observable by the tester due to possible limitations in observability imposed by, e.g. an intermediate communication medium between tester and IUT. Figure 8 gives an abstract view of the test architecture[ITUZ500].

Figure 8 Test Architecture Defined by ITU-T Z.500



This test architecture consists of:

- a tester;
- an Implementation Under Test (IUT);
- a test context;
- Points of Control and Observation (PCO);
- Implementation Access Points (IAP).

The tester is the implementation of a test suite. It carries out the experiments by executing the test cases and observing the results. The tester communicates with the Test Context via the PCOs, and indirectly with the IUT via the Test Context. The tester can be sub-structured in different components (e.g. in a Lower Tester and an Upper Tester).

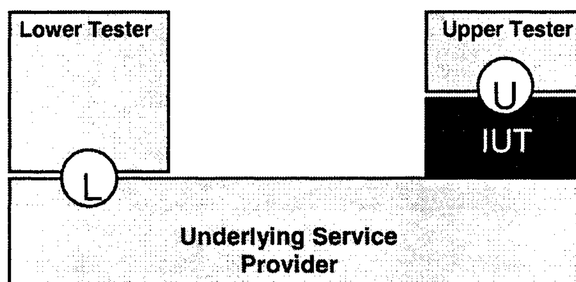
The Test Context is the system in which the IUT is embedded, and via which the IUT communicates with the tester. It relates events that occur at PCOs in the communication between the Test Context and the tester, to events that occur at the IAPs in the communication between the Test Context and the IUT.

An Implementation Access Point (IAP) is an interaction point in the test architecture where the IUT interacts with its environment, i.e. the Test Context, and via the Test Context (indirectly) with the tester. In certain cases, IAPs and PCOs may coincide.

2.4.2. Test Architecture Defined by ISO 9646

Test architecture in ISO 9646[ISO9646] is called test configuration. Figure 9 shows the basic conceptual test configuration. The upper and the lower tester connect to the IUT through PCOs, send stimuli to the IUT, and check the behavior accordingly.

Figure 9 Test Architecture Defined by ISO 9646



2.4.3. Test Case and Test Suite

The formal description of the *tester(s)* is given by a *test suite*. A *test suite* specifies the entire set of experiments which are to be executed by the *tester*. The *tester* is the implementation of the *test suite*.

The experiments that constitute a *test suite* are called *test cases.* Each *test case* specifies the behavior of the *tester* in a separate experiment that tests an aspect of the IUT, and that leads to an observation and a verdict.

A *test case* consists of a *test purpose* and a *test case behavior.*

As opposed to formal test purposes, a prose description of the objective of testing is called the informal or natural language test purpose. In TTCN, this natural language test purpose is referred to as *test purpose.* In this thesis, we refer *test purpose* the same way as in TTCN.

2.4.4. Test Case Behavior

The dynamic behavior of a test case is referred to as *test case behavior. Test case behavior* is concerned with observing and controlling sequences of interactions between the tester and the IUT. It specifies a sequence of stimuli to send to the IUT and expected responses from the IUT. It often needs to store and to transfer temporary data, to take alternative actions and to repeat actions. We refer them as storing and transferring data, Flow Control and Test Step Repetition. A test case ends when a final verdict is assigned.

2.4.5. Observation and Reporting

At the end of each test case, a *verdict* should be assigned. The value of the *verdict* can be *pass, fail or inconclusive* depending on whether the responses from the IUT are as expected.

In many instances it is not possible, or even desirable, to specify that the expected received signal carries a specific parameter (value of PDU in ISO9646). It may be more appropriate to say that a match occurs if the received value falls within certain boundaries or fulfils certain conditions. To determine whether a response is as expected, a matching mechanism must be defined in the test specification language.

2.4.6. Key Requirements of Test Specification Languages

To summarize what we discussed in previous sections, the architecture of a test system can be defined as a *tester(s)* running *test cases* against an IUT. The entities of this test system include a *tester(s)*, an *IUT*, a *Test Context*, *PCOs* and *IAPs* (Section 2.4.1 & Section 2.4.2). Since the implementations under test in this thesis are SDL specifications, we use SUT (Specification under Test) instead of IUT.

The *tester* is the implementation of a *test suite*, which is a collection of *test cases* (Section 2.4.3). The purposes and behaviors of the test cases need to be specified and the actually execution results need to be captured and matched against the specification (Section 2.4.4 and 2.4.5).

These lead to the key requirements of a test specification language:

- Test Architecture
  - Tester(s), SUT and Test Context
  - Connection between Tester(s) and SUT (PCOs and IAPs)
  - Communication between Tester(s) and SUT

- Organization and Management of Tests

- Test Case Behavior
    - Sending Stimuli to the SUT
    - Receiving Responses from the SUT
    - Storing and Transferring Data
    - Flow Control
    - Test Step Repetition

- Observation and Reporting
    - Checking Responses
    - Measuring the Timing of Responses
    - Assigning and Handling of Verdicts

In next chapter, we will analyze the suitability of SDL for testing specifications based on these key requirements.

CHAPTER 3. SDL: OVERVIEW AND SUITABILITY FOR TEST SPECIFICATIONS

As we have already identified in Chapter 1, SDL-based test language has many advantages over other test languages for testing SDL specifications. Based on the key requirements for test languages identified in Chapter 2, in this chapter, we analyze the suitability of SDL for test SDL specifications and the limitations it has starting from an overview of SDL.

3.1. Overview: Basic Concepts and History

Specification and Description language (SDL)[ITUZ100] is an object-oriented, formal, and graphical language defined by the International Telecommunications Union–Telecommunications Standardization Sector (ITU–T) (formerly CCITT) as recommendation Z.100. The language is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals.

Systems described in SDL consist of one or more processes running simultaneously which communicate with each other via signals. Each process is described by an extended finite state machine.

3.1.1. History

The development of SDL started in 1972. A study group within CCITT representing several countries and large telecom companies like Bellcore, Ericsson, and Motorola began research on a standard specification language for the telecommunications industry. The first version of the language was issued in 1976, followed by new versions in 1980, 1984, 1988, 1992, and 1996. The latest version is SDL-2000, which gives better support for object modeling and for code generation.

3.1.2. Basic Concepts

The basic theoretical model of an SDL system consists of a hierarchical set of extended finite state machines (FSMs) that run in parallel. These machines are independent of each other and communicate with discrete signals.

An SDL system consists of the following components:

- structure—system, block, process, and procedure hierarchy
- communication—signals with optional signal parameters and channels (or signal routes)
- behavior—processes
- data—abstract data types (ADT) and ASN.1
- inheritance—describing relations and specialization

The following subsections introduce the basic concepts.

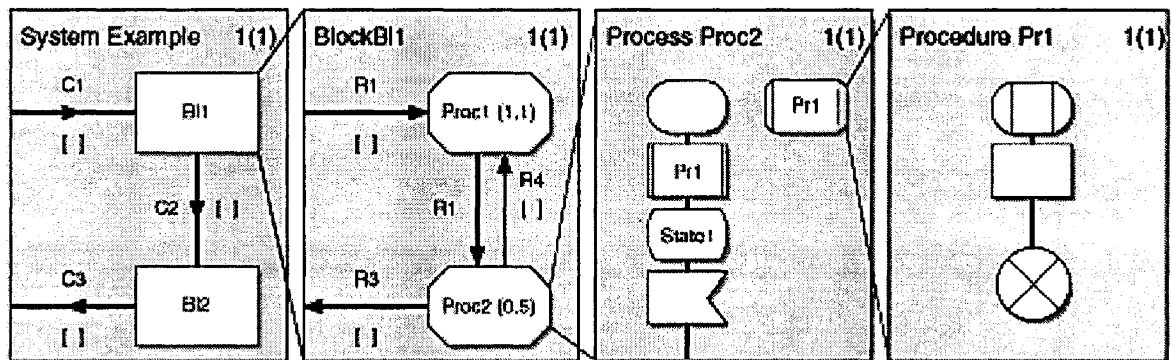
Structure

SDL comprises four main hierarchical levels:

- system
- blocks
- processes
- procedures

Figure 10 The Structural View of an SDL System

[IEC]



Each SDL process type is defined as a nested hierarchical state machine. Each sub-state machine is implemented in a procedure. Procedures can be recursive.

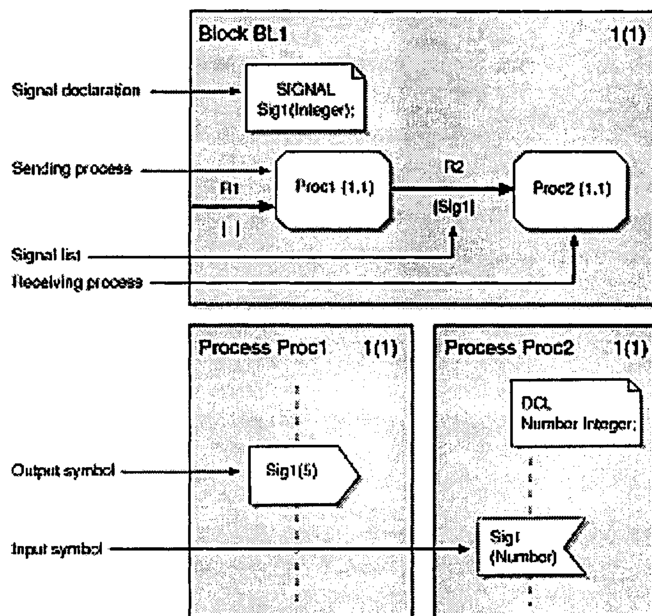
A set of processes can be logically grouped into a block (that is, subsystem). Blocks can be nested inside each other to recursively break down a system into smaller and maintainable encapsulated subsystems. These break-down mechanisms are important for large team development effort, and SDL simplifies this by also providing clear interfaces between subsystems.

Communication

SDL defines clear interfaces between blocks and processes by means of a combined channel and signal route architecture. This communication architecture with formally clear signal interfaces simplifies large team development and ensures consistency between different parts of a system.

Figure 11 Signals Travel through Channels and Signal Routes

[IEC]



Behavior

The dynamic behavior in an SDL system is described in the processes. The system/block hierarchy is only a static description of the system structure. Processes in SDL can be created at system start or created and terminated at run time. More than one instance of a process can exist if specified. Each instance has a unique process identifier (PId). This

makes it possible to send signals to individual instances of a process. The concept of processes and process instances that work autonomously and concurrently makes SDL a true real-time language.

Data

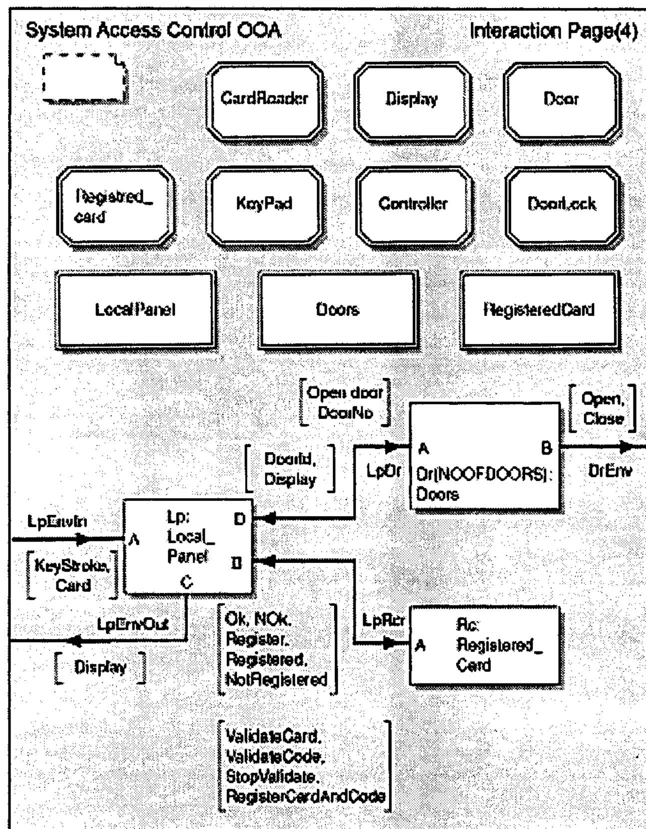
SDL accepts two ways of describing data, abstract data type (ADT) and ASN.1. The integration of ASN.1 enables sharing of data between languages, as well as the reuse of existing data structures.

The ADT concept used within SDL is very well suited to a specification language. An abstract data type is a data type with no specified data structure. Instead, it specifies a set of values, a set of operations allowed, and a set of equations that the operations must fulfill.

Inheritance

The OO (Object-Oriented) concepts of SDL give the user powerful tools for structuring and reuse. The concept is based on type declarations. Type declarations can be placed anywhere, either inside the system close to their context, or at system level. Figure 12[IEC] shows an access control system with block and process types at system level. Type declarations can also be placed in packages outside the system, for sharing with other systems.

Figure 12 System with Type Declarations



3.1.3. SDL-2000

SDL-2000 is the latest version of SDL. In SDL-2000, some features that were not strongly supported by tools were removed. Object modeling in SDL was strengthened and better support given for programming directly in SDL. In particular the data model was revised to give such features as global data and referenced data objects. The structuring features (blocks and processes) were harmonized into an agent concept. Support for ASN.1 was strengthened so that the use of ASN.1 modules with SDL no longer requires any change main body of the language.

To enable these changes, many elements of old SDL models may need changes for SDL-2000 tools. As of May 2004, there is no commercial tool fully supporting SDL-2000.

3.1.4. Support Tools

The precision and formality of SDL provide the possibility to build tools for the simulation of SDL specifications and for the validation of formal characteristics. It also provides the possibility for tool-supported code compilation into lower level languages, such as C/C++. This means that the SDL specification can be translated into an executable application without manual coding, leading to shortened development time and increased quality.

However, as we discussed in previous section, due to its complexity, the latest version SDL-2000 has no tool support after it has been introduced for three years. Without tool support, the language cannot be learned, accepted and used.

At the same time, the existing tools supporting older versions are also expensive, which prevents students, individual users, or small organizations from learning it or adopting it.

A simplified version of SDL with validation features will encourage tool manufactures to develop cheaper tools, and this will eventually turn the potential users into its real users.

3.2. Suitability and Limitations of SDL for Test Specifications

Experiments showed that SDL has the potential to be a test specification language[Bhas+01][Lopez+03][Tret+92]. This section analyzes the suitability of SDL as such a language by mapping the key requirements to existing SDL features. This mapping can identify the useful testing features and limitations of SDL and they will be used as the basis for the definition of SIMPL-T.

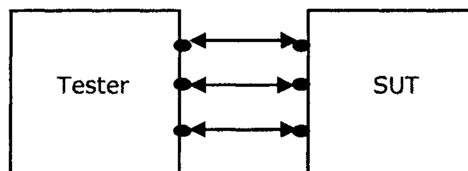
As we have discussed earlier, SDL has evolved through many different versions. The latest version is SDL-2000, which made revolutionary changes to the older versions. In this section, we discuss basic SDL concepts, which are ideally version independent. When different versions contain the same concept but different semantics, the latest version is used.

The work of this thesis is part of and based on draft documents of the *SDL task force*[SDLTF]. In these draft documents, some concepts have slightly different semantics from what they have in original (traditional) versions. When these new semantics are more suitable for our discussion, we refer them to "task force definitions" or "task force concepts", and a brief justification is given. Otherwise, traditional terms and semantics are used and assumed.

3.2.1. Test Architecture – Tester(s), SUT and Test Context

We assume the architecture of the SDL-based testing system can be defined as shown in Figure 13. The entities of the test architecture can be defined in an SDL specification using the same constructs as for the original system specification.

Figure 13 A SDL-based Test System Architecture



SUT

Since we only consider black-box testing, the internal behaviour of the SUT is not tested. Therefore the SUT is assumed not to be an SDL process. It can be either an SDL system, or an SDL block.

Tester

The *Tester* represents the environment of the SUT. It can be specified using the SDL *block* construct. The behaviours of the *Tester* can be defined by the processes of the block.

Test Context

Since the SUT can communication with the *Tester* directly, the *Test Context* and the SUT coincide.

3.2.2. Test Architecture – Connection between Tester and SUT (PCOs and IAPs)

The *Tester* and SUT can connect to each other through SDL *gates* and *channels.*

Gates are defined in 8.1.5 of [ITUZ100] as "in agent types (block types, process types) or state types and represent connection points for channels, connecting instances of these types (as defined in 8.1.3) with other instances or with the enclosing frame symbol". "It is possible also to define gates in agents and composite states and this represents a notation for specifying that the considered entity has a named connection point."

SDL *channels* are FIFO-queues. They are considered as reliable and order-preserving communication links[ITUZ100]. So they can be used to model PCOs in accordance with ISO 9646 and ITU-T Z.500.

Since Test Context and SUT coincide in the architecture, PCOs and IAPs coincide.

3.2.3. Test Architecture – Communication between Tester and SUT

Since the *Tester* and SUT are both SDL blocks, and the test system they formed is very much like an original SDL specification, naturally they can communicate by signal exchange.

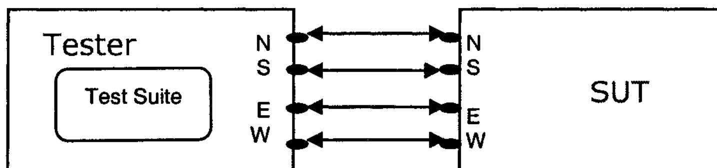
3.2.4. Justification of the Test System Architecture

The interface between the *Tester* and the SUT is actually the environment of the SUT. The gates and signals are the same from both sides of this interface, so the interface is symmetrical. Therefore the Tester has the same environment as the SUT.

This model implies that the *Tester* has the same number of gates as the SUT and that the signals used by the *Tester* are the same as by the SUT (only the signals in each side of the interface travel in opposite direction).

To implement this model, the *Tester* can be defined as an SDL block with the required number of gates. The gates of the SUT can be mapped and connected to the corresponding gates of the *Tester.* The signals of each gates in the *Tester* can be defined mapping the signals in the corresponding gates of the SUT, but in opposite direction (Figure 14).

Figure 14 Connecting the Tester to the SUT



Since the dynamic behavior in an SDL system is described in the processes, rather than in blocks, the communications (signal exchanges) may only occur between processes. These signals may travel within one block through signal routes, or they may travel between

blocks through channels. Only the latter appear as these blocks communicate with each other. An SDL system communicates with its environment in the same way.

SDL blocks communicate with each other by exchanging signals. These signals are atomic events. They trigger state transitions (to other states or to themselves).

Since a *Tester* can be defined as an SDL block, it can connect with the SUT through gates and channels, and it communicates with the SUT through signal exchange. Therefore, the assumption concerning the SDL-based test system architecture (see Figure 13) holds.

3.2.5. Organization and Management of Tests

Studies have shown that SDL can be used to specify tests[Bhas+01][Tret+92]. A test case may be modeled as an SDL procedure and a test suite as a process. Then the order of test case execution may be defined in the process definition.

Since SDL is intended for specification and design rather than tests, it does not have the constructs of test case, test suite, or test purpose. It also does not have an explicit mechanism to allow test cases to be organized and managed.

3.2.6. Test Case Behavior - Send Stimuli to the SUT and Receive Response from the SUT

Although SDL does not have a test case construct, test case behavior can be specified using existing SDL features.

Within the framework defined, sending a stimulus to the SUT is simply a matter of using the SDL *OUTPUT* construct to send a signal to the SUT. In SDL, *OUTPUT* is used to send signals, and possibly contained values as parameters, to other processes.

In a specific test, a signal can be used from the available signal lists defining the environment of the tests. The parameters for the signal can be chosen according to the requirements of this test.

As in sending a stimulus to an SUT, specifying an expected response from the SUT can simply use the SDL *INPUT* construct. When an expected response is received, another stimulus may be sent to the SUT, or a verdict may be assigned and the test terminated.

Figure 15 is an example from[Prob02]. It shows how to use TTCN to test an SDL specification. Figure 16 rewrites the Test Case 1 of Figure 15 using SDL INPUT and OUTPUT constructs.

Figure 15 An Example of Using TTCN to Test an SDL Specification

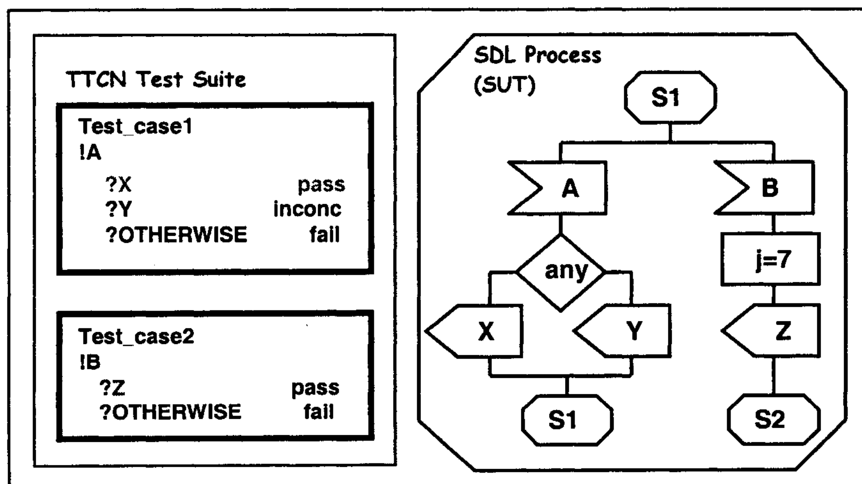
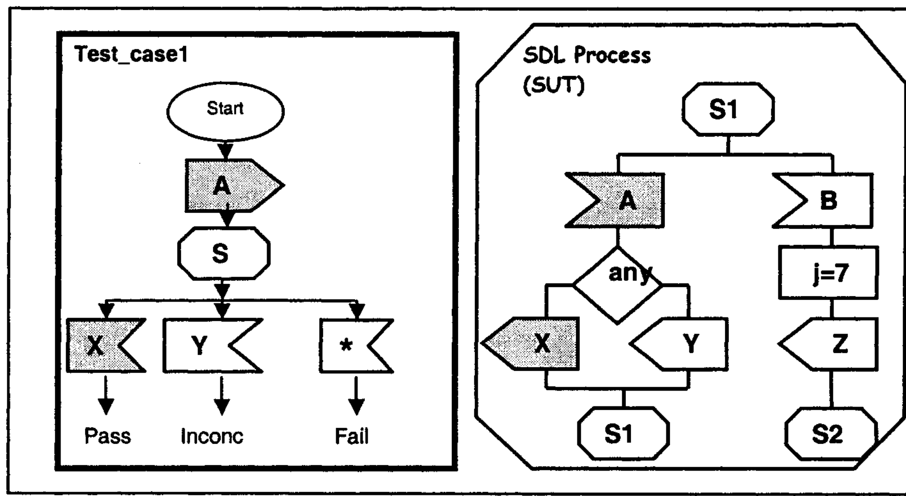


Figure 16 Using SDL to Send Stimuli to and Receive Responses from the SUT



3.2.7. Test Case Behavior – Storing and Transferring Data

When temporary data needs to be stored, SDL *VARIABLE* can be used.

A *VARIABLE* in SDL is defined as an object that takes on a specific value of the sort of the variable at a given time. With time, the value may change.

SDL accepts two ways of describing data, abstract data type (ADT) and ASN.1[ITUX680]. The integration of ASN.1 enables sharing of data between languages, as well as the reuse of existing data structures.

Since data can be transferred between processes as parameters carried by signals, during a test execution, data can be passed between responses and stimuli, and can be reused by storing it in *VARIABLEs*.

Therefore, SDL data type and *VARIABLE* concepts are well suited for test specifications.

3.2.8. Test Case Behavior – Flow Control

Using SDL *DECISION* construct, alternative actions can be taken in a test case. Depending on variables used for counting iterations or for storing data unloaded from parameters, the sequence of stimuli sent can be influenced and different verdicts can be assigned.

We use an example to show the usage of SDL *VARIABLE* and *DECISION* in testing. We modify the SDL diagram shown in Figure 15 and the corresponding *Test Case 2* for this purpose.

Suppose in the SDL diagram, the output signal Z carries data j (j=7 in this case). In *Test Case 2,* if signal Z is received and the data it carries is integer "7", it passes; otherwise it fails (Figure 17). Figure 18 rewrites the *Test Case 2* of Figure 17 using SDL *Decision* construct.

Figure 17 Example of Output Signal Carrying Data

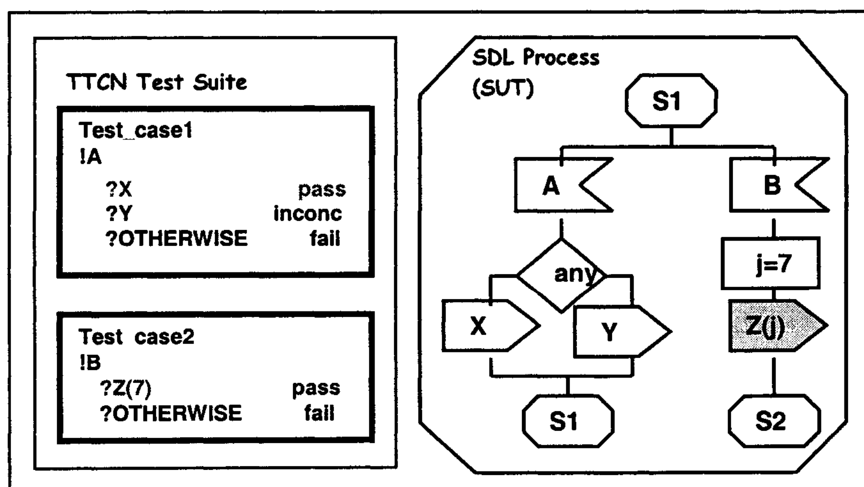
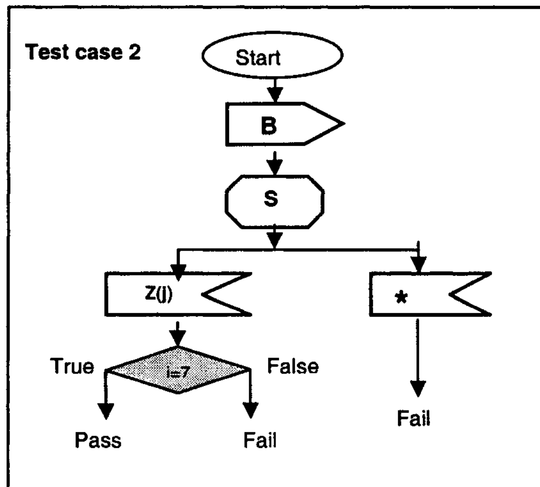


Figure 18 SDL Decision Construct Used in Testing



3.2.9. Test Case Behavior – Test Step Repetition

When a test or several tests have sequences of actions which occur more than once, the SDL *procedure* can be used.

The *procedures* in SDL are similar to the ones known in programming languages. *Procedures* as part of processes define patterns of behavior that the process may execute at several places or several times during its life-time.

A *procedure* can be declared to be "local" or "remote". If it is declared to be "remote", it can be used in more than one test case.

3.2.10. Observation – Checking Responses

Using SDL can send stimuli to an SUT, and receive responses from an SUT. However, receiving an actual response is not the same as specifying the expected response. The ability to specify the expected events to be received, i.e. the expected response of an SUT, and checking the actual responses against them are an important part of a test language.

In terms of SDL systems, an expected event includes

- Signal name
- The gate on which this signal should arrive
- The expected values of the parameters.

If a received event matches an expected event, the associated sequence of actions can be executed, such as sending further signals to the SUT, storing data, starting timers, or assigning a verdict.

SDL *INPUT* construct allows an expected signal name to be specified, but it does not allow the expected gate and expected values of the parameters to be specified.

- ***Expected Gate***

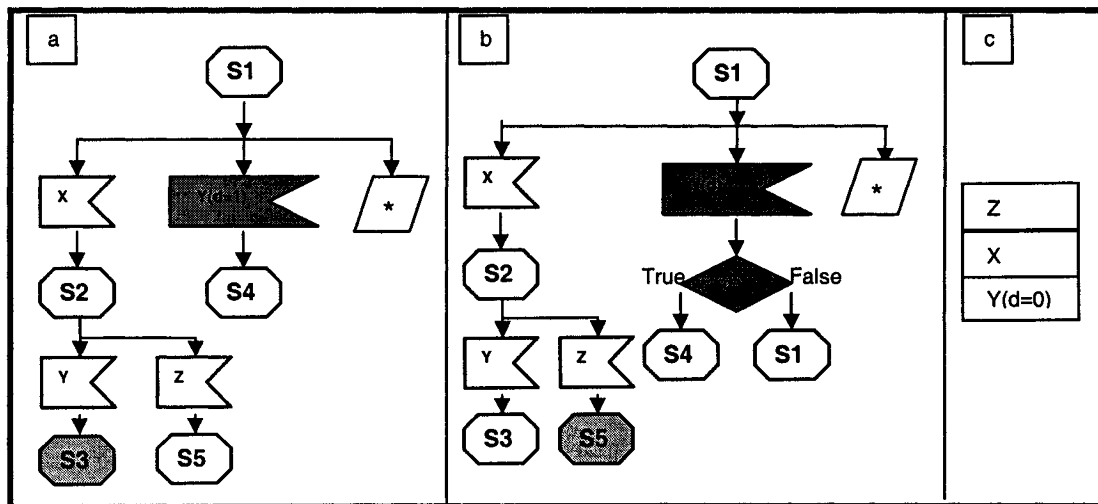
Since the same signal name can arrive from different gate, SDL has no way to specify it using *INPUT* construct, or any other mechanism.

- ***Expected Values of the parameters***

SDL does not allow the expected value of parameters to be specified inside an *INPUT* construct. It does allow the values to be unloaded into *VARIABLES* and use the *DECISION* construct to influence the flow. But this can only happen once the signal name has matched and the decision to execute the associated actions has been taken (the signal is removed from the input queue). This is not the same as specifying it inside an *INPUT* construct. In some cases, it leads to different actions. Figure 19 uses an example to show the difference.

- Figure 19 "c" shows the input queue, where the bottom is the front of the queue.
- Figure 19 "a" shows the case in which the expected value of a parameter can be specified inside an *INPUT* construct. The end state is "S3".
- Figure 19 "b" shows the case in which the parameter is unloaded into a *VARIABLE* and the *DECISION* construct is used. The end state is "S5".

Figure 19 How Checking of Parameters Makes a Difference



In summary SDL provides all the functionality needed to check responses and execute associated actions, except for specifying the expected gate and parameters.

3.2.11. Observation – Measuring the Timing of Responses

Time is an important aspect in all real-time systems. Using SDL *TIMER* it is possible to test timer events in an SUT, i.e. to monitor the response time of the SUT.

SDL defines time and timers in an abstract manner. SDL time is abstract in the sense that it can be efficiently mapped to the time of the target system, be it an operating system timer or hardware timer. This makes it possible to simulate time in SDL models before the target system is available. This also makes it possible to monitor the response time of any SUT.

An SDL process can set *TIMERs* that expire within certain time periods resulting in *time-outs* when exceptions occur. This makes it possible to measure and control response times from other processes and systems.

When an SDL *TIMER* expires, the process that started the *TIMER* receives a notification signal in the same way as it receives any other signal. Actually an expired *TIMER* is treated in exactly the same way as a signal.

To monitor the response time of an SUT, a *TIMER* can be set to a certain value which is the allowed response time from the SUT. This *TIMER* is started (*SET* operation in SDL) when a stimulus is sent to the SUT. If the response is received before the *TIMEOUT* signal, this step passes, otherwise, it fails.

We use the same example used in Figure 15 to illustrate how *TIMER* can be used to monitor the response time of the SUT.

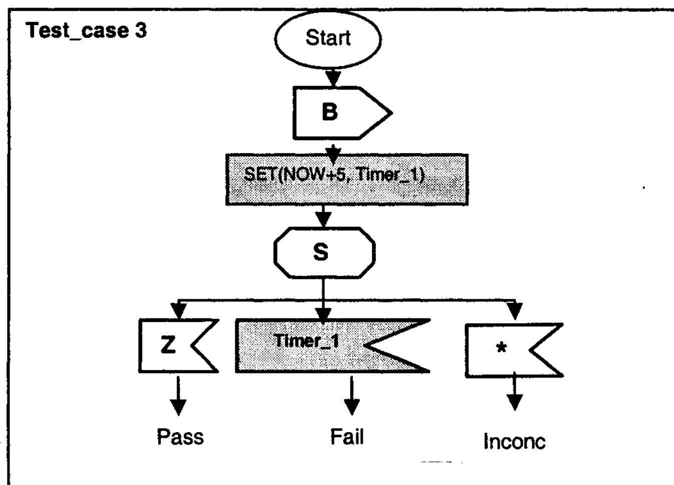
Since this example only shows a small part of the system -- one of the SDL processes, we can add hypothetic user requirements for the purpose of showing how timers can be used.

- *Requirement R1: After sending signal B to the system, signal Z should be received within 5 time units.*

(E.g. In an intersection, after a pedestrian button is pressed, the lights in the corresponding direction are turned green in 5 seconds.)

Test_Case 3 is a test case written in SDL to test *Requirement R1* (**Figure 20**).

Figure 20 Using a Timer to Monitor the Response Time of the SUT



To monitor whether the response is within the range of a minimum and maximum time, two *TIMER*s can be set. If the response is received after the first *TIMER TIMEOUT* and before the second *TIMER TIMEOUT*, the test case passes, otherwise, it fails (See Figure 21 in next section).

3.2.12. Assigning and Handling of Verdicts

Verdict is an indication whether the SUT responded as expected. It can have the value of *pass, fail, inconclusive,* and *error*, as used in TTCN. At the end of a test case execution, a *verdict* should be assigned.

In Tretmans' approach[Tret+92], the *verdict* assignments are modeled by *VARIABLEs* associated to each test case. The *verdict* assignments that result from test case executions were passed to the environment via signals.

SDL does not have a *VERDICT* construct. It does not provide a mechanism for the *verdicts* to be assigned, recorded, and reused. Especially when a preliminary result needs to be formed and used to continue a test, SDL obviously has its limitations.

3.2.13. An Example Test Case Written in SDL

As we have discussed in previous sections, SDL features are not sufficient to write complete test cases, however, many SDL constructs can be directly used in test cases. In this section, we give an example to illustrate how SDL *INPUT, OUTPUT*, data, *VARIABLE, TIMER* and *DECISION* can be used in a (incomplete) test case.

We use the same example used in Figure 15, and add an additional requirement.

- *Requirement R2: After sending signal B to the system, signal Z should be received within the range of 5 time units to 9 time units.*

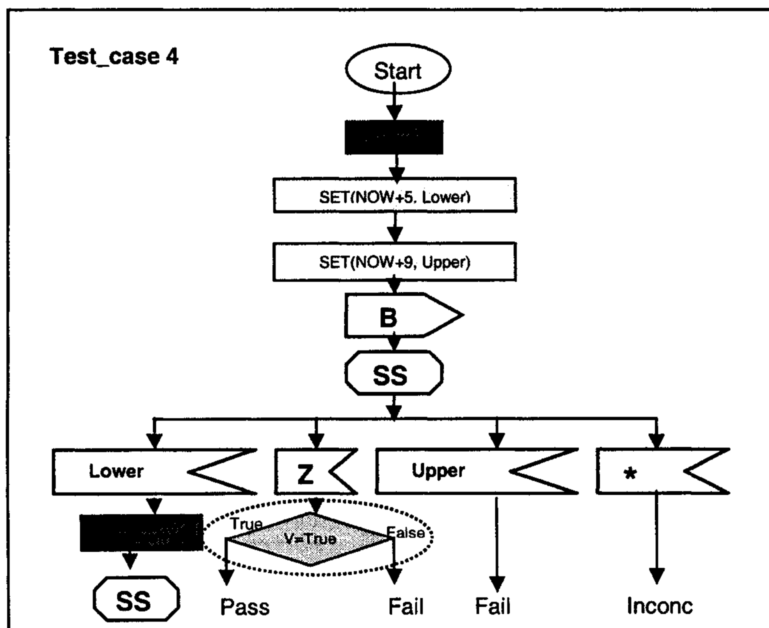
Test_Case 4 is a test case written in SDL to test *Requirement R2* (Figure 21).

In this test case, two *TIMER*s are used, *Lower* and *Upper*. *Lower* is the lower bound of the responding time, which is set to 5 time units. *Upper* is the upper bound of the responding time, which is set to 9 time units.

A *VARIABLE V* of sort (type) *Boolean* is used to record whether the lower bound of the responding time has reached. Its value is set to *False* at the beginning of the test. When *Lower* is received (timeout), *V*'s value is set to *True*.

When signal *Z* is received, the value of *V* is check using a *DECISION* construct. If *V=True*, it means the lower bound of the responding time has reached; if *V=False*, it means otherwise.

Figure 21 How Existing SDL Features Used in a Test Case



3.2.14. Summary

The discussions in this chapter show that, although SDL has its limitations, a significant proportion of the key requirements of test specification languages can be mapped onto existing SDL features. This is not surprising and simply reflects the overlap in requirements specification and test specification. This mapping is summarized in Table 1.

Table 1 Mapping Key Requirements onto SDL Features

Key Requirements	**SDL Features**
Test Architecture – Tester, SUT & Test Context	SDL Blocks
Test Architecture - Connecting between Tester and SUT (PCOs & IAPs)	Gate & Channel
Test Architecture - Communication between Tester and SUT	Signal Exchange
Organization and Management of Tests	**Not Supported**
Test Case Behaviour - Sending Stimuli to SUT	Output
Test Case Behaviour – Receiving Responses from SUT	Input
Test Case Behaviour - Storing and Transferring data	Variable & Data Type
Test Case Behaviour – Flow Control	Decision
Test Case Behaviour - Test Step Repetition	Procedure
Observation - Checking Responses	**Partially Supported**
Observation - Measuring the Timing of Responses	Timer
Assigning and Handling of Verdicts	**Not Supported**

CHAPTER 4. SIMPL-T: SDL INTENDED FOR MANAGEMENT AND PLANNING OF TESTS

As we already discussed, requirements specifications must be tested, and TTCN, being more powerful than needed, is not a perfect language for testing SDL specifications. SDL, although not providing all the testing features, is a good basis for an alternative. In this chapter, we define a simple test specification language – SIMPL-T (SDL Intended for Management and Planning of Tests) for testing SDL specifications based on SDL with minimal extensions of testing features.

4.1. Overview of Approach and Scope of SIMPL-T

Instead of discussing all the aspects as a complete language, we are going to reuse as many SDL concepts and features applicable to testing as possible. Therefore, we base our work on previous discussions. We first summarize the key requirements as a test specification language, and summarize which necessary test concepts are already in SDL and which need to be provided by SIMPL-T (**Table** 2).

As we have already discussed in Chapter 3, SDL has evolved many different versions. The SDL concepts SIMPL-T used are basic concepts, which are SDL-version

independent. When different versions have different semantics, the semantics discussed in Chapter 3 are used.

For the SDL concepts and features which are useful for testing but not shown in **Table** 2, such as concurrency and save construct, they are implicitly inherited by SIMPL-T.

Table 2 Key Testing Requirements and Extensions in SIMPL-T

Key Requirements	**SDL Features**
Test Architecture – Tester, SUT & Test Context	SDL Blocks
Test Architecture - Connecting between Tester and SUT (PCOs & IAPs)	Gate & Channel
Test Architecture - Communication between Tester and SUT	Signal Exchange
Organization and Management of Tests	**SIMPL-T Extension**
Test Case Behaviour - Sending Stimuli to SUT	Output
Test Case Behaviour – Receiving Responses from SUT	Input
Test Case Behaviour - Storing and Transferring data	Variable & Data Type
Test Case Behaviour – Flow Control	Decision
Test Case Behaviour - Test Step Repetition	Procedure
Observation - Checking Responses	**SIMPL-T Extension**
Observation - Measuring the Timing of Responses	Timer
Assigning and Handling of Verdicts	**SIMPL-T Extension**

Table 2 shows that the testing-specific concepts which are not fully supported in SDL are:

- Organization and Management of Tests
- Checking Responses
- Assigning and Handling of Verdicts

These concepts need to be defined as necessary extensions to complete SIMPL-T. To make these definitions reasonable, we studied how they are defined and implemented in TTCN (see Appendix B).

TTCN is a feature-rich language, which has a much wider application domain than SIMPL-T, and TTCN-3 is not only used for conformance testing. Therefore, TTCN features had to be general and independent of the language used for implementing the SUT. As a result it is relatively complex.

When we reused TTCN concepts to define SIMPL-T, we modified and simplified them when necessary to assure the simplicity of SIMPL-T.

In the definition of each of these extensions, we first discuss the corresponding TTCN concepts, and then define the SIMPL-T semantics base on them, followed by the syntax of SIMPL-T defined in BNF. When necessary, examples are also given to illustrate the discussed concepts or syntax.

In the syntax definitions, applicable SDL definitions are not redefined. They are highlighted and referred to [ITUZ100].

4.2. Organization and Management of Tests

4.2.1. TTCN Test Suite Structure

As already identified, SDL does not support the concept of a test, or the organization of tests into a larger structure. However TTCN explicitly supports these concepts (see Appendix B).

While SDL deals with systems, TTCN deals with *test suites*. A test suite is a top level entity containing all the information related to a set of tests, arranged as:

- Overview
- Declarations
- Constraints
- Behaviour

This arrangement allows information to be quickly located, and it separates the declaration of information from the use of it.

In TTCN, the individual tests of a test suite are defined in the behaviour section and called *test cases*. The test cases that belong together can be defined in *test groups*, which can be nested. Each test case has a defined *test purpose*, which is a textual description of the objectives of the test.

The dynamic part contains all the test cases, test steps, and defaults. This part defines the sequence of events to be sent to or expected from the SUT, at what point a verdict can be assigned and when the test should be terminated.

4.2.2. SIMPL-T Test Suite

- Test Suite

Test Suite is defined as a SIMPL-T construct (see Appendix C).

Since SDL does not have the concept of ASP, PDU, CM, or complicated structure like in TTCN, the *Constraints* part is not needed (Specifying constraints values and matching mechanisms are discussed in Section 4.3.2.). Moreover, since SDL test suite is simpler than TTCN test suite, the *Overview* part is not necessary either.

Therefore, a SIMPL-T *Test Suite* may contain only two parts:

- *Declarations*
- *Behaviour.*

The *Declarations* part is concerned with the declaration of all the *Test Suite* components, including:

- Gates and Signal List Declarations
- Variable Declarations
- Constant Declarations
- Timer Declarations
- Procedure Declarations
- Test Group Declarations
- Test Case Declarations

These declarations can be implemented as they are normally done in SDL.

The *Behaviour* part defines *Test Groups* and *Test Cases*.

- Test Group

Test Group is defined as a SIMPL-T construct (see Appendix C).

A *Test Group* is simply a container which contains a block of *Test Cases*. It does not have any behavior by itself.

- Test Case

Test Case is defined as a SIMPL-T construct (see Appendix C).

The behavior of a *Test Case* can be considered as a special SDL process. This process always stops itself (use *STOP* construct) after a *final verdict* is formed, while normal SDL processes do not stop until the whole system is forced to terminate. Therefore, no additional definition needs to be done regarding *Test Case behavior.*

- Test Purpose

Test Purpose is defined as a SIMPL-T construct (see Appendix C).

Since a *Test Purpose* is only a textual description of the objectives of the test, its content doesn't influence the behavior of the test at all.

- Explicit Test Step

Since an *Explicit Test Step* acts exactly like an SDL *procedure*, defining a new SIMPL-T construct for *Test Step* is not necessary. When repeated explicit test steps needed, an SDL *procedure* can be defined and called by the *Test Cases*.

- Summary

A SIMPL-T *Test Suite* consists of nested *Test Groups* containing *Test Cases*, each with a defined *Test Purpose* and *Test Case behavior.*

4.2.3. Syntax of Test Suite in SIMPL-T

```
Testsuite_Definition ::=     "TESTSUITE" TestsuiteName ";"
                             [ Gate_Definition ]
                             [ Testsuite_Component ]
                             "ENDTESTSUITE;"

Gate_Definition ::=   "GATE" GateName ";"
                      [ In_Signal_List ] ";"
                      [ Out_Signal_List ]";"

In_Signal_List ::=    Signal_Identifier
                      [ "," In_Signal_List ]

Out_Signal_List ::=   Signal_Identifier
                      [ "," Out_Signal_List ]

Testsuite_Component ::=     ([Signal_Definition]
                            [Signal_List_Definition]
                            [Asn1type_Definition]
                            [Timer_Definition]
                            [Variable_Definition]
                            [Synonym_Definition]
                            [Procedure_Definition]
                            [Test_Group_Definition]
                            [Test_Case_Definition] )
                            [Testsuite_Component ]
```

Test_Group_Definition ::= "TESTGROUP" TestGroupName ";"
Test_Case_Definition_List
"ENDTESTGROUP;"

Test_Case_Definition_List ::= (Test_Case_Definition | Test_Group_Definition)
[Test_Case_Definition_List]

Test_Case_Definition ::= "TESTCASE" TestCaseName ";"
"TESTPURPOSE" Test_Purpose_Description ";"
Behavior_Definition
"ENDTESTCASE;"

Note:

- **Testsuite_Definition** is defined parallel to **Process** definition as part of the **Block** definition in [ITUZ100], which is not repeated here.
- **Test_Purpose_Description** is simply a text description of the test purpose.
- **Behavior_Definition** is basically EFSM, which has been defined in the *Process* section in SDL[ITUZ100]. The test-specific behaviour of a state machine, such as verdict, will be discussed in later sections in this chapter.

4.2.4. Examples

This section uses an example to show the hierarchical structure of a test suite defined in SIMPL-T.

This example shows a test suite Door_Control. In this test suite, there are two test groups: Normal_Scenario and Abnormal_Scenario. Open_Door is a test case located in Normal_Scenario test group.

Figure 22 An Example Test Suite Defined in SIMPL-T

```
TESTSUITE Door_Control;
  ......
  ...... (gate, signal, variable, timer... declarations. They are the same as in SDL)
  ......
  TESTGROUP Normal_Scenario;
      TESTCASE Open_Door ;
            TESTPURPOSE "Open the door when it is closed" ;
            ...... (Behavior_Definition)
      ENDTESTCASE;
      ......
  ENDTESTGROUP
  TESTGROUP Abnormal_Scenario
      ......
  ENDTESTGROUP
ENDTESTSUITE;
```

4.3. Checking Responses

As previously identified, SDL allows the expected signal name to be specified, but it does not allow the expected gate or the expected parameters to be specified inside an *INPUT* construct.

TTCN does support an extensive mechanism to specify both the PCO (TTCN equivalent of gate) and the ASP/PDU (TTCN equivalent of both signal name and parameters) (see Appendix B).

4.3.1. Specifying Expected Gate

Since the same signal name can arrive from different gate, where a signal comes from has to be identified.

In TTCN, when an expected event is specified in a scenario, the PCO is specified at the same time, eg. PCO1: ? PDU (Param1). This is symmetrical to the way an event to be sent is specified, which corresponds to the *OUTPUT* construct in SDL.

In an SDL specification, a *PId* can be used to identify from which process a signal is sent. Since validation uses black-box techniques, the Tester is not supposed to know what processes exist inside an SUT, so it does not know the *PId*s of the processes. Therefore, it has no way to specify where a signal comes from.

However the SDL *OUTPUT* construct does allow the gate (or channel) to be specified using the *OUTPUT VIA* construct (Figure 23).

Figure 23 SDL "OUTPUT VIA" Construct

```
STATE S1;
   INPUT A;
   OUTPUT B VIA Gate1;
NEXTSTATE;
```

We define a new construct ***INPUT VIA*** in SIMPL-T to specify the expected gate/channel (Figure 24) (see Appendix C).

Figure 24 "INPUT VIA" Construct (NEW)

```
STATE S1;
   INPUT A VIA Gate1;
NEXTSTATE S2;
```

4.3.2. Specifying Expected Values of Parameters and Matching Mechanism

As we discussed in Chapter 3, SDL does not allow the expected value of parameters to be specified inside an *INPUT* construct, and saving the parameters into variables cannot always solve the problem, while TTCN *RECEIVE* events and matching mechanism can handle this very well. (see Appendix B).

4.3.2.1. TTCN *RECEIVE Events and Matching Mechanisms*

In TTCN, if a constraint is to be used to construct the values of ASP parameters or PDU fields that a received ASP or PDU shall match, it shall contain only specific values evaluated, or special matching mechanisms where it is not desirable, or possible, to specify specific values. The matching mechanisms specify other ways of matching than "equal to a specific value"[ETSTR9905a].

An incoming ASP and/or PDU match a constraint used in a RECEIVE event if, and only if, all the following conditions are met:[ETSTR9905a]

a) all the ASP parameters and/or PDU fields are of the type specified in the ASP and/or PDU definitions;
b) the value, alphabet and length satisfies any restriction associated with the type;
c) the ASP parameter and/or PDU field values correctly match those of the constraint;
d) for PDUs, the correct decoding of the PDU has taken place, taking into account applicable encoding rule defaults and overrides; if encoding rules other than those specified for the constraint have been used to encode the received PDU, then that received PDU will not match.

In the case of substructured ASPs and/or PDUs, either using Structured Types or ASN.1, the above rules shall apply to the fields of the substructure(s) recursively.

Matching mechanisms[ETSTR9905a]

An overview of the supported matching mechanisms is shown in Table 3[ETSTR9905a], including the special symbols and the scope of their application. The left hand column of this table lists all the ASN.1 types and TTCN equivalent types to which these matching mechanisms apply. The matching mechanisms in the horizontal headings are arranged in four groups:

a) specific values;
b) special symbols that can be used instead of values;
c) special symbols that can be used inside values;
d) special symbols which describe attributes of values.

Some of the symbols may be used in combination (for more detailed see Appendix B).

The shaded area in Table 3 indicates the mechanisms that apply to both predefined TTCN and ASN.1 types.

Table 3 TTCN Matching Mechanisms

	VALUE	INSTEAD OF VALUE								INSIDE VALUE			ATTRIBUTES	
TYPE	Specific Value	Complement	Omit (-)	AnyValue (?)	AnyOrOmit (*)	ValueList	Range	SuperSet	SubSet	AnyOne (?)	AnyOrNone (*)	Permutation	Length	IfPresent
BOOLEAN	•	•	•	•	•	•								•
INTEGER	•	•	•	•	•	•	•							•
ENUMERATED	•	•	•	•	•	•								•
BITSTRING	•	•	•	•	•	•				•	•		•	•
OCTETSTRING	•	•	•	•	•	•				•	•		•	•
HEXSTRING	•	•	•	•	•	•				•	•		•	•
CHARSTRINGS	•	•	•	•	•	•				•	•		•	•
SEQUENCE	•	•	•	•	•	•								•
SEQUENCE OF	•	•	•	•	•	•				•	•	•	•	•
SET	•	•	•	•	•	•								•
SET OF	•	•	•	•	•	•		•	•	•	•		•	•
ANY	•	•	•	•	•	•								•
CHOICE	•	•	•	•	•	•								•
OBJECT ID	•	•	•	•	•	•								•

In a constraint specification, the matching mechanisms may replace values of single ASP parameters or PDU fields or even the entire contents of an ASP or PDU.

4.3.2.2. SIMPL-T INPUT and Matching Mechanism

- ***INPUT* construct**

We define the SIMPL-T *INPUT* construct as (an enhancement to the SDL *INPUT* construct):

***INPUT* signal_name (*parameter_list*) *VIA* gate/channel**

The *parameter_list* can be either specific values or specific matching mechanisms.

- **Matching Mechanism**

SIMPL-T matching mechanisms are defined based on simplified TTCN matching mechanisms (Appendix C).

Each parameter in the *parameter_list* has a data-type. For simple data-types the following matching mechanisms apply:

- specific value – Variable, Constant, or Expressions
- a range of values - denoted by "Value1 : Value2"
- list of values - denoted by " (Value1, Value2, Value3……)"
- wildcard value, matching all values – denoted by "?"

- omitted value, for missing parameters – denoted by "-"
- General wildcard value, matching all values and missing parameters – denoted by "*"
- Default value – used if no value is specified

As parameters are optional in SDL, if the omitted value is specified ("-"), it means that the parameter is not allowed to be present in the signal. The general wildcard ("*") matches any value for the parameter, whether present or not.

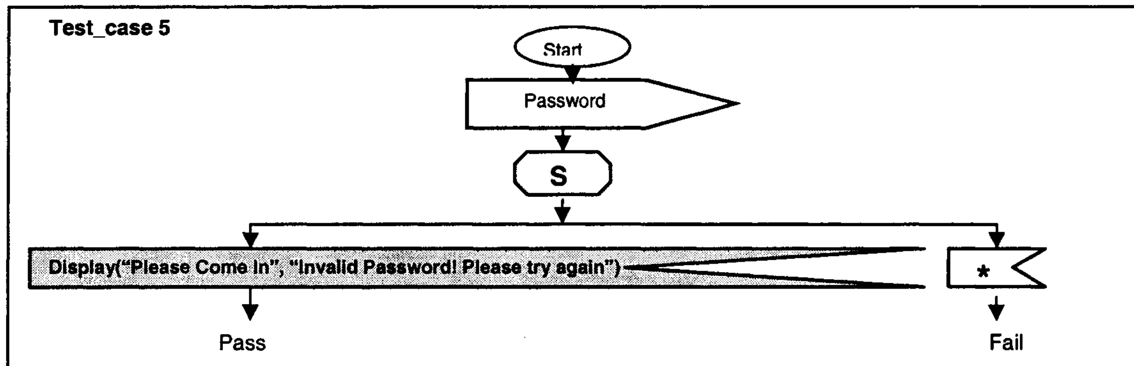
If the parameter has a structured type, such as CHOICE or SEQUENCE, the ASN.1 value notation can be used to specify a specific value, range or list of values.

The wildcards can be used within the value notation for specific fields and the "omitted value" can be used for optional fields in a SEQUENCE, to specify that an optional field must be omitted.

Figure 25 shows an example of how these matching mechanisms can be implemented in SIMPL-T. In this example, an electronic door will display a message on the control panel after a user enters a password. If the password is valid, the message is "Please Come In". Otherwise, the message is "Invalid Password! Please try again".

In the test case shown in **Figure 25**, after sending a signal *Password*, the test case expects a signal *Display* carrying a parameter – display message. This message can be either *"Please Come In"*, or *"Invalid Password! Please try again"*. They are specified as a list of values of the parameter carried by signal *Display*.

Figure 25 Specify the Values of a Parameter in an *INPUT* Construct



- **Unmatched signals**

Inherited from SDL, SIMPL-T disregards the unmatched signals by default. This means these unmatched signals are removed from the input queue and thrown away. If these unmatched signals are needed at later time, SIMPL-T can save them in the queue using *Save* construct, which is also inherited from SDL.

- **Overlapped signals in a test case**

There are two types of overlapped signals: (1) the same signal arriving from different gates/channels; (2) the parameters carried by the same signal have different values and the values have overlap. Figure 26 and Figure 27 show examples of these two types of overlaps.

Figure 26 Overlapped signals: the same signal arriving from different gates/channels

The signals shown in Figure 28 are not considered as overlap in SIMPL-T, since signals from different gates are considered different signals, and they can not arrive at the same time because only one FIFO queue.

Figure 27 Overlapped signals: the values of parameters have overlap

The inputs shown in Figure 27 are not allowed in SIMPL-T. Since SIMPL-T alternatives are not ordered (prioritized), this overlap can cause nondeterministic behaviours.

4.3.3. Syntax of SIMPL-T Input

Input_Definition ::= "INPUT" Input_Signal
"VIA" Origination_Name ";"

Origination_Name ::= Channel_Identifier
| Gate_Identifier

Input_Signal ::= "*" | (Signal_Identifier [Parameter_List])

Parameter_List ::= "(" Parameter ["," Parameter] ")"

Parameter ::= " - "
| " ? "
| " * " |
Value_List
| Value_Range
| Empty

Empty ::= ""

Value_List ::= "(" Value ["," Value_List] ")"

Value_Range ::= Value " : " Value

Value ::= **Expression**

Note: Expression is defined in [ITUZ100].

4.4. Assigning and Handling of Verdicts

SDL does not have a *Verdict* construct, or a mechanism for the verdicts handling, while TTCN has good mechanisms to handle verdicts. We will use exactly the same mechanisms in SIMPL-T (see Appendix B and Appendix C).

4.4.1. Verdict handling in TTCN

There are two mechanisms in TTCN that provide assignment of verdicts to a test case. These mechanisms are [ISO9646][Wiles] :

- preliminary results;
- explicit final verdicts.

4.4.1.1. The result variable

TTCN has a predefined test case variable, known as the *result variable*, called *R*. This variable may be used in expressions and the verdict column of a behavior description. It is used to store any intermediate result and has the following characteristics:[Wiles]

- A *preliminary result* does not terminate execution of a test case;
- it may appear in expressions as a read-only variable, i.e. it may not be used on the left hand side of an assignment;
- it may only take one of the values: *pass, fail, inconc* or type definition. These values are predefined identifiers, and are case sensitive;
- changes are made to its value by entries in the verdicts column;
- at the start of a test case *R* is bound to the value type definition.

4.4.1.2. Preliminary results

The value of *R* is changed by recording a *preliminary result* in the verdicts column. A preliminary result may be one of the following:

- (P) or (PASS), meaning that some aspect of the test purpose has been achieved;
- (I) or (INCONC), meaning that something has occurred which makes the test case inconclusive for some aspect of the test purpose;
- (F) or (FAIL), meaning that a protocol error has occurred or that some aspect of the test purpose has resulted in failure.

Preliminary results have an order of precedence, for example:

- if *R* has the value *fail* and a *preliminary result* (*PASS*) is encountered in the verdict column, then *R* cannot be changed to *pass* and it will remain bound to *fail*. On the other hand, if *R* has the value *pass* and a *preliminary result* (*FAIL)* is encountered in the verdict column, then *R* is bound to the value *fail.*

The table below [Wiles] (Table 4) shows how *R* may be changed according to the precedence rules:

Table 4 Calculation of the Result Variable *R*

Current	Preliminary Result		
Value of R	(PASS)	(INCONC)	(FAIL)
none	pass	inconc	fail
pass	pass	inconc	fail
inconc	inconc	inconc	fail
fail	fail	fail	fail

4.4.1.3. Final verdicts

Execution of a test case is terminated either by:

- reaching a leaf of the test case behavior tree; and/or
- an explicit *final verdict* on the behavior line (i.e. in the verdict column).

A *final verdict* may be one of the following:

- P or PASS, meaning that a *pass* verdict is to be recorded;
- I or INCONC, meaning that an *inconclusive* verdict is to be recorded;
- For FAIL, meaning that a *fail* verdict is to be recorded;
- the predefined variable *R*, meaning that the value of *R* is to be taken as the *final verdict,* unless the value of *R* is none in which case a test case error is recorded instead of a *final verdict.*

If no explicit *final verdict* is reached, then the *final verdict* is the value of *R*: If *R* is still bound to the value none then this is a test case error.

The *final verdict* must be consistent with the value of *R*. For example:

- if *R* has the value *fail* and an explicit *final verdict* PASS is encountered in the verdict column, then a *final verdict* of *fail* and not *pass* should be recorded. On the other hand, if *R* has the value *pass* and an explicit *final verdict* FAIL is encountered in the verdict column, then a *final verdict* of *fail* should be recorded.

The table below [Wiles] (Table 5) shows how the *final verdict* should be recorded according to the value of *R*:

Table 5 Calculation of the Final Verdict

Current Value of R	Final Verdict (PASS)	(INCONC)	(FAIL)	R
none	pass	inconc	fail	error
pass	pass	inconc	fail	pass
inconc	error	inconc	fail	inconc
fail	error	error	fail	fail

In summary, these TTCN verdict handling concepts are suitable for tests written in SIMPL-T, and in principle the TTCN semantics can be incorporated into SIMPL-T without any modification.

4.4.2. Syntax of SIMPL-T Verdict Assigning and Handling

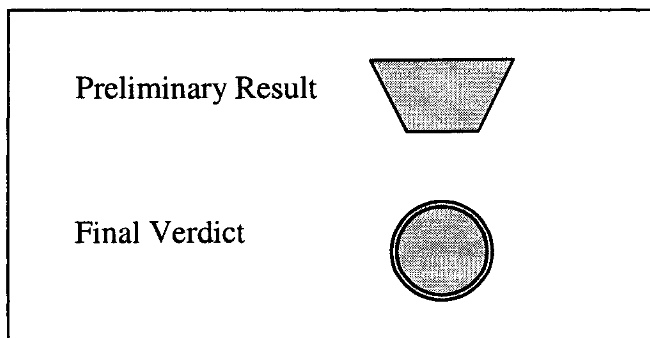
Textual (PR) Representation:

Verdict_Pre ::= "PRELIMINARY_VERDICT"
"(" Verdict ")"

Verdict_Fin ::= "FINAL_VERDICT"
"(" Verdict ")"

Verdict ::= "PASS"
| "INCONC"
| "FAIL"
| "ERROR"

Graphical (GR) Representation:



4.5. An Example of a Complete Test Case Written in SIMPL-T

In this section, we use an example to illustrate how SIMPL-T is used to write a complete test case. This example uses the existing SDL constructs *input, output, data, variable, decision, timer and procedure.* It also uses the new SIMPL-T features *Preliminary Result, Final Verdict, input via,* specification of the values of a parameter inside an *input* construct, and *matching mechanism.*

We use the same example used in Figure 21, and modify the requirement:

- *Requirement R3: The environment sends signal B to the system, and receives signal Z within the range of 5 time units to 9 time units. Signal Z should arrive at Gate1 and carry a parameter of sort Integer. The value of the parameter is between 1 and 100.*

Test_Case 6 is a test case written in SIMPL-T to test *Requirement R3* (Figure 28).

In this test case diagram, the expected signal Z is specified inside an *input* symbol, together with the expected gate on which it arrives *(Gate1)* and the expected parameter it carries. This expected parameter is not a specific value, but a range of integer value between 1 and 100, which is specified as "1:100".

The new symbols of *preliminary result* and *final verdict* are also used in this test case. At the beginning of the test case, it initiates the SUT first by calling a procedure *Initiation.* The procedure *Initiation* (Figure 29) sends a *Reset* signal to the SUT and expects a response signal *ResetACK* within 4 time units. If it is received within the expected time, this procedure passes; if *ResetTimer* timeout is received, it fails; otherwise, it is inconclusive.

Note that the verdict assigned in the procedure is a *preliminary result.* Its value influences the *final verdict.* For example, if the procedure *Initiation* fails, it means the SUT cannot reset properly. Therefore, even if the *final verdict pass* is encountered (signal *Z* is received at the proper time, on the proper gate and carries the proper parameter), this test case still fails, because the test case may have started at the wrong state (Reset failed). Whatever happens afterwards cannot confirm anything.

Figure 28 An Example Test Case Written in SIMPL-T

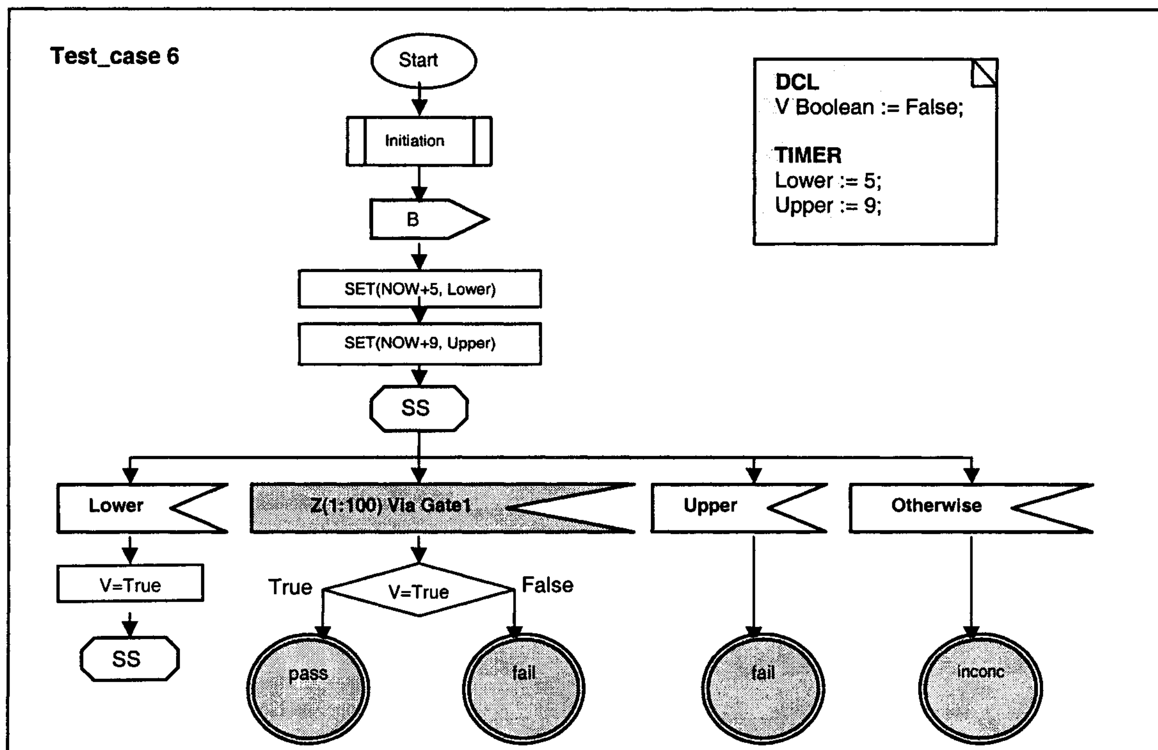
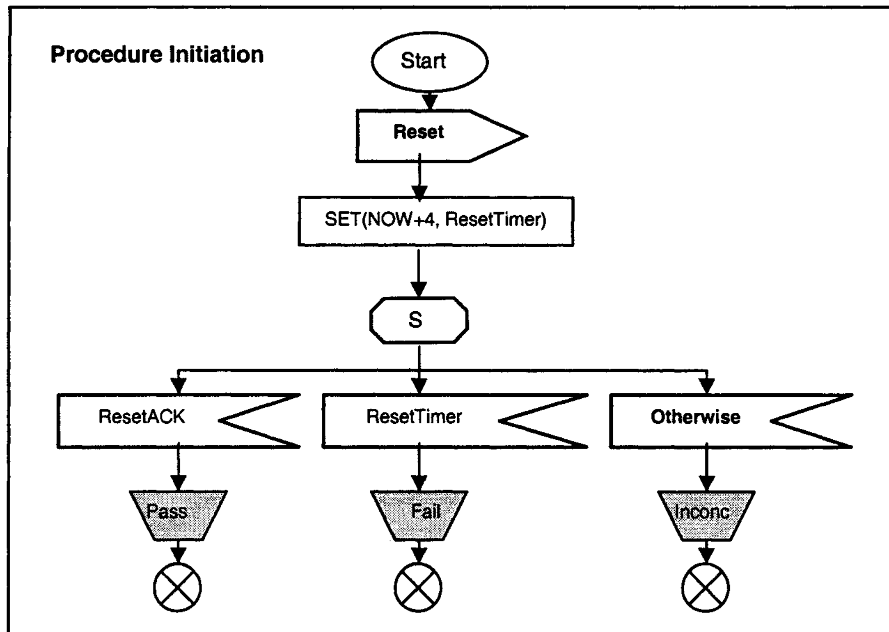


Figure 29 A Procedure Definition with Preliminary Results



4.6. Summary

This chapter defined a new test specification language SIMPL-T for SDL specifications. It recalled the key test requirements defined in Chapter 2 and used them as the benchmark for SIMPL-T definitions. Based on the SDL existing testing features identified in Chapter 3, most of these key requirements are already supported by SDL and they are reused without modification as part of SIMPL-T. For the requirements which SDL does not support, namely *Organization and Management of Tests, Checking Responses, and Assigning and Handling of Verdicts,* their semantics is defined based on TTCN concepts. Syntax is also defined in BNF form.

SIMPL-T, as an extension to SDL, is defined the same way as in SDL standard[ITUZ100]. Therefore, it is very much like SDL. The test cases written in SIMPL-T are very much like SDL processes. Therefore SIMPL-T should be easily understandable to persons familiar with SDL. As well, SIMPL-T is immediately useful for SDL users, including analysts, designers, developers and testers.

CHAPTER 5. CASE STUDY: TESTING A TRAFFIC CONTROLLER SPECIFICATION USING SIMPL-T

This chapter uses a case study to illustrate how the SIMPL-T concepts defined in this thesis are used in developing and testing a Reference Specification for a reactive application – a traffic controller. It also demonstrates a successfully developed test tool SAFIRE based on SIMPL-T concepts.

During the cooperation with SOLINET[SOLINET], the theoretical ideas of SIMPL-T were refined, developed and incorporated as new features in the SAFIRE tool[SOLINET]. These features were then validated in the Reference Specification for the SDL'03 Design Contest[SDLDC03].

The successful implementation and validation of the Reference Specification confirmed the theoretical basis of SIMPL-T and provided further ideas for future development.

This case study summarizes work carried out during the period Oct 2002 until June 2003, in preparation for the 11th SDL Forum conference[SDLCon11] held from July 1st to July 4th, 2003.

Case Study Research Plan

The goals of the case study are:

1) To confirm that the language SIMPL-T can be used to write realistic test cases to test SDL specifications, is easy to use, and yields test cases which are simple and easy to understand.
2) To confirm that the language SIMPL-T can be incorporated into test tools.

The steps are:

1) Overview the *Task Force Concepts* used in the SAFIRE tool and the case study
2) Gather the predefined requirements for a realistic contest application (SUT)
3) Specify the application using SDL, and justify it as a realistic SDL specification and SUT
4) Write a SIMPL-T test suite for testing the SUT
5) Execute the test suite using the SAFIRE tool
6) Observe the test results, and correct errors in either the SDL specification or the SIMPL-T test cases as necessary according to test results
7) Give observations and a preliminary assessment of the value of the case study

5.1. Background: The Reference Specification for SDL'03 Design Contest

The purpose of the SDL'03 design contest was to encourage debate about the use of SDL in real applications. The issues which came up at the SDL'03 conference included the core features of SDL, the benefits of simple designs and automated validation using SDL.

The task for the SDL'03 design contest was to specify a traffic light controller using SDL. The purpose of the Reference Specification was to confirm that the contest requirements were complete and feasible before they were released to the contest participants. The Reference Specification was also used to define the evaluation guidelines for the contest. As a result of applying this work, two requirements errors were discovered and corrected. Without the correction, the requirements would have been unrealistically difficult to fulfill.

The Reference Specification was implemented in the SAFIRE environment. The work of the Reference Specification includes:

- Specifying the traffic controller using SDL, which is the SUT
- Writing a SIMPL-T test suite for testing the traffic controller SDL specification
- Executing the test suite using SAFIRE tool
- Correcting errors in the specification and in the test cases according to test results

This Reference Specification had to consider the available features of the SAFIRE tool and the concepts the tool based on. The concepts SAFIRE used were mostly the *SDL Task Force[SDLTF]* Concepts wherever they are different from traditional SDL concepts.

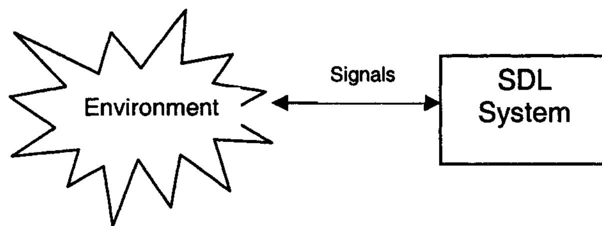
To avoid confusion, the next section explains the concepts used in the SAFIRE environment and the traffic controller specification, and how they are referred to in the rest of this chapter.

5.2. *Task Force* Concepts Used in SAFIRE

5.2.1. A System and Its Environment

SDL sees the world as divided into two parts: the system and its environment. The system is the object under consideration. Everything outside the system is part of the environment. The SDL specification defines how the system reacts to its environment through signal exchange (Figure 30)[Ells+97].

Figure 30 SDL System and Its Environment



Although the environment is not specified formally, some assumptions about the environment are usually made. For example, the environment has to interact with the system. As a normal behavior, when it receives a signal from the system, it should respond, and this response takes finite time duration. As an example, allowing the system to wait forever for a response from the environment is not the environment's normal behavior.

To test an SDL specification, the test tool should provide a means of simulating the environment and of sending signals to the SDL system. Telelogic TAU[Telelogic]

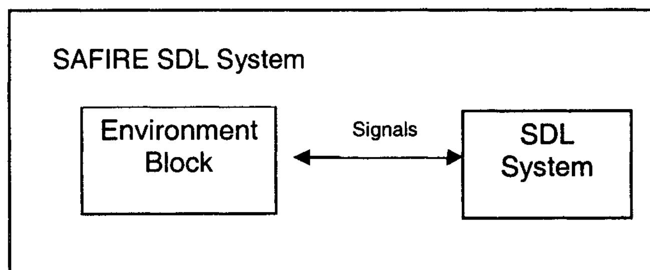
provides a *Simulator* to allow sending signals to the system manually. SAFIRE provides a model which makes this simulation more automatic.

Specifically, SAFIRE allows the environment to be modeled as an SDL block (Figure 31). We name it *Environment Block* to make it easy to refer to. This model makes an SDL *open system* look like a *closed system.* Therefore, it is easier to manage, control, and test.

The SDL Task Force defines an SDL System as: "A system is a complete, self-contained state machine without any gates and can be executed."[SDLTF]

Figure 31 shows a SAFIRE SDL system model. The screen shot diagrams of *Traffic Controller SDL System* (Figure 35) will be shown this way in Section 5.4.2. However when we discuss this system, we only refer to the *SDL System* itself (not including the *Environment Block*).

Figure 31 SAFIRE SDL System



In Section 5.5.2, we will show how the SAFIRE SDL System/Environment architecture is mapped onto a testing architecture.

5.2.2. Block, Process and Agent

Traditionally an SDL *process* is a state machine which has specific behaviors. A set of processes can be grouped into a block. A *block* is a container which does not have its own behavior. As a result, many blocks may have only one process inside to follow this rule. This makes the diagrams more complicated and generates unnecessary editing work.

The introduction of *Agent in* SDL-2000 solved this problem[ITUZ100]. Although there are still *Block Agent* and *Process Agent*, the boundary between Block and Process is fading, and a *block agent* can now have its own state machines.

In the same fashion, the *SDL Task Force* does not distinguish between *block* and *process*. Instead, they are both called *Instance*.

In this thesis case study, when an agent is discussed as a black box, it is referred to as a block, while when the same agent's actual behavior is discussed, it is referred to as a process.

5.2.3. Class, Agent Type and Agent

In object-oriented (OO) programming, a class is a software module that represents and defines a set of similar objects, its instances. All the objects with the same properties and behaviors are instances of one class[Leth+01]. In Java and C++, all objects should be defined as classes and instantiated when they are used.

Type (agent Type) in SDL acts the same way as *Class* in OO programming. The only difference is that SDL allows the independent existence of agents.

The SDL Task Force pushes the *Type* concept one step closer to OO *Class*. It requires that all Agents (Process and Block) have to be defined as Agent Type, and instantiated when used. We will see examples of *Agent Type* in the Traffic Controller SDL System diagrams in later sections.

5.3. Summary of SDL'03 Design Contest Requirements

The challenge for the SDL '03 design contest was to design a traffic light controller with the following characteristics[SDLFO]:

1. The traffic lights are located at a traffic intersection arranged as two roads crossing with one lane in each direction.
2. Each direction has a set of three lights with different colors; red means stop, green means go. The transition from go to stop is indicated by yellow. The lights at opposite sides of the intersection have the same color.
3. Each set of three lights is a single unit with a control interface to select the color red, yellow or green. Only one of the three lights is on at any time. Note: The lights unit is not part of the controller.
4. There are no lanes for turning, but turning left and right is allowed when green otherwise not (right turn on red not allowed).
5. Each lane arriving at the intersection has a single sensor, which has two states: traffic and no traffic. Note: The sensor indicates the transition from one state to another and is not part of the controller.
6. The lights spend a maximum time in any state, i.e. traffic from the left or right will not be permanently be blocked by continual traffic in the other directions. The transition time is configurable without recompilation.

7. If traffic is waiting in a red direction and the green direction is free of traffic a transition will be initiated after a configurable delay. If traffic arrives in the green direction during this time, the transition is deferred.
8. Traffic is not allowed to enter the intersection unless the exit is free. It is assumed the configurable transition time is always greater than the time needed for traffic to cross the intersection.
9. The controller is connected to a reset switch, which can be used to reset the controller to the starting state. The starting state is defined to be green for one road and red for the other.
10. Reserved for future extensions.

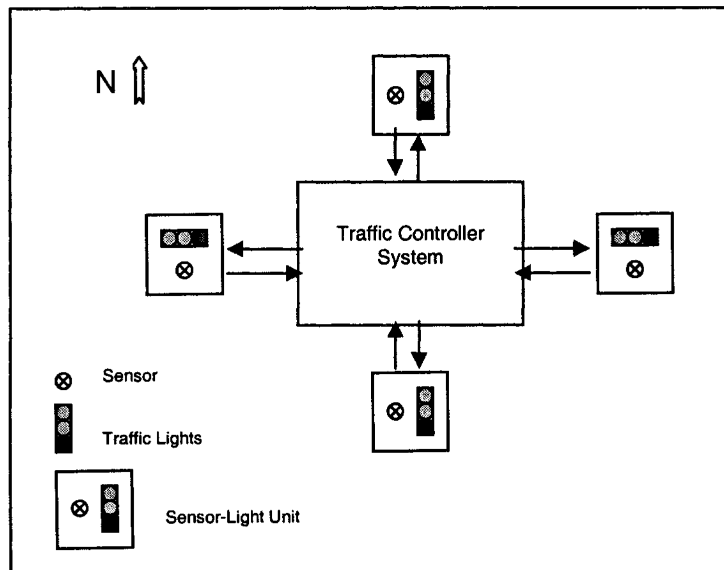
The requirements are to be captured faithfully by an SDL specification submitted by each team of contest participants. These submitted specifications are judged according to the satisfaction of guidelines released before the contest presentations based on the Reference Specification provided by this author.

5.4. Specifying the Traffic Controller System Using SDL (SUT)

5.4.1. The System and Its Environment

The traffic controller system is specified in SDL. It is connected to four sets of *Sensor-Light Units* (refer to East (E), West (W), North (N) and South (S)). Each of these units consists of a sensor and a set of three traffic lights. The sensors provide traffic conditions of each corner of the intersection, and the traffic lights displays the status of the lights at each direction. All the *Sensor-Light Units* together form the environment of the system (Figure 32).

Figure 32 The Traffic Controller System and Its Environment



5.4.2. System Architecture

The system consists of three blocks: the NS processor, the EW processor, and the controller.

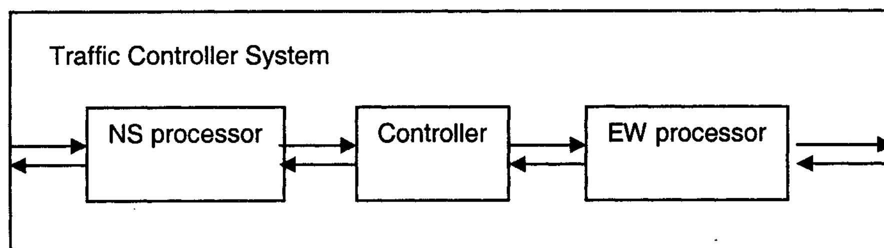
The NS processor connects to the controller, the *Sensor-Light unit* at the north direction and the *Sensor-Light unit* at the south direction. It collects the traffic condition signals from both sensors, process them and notify the controller. It also receives traffic control signals from the controller and distributes them to the two sets of lights.

The EW processor functions exactly the same way as the NS processor, but connects to the *Sensor-Light Units* at the east and west directions instead.

The controller connects to the NS processor and the EW processor. It collects traffic condition signals from both processors, makes decisions regarding traffic control, and sends traffic control signals to the processors.

The system architecture is shown in Figure 33.

Figure 33 Traffic Controller System Architecture



In this design, the controller connects to the environment through the two processors. The two processors connect and communicate with the *Sensor-Light Units*. This partitions the traffic control work into three parts and simplifies the behavior of the controller.

Since the lights at the opposite sides of the intersection have the same color (as indicated by point 2 in the requirements), traffic at either side of these two directions should cause the same effect to the traffic control decisions. Therefore, the *Sensor-Light Units* at opposite sides can be connected and processed by the same processor.

Additionally, since the EW processor functions exactly the same way as the NS processor, they can be instantiated from the same block type as shown in Figure 34.

Figure 34 System Architecture with Block Type Definition

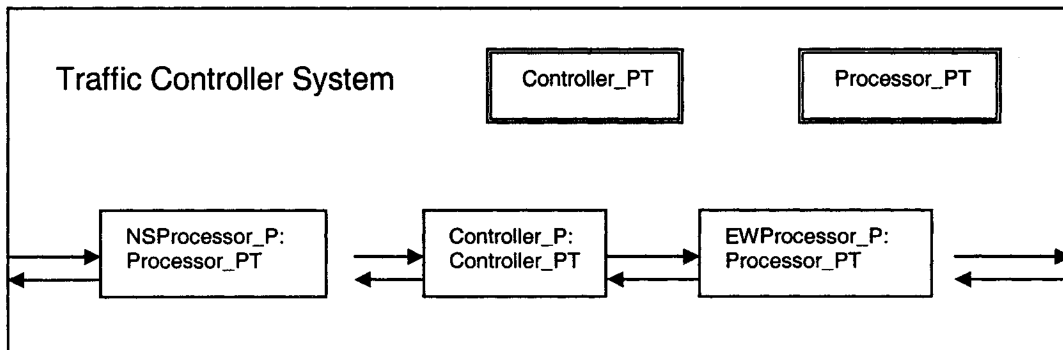
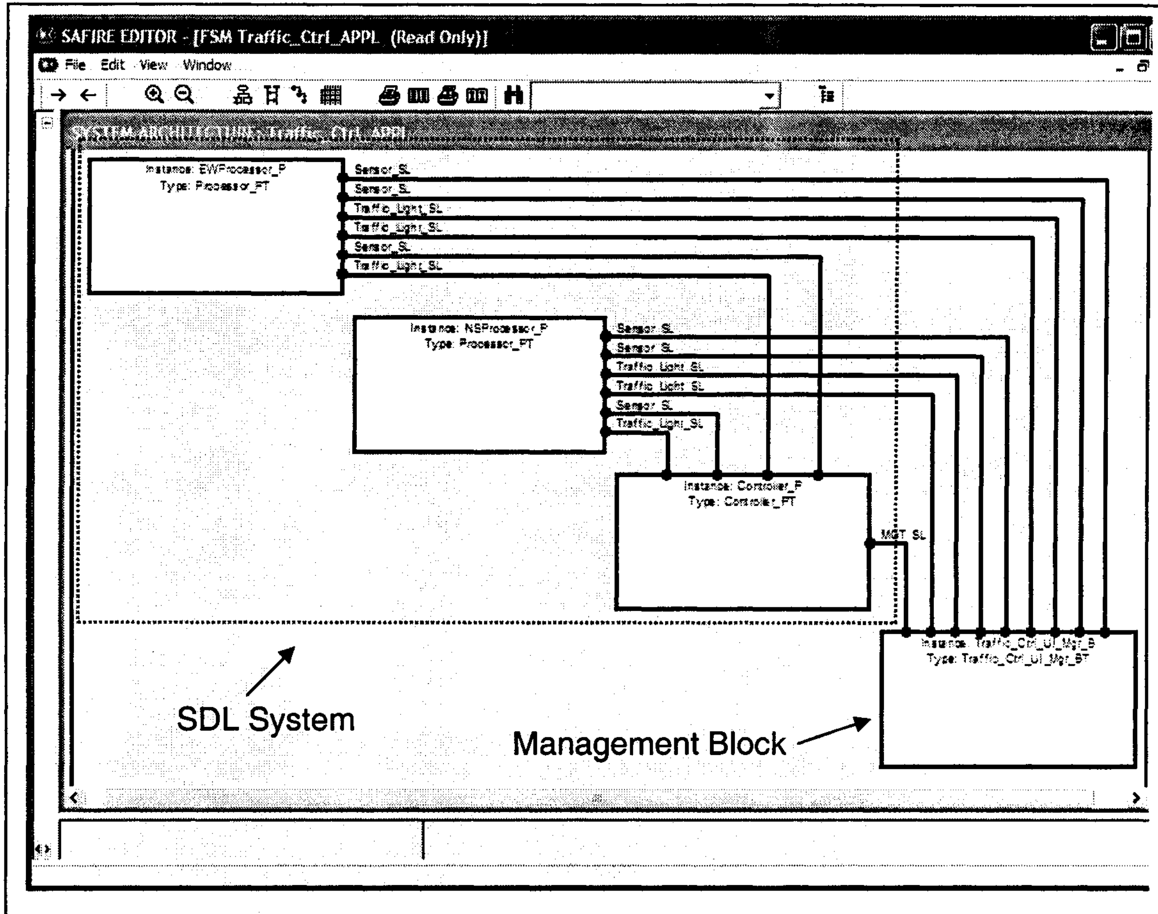


Figure 35 shows the *Traffic Controller SDL System* implemented in SAFIRE environment. In this diagram, the right-bottom corner block is the Environment Block, which simulates the environment of the rest of the SDL system. This is an example of the key concept described in Section 5.2.1.

Figure 35 Traffic Controller SDL System Implemented in the SAFIRE Environment



5.4.3. The Processor Agent Type

The responsibility of each Processor is to collect traffic information from both sensors it connected to and to pass the information to the controller.

The *Processor* Agent has six gates:

- Sensor_A_G: Connect to *A Sensor*, to collect traffic condition signals from A direction

- Sensor_B_G: Connect to *B Sensor*, to collect traffic condition signals from B direction
- Ctrl_Sensor_G: Connect to the *Controller*, to send traffic condition signal to the *controller*
- Ctrl_Light_G: Connect to the *Controller*, to receive traffic control signal from the *controller*
- Light_A_G: Connect to *A Lights*, to send traffic control signal to A direction
- Light_B_G: Connect to *B Lights*, to send traffic control signal to B direction

Traffic Condition Signals are:

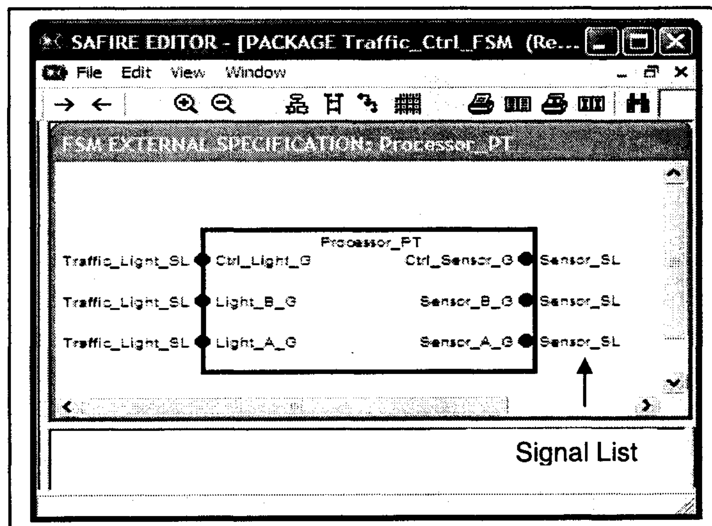
- Traffic
- No Traffic

Traffic Control Signals are:

- Green
- Yellow
- Red

The *Processor* agent type, its gates and associated signal lists are shown in Figure 36.

Figure 36 Processor Agent Type, Its Gates and Associated Signal Lists



As a process, the *Processor* has 4 states:

- No_Traffic,
- A_Traffic_B_No_Traffic,
- A_No_Traffic_B_Traffic,
- A_Traffic_B_Traffic

The state chart in Figure 37 shows the relationship among these states. Figure 38 shows an example of the Processor state diagram.

Figure 37 State Chart of the Processor

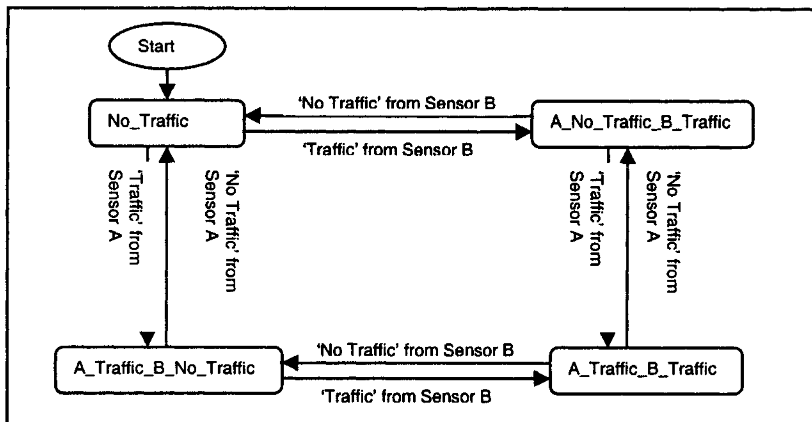
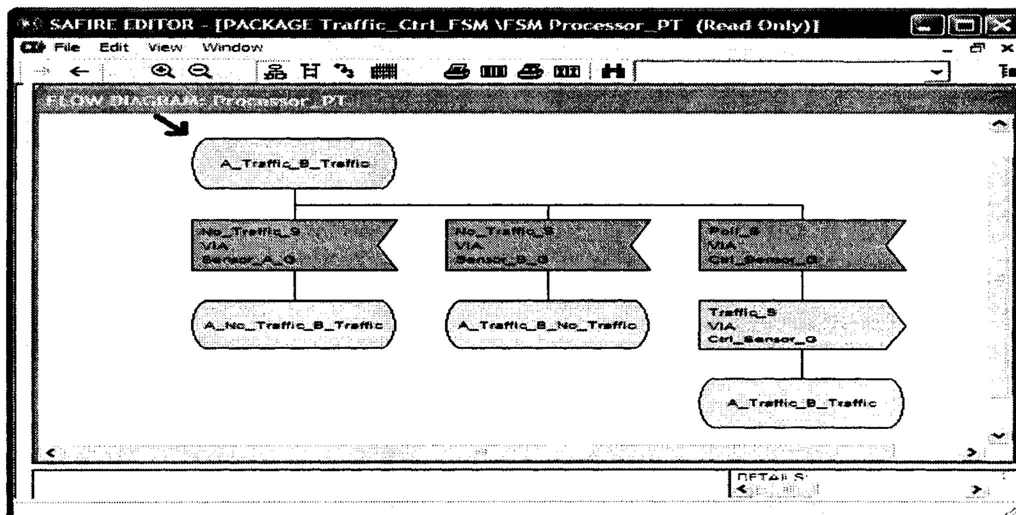
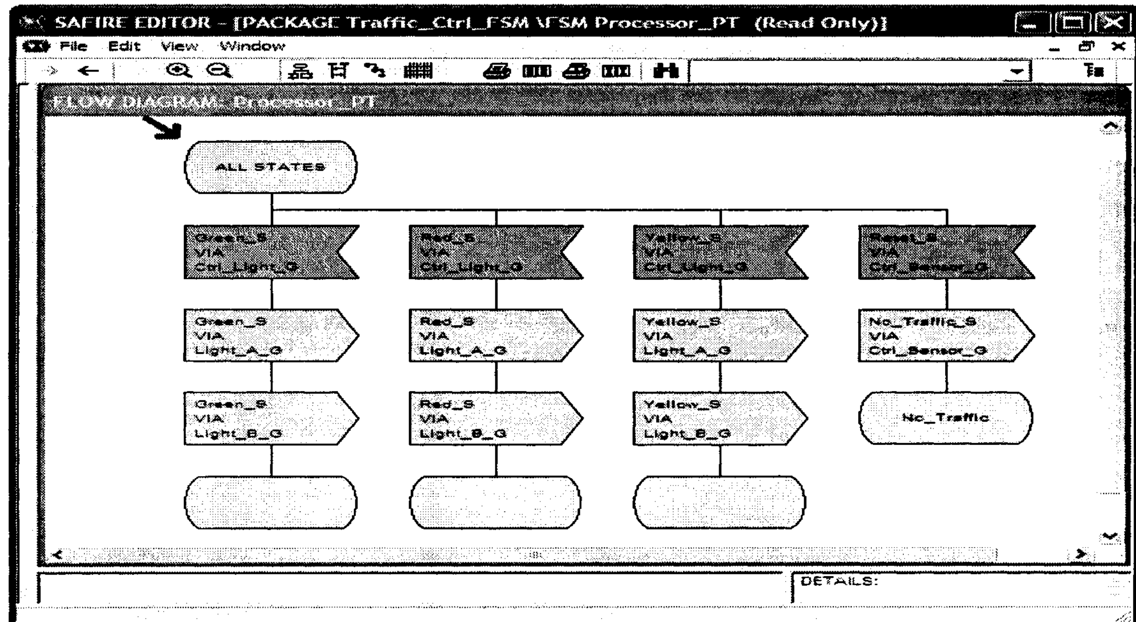


Figure 38 Processor State Diagram – Transition Triggered by Signals from the Sensors



The Processor also receives traffic control signals from the controller and passes them to the lights. This happens the same way at all states, and it does not trigger a state transition to a new state (Figure 39).

Figure 39 Processor State Diagram – Pass Signals from the Controller to the Lights



5.4.4. The Controller

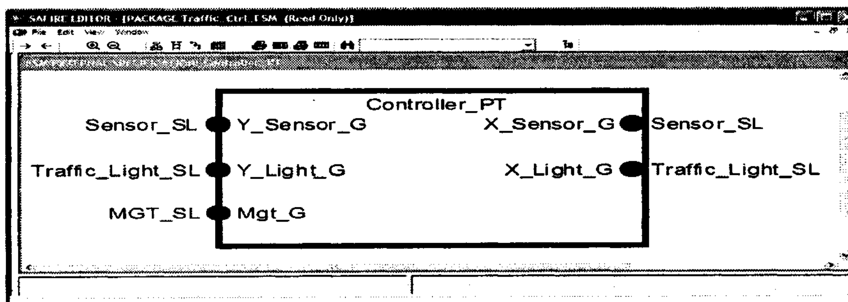
The responsibility of the *Controller* is to collect traffic condition signals from the two *Processors* it is connected to, to make decisions regarding traffic control, and to send traffic control signals to the *Processors*.

The *Controller* has five gates:

- X_Sensor_G: Connect to the Ctrl_Sensor_G of the *EWProcessor* to collect traffic condition information of east-west direction.
- X_Light_G: Connect to the Ctrl_Light_G of the *EWProcessor* to send traffic condition signals to the lights at east-west direction.
- Y_Sensor_G: Connect to the Ctrl_Sensor_G of the *NSProcessor* to collect traffic condition information of north-south direction.
- Y_Light_G: Connect to the Ctrl_Light_G of the *NSProcessor* to send traffic condition signals to the lights at north-south direction.
- Mgt_G: Connect to the Environment Block for Exception handling.

The *Controller* agent, its gates and associated signal lists are shown in Figure 40.

Figure 40 Controller Agent Type, Its Gates and Associated Signal Lists



The *Controller* process has ten states. They are shown in Figure 41. Figure 42 shows an example of the Controller Process diagram.

Figure 41 The Ten States of the Controller Process Agent

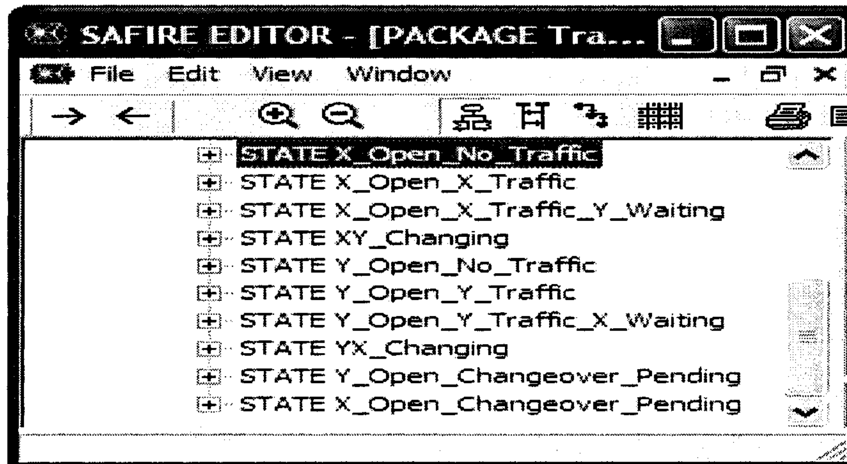
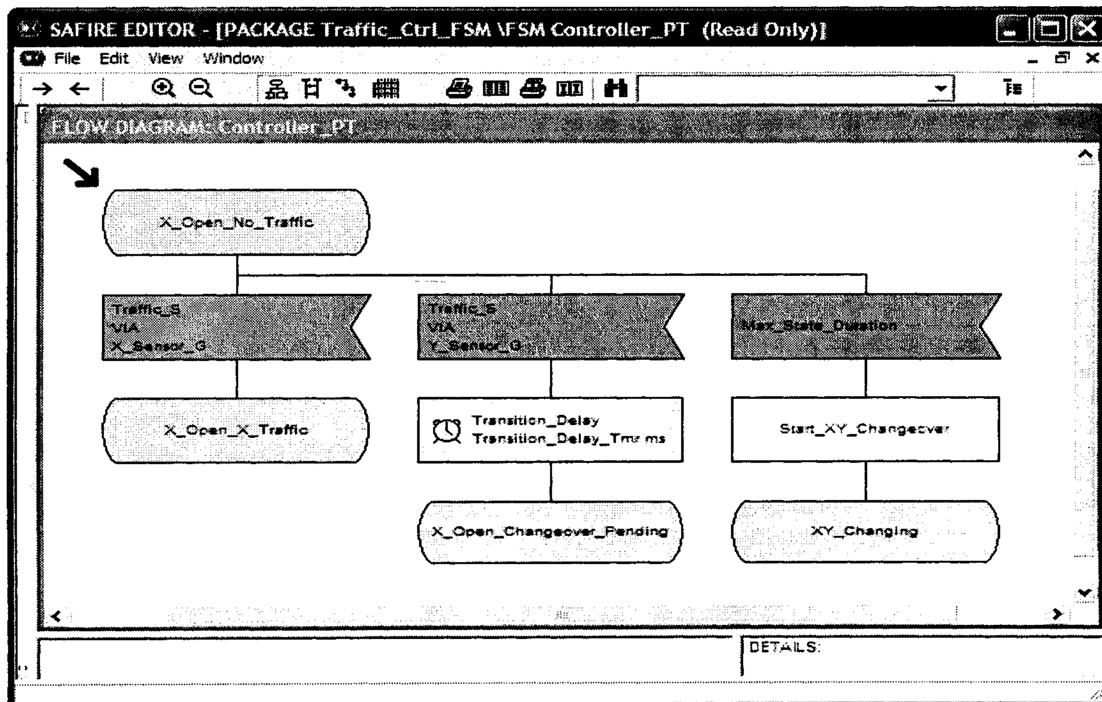


Figure 42 An Example of the Controller Process State Diagram



5.5. Test System Configuration in the SAFIRE Environment

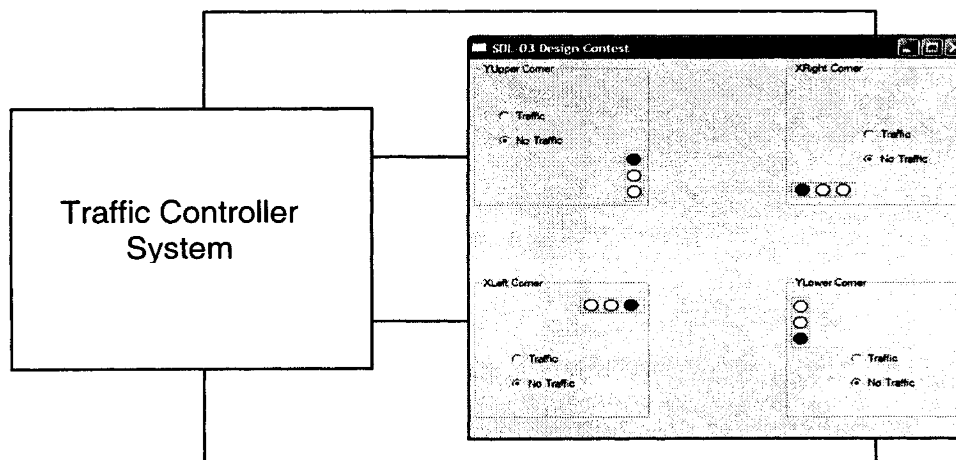
Now, starting from this section, we discuss the validation of the Reference Specification under Test (SUT). Since validation of a specification uses Black-Box techniques, we do not have to know implementation details. We do need to know its connection points (gates) to the environment and the signal lists associated with these gates.

5.5.1. A User Interface for Interactive Simulation

At the time the Reference Specification was built, the SAFIRE tool did not completely support automatic test case execution yet. The SAFIRE engineers built a temporary user interface for testing the Reference Specification.

This user interface simulates the environment of the traffic controller SDL system. It provides the traffic conditions at each lane and displays the status of the lights (Figure 43).

Figure 43 The Traffic Controller SDL System (SUT) and A Temporary User Interface



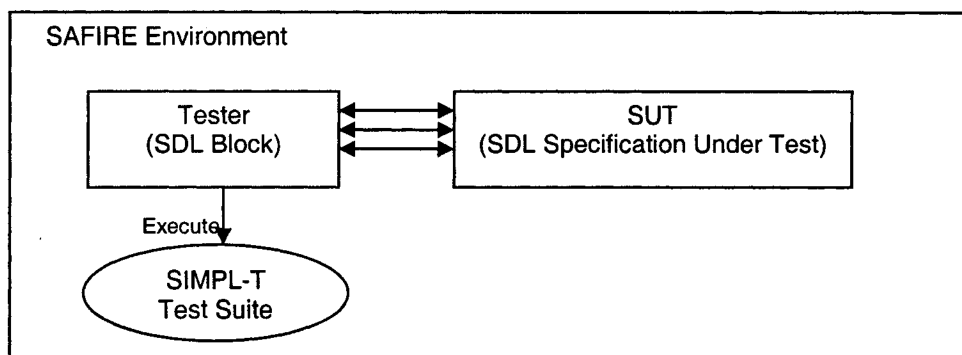
This configuration allowed the basic functionality to be tested interactively in an informal manner. However, manual simulation is not enough for formal validation. It does not assure reliability in test execution, it does not provide repeatability, and test results cannot be automatically documented.

5.5.2. The Test System Configuration in the SAFIRE Environment

For formal validation an automated testing configuration should be used, where test scenarios can be executed automatically by a Tester connected to the SUT.

In this case the SUT is still the Traffic Controller SDL System. Instead of being connected to a user interface representing its environment, the SUT has to be connected to the Tester, which can run test cases automatically. Figure 44 shows the test system configuration in the SAFIRE environment.

Figure 44 The Test System Configuration in the SAFIRE Environment



5.6. Test Suite Design in SIMPL-T

A test suite was designed in SIMPL-T by:

- *Defining the Test Interface*

The interface between the Tester (test suite) and the SUT was defined via the declarations of the gates which connect the Tester and the SUT and the associated signals.

- *Identifying Reusable SUT Components*

Since SIMPL-T is an extension to SDL, theoretically some SDL components in the SUT should be reusable by the test suite. The SAFIRE environment supports this reusability. Therefore, for efficiency, the reusable SUT components were identified before the test suite was developed.

These reusable components include signals (signal lists), user-defined data types and timers.

The internal signals of the SUT are not visible to the Tester, but all the external signals (signals on the interface to the environment) can be reused by the test suite.

The Traffic Controller SDL System does not have any user-defined data types. Otherwise, they can be reused.

Two types of timers are normally used in SDL specifications, one for timing events (timing behavior), the other for exception handling. The exception handling timers are normally used internally, while the system behavior timers should be visible to the Testers, and they should be reused in the test suites.

In the Traffic Controller SDL System, 3 timers were reused. They are: the max time duration any light can stay green, the time duration all lights stay yellow, and the minimum time duration any light stays green.

- *Declaring Timers*

To monitor response time, handle transmission errors and signal loss and avoid deadlock, timers should be used. A number of timers were used in this test suite. For example, a guard timer was used to assure that the tester will not wait forever for a response. This timer should be set long enough to allow time for signal transition, process and actions, and it should be short enough for efficiency of testing. In our case, it was set to 3 times the "max time duration any light can stay green".

- *Defining Procedures*

A number of procedures were defined as setup initiation, test steps and preambles. By moving groups of actions into procedures, test cases look neat and less complicated.

- *Defining Test Cases*

Since requirements are written in natural language, they are often ambiguous and incomplete. Before test cases were defined, the requirements were examined to check correctness and completeness. As a result, an error in the requirements was found.

After clarifying and complete the requirements, a test case was designed for each item of the requirements.

5.7. Test Suite Development using SIMPL-T in SAFIRE Environment

The test suite was written in SIMPL-T using the SAFIRE Editor, which has a graphical user interface using the SIMPL-T notation. The Editor had also been enhanced to support the features defined in SIMPL-T.

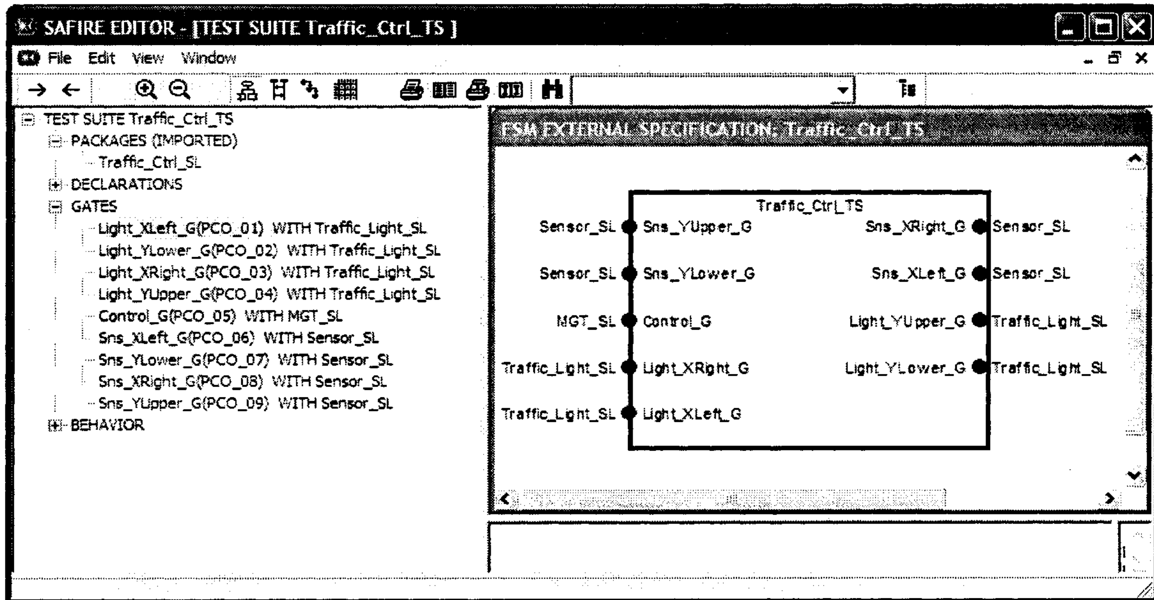
5.7.1. Gates and Signal List Declarations

The interface between the Tester and the SUT is actually the environment of the SUT. The gates and signals are the same from both sides of this interface, so the interface is symmetrical. Therefore the Tester has the same environment as the SUT.

This model implies that the Tester has the same number of gates as the SUT and that the signals used by the Tester are the same as by the SUT (only the signals in each side of the interface travel in opposite direction).

To implement this model, the Tester can be defined as an SDL block with the required number of gates. The gates of the SUT can be mapped and connected to the corresponding gates of the Tester. And the signals of each gates in the Tester can be defined mapping the signals in the corresponding gates of the SUT, but in opposite direction (Figure 45).

Figure 45 The Gate and Signal List Declarations in a Test Suite



5.7.2. Timer and Variable Declarations

Timers are declared to test time events and to monitor the responding time from the SUT. Variables are declared to store temporary values. Here are some examples:

- *Timer: Max_State_Duration, Transition, Transition_Delay*

Max_State_Duration is the max time duration that any light can stay green before it changes. This is set for requirement 6 “The lights spend a maximum time in any state”.

Transition is the time duration that a light stays yellow.

Transition_Delay is the minimum time duration that a light stay green. It is also used to set delay after a "Change light" decision is made. This delay makes the "transition defer" in requirement 7 possible.

These three timers should be used in the SUT and visible to the Tester. Therefore, these values of these timers in the test suite are set to the values used in the SUT. To do so, three variables are used to record these values from the parameters carried by a signal from the SUT.

- *Timer: Max_TC_Duration*

This is a guard timer used to monitor the responding time from the SUT. It assures the Tester will not wait forever for a response.

In an SDL specification, the signal exchange time, especially through signal route, is ignored, but in testing, the process time and transmission time must be taken into account when the guard timer is set.

Max_TC_Duration should be set long enough to allow waiting, delay, transition and process time, therefore, it is set to 3 times the value of *Max_State_Duration.*

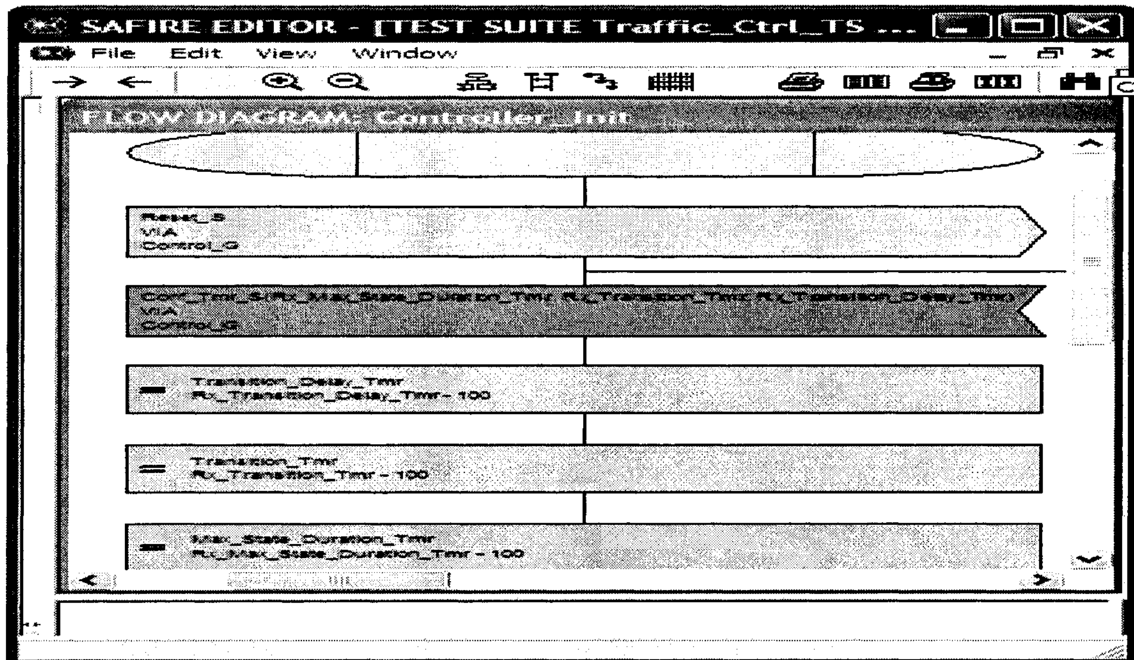
5.7.3. Procedure Declarations

Procedures are used for test setup (initiation), test steps and preambles. Here are some examples:

- *Procedure: Controller_Init*

This procedure defines the first step of every test case. It sends a reset signal to the SUT, and receives a reset confirmation. This confirmation signal carries the value of the three timers we discussed earlier, and these values are saved in variables (
Figure 46).

Figure 46 Part of the *Controller_Init* Procedure Diagram



- *Procedure: Open_Y_Direction*

This procedure defines the test step *Open_Y_Direction.* In this step, the *Tester* should receive four signals from the SUT to set X direction to red, and Y direction to green. Since these four signals may arrive in any order, all the possibilities of different order have to be considered.

If any other signals arrive on any gates, they are disregarded. If the *Max_TC_Duration* timer-out is received, which means the four expected signals did not all arrive within reasonable time, this test step fails.

Note that in a test step, a verdict is not a *final verdict*, but a *preliminary result.* SAFIRE does not use the symbol defined in SIMPL-T, but a square for a *preliminary result* (Figure 47).

Figure 47 Procedure *Open_Y_Direction*

5.7.4. Test Cases

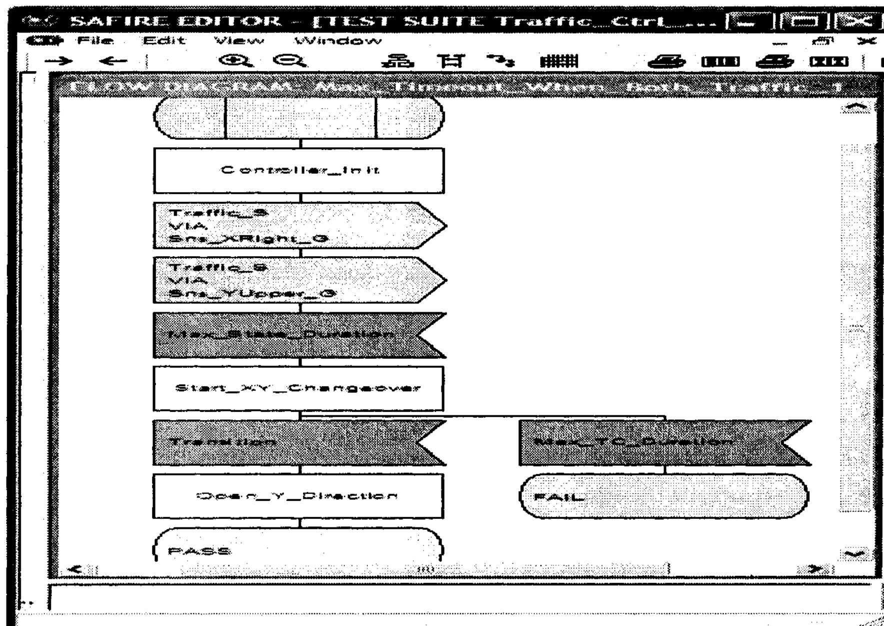
Test cases are defined in SAFIRE. They are designed to test normal scenarios and race scenarios. Here are some examples:

- *Test Case: Max_Timeout_When_Both_Traffic*

Test purpose: Test whether the lights spend a maximum time in any state.
Pre-amble: X direction is green, and Y direction is red. Both directions have traffic.
Post-amble: X direction is red and Y direction is green.

This test case uses the *Controller_Init* procedure to set the pre-amble. Then it uses the *Start_XY_Changeover* procedure to set all the lights to yellow. After a delay (*Transition* timer timeout), it uses the *Open_Y_Direction* procedure to set the lights in X direction to red and in Y direction to green (Figure 48).

Figure 48 Test Case *Max_Timeout_When_Both_Traffic*



- *Test Case: XY_Transition_Defer*

Test purpose: If traffic is waiting in a red direction and the green direction is free of traffic, a transition will be initiated after a configurable delay. If traffic arrives in the green direction during this time, the transition is deferred.

Pre-amble: X direction is green without traffic, and Y direction is red with traffic waiting.

Post-amble: X direction is still green and Y direction is still red.

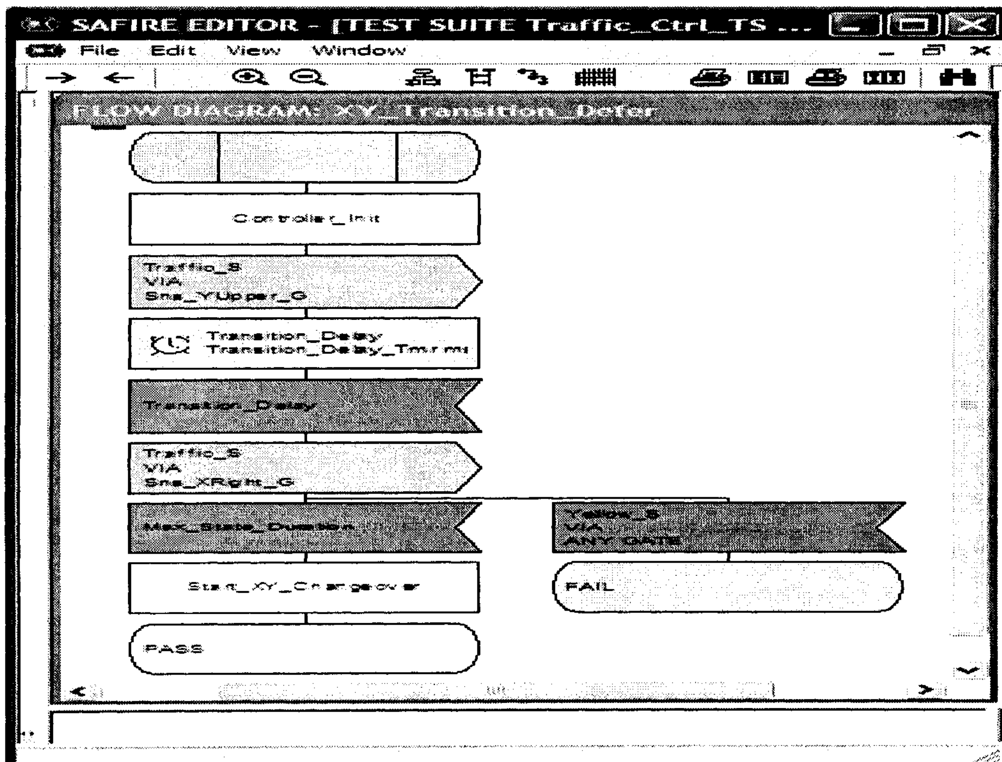
This test case sets up the pre-amble by calling the *Controller_Init* procedure and sending traffic signal to the SUT in Y direction.

Race Scenario:

After minimum time duration *(Transition_Delay),* the lights in X direction are supposed to change to yellow. At this moment, traffic arrives in X direction. This is a typical race scenario. Will the controller continue its decision for the transition, or will it change its mind and defer this transition because of the arriving traffic in X direction?

To test this race scenario, if the *Tester* receives a *Max_State_Duration* timeout signal before a Yellow_S signal, it means the transition is deferred and the lights did not change until the max time duration *(Max_State_Duration)* has reached. Therefore, this test case passes. Otherwise, it fails (Figure 49).

Figure 49 Test Case *XY_Transition_Defer*

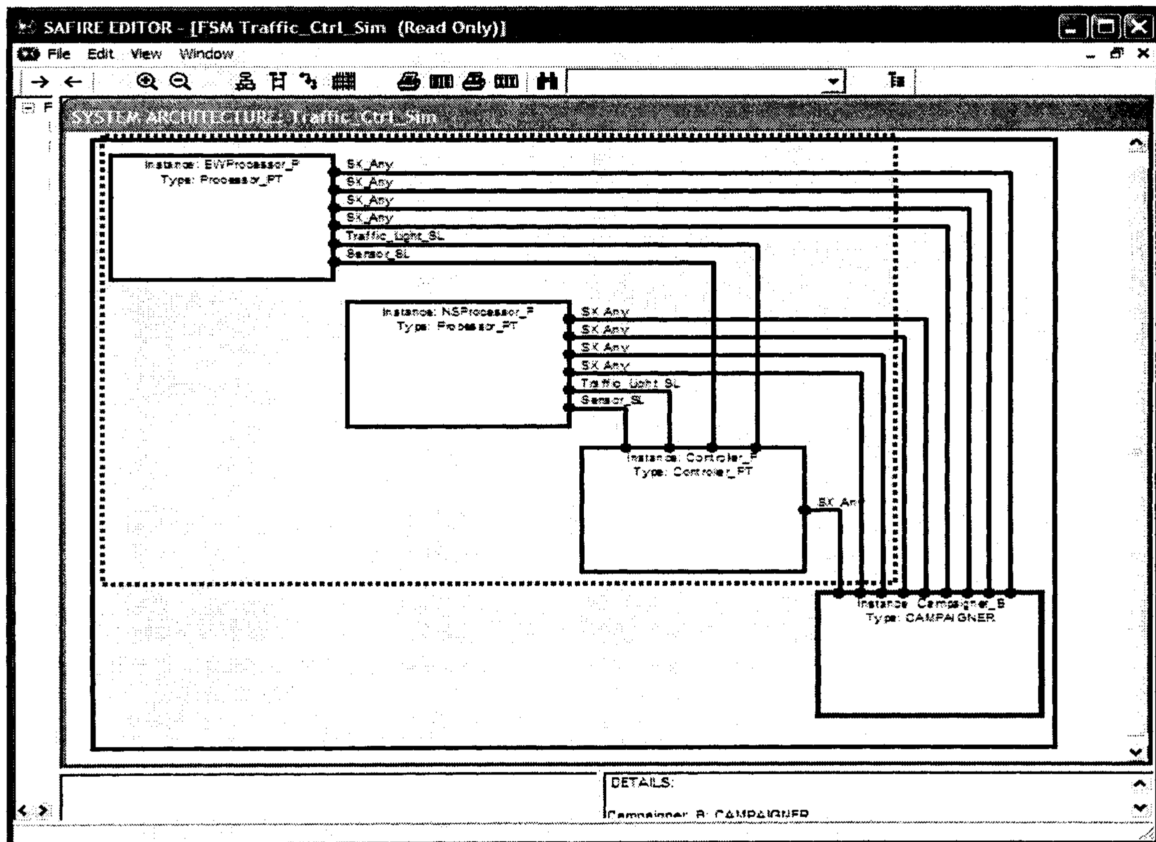


5.8. *Test Harness* and Tester-SUT Connection

The Tester is called *Campaigner* in SAFIRE, which is a predefined SDL block type. It can be instantiated from the SAFIRE library, and connected to the SUT.

To do this, a system is created in SAFIRE containing an instance of the Tester and an instance of the SUT. The gates on the SUT are connected to the gates on the Tester, which in effect are the environment of the test suite. This system is called the *Test Harness* in SAFIRE, which is shown in Figure 50. In this diagram, the *Campaigner* is the Tester, and it replaced the *Environment Block* in the system diagram of Figure 35.

Figure 50 Test Harness in SAFIRE



5.9. Tester and Test Case Execution

The Tester used for validating Reference Specification was the SAFIRE *Campaigner*, which had been enhanced with the features presented earlier in this thesis to manage the execution of test suites written in SIMPL-T.

The *Campaigner* has a user interface for selecting test cases from a test suite (Figure 51) and for executing them as a batch automatically (Figure 52).

Figure 51 SAFIRE CAMPAIGNER – Select Test Cases

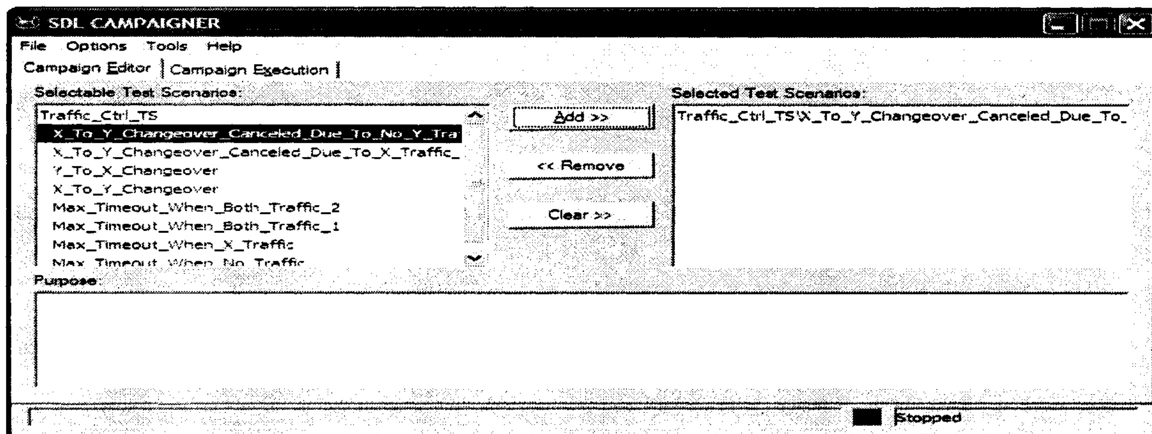
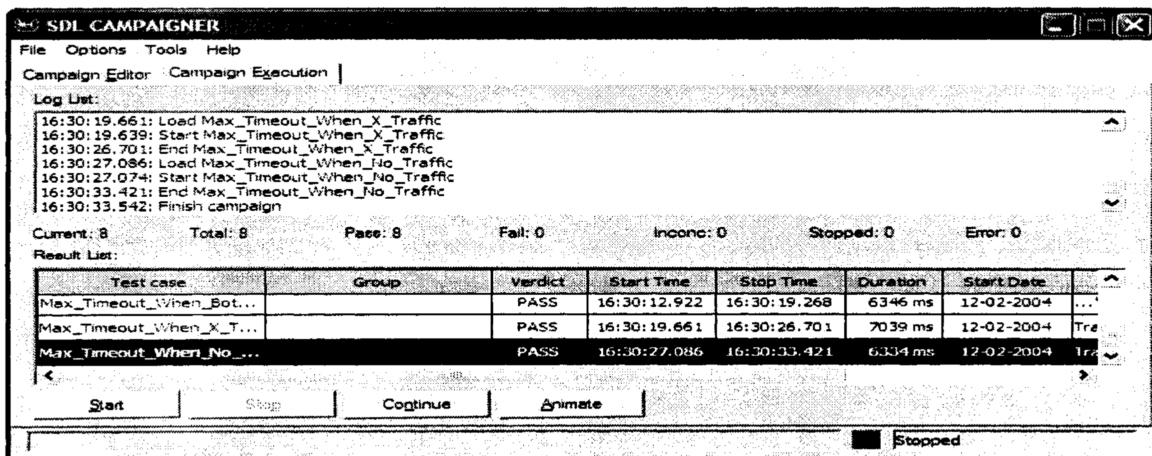


Figure 52 SAFIRE CAMPAIGNER – Automatic Execution of Selected Test Cases



5.10. Test Results Observations and Modifications Made

As a result of the extensions made in Chapter 4 for organization of tests, the SAFIRE Campaigner can automatically generate a test report of the SIMPL-T test cases executed and the verdict of each test case.

Originally 11 test cases were written. They covered all the scenarios possible directly and indirectly from the user requirements. After execution, 8 of them are passed and 3 are failed.

Among the 3 failed test cases, one of them has an error in the test case. After correction, it is a redundant test case (basically the same as another test case). Two other failed test cases correspond to errors in the Reference SDL Specification Under Test (SUT).

After correcting the Reference Specification and removing the redundant test case, all 10 remaining test cases were passed.

5.11. Observations and Assessment of Case Study

In this case study, a realistic, real-time application – a traffic controller - was specified using SDL. To test this SDL specification, a test suite was written using SIMPL-T and executed successfully, finding two errors in the specification.

The test cases written in SIMPL-T in this case study are intuitive and easy to understand. As a result, an error in one of the test cases was easily discovered.

Writing test cases is easy using SIMPL-T. It would take more time if TTCN were used, since all signals defined in the SUT are reused in the SIMPL-T test suite, but they cannot be reused in TTCN.

This case study also confirmed that the language SIMPL-T can be incorporated into the development of a commercial test tool. In fact, the successful test tool SAFIRE is based on SIMPL-T concepts. Both specification and validation of this Traffic Controller SDL System were accomplished via SDL, SIMPL-T and the SAFIRE environment.

The case study involved roughly 3 person months of a SDL/testing expert, but much of this time was involved in tool development and trail. With the current capabilities of SAFIRE, a similar project could be completed in less than one person month.

In summary, SIMPL-T, supported by SAFIRE, and part of the foundation of SAFIRE, facilitates both building testable specifications in SDL and also validating (testing) these specifications in a cost-effective manner. A complete evaluation and assessment of the value of this language is given in the next chapter.

CHAPTER 6. EVALUATION OF APPROACH

This chapter evaluates SIMPL-T through the following means:

- analyzing experience gained from the case study
- comparing SIMPL-T to other approaches
- assessing its industrial applicability
- assessing its support for important test concepts

6.1. Strengths and Weaknesses of SIMPL-T from Case Study

- ***Strengths***

SIMPL-T is very easy to use when writing test cases for the Traffic Controller SDL System (SUT). This is very similar to writing an SDL specification, except for some new SIMPL-T syntax and semantics.

The test cases written in SIMPL-T are very much like SDL processes. They are intuitive and easy to understand.

In summary, for SDL users, SIMPL-T is nothing new but simpler with minor intuitive extensions.

- ***Weaknesses***

The case study was conducted in the SAFIRE environment. Thus it was limited to the available tool concepts and features.

These different concepts have been discussed in section 5.2 and include:

- o Test group organization – This was not well supported. Therefore, the test suite is completely flat (i.e. has no structure). This feature has now been added.
- o *Save* Construct – It was not supported in SAFIRE. Therefore, Ordering Problem exists in the case study (for details see section 6.4). To solve this, additional procedures were used to make the test cases appear less complicated.

6.2. Comparisons with TTCN-based Approaches

To evaluate SIMPL-T, we compare its use to the use of TTCN, the international standard testing language, for validating SDL specifications.

1) TTCN has no advantage over SIMPL-T in testing SDL specifications

It seems that TTCN should be a better choice to test SDL specifications, not only because TTCN is a standard test language and it has been used traditionally, but also because there are some tools in the market which support automatic TTCN test case generation from SDL specifications. However, this is a misunderstanding.

As we stated in Chapter 1, automatically generated test cases from a specification cannot be used to validate the specification itself. What these test cases can do is to test

whether the system is implemented according to the specification, i.e., to test conformance of the implementations to the specifications.

To validate a requirements specification against user requirements, independent test cases have to be written manually, regardless of language used. Therefore, in this sense, TTCN has no advantage over SIMPL-T.

2) For SDL users, SIMPL-T is more intuitive and easier to understand than TTCN

Since SIMPL-T reuses many SDL features, and the extensions defined in the same fashion as in SDL, it should take very little effort for SDL users to learn SIMPL-T. While TTCN is a completely different language and it is powerful and feature rich. Thus it should take much more effort for SDL users to learn it.

3) For testing SDL specifications, SIMPL-T is more efficient and reliable than TTCN

Since SIMPL-T is an extension to SDL, signals, user-defined data types and many other components defined in the SDL specifications can be reused in the SIMPL-T test suite. But they cannot be reused in TTCN test suite.

This reusability not only saves time for the testers, but makes the test cases more reliable than redefining them manually, which is error-prone and reduces reliability.

For example, many telecommunications standards from ITU-T were developed as executable SDL specifications. There are many type definitions and declarations involved. The test suite independent types, such as ASP and PDU type definitions, defined in the SDL specifications can be imported and reused in the SIMPL-T test suite.

As stated by Dr. Ostap Monkewich, Vice Chairman of the ITU-T Study Group 17[ITUSG17], a reasonably complex protocol may have 30 to 40 PDUs and 6 to 10 ASPs. It should take about one hour to enter correctly one PDU into the test suite from the protocol standard if a tool is used. An ASP would take about the same effort. Therefore, in total, it may take up to 40 hours to re-enter these PDU/ASP definitions. If SIMPL-T is used, all these hours could be saved.

4) TTCN Undefined-object handling mechanism is not needed in SIMPL-T

TTCN *Otherwise* statement can be used to handle the objects that may not normally be recognized as proper ASPs or PDUs, due to the fact that the IUT may not be working correctly.

SDL also has an *Otherwise* statement, which is defined to handle the signals which are not explicitly specified in a specific state. It can be useful in SIMPL-T to handle the responses from the SUT which are not explicitly specified in a state of a test case.

However, the *Otherwise* statement in SDL is different from in TTCN, because the un-specified signals handled by the *Otherwise* statement in SDL are actually well defined signals, rather than improper defined (or undefined) ASPs/PDUs.

Since the atomic events SDL handles are signals, which have to be defined (declared) before use, improper defined or undefined object (or signals) cannot exist in SDL (SUT). These objects cannot be sent into an SDL system from its environment either. Therefore, this type of undefined-object handling mechanism is not needed in SIMPL-T.

5) TTCN structure types and accessing components of complex types are not needed in SIMPL-T

TTCN structure types are used to substructure ASPs, PDUs, CMs and other structure types. TTCN allows the use of individual components of complex types, such as a single PDU field, or an individual *BIT* in a *BITSTRING*, as operands in expressions, as the left hand side of an assignment, or references in a *SEND* or *RECEIVE* statement.

Since the atomic events SDL handles are signals rather than bit-strings of ASP/PDU as in TTCN, SDL does not handle structure types or access a component of a date type. Since these issues do not appear in SDL specifications, as its test language, SIMPL-T does not need to handle them either.

In summary, SIMPL-T is a more suitable language than TTCN to test SDL specifications.

6.3. Industrial Applicability

During the co-operation with SOLINET[SOLINET], the theoretical ideas of SIMPL-T were refined, developed and incorporated as new features in the SAFIRE tool. These features were then used in the *Traffic Controller* application.

Since the work on the *Traffic Controller* was carried out in parallel to the further development of SAFIRE, and the software versions used were often pre-release status, in effect the Reference Specification was the first test project for SAFIRE, and as a result, problems in the software were found, reported and fixed during several iterations of development.

The successful implementation and validation of the *Traffic Controller* in the SAFIRE environment confirmed the theoretical basis of SIMPL-T, it also confirmed that SIMPL-T can be incorporated into the development of the commercial tool SAFIRE.

6.4. Assessing Support for Important Test Concepts

6.4.1. Ordering Problem in Test Scenarios

Ordering Problem in this thesis means the situation that in a test case, two or more signals can arrive in arbitrary order and the order is irrelevant, but the test language does not have a mechanism to specify this. Probert[Prob+99] and Li[Li+04] discussed this problem in more detail.

TTCN can solve the *Ordering Problem* by explicitly providing a number of alternative events. However, when the number of events increases, the number of possibilities increases exponentially[Prob+99], which is called *Event Explosion*. TTCN-3 has solved this problem, but we address it as well here, because earlier versions of TTCN, still in use, did not.

SIMPL-T solves the *Ordering Problem* in a way closer to SDL by using the *save* construct inherited from SDL. In SDL, every process has an FIFO input queue. Signals in the input queue are consumed in the arriving order. By default, when a signal in the front of the queue is not expected in the current state, it is removed and disregarded. When the *save* construct is used, the signal associated with the *save* is saved in the queue until next state. This solved the *Ordering Problem* and does not cause *Event Explosion*.

However, during the *Task Force* discussions, many SDL experts consider *save* to be an inelegant solution because it reorders signals with very limited transparency[Li+04]. Therefore, it is not included in the *SDL Subset*.

The author of this thesis proposed to the *SDL Task Force* a new notion of *Combined Event[Li+04]*. A *Combined Event* consists of several events that can happen in any order.

This new construct is not only useful in SIMPL-T, but also an elegant solution for the *Ordering Problem* in SDL specifications. It will be discussed further in the SAM'04 workshop[SAM04] and may be also added as a new SDL construct.

6.4.2. Concurrency

TTCN, since version TTCN-2, supports concurrency through Main Test Component(MTC) and Parallel Test Components(PTCs) configuration. The MTC creates, starts, and stops the PTCs, and the PTCs run in parallel[ETS2003][Grab+03].

SDL supports concurrency by nature. The behavior of an SDL system is the joint behavior of all the process instances contained in the system, and all process instances exist in parallel with equal rights. A process can create another process, but the offspring and its parent have equal rights. There is no hierarchy association among process instances. All processes and process instances work autonomously and concurrently[ITUZ100].

SIMPL-T can also support concurrency easily. A SIMPL-T test case has the characteristics similar to an SDL process. Similar to TTCN MTC/PTC mechanism, a SIMPL-T test case (referred to as *main process*) can create other processes, and these created processes can work concurrently with their parent. They can stop themselves or be stopped by the main process. The verdicts assigned in these processes should be preliminary results and only the main process can assign final verdict.

Therefore, SIMPL-T can naturally support concurrency and no extensions are needed.

6.4.3. Race Condition

Race Condition in this thesis describes the situation in which two or more events start at the same time, and the order of their happening causes different outcomes. Therefore, the result triggered by these events is out of the control of the user.

Race Conditions exist in all real-time systems. A typical example is: A and B both call C at exactly the same time. In the users' perspective, this situation is indeed a race condition, because it is out of the users' control which one (A or B) will be connected to C.

However, in the perspective of SDL specifications, this situation is not a *Race Condition* any more. Since an SDL process has only one input queue, two signals cannot enter it at the same time regardless of the times they were sent. If every possible order of their arrival is considered in the SDL specification, all the possible results can be specified.

When SDL specifications are tested, the situation is more complicated. Two types of *Race Conditions* may occur: (1) Two (or more) external events race; (2) An external event(s) races against an internal action(s).

(1) Two (or more) external events race

SIMPL-T can handle this type of *Race Condition* the same way as SDL does. With all the possible orders of the race events, if each case is covered by a test case, this type of *Race Condition* can be easily tested.

(2) An external event(s) races against an internal action(s)

A good example of this type of *Race Condition* is the test case *XY_Transition_Defer* we discussed in the case study in Chapter 5. In this example, at the moment the lights are going to change, traffic arrives to potentially interfere with the change. Will the lights change or not?

To specify this situation in an SDL specification is simply to consider the two possibilities that *traffic* arrives both before and after the lights changed.

However, to test it is not so intuitive since the tester cannot see what is going on inside the SUT and cannot easily control which moment the *traffic* signal arrives in the SUT. Although SDL can ignore the process time and transmission time in a specification, SIMPL-T cannot do so. And these time durations are not predictable.

Since SIMPL-T tests SDL specifications rather than implementations, and timing concepts in SDL are abstract, time durations in the SUT can be reconfigured for testing purposes. As long as all the related timers are set according to the same proportions as in the original SDL specification, the system interactions being tested should be the same.

To control the arrival time of a stimulus, the related timers can be set long enough so that the SUT acts in slow motion. For example, if the *traffic* signal should arrive while the light is yellow and the normal time duration of yellow is only 10 seconds, then it would be difficult to make sure the *traffic* signal arrives during this time. SIMPL-T allows this time duration reconfigured, to 2 minutes for example, for testing purpose, so that the *traffic* signal can arrive on time.

In summary, SIMPL-T can test *Race Conditions* in SDL specifications.

6.5. Overall Assessment of Our Approach and the SIMPL-T language

This thesis starting from what is required in a test language defined a new test language SIMPL-T based on existing SDL features and concepts. It used a case study to demonstrate how this new language can be used by testers and tool developers. This language has its strengths and limitations, which are summarized in Table 6.

Table 6 The Strengths and Limitations of SIMPL-T Comparing to TTCN

+ Strength - Weakness = Same / Not needed

Number	Description	SIMPL-T	TTCN
1	Easy to Learn for SDL Users	+	-
2	Simplicity of Test Suites	+	-
3	Support Reusability -- Efficiency and Reliability	+	-
4	Tool Support	+	-
5	Handle Ordering Problem	+	-
6	Handle Race Condition	=	=
7	Handle Concurrency	=	=
8	Scalability	-	+
9	Number of Available Features	-	+
10	Support testing non-SDL Specifications	-	+
11	Undefined-object (Exception) Handling	/	+
12	Structure Type Support & Access Components of Complex Types	/	+

Strengths

1. *It is easy to learn and use for SDL users*

   Since it is defined as minimal extensions to SDL, its features and concepts are consistent to SDL. It should take almost no extra effort for SDL users to learn SIMPL-T.

2. *It is much simpler than TTCN*

   Since TTCN has a much wider scope and more general purposes than SIMPL-T, while SIMPL-T is defined specifically for testing SDL specifications, TTCN test suite is much more complicated than SIMPL-T test suite for testing SDL specifications.

3. *Data sharing and reusability make it more efficient and reliable than TTCN*

   Since SIMPL-T and SDL are in the same family, signals, user-defined data types and other components defined in the SDL specifications can be reused in the SIMPL-T test suites, but they cannot be reused in TTCN test suites. Redefining these can be tedious, time consuming and error-prone.

4. *SIMPL-T is confirmed it can be incorporated into test tools*

   The simplicity of SIMPL-T makes it easy to be supported by tools. The SAFIRE environment was successfully developed in parallel with the development of the language SIMPL-T. In contrary, TTCN took much longer before full support tools were developed. While SDL-2000 has been standardized for years, there is still no any commercial tool fully supporting it.

5. *SIMPL-T can handle the Ordering Problem well*

   For details see section 6.4.

6. *SIMPL-T can handle Race Conditions well*

   For details see section 6.4.

7. *SIMPL-T can handle Concurrency well*

   For details see section 6.4.

Limitations

8. *Scalability*

   SIMPL-T is defined with *SDL Subset* in mind. Therefore, it is intended for relatively small projects.

9. *Number of Available Features*

   SIMPL-T is defined as a **simple test language** with only essential functionality. Therefore, many otherwise useful features, such as the PICT/PIX auto test selection mechanism, are not included.

10. *Support for Testing Non-SDL Specifications*

    SIMPL-T is defined to test SDL specifications. It cannot be used directly to test specifications written in other languages.

Other Important Issues

11. *TTCN Undefined-object(Exception) handling mechanism is not needed in SIMPL-T*

    Since the atomic events SDL handles are signals, which have to be defined (declared) before used, the improper defined or undefined object (or signals) cannot exist in SDL. And these objects cannot be sent into an SDL system from its environment either. Therefore, the undefined-object handling mechanism is not needed in SIMPL-T.

12. *TTCN structure types and accessing components of complex types are not needed in SIMPL-T*

    Since the atomic events SDL handles are signals rather than bit-strings of ASP/PDU as in TTCN, SDL does not handle structure types or access a component of a date type. Therefore, as its test language, SIMPL-T does not need to handle them.

Summary

This section summarized the strengths and weaknesses of SIMPL-T as a test specification language. In the next and final chapter, we conclude and suggest areas for future work.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.1. Summary

This thesis defined a new language SIMPL-T for testing SDL specifications. It was defined using existing SDL features with minimal, but necessary extensions.

This thesis first derived the key requirements of a test language by analyzing international conformance testing standards[ISO9646][ETS2003][ITUZ500]. It then mapped these requirements to SDL features and suggested how these SDL features can be used in testing.

The SDL language elements useful for test specifications are only part of the SDL language and correspond closely to the subset identified by the *SDL Task Force[SDLTF]*, which defines "the simplest, useful subset of SDL and necessary extensions for test requirements".

However, since it was designed to be a requirements specification language, SDL does not have all the concepts and features needed for a test language. In particular, it does not support three fundamental test requirements, namely, *management and organization of tests, checking responses, and assigning verdicts*. Corresponding TTCN semantics were

analyzed, and incorporated as extensions to SDL to complete the definition of the semantics and syntax of SIMPL-T.

Finally a case study was presented of a realistic application – a traffic controller. It was specified using SDL and tested using SIMPL-T in SAFIRE environment. It demonstrated how SIMPL-T can be used by testers and how easy it is. It confirmed the feasibility of the theoretical concepts of the language. It also confirmed its feasibility for tool development.

7.2. Contributions of Thesis

In this thesis, we:

- Identified, derived and justified the key requirements of a test language.
- Identified what SDL concepts and features can be used for testing.
- Suggested how SDL features can be used for testing.
- Analyzed TTCN as an alternative test language and how the fundamental concepts can be redefined in SIMPL-T.
- Defined extensions to SDL to complete the definition of the semantics and syntax of SIMPL-T.
- Assessed how SIMPL-T addresses important test issues, namely *Race Condition, Concurrency*, and *Ordering Problem.*
- Proposed a new construct *combined event* in SDL as a solution for *Ordering Problem.*
- Presented a significant case study confirming these concepts with a realistic application.

- Confirmed, in cooperation with an industrial partner, SOLINET, the theoretical concepts of SIMPL-T by the successful development of an SDL-specification test tool SAFIRE.

SIMPL-T was not defined as a replacement for SDL-2000 in the domain of requirements specification, nor for TTCN-3 in the domain of validation. It was defined as a simple and easy to use language only for testing SDL specifications. It leads to a number of possible benefits:

- SIMPL-T is easier to learn than a dedicated test language, allowing users to quickly become familiar with it and master it. This has potential to generate wider interest in testing.
- Together with the *SDL Subset* being defined by the *SDL Task Force*, SIMPL-T makes it easier to write SDL specifications and validate them, as compared to SDL and TTCN. This can generate wider interest in SDL.
- SIMPL-T is a simple basis for tool manufacturers to build on, with potential for lower development costs and therefore lower cost to users, thus also leading to wider interest in development of support tools.

This iterative pattern of benefits, namely, the cycle of lower cost, leading to wider interest, leading to lower cost has already been seen in the widespread acceptance of Java and Linux, for example.

Just as TTCN-3 GFT is a standardized MSC based presentation format, SIMPL-T can also be used as an SDL-like state based presentation format of TTCN-3.

7.3. Future Work

The work of this thesis has been submitted to the *SDL Task Force[SDLTF]* for inclusion in its document, defining "the simplest, useful subset of SDL and necessary extensions for test requirements".

This is the basis of an ongoing discussion in the *SDL task force*, which is reviewing the contributions and will present the results to the SDL Forum[SDLFO] at the SAM'04 workshop[SAM04] in Ottawa.

Although this thesis has defined the semantics and syntax of SIMPL-T, only the essential functions were defined to make it a simple testing language. For relatively large projects, some extensions may be needed to make it easier to use.

Some concepts, such as concurrency were discussed in this thesis. More work may be needed in this area. A new concept of *combined event* (Section 6.4.1.) was proposed in this thesis. However, because it will change SDL itself rather than as extension, confirmations are needed from the SDL society before going into more details. Therefore, it has to be left as future work.

REFERENCES

[Alhir99] S. S. Alhir, "Understanding the Unified Modeling Language (UML)", Published in "Methods & Tools" (April 1999) - An international software engineering digital newsletter published by Marting & Associates.

[Amyot01] D. Amyot, "Specification and Validation of Telecommunications Systems with Use Case Maps and LOTOS", PHD Thesis, University of Ottawa (2001)

[Amyot03] D. Amyot, "Introduction to the User Requirements Notation: Learning by Example", Communication Networks, Volume 42, Issue 3 (June 2003) pp. 285-301.

[Bauer01] N. Bauer, "Deployment of SDL Systems Using UML", Proceedings of the 10th International SDL Forum, Copenhagen, Denmark (June, 2001), pp.107-122.

[Bhas+01] R. G. Bhaskar, A. Vimal (Motorola India Electronics Ltd), "SDL Based Test Automation for Real Time Systems Testing", Paper Presentations of the 3rd Annual International Software Testing Conference, India (2001), available at: http://www.softwaredioxide.com/Channels/events/testing2001/Presentations/Bhaskar_motorola.pdf

[Bjor03] M. Björkander, C. Kobryn, "Architecting Systems with UML 2.0", IEEE Software (July/August, 2003), pp.57-61.

[Bolo+87] T. Bolognesi, E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems 14 (1) (1987) pp. 25-59.

[Buhr98] R. J. A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems", IEEE Transactions on Software Engineering. Vol. 24, No. 12, (December, 1998), pp.1131-1155.

[DaV] www.DaVinci-Communications.com

[Ells+97] J. Ellsberger, D. Hogrefe, A. Sarma, "SDL Formal Object-oriented Language for Communication Systems", Prentice Hall Europe, 1997.

[Erik+98] H. E. Eriksson, M. Penker, "UML Toolkit", John Wiley & Sons, Inc., 1998.

[ETS2003] ETSI. "Methods for Testing and Specification; the Testing and Test Control Notation version 3 (TTCN-3); Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI)", 2003. Available at http://www.etsi.org/

[ETS9306] ETSI, "Methods for Testing and Specification (MTS); Semantical relationship between SDL and TTCN, A common semantics representation", June, 1993

[ETSEG9902] ETSI, EG 201 383 V1.1.1 (1999-02), "Methods for Testing and Specification (MTS); Use of SDL in ETSI deliverables; Guidelines for facilitating validation and the development of conformance tests"

[ETSEG9905] ETSI, EG 201 015 V1.2.1 (1999-05): "Methods for Testing and Specification (MTS); Specification of protocols and services; Validation methodology for standards using Specification and Description Language (SDL); Handbook".

[ETSETR95] ETSI, ETR 184 (1995): "Methods for Testing and Specification (MTS); Overview of validation techniques for European Telecommunication Standards (ETSs) containing SDL".

[ETSTR9807] ETSI, TR 101 279 V1.1.1 (1998-07), "Experiences of the application of SDL and CATG tools for the development of Abstract Test Suites (ATSs)"

[ETSTR9905] ETSI, TR 101 680 V1.1.1 (1999-05), "Methods for Testing and Specification (MTS): A harmonized integration of ASN.1, TTCN and SDL"

[ETSTR9905a] ETSI. TR 101 666 V1.0.0. (1999-05), "Information technology - Open Systems Interconnection - Conformance testing methodology and framework; The Tree and Tabular Combined Notation (TTCN) (Ed. 2++)"

[Grab+03] J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, A. Wiles and C. Willcock, "An Introduction to the Testing and Test Control Notation (TTCN-3)", Computer Networks, Volume 42, Issue 3 (2003), pp. 375-403.

[Haugen00] O. Haugen, "MSC-2000: Interacting with the Future", Telektronik Journal 4.2000 (Languages for Telecommunications Applications) pp. 54-61.

[He+03] Y. He, D. Amyot, A. Williams, "Synthesizing SDL from Use Case Maps: An Experiment", Proceedings of 11th International SDL Forum, Stuttgart, Germany (July, 2003), pp.117-136.

[Hydb01] O. Hydbom. "Notations That May Help to Minimize H/W-S/W Integration Pain", UK Design Forum (March, 2001)

[IEC] International Engineering Consortium (http://www.iec.org/online/tutorials/sdl/)

[ISO8807] ISO (1989), Information Processing Systems, Open Systems Interconnection, LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, IS 8807, Geneva.

[ISO9646] ISO/IEC 9646 (1994): "Information technology – Open Systems Interconnection – Conformance testing methodology and framework".

[ITUSG17] ITU-T Study Group 17 website:
http://www.itu.int/ITUT/studygroups/com17/index.asp

[ITUSG7] ITU-T Study Group 7: "The Evolution of TTCN",
available at: http://www.itu.int/ITU-T/studygroups/com07/ttcn.html

[ITUX680] ITU-T Recommendation X.680 (2002) | ISO/IEC 8824-1:2002: "Abstract Syntax Notation One (ASN.1), Specification of Basic Notation"

[ITUZ100] ITU-T, Rec. Z/100, "Specification and description language (SDL)", 2002

[ITUZ109] ITU-T Recommendation Z.109 (1999), "SDL Combined with UML".

[ITUZ120] ITU-T Recommendation Z.120 (11/1999): "Message Sequence Chart (MSC)".

[ITUZ150] ITU-T Recommendation Z.150 (2003), "User Requirements Notation (URN) – Language Requirements and Framework".

[ITUZ152] ITU-T, URN Focus Group: Draft Rec. Z.152 (2002) "UCM: Use Case Map Notation (UCM)", Geneva, Switzerland.

[ITUZ500] ITU-T Recommendation Z.500 (1997), "Framework on Formal Methods in Conformance Testing".

[Knig93] K. G. Knightson, "OSI Protocol Conformance Testing", McGraw-Hill, Inc., 1993.

[KobrUML01] C. Kobryn, "UML 2001: A Standardization Odyssey", Communications of the ACM, vol. 42, no. 10, (October, 1999), pp.29-37.

[KobrUML20] C. Kobryn, "Designing a More Agile UML 2.0", Telelogic Publications, available at: http://www.telelogic.com/news/publications/uml_models/index.cfm

[Leth+01] T. Lethbridge, R. Laganiere, "Object-Oriented Software Engineering", McGraw-Hill Education, 2001.

[Li+04] Qing Li, A. Prinz, W. Skelton and A. Yiannakoulias, "Back to the Basics", Accepted by the IFIP SAM'04 Workshop, Ottawa, Canada (June, 2004)

[Li+04a] Qing Li, R. Probert, W. Skelton and Y. Xu, "SIMPL-T: A Simple Language for Testing SDL Specifications", Accepted by the IFIP SAM'04 Workshop, Ottawa, Canada (June, 2004)

[Lopez+03] J. Lopez, J.J. Ortega, J.M. Troya, "Applying SDL to Formal Analysis of Security Systems", Proceedings of the 11th International SDL Forum, Stuttgart, Germany (July 2003), pp.300-316.

[Mans+01] N. Mansurov and R. Probert, "A Scenario-based Approach to Evolution of Telecommunications Software", IEEE Communications Magazine, Oct 2001, pp. 94-100, available at: www.ispras.ru/groups/case/downloads/scenario_based.pdf

[Mans+01a] N. Mansurov and R. Probert, "Improving time-to-market using SDL tools and techniques", Computer Networks, 35 (2001), pp. 667-691.

[Mans+99] N. Mansurov, R. Probert, "Dynamic scenario-based approach to re-engineering of legacy telecommunication software", Proceedings of the 9th SDL Forum, Montreal, Canada (June 1999) pp. 325-340.

[OMG03] OMG Unified Modeling Language Specification, Version 1.5, March, 2003

[Prob02] R. Probert, "Opportunities for Incorporating Formal Validation Methods in Industrial Software Development", (July 4, 2002), Systems & Software Engineering

Research Group Seminars, London South Bank University, UK, available at: www.cafm.sbu.ac.uk/seminars/sse/SDLMSCTTCNprobert.ppt

[Prob82] R. Probert, "Grey-box (design based) testing techniques", Proceedings of 15th Hawaii International Conference on System Science (1982), pp. 94-102.

[Prob+01] R. Probert, H. Ural, A. Williams, "Rapid generation of functional tests using MSCs, SDL and TTCN," Computer Communications ,Vol. 24, No. 3-4, (February 15, 2001), pp. 374-393.

[Prob+91] R. Probert, K. Saleh, "Synthesis of communications protocols: survey and assessment", IEE Transactions on Computers 40 (4) (1991) pp. 468-476.

[Prob+92] R. Probert, O. Monkewich, "TTCN: the international notation for specifying tests for communications systems", Computer Networks ISDN Syst. 23 (5) (1992) pp. 417-438.

[Prob+99] R. Probert, A. Williams, "Fast Functional Test Generation using an SDL model," in Proceedings of the 12th annual International Workshop on the Testing of Communicating Systems (IWTCS '99), Budapest Hungary, September 1999, pp. 299-315.

[Prob+99a] R. Probert, K. Saleh, W. Li, "High-yield requirements capture for electronic commerce software" Proceedings of the International Symposium on Electronic Commerce, and the 2nd International Workshop on Technological Challenges of Electronic Commerce, China, (1999).

[Saleh+99] K. Saleh, R. Probert, K. Al-Saqabi, "Recovery of CFSM-based protocol and service design from protocol execution traces", Information and Software Technology, 41(1999), pp. 839-852.

[SAM04] SAM'04 workshop: http://www.site.uottawa.ca/sam04/cfp.html

[Schu+02] S. Schulz and T. Vassiliou-Gioles, "Implementation of TTCN-3 Test Systems using the TRI", IFIP 14th International Conference on Testing of Communicating Systems - TestCom 2002 (March 2002).
available at: www.testingtech.de/technology/TestCom2002-TRI.pdf.

[SDLCon11] 11th SDL Forum Conference: http://www.sdl-forum.org/Events/SDL11.htm

[SDLDC03] SDL'03 Design Contest: http://www.solinet.com/sdl_design_contest.htm

[SDLFO] SDL FORUM: http://www.sdl-forum.org

[SDLTF] SDL Task Force: http://www.sdl-task-force.org/

[SOLINET] SOLINET: http://www.solinet.com

[Stir] http://www.cs.stir.ac.uk/~kjt/research/well/

[Telelogic] Telelogic: http://www.telelogic.com

[TGeni] http://www-lor.int-evry.fr/~sdl97/TestGen-SDL/

[Tret+92] J. Tretmans, "A Formal Approach to Conformance Testing", PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[TSWP] Testing Standards Working Party, "Glossary of Software Testing terms" (http://www.testingstandards.co.uk/living_glossary.htm)

[TTech] www.testingtech.de

[UCM] UCM website: http://www.usecasemaps.org/index.shtml

[Wiki] Wikipedia website: http://en2.wikipedia.org/wiki

[Wiles] A. Wiles, "The Tree and The Tabular Combined Notation – A Tutorial", available at: http://www.etsi.org/ptcc/ptccdownloads.htm

[Wvong98] R. Wvong, "A New Methodology for OSI Conformance Testing Based on Trace Analysis", Master's thesis, University of British Columbia, (1988)

APPENDIX A: SDL TASK FORCE

A.1. Introduction

The SDL Task Force is a group of technical specialists working to identify the simplest, useful subset of SDL.

The SDL Task Force was formed during the SDL'03 Conference following a series of comments from participants about how complex SDL had become, and has been recognized by the Annual General Meeting of the SDL Forum Society.

At the SDL Forum SAM Workshop held in Wales 2002, it had already been discussed that there was a need to go "back to basics". As a result of the repeated interest, a task force committee was formed by interested society members. A corresponding proposal "to define the simplest, useful SDL subset" was submitted to the SDL Forum society and approved at the Annual General Meeting by 27 votes with 3 abstaining and no objections.

The scope of the task force also includes the necessary enhancements needed to address automated presentation, validation, ASN.1 support and methodology.

The SDL Task Force Committee is formally a consortium of the founding members, with editorial responsibility for the result. The task force itself is open for any member of the SDL Forum to join. Both the task force and its mandate are officially recognized by the SDL Forum society, which is itself officially recognized by the ITU-T.

The committee was intentionally limited to 5 to allow it to focus on its mandate. There are two commercial organizations (SOLINET & TELETEL), two universities (Athens, winner of the SDL'03 Design Contest, & Ottawa, the previous year's winner) and one SDL Forum board member (Andreas Prinz).

The universities are looking at the theoretical issues and case studies, the commercial organizations are handling the requirements and tools, the SDL Forum will ensure the task force stays within scope and prepare the way for standardization by the ITU-T.

A.2. Task Force Editorial Committee

- William Skelton, SOLINET (Lead)
- Vangelis Kollias, TELETEL
- Alkis Yiannakoulias, University of Athens
- Qing Li, University of Ottawa
- Andreas Prinz, DResearch (SDL Forum board member)

A.3. Proposal To AGM Assembly SDL '03

"Task force to identify the simplest, useful SDL subset"

Background

SDL originated as a documentation aid to help visualise state machines, both in terms of architecture and execution.

Although there is a strong interest in the subject, SDL has not found the wide acceptance in the developer community that other languages, such as C++ and Java, have enjoyed.

This may be related to the slow cycle of language evolution and standardisation, as well as the relatively specific areas of application. It may also be because many users still only use SDL for documentation purposes.

SDL 2000 was a serious attempt to catch up, but the tool suppliers have not been enthusiastic in supporting the wide-range of enhancements and some opinions can be heard that SDL is now too complex for many users to understand.

Proposal

Building on the suggestions originally raised at the SAM workshop last June, SOLINET, proposes the SDL Forum establishes a task force to identify the simplest, useful subset of SDL.

Scope

The task force should only address the issue of the SDL subset; the parallel evolution of SDL 2000 is out of scope. However, enhancements considered to be essential in making the subset useful should be considered. As a framework the task force will prioritise:

- the graphical representation, ensuring auto-layout is possible
- test capabilities, such as SDL based test scripts
- ASN.1 (1994) support, including encoding/decoding of PDUs
- associated methodology issues, such as maximum integration of tool chain.

Organization

The task force is open to participation from any SDL Forum member and will be run by a committee with editorial responsibility. The proposed committee is

- William Skelton, SOLINET (Lead)
- Vangelis Kollias, Teletel
- Alkis Yiannakoulias, University of Athens
- Qing Li, University of Ottawa

Reporting

The task force progress will be published on the Internet for interested parties to build on, independently of any pending standardization activities. An interim status report will be published after 6 months, and a final report will be presented at the next SAM workshop (June 2004).

APPENDIX B: TTCN STANDARD REFERENCE

The content of this appendix is from [ETSTR9905a]:

ETSI. TR 101 666 V1.0.0. (1999-05), "Information technology - Open Systems Interconnection - Conformance testing methodology and framework; The Tree and Tabular Combined Notation (TTCN) (Ed. 2++)"

B.1. TTCN Test Suite

9. TTCN test suite structure

9.1 Introduction

TTCN allows a test suite to be hierarchically structured in accordance with ISO/IEC 9646-1, 8.1. The components of this structure are:

a) Test Groups;
b) Test Cases;
c) Test Steps.

A TTCN test suite may be completely flat (i.e., have no structure) in which case there are no Test Groups.

TTCN allows the use of Test Step Groups and Default Groups, similar to the concept of Test Groups, in order to structure Test Steps and Defaults hierarchically. This hierarchical structure is optional.

9.2 Test Group References

TTCN supports a naming structure that shows a conceptual grouping of Test Cases. Test Groups can be nested. Test Cases can also be stand-alone (see ISO/IEC 9646-1, clause 8, figure 9). The Test Group References define the structure of the test suite.

9.3 Test Step Group References

Test steps may be explicitly identified in TTCN and used to structure Test Cases and other Test Steps. Alternatively Test Steps may be implicit within the behaviour description of a Test Case. Explicit Test Steps may be specified either

- locally within a Test Case or Test Step behaviour description; or
- globally within a Test Step Library, which may be hierarchically structured into Test Step Groups.

NOTE: For example, a preamble may consist of just a few statement lines within a behaviour description of the Test Case, in which case it is implicit. Alternatively, a preamble may be explicitly specified with its own behaviour description. If such an explicit preamble is only of use within one Test Case, then it may be specified locally within that Test Case, but if it is of use in several Test Cases then it should be specified in the Test Step Library.

Local Test Steps are identified simply by a tree identifier. Global Test Steps are identified by a Test Step identifier. Global Test Steps also have a Test Step Group Reference, which shows the position of a Test Step in the Test Step Library. The structure of the Test Step Library is independent of the structure of the test suite.

9.4 Default Group References

Default behaviours (if any) are located in a Default Library.

A Default Group Reference specifies the location of the Default in the Default Library, which may be hierarchically structured. The Default Library has no influence on the test suite structure itself.

9.5 Parts of a TTCN test suite

An ATS written in TTCN shall have the following four sections in the order indicated:

a) Suite Overview (see clause 10),
which contains the information needed for the general presentation and understanding of the test suite, such as test references and a description of its overall purpose;

b) Import Part (see 10.8),
which contains the declarations of the objects used in the test suite or module that are imported from a source object;

c) Declarations Part (see clause 11),
which contains the definitions or declarations of all the components that comprise the test suite (e.g., PCOs, Timers, ASPs, PDUs, and their parameters or fields);

d) Constraints Part (see clause 12, 13, 14),
which contains the declarations of values for the ASPs, PDUs, and their parameters used in the Dynamic Part. The constraints shall be specified using:

1) TTCN tables; or
2) the ASN.1 value notation; or
3) both TTCN tables and the ASN.1 value notation.

e) Dynamic Part (see clause 15),
which comprises three sections that contain tables specifying test behaviour expressed mainly in terms of the occurrence of ASPs or PDUs at PCOs. These sections are:

1) the Test Case dynamic behaviour descriptions;
2) a library containing Test Step dynamic behaviour descriptions (if any);
3) a library containing Default dynamic behaviour descriptions (if any).

11. Declarations Part

11.1 Introduction

The purpose of the declarations part of the ATS is to define and declare all the objects used in the test suite. The following objects of an ATS referenced from the overview part, the constraints part and the dynamic part shall have been declared in the declarations part. These objects are:

a) definitions:
   1) Test Suite Types (see 11.2.3);
   2) Test Suite operations (see 11.3.4);
b) parameterization and selection of Test Cases:
   1) Test Suite Parameters (see 11.4);
   2) Test Case Selection Expressions (see 11.5);
c) declarations/definitions:
   1) Test Suite Constants (see 11.6 and 11.7);
   2) Test Suite Variables (see 11.8.1);
   3) Test Case Variables (see 11.8.3);
   4) PCO types (see 11.9);
   5) PCOs (see 11.10);
   6) CPs (see 11.11);
   7) Timers (see 11.12);
   8) Test Components (see 11.13.1);
   9) Test Component Configurations (see 11.13.2);
   10) ASP types (see 11.14);
   11) PDU types (see 11.15);
   12) Encoding Rules (see 219);
   13) Encoding Variations (see 11.16.2);
   14) Invalid Field Encodings (see 11.16.3);
   15) CM types (see 11.17);
   16) Aliases (see 11.21).

B.2. Constraints for RECEIVE Events and Matching Mechanisms

12.6 Constraints for RECEIVE events

12.6.1 Matching values

If a constraint is to be used to construct the values of ASP parameters or PDU fields that a received ASP or PDU shall match, it shall contain only specific values evaluated as explained in 12.6.3, or special matching mechanisms where it is not desirable, or possible, to specify specific values. The matching mechanisms specify other ways of matching than "equal to a specific value".

An incoming ASP and/or PDU matches a constraint used in a RECEIVE event if, and only if, all the following conditions are met:

e) all the ASP parameters and/or PDU fields are of the type specified in the ASP and/or PDU definitions;
f) the value, alphabet and length satisfies any restriction associated with the type;
g) the ASP parameter and/or PDU field values correctly match those of the constraint;
h) for PDUs, the correct decoding of the PDU has taken place, taking into account applicable encoding rule defaults and overrides; if encoding rules other than those specified for the constraint have been used to encode the received PDU, then that received PDU will not match.

In the case of substructured ASPs and/or PDUs, either using Structured Types or ASN.1, the above rules shall apply to the fields of the substructure(s) recursively.

NOTE: If a RECEIVE event is qualified by a Boolean expression, then a successful match means that both the incoming ASP and/or PDU must match the constraint and that the qualifier must evaluate to TRUE.

12.6.2 Matching mechanisms

An overview of the supported matching mechanisms is shown in table 6, including the special symbols and the scope of their application. The left hand column of this table lists all the ASN.1 types and TTCN equivalent types to which these matching mechanisms apply. The matching mechanisms in the horizontal headings are arranged in four groups:

e) specific values;
f) special symbols that can be used instead of values;
g) special symbols that can be used inside values;
h) special symbols which describe attributes of values.

Some of the symbols may be used in combination, as detailed in the following clauses.

The shaded area in table 6 indicates the mechanisms that apply to both predefined TTCN and ASN.1 types.

Table 6: TTCN Matching Mechanisms

	VALUE	INSTEAD OF VALUE								INSIDE VALUE			ATTRIBUTES	
TYPE	Specific Value	Complement	Omit (-)	AnyValue (?)	AnyOrOmit (*)	ValueList	Range	SuperSet	SubSet	AnyOne (?)	AnyOrNone (*)	Permutation	Length	IfPresent
BOOLEAN	•	•	•	•	•	•								•
INTEGER	•	•	•	•	•	•	•							•
ENUMERATED	•	•	•	•	•	•								•
BITSTRING	•	•	•	•	•	•				•	•		•	•
OCTETSTRING	•	•	•	•	•	•				•	•		•	•
HEXSTRING	•	•	•	•	•	•				•	•		•	•
CHARSTRINGS	•	•	•	•	•	•				•	•		•	•
SEQUENCE	•	•	•	•	•	•								•
SEQUENCE OF	•	•	•	•	•	•				•	•	•	•	•
SET	•	•	•	•	•	•								•
SET OF	•	•	•	•	•	•		•	•	•	•		•	•
ANY	•	•	•	•	•	•								•
CHOICE	•	•	•	•	•	•								•
OBJECT ID	•	•	•	•	•	•								•

In a constraint specification, the matching mechanisms may replace values of single ASP parameters or PDU fields or even the entire contents of an ASP or PDU.

NOTE: When these matching mechanisms are used singly or in combination, many protocol restrictions can be specified in the constraints, thereby avoiding undesirable computation details in the behaviour part.

12.6.3 Specific Value

This is the basic matching mechanism. Specific values in constraints are expressions. Unless otherwise specified, a constraint ASP parameter or PDU field matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field has exactly the same value as the value to which the expression in the constraint evaluates.

Two values of a tabular ASP, PDU or Structured Type, or of ASN.1 SEQUENCE or SEQUENCE OF are considered the same if each of their parameters fields or elements match and are in the same order. For ASN.1 SET and SET OF types two values are the same if they have the same number of elements, and each element in one value matches exactly one element in the other value. The elements in a SET or SET OF type value need not be in the same order to match.

12.6.4 Instead of Value

12.6.4.1 Complement

Complement is an operation for matching that can be used on all values of all types. Complement is denoted by the keyword COMPLEMENT followed by a list of constraint values. Each constraint value in the list shall be of the type declared for the ASP parameter or PDU field in which the Complement mechanism is used.

A constraint ASP parameter or PDU field that uses Complement matches the corresponding ASP parameter or PDU field if and only if the incoming ASP parameter or PDU field does not match any of the values listed in the ValueList.

12.6.4.2 Omit

Omit is a special symbol for matching that can be used on values of all types, provided that the ASP parameter or PDU field is optional.

In ASN.1 constraints it is also possible to simply leave out an OPTIONAL ASP parameter or PDU field instead of using OMIT explicitly.

NOTE: In tabular constraints, all parameters, fields and elements are considered to be implicitly optional, and hence may be omitted using Omit. In ASN.1 constraints, parameters, fields and elements which are not explicitly marked as OPTIONAL in the type definition are mandatory and cannot be omitted without violating the type definition. If such a parameter, field or element needs to be omitted from a particular constraint, either another type needs to be defined in which that parameter, field or element is explicitly marked as OPTIONAL (perhaps by marking everything as OPTIONAL), or an Invalid Field Encoding needs to be applied to that parameter, field or element, with the effect of omitting it from the encoding.

In tabular constraints Omit shall be denoted by dash (-). In ASN.1 constraints Omit is denoted by OMIT. An Omit symbol in a constraint is used to indicate that an optional ASP parameter or PDU field shall be absent.

12.6.4.3 AnyValue

AnyValue is a special symbol for matching that can be used on values of all types. In both tabular and ASN.1 constraints AnyValue is denoted by "?".

A constraint ASP parameter or PDU field that uses AnyValue matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field evaluates to a single element of the specified type.

12.6.4.4 AnyOrOmit

AnyOrOmit is a special symbol for matching that can be used on values of all types, provided that the ASP parameter or PDU field is declared as optional. In both tabular and ASN.1 constraints AnyOrOmit is denoted by "*".

NOTE: The symbol "*" is used for both AnyOrOmit and AnyOrNone. Ambiguity in interpretation is resolved by the requirements in 12.6.4.4 and 12.6.5.2.

A constraint ASP parameter or PDU field that uses AnyOrOmit matches the corresponding incoming ASP parameter or PDU field if, and only if, either the incoming ASP parameter or PDU field evaluates to any element of the specified type, or if the incoming ASP parameter or PDU field is absent.

12.6.4.5 ValueList

ValueList can be used on values of all types. In both tabular and ASN.1 constraints. ValueLists are denoted by a parenthesized list of values separated by commas.

A constraint ASP parameter or PDU field that uses a ValueList matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field value matches any one of the values in the ValueList. Each value in the ValueList shall be of the type declared for the ASP parameter or PDU field in which the ValueList mechanism is used.

12.6.4.6 Range

Ranges shall be used only on values of INTEGER type. A range is denoted by two boundary values, separated by ".." or TO, enclosed by parentheses. A boundary value shall be either

a) INFINITY or -INFINITY;
b) an expression that evaluates to a specific INTEGER value.

The lower boundary shall be put on the left side of the ".." or TO, the upper boundary at the right side. The lower boundary shall be less than the upper boundary.

A constraint ASP parameter or PDU field that uses a Range matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field value is equal to one of the values in the Range.

12.6.4.7 SuperSet

SuperSet is an operation for matching that shall be used only on values of SET OF type. SuperSet shall be used only in ASN.1 constraints. SuperSet is denoted by SUPERSET.

A constraint ASP parameter or PDU field that uses SuperSet matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field contains at least all of the elements defined within the SuperSet, and may contain more. The argument of SuperSet shall be of the type declared for the ASP parameter or PDU field in which the SuperSet mechanism is used.

12.6.4.8 SubSet

SubSet is an operation for matching that can be used only on values of SET OF type. SubSet shall be used only in ASN.1 constraints. SubSet is denoted by SUBSET.

A constraint ASP parameter or PDU field that uses SubSet matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field contains only elements defined within the SubSet, and may contain less. The argument of SubSet shall be of the type declared for the ASP parameter or PDU field in which the SubSet mechanism is used.

12.6.5 Inside Values

12.6.5.1 AnyOne

AnyOne is a special symbol for matching that can be used within values of string types, SEQUENCE OF and SET OF. In both tabular and ASN.1 constraints AnyOne is denoted by "?".

Inside a string, SEQUENCE OF or SET OF a "?" in place of a single element means that any single element will be accepted. If the symbol "?" is needed within a CharacterString as a character, it shall be indicated by "\?". If the symbol "\" is needed within a CharacterString as a character, it shall be indicated by "\\".

12.6.5.2 AnyOrNone

AnyOrNone is a special symbol for matching that can be used within values of string types, SEQUENCE OF and SET OF. In both tabular and ASN.1 constraints AnyOrNone is denoted by "*".

If a "*" appears at the highest level inside a value of string type, SEQUENCE OF or SET OF, it shall be interpreted as AnyOrNone.

NOTE: This rule prevents the otherwise possible interpretation of "*" as AnyOrOmit that replaces an element inside the string, SEQUENCE OF or SET OF.

Inside a string, SEQUENCE OF or SET OF a "*" in place of a single element means that either none, or any number of consecutive elements will be accepted. The "*" symbol matches the longest sequence of elements possible, according to the pattern as specified by the symbols surrounding the "*". If the symbol

"*" is needed within a CharacterString as a character, it shall be indicated by "*". If the symbol "\" is needed within a CharacterString as a character, it shall be indicated by "\\".

12.6.5.3 Permutation

Permutation is an operation for matching that can be used only on values inside a value of SEQUENCE OF type. Permutation shall be used only in ASN.1 constraints. Permutation is denoted by PERMUTATION.

Permutation in place of a single element means that any series of elements is acceptable provided it contains the same elements as the value list in the Permutation, though possibly in a different order. If both Permutation and AnyOrNone are used inside a value, the AnyOrNone shall be evaluated first. Each element listed in Permutation shall be of the type declared inside the SEQUENCE OF type of the ASP parameter or PDU field.

12.6.6 Attributes of values

12.6.6.1 Length

Length is an operation for matching that can be used only as an attribute of the following mechanisms: Complement, AnyValue, AnyOrOmit, AnyOne, AnyOrNone, Permutation, SuperSet and SubSet. It can be used in conjunction with the IfPresent attribute.

In both tabular and ASN.1 constraints, length may be specified as an exact value or range in string values and SEQUENCE OF or SET OF values, according to 11.18. The units of length are to be interpreted according to table 5. The boundaries shall be denoted by expressions which resolves to specific non-negative INTEGER values. Alternatively, the keyword INFINITY can be used as a value for the upper boundary in order to indicate that there is no upper limit of length.

The length specifications defined for the ASP parameter or PDU field type in the Test Suite Type definitions shall not conflict with the length specifications in the ASP or PDU constraint, i.e., the set of strings defined by a length restriction in an ASP or PDU constraint shall be a true subset of the set of strings defined by the ASP or PDU definition.

A constraint ASP parameter or PDU field that uses Length as an attribute of a symbol matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field matches both the symbol and its associated attribute. The length attribute matches if the length of the incoming ASP parameter or PDU field is greater than or equal to the specified lower bound and less than or equal to the upper bound. In the case of a single length value the length attribute matches only if the length of the received ASP parameter or PDU field is exactly the specified value.

In the case of an omitted parameter, field or element, Length is always considered as matching. Hence, with Omit it is redundant and with AnyOrOmit and IfPresent it places a restriction on the incoming value, if any.

12.6.6.2 IfPresent

IfPresent is a special symbol for matching that can be used as an attribute of all the matching mechanisms, provided the type is declared as optional. In both tabular and ASN.1 constraints IfPresent is denoted by IF_PRESENT.

A constraint ASP parameter or PDU field that uses an IfPresent symbol as an attribute of another symbol matches the corresponding incoming ASP parameter or PDU field if, and only if, the incoming ASP parameter or PDU field matches the symbol, or if the incoming ASP parameter or PDU field is absent.

NOTE: The AnyOrOmit symbol (*) has exactly the same meaning as ? IF_PRESENT

B.3. VERDICT

15.17 Verdicts

15.17.1 Introduction

Entries in the verdict column in Dynamic Behaviour tables shall be either:

- a preliminary result, which shall be given in parentheses;
- or an explicit final verdict.

An entry, of either type, shall not occur on an empty line, or on the following TTCN statements:

a) an ATTACH construct;
b) a REPEAT construct;
c) a GOTO;
d) an IMPLICIT SEND.

NOTE: During Test Case execution, whenever an entry in a behaviour tree occurs for which there is a corresponding entry in the verdict column of the abstract Test Case, that verdict column information is intended to be recorded in the conformance log in such a way that it is associated with the record of that entry in the behaviour tree.

15.17.2 Preliminary results

A predefined variable called R, of the predefined type R_TYPE, is available to each Test Case to store any intermediate results. These values are predefined identifiers and as such are case sensitive.

R may be used wherever other Test Case Variables may be used, except that it shall not be used on the left-hand side of an assignment statement. Thus, it is a read-only variable, except for the changes to its value caused by entries in the verdict column (as specified below).

If a preliminary result is to be specified in the verdict column it shall be one of the following:

a) (P) or (PASS), meaning that some aspect of the test purpose has been achieved;
b) (I) or (INCONC), meaning that something has occurred which makes the Test Case inconclusive for some aspect of the test purpose;
c) (F) or (FAIL), meaning that a protocol error has occurred or that some aspect of the test purpose has resulted in failure.

NOTE 1: PASS or P, FAIL or F and INCONC or I are keywords that are used in the verdicts column only. The predefined identifiers pass, fail, inconc and none are values that represent the possible contents of the predefined variable R. These predefined identifiers are to be used for testing the variable R in behaviour lines only.

Whenever a preliminary result is recorded, because the corresponding entry in the behaviour tree is executed, then the value of the predefined Test Case Variable R shall be changed according to the following table:

Table 7: Calculation of the variable R

Current Value of R	Entry in verdict column		
	(PASS)	(INCONC)	(FAIL)
none	pass	inconc	fail
pass	pass	inconc	fail
inconc	inconc	inconc	fail
fail	fail	fail	fail

NOTE 2: Thus, the order of precedence (lower → higher) is: N, P, I, F. Even if R has value fail it can be useful to record a preliminary result of P or I in order to record in the conformance log that a P or I is appropriate for some aspect of the test purpose, despite the fact that this will not change the value of R.

15.17.3 Final verdict

If an explicit final verdict is to be specified in the verdict column, it shall be one of the following:

a) P or PASS, meaning that a pass verdict is to be recorded;
b) I or INCONC, meaning that an inconclusive verdict is to be recorded;
c) F or FAIL, meaning that a fail verdict is to be recorded;
d) the predefined variable R, meaning that the value of R is to be taken as the final verdict, unless the value of R is none in which case a test case error is recorded instead of a final verdict.

Table 8: Calculation of the final verdict R

Current Value of R	Entry in verdict column			
	PASS	INCONC	FAIL	R
none	pass	inconc	fail	*error*
pass	pass	inconc	fail	pass
inconc	*error*	inconc	fail	inconc
fail	*error*	*error*	fail	fail

Whenever, during execution of a Test Case, an explicit final verdict is specified, then this terminates the Test Case. For compliance with ISO/IEC 9646-2, an explicit final verdict should be specified only if the Test Case has returned to a suitable stable testing state (e.g., the idle testing state).

NOTE 1: The termination of the Test Case caused by the specification of an explicit final verdict is necessary, for example, if the stable state is reached in an attached Test Step when subsequent behaviour is specified in the calling tree.

If the leaf of the behaviour tree is reached without an explicit final verdict being specified, then the final verdict is determined as for case d) above (i.e., as if R had been put in the verdict column).

If an explicit final verdict other than R is to be recorded, then that verdict shall be compared with the value in R to determine whether or not they are consistent. If R is fail then a final verdict of PASS or INCONC shall be regarded as inconsistent; if R is inconc then a final verdict of PASS shall be regarded as inconsistent. If there is one of these inconsistencies, then it is a test case error.

NOTE 2: In such a case, "Test Case Error" should be recorded in the conformance log.

APPENDIX C: SEMANTICS OF SIMPL-T

C.1. Organization and Management of Tests

C.1.1. Test Suite

Test Suite is defined as a SIMPL-T construct. It contains two parts:

- *Declarations*
- *Behaviour*

C.1.2. Declarations

The *Declarations* part is concerned with the declaration of all the *Test Suite* components, including:

- *Gates and Signal List Declarations*
- *Variable Declarations*
- *Constant Declarations*
- *Timer Declarations*
- *Procedure Declarations*

- *Test Group Declarations*
- *Test Case Declarations*

These declarations can be implemented as they are normally done in SDL.

C.1.3. Behavior

The *Behaviour* part defines *Test Groups* and *Test Cases*.

C.1.4. Test Group

Test Group is defined as a SIMPL-T construct. It is simply a container which contains a block of *Test Cases*. It does not have any behavior by itself.

C.1.5. Test Case

Test Case is defined as a SIMPL-T construct. The behavior of a *Test Case* can be considered as a special SDL process. This process always stops itself (use SDL *STOP* construct) after a *final verdict* is formed, while normal SDL processes don't stop until the whole system is forced to terminate. No additional definition needs to be done regarding *Test Case behavior.*

C.1.6.Test Purpose

Test Purpose is defined as a SIMPL-T construct. Since it is only a textual description of the objectives of the test, its content doesn't influence the behavior of the test at all.

C.1.7. Explicit Test Step

Since an *Explicit Test Step* acts exactly like an SDL *procedure*, defining a new SIMPL-T construct for *Test Step* is not necessary. When repeated explicit test steps needed, an SDL *procedure* can be defined and called by the *Test Cases*.

C.1.8. Summary

A SIMPL-T *Test Suite* consists of nested *Test Groups* containing *Test Cases*, each with a defined *Test Purpose* and *Test Case behavior.*

C.2. Specifying Expected Gate a Response Arrives on

When an expected response is specified, the gate on which the signal arrives needs to be specified.

INPUT VIA is defined as a SIMPL-T construct. It is defined as an enhancement to SDL *INPUT* construct and used to specify on which *gate* or via which *channel* an expected response arrives.

Figure 53 "INPUT VIA" Construct

```
STATE S1;
  INPUT A VIA Gate1;
NEXTSTATE S2;
```

C.3. Input and Matching Mechanism

SIMPL-T *INPUT* construct is defined as (an enhancement to the SDL *INPUT* construct):

***INPUT* signal_name *(parameter_list) VIA* gate/channel**

The *parameter_list* can be either specific values or specific matching mechanisms. Each parameter in the *parameter_list* has a data-type. For simple data-types the following matching mechanisms apply:

- specific value – Variable, Constant, or Expressions
- a range of values - denoted by "Value1 : Value2"
- list of values - denoted by " (Value1, Value2, Value3......)"
- wildcard value, matching all values – denoted by "?"
- omitted value, for missing parameters – denoted by "-"
- General wildcard value, matching all values and missing parameters – denoted by "*"
- Default value – used if no value is specified

As parameters are optional in SDL, if the omitted value is expected ("-"), it specifies that the parameter is not allowed to be present in the signal. The general wildcard ("*") matches any value for the parameter, whether present or not.

If the parameter has a structured type, such as CHOICE or SEQUENCE, the ASN.1 value notation can be used to specify a specific value, range or list of values.

The wildcards can be used within the value notation for specific fields and the "omitted value" can be used for optional fields in a SEQUENCE, to specify that an optional field must be omitted.

C.4. Assigning and Handling of Verdicts

Assigning and handling of verdicts in SIMPL-T reuse all the concepts defined in TTCN. (See Appendix B.3.)

APPENDIX D: SYNTACTIC DEFINITION OF SIMPL-T

This appendix only lists the extensions defined. The reused SDL constructs can be found in[ITUZ100], which are not repeated here.

Table 7 defines the syntactic meta-notation used to specify the extended BNF grammar for SIMPL-T (henceforth called BNF):

Table 7 The SIMPL-T Syntactic Meta-notation

::=	is defined to be
abc xyz	abc followed by xyz
|	alternative
[abc]	0 or 1 instances of abc
(...)	textual grouping
abc	the non-terminal symbol abc
"abc"	a terminal symbol abc

D.1. Test Suite

```
Testsuite_Definition ::=    "TESTSUITE" TestsuiteName ";"
                            [ Gate_Definition ]
                            [ Testsuite_Component ]
                            "ENDTESTSUITE;"

Gate_Definition ::=   "GATE" GateName ";"
                      [ In_Signal_List ] ";"
                      [ Out_Signal_List ]";"

In_Signal_List ::=    Signal_Identifier
                      [ "," In_Signal_List ]

Out_Signal_List ::=   Signal_Identifier
                      [ "," Out_Signal_List ]

Testsuite_Component ::=     ([Signal_Definition]
                            [Signal_List_Definition]
                            [Asn1type_Definition]
                            [Timer_Definition]
                            [Variable_Definition]
                            [Synonym_Definition]
                            [Procedure_Definition]
                            [Test_Group_Definition]
                            [Test_Case_Definition] )
                            [Testsuite_Component ]
```

Test_Group_Definition ::= "TESTGROUP" TestGroupName ";"
Test_Case_Definition_List
"ENDTESTGROUP;"

Test_Case_Definition_List ::= (Test_Case_Definition | Test_Group_Definition)
[Test_Case_Definition_List]

Test_Case_Definition ::= "TESTCASE" TestCaseName ";"
"TESTPURPOSE" Test_Purpose_Description ";"
Behavior_Definition
"ENDTESTCASE;"

Note:

- **Testsuite_Definition** is defined parallel to **Process** definition as part of the **Block** definition in [ITUZ100], which is not repeated here.
- **Test_Purpose_Description** is simply a text description of the test purpose.
- **Behavior_Definition** is basically EFSM, which has been defined in the *Process* section in SDL[ITUZ100]. The test-specific behaviour of a state machine, such as verdict, is defined in *Verdict* section.

D.2. Input

(Replace *INPUT* defined in[ITUZ100])

Input_Definition ::= "INPUT" Input_Signal
"VIA" Origination_Name ";"

Origination_Name ::= Channel_Identifier | Gate_Identifier

Input_Signal ::= "*" | (Signal_Identifier [Parameter_List])

Parameter_List ::= "(" Parameter ["," Parameter] ")"

Parameter ::= " - "
| " ? "
| " * "
| Value_List
| Value_Range
| Empty

Empty ::= ""

Value_List ::= "(" Value ["," Value_List] ")"

Value_Range ::= Value " : " Value

Value ::= **Expression**

Note: Expression is defined in [ITUZ100].

D.3. Verdict

Textual (PR) Representation:

Verdict_Pre ::= "PRELIMINARY_VERDICT"
"(" Verdict ")"

Verdict_Fin ::= "FINAL_VERDICT"
"(" Verdict ")"

Verdict ::= "PASS"
| "INCONC"
| "FAIL"
| "ERROR"

Graphical (GR) Representation: