DUBLIN CITY UNIVERSITY
FACULTY OF ENGINEERING AND DESIGN

DESIGN AND IMPLEMENTATION OF A TTCN TO C TRANSLATOR

Author Kenneth Cunnıngham B Eng
Supervisor . Dr Tommy Curran
Date 29th July 1993

Thıs thesıs was submıtted to the school of electronıc engıneerıng of Dublın Cıty Unıversıty ın fulfilment of the requırement of Master of Engıneerıng degree

DECLARATION

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Engineering (M Eng) is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work

Signed [illegible]
Candidate

I D No 91700965

Date July 1993

Acknowledgements

I wish to take this opportunity to thank my supervisor, Dr Tommy Curran for his guidance and suggestions throughout this project Also the staff at NETC Eolas, in particular, Patrick Walsh and Des Keane for giving me the opportunity of working on the CTS II bis project which enabled me to get up to speed with TTCN and the conformance testing process in a practical manner Finally, I wish to thank my fellow post-graduates for their support over the past two years, and indeed for some the past six

Abstract

The conformance testing of a protocol implementation, may be logically divided into, the specification of the abstract test suite (ATS) from a formal description of the protocol, and the subsequent derivation of the executable test suite (ETS) from the ATS specification Our concern here is with the latter step, in particular, the automatic derivation of an ATS expressed in the Tree and Tabular Combined Notation (TTCN) to an executable C language equivalent This process is currently a manual one, and as a consequence is error prone, time consuming, often repetitive and not necessarily consistent To overcome these problems, there exists the real need for a computer aided, and if possible, fully automatic solution

This study describes the design and implementation of a fully working TTCN subset to C language translator, which takes a TTCN ATS and produces an equivalent ETS, with a minimal amount of manual intervention The methodology used is logically divided into three stages direct TTCN to C language mappings, implementation issues, including the generation of additional code to drive the above mappings, and test system implementation issues The system was tested using parts of an ETSI ISDN LAPD ATS and the results showed considerable time savings against a similar manual implementation

In conclusion, suggestions are provided to the further development of the TTCN to C translator system, and discussion is given to the application of this tool to a complete conformance testing system

Table of Contents

1. Introduction

The conformance testing of protocols is not concerned with performance or efficiency issues, but with the determination of the extent of adherence of protocol implementations, to the standards that they claim to implement The importance of conformance testing is paramount, since any vendor can implement a layer of a protocol, and claim conformance for their product The work group ISO TC97/SC21/WG 16-1 was formed to standardise all aspects of ISO protocol implementations The Tree and Tabular Combined Notation (TTCN) arose from this group, as the standardised abstract notation for the specification of abstract conformance test suites TTCN is now a fundamental component in the conformance testing process

The specification of abstract test suites (ATSs) in TTCN is, however, only half of the conformance testing process ATSs written in TTCN are complete protocol test specifications in their own right, but are written in a manner that is test system independent The ATS must be then transformed into an equivalent executable test suite (ETS), which involves a translation and an implementation of the ATS on a conformance test system, this step constitutes the second half of the conformance testing process, and at present is a manual one Since ATSs are voluminous specifications, the task of translating them manually is time consuming, repetitive and error prone Therefore, to maintain consistency in ATS translations, to reduce errors, and moreover, to reduce translation costs, there is a real need for automation

This study seeks to provide a solution to the automatic derivation of ETSs from their corresponding ATSs In particular, it will outline a methodology, and will implement a translator, to translate an ATS specified in TTCN to a C language equivalent

This is a relatively new area of research, with study only beginning in the late eighties, by work groups within Swedish Telecom and the NIST in the United States, to name but two To date, less than a handful of papers on the subject of a TTCN to C translator have been published Most development work seems to have been for in-house, as opposed to commercial, translators In early 1992, the international standardised version of TTCN became available, thus opening the way for standardised tools for developing ATSs in TTCN, one such tool is the ITEX-DE Current research into the development of TTCN to C utilities as part of many of these tools, is still under way

The approach taken in this study is to design a TTCN translator for the LAPD protocol of an integrated services digital network (ISDN) This target is merely a starting point for further study on the application of the translator to higher layer, and other, protocols The LAPD protocol provides a good basis for a wide spectrum of TTCN constructs The required input to the TTCN to C translator is a file of a standardised TTCN ATS in the machine processable format Output is in the form of C files This generated C code is readable, modular, reusable, and above all, executable All generated code is indented, and where possible maintains the structure of the original TTCN The system is built, such that extensions to the TTCN subset chosen to implement the translator, may be made with a minimum of programming overhead Where possible the translator is test system independent The actual implementation of communication routines, timers, etc is, however, DCT-S test system specific If a test system other than the DCT-S was used to implement these routines, then it is only the test system interface that would require modification What exists is a complete TTCN to executable language mapping via the C programming language

Chapter two introduces TTCN in both its forms, and discusses tools for TTCN ATS development The application of TTCN to the conformance testing process is discussed, and research into the application of formal description techniques (FDTs) to conformance testing, and their application to TTCN, is considered Chapter three discusses the derivation of ETSs from ATSs A methodology for automatically deriving ETSs, and its application to the TTCN to C translator, is described A model of the translator that remains the focus of the subsequent chapters is illustrated at the end of chapter three Chapter four details the TTCN to C mappings that are implemented in the translator, and a discussion of two tools that were useful in the development of the translator is given Chapter five focuses on the software needed to implement a working TTCN to C translator Further mappings that are not necessarily derivable from the ATS, but are nonetheless prerequisites for execution of the ATS, are also discussed Chapter six then focuses on the test system and the interface needed to communicate with it The TTCN to C translator modules are then pulled together, and the translator model introduced at the end of chapter three is expanded and discussed in the context of a conformance testing session Chapter seven concludes the thesis, and provides recommendations for further study in regard to this system, and indeed conformance testing as a whole

2. Tree and Tabular Combined Notation (TTCN)

2 1 An Overview

The Tree and Tabular Combined Notation (TTCN) is an informal test notation, standardised by the International Standards Organisation (ISO), for use in the precise specification of Open System Interconnect (OSI) abstract conformance test suites It is an informal test notation with clearly defined semantics TTCN differs from the more familiar formal description techniques (FDTs), having informally, as opposed to formally, defined semantics

The conformance testing methodology and framework ISO9646 [1] was developed and standardised over a period of eight years It is a five part document Part one defines conformance in the context of OSI, part two defines abstract test suite (ATS) specification, and part three defines TTCN The other parts are concerned with test realisation and test laboratory procedures TTCN began with the specification of the X 25 networking protocol, which proved that TTCN could be effectively used to specify complex conformance test suites, in accordance with the international standard

TTCN combines a tree notation with a tabular representation The tree notation is used to describe the events which can occur as alternatives to a previous event The tabular component is used to simplify the representation of all static elements such as data types, protocol data unit (PDU) and abstract service primitive (ASP) formats, timers, and the verdicts associated with test events, in the TTCN ATS The dynamic behaviour description addresses many important aspects of conformance testing such as, modularity support in terms of test cases, test steps, and default dynamic behaviour, sophisticated timer management etc The structure and syntax of TTCN is entirely directed towards describing desirable sequences of interactions between the entities involved in the conformance testing process

One might ask , "why the need for another test notation ?" The main argument favouring TTCN is that it's primary focus is on testing TTCN boasts a compact way of specifying dynamic protocol behaviour It is ideally suited to the active testing of reactive systems, i e the technique of putting a system under test (SUT) into a controlled environment and sending specific stimuli to it, aimed at checking specific aspects of the system TTCN provides a test notation, that standardises the representation of sequences of test events that make up test cases, and subsequently the representation of test cases within standardised test suites, in a way that is independent of test methods and protocol layers Moreover, TTCN reflects the testing methodology described in ISO9646

2 2 Abstract test suite specification using TTCN

An abstract conformance test suite consists of a number of test cases which test an implementation for conformance It is a top down hierarchical structure, the central component of which is the test case Test cases may be grouped with other test cases to form test groups Test groups may be subsequently grouped together to form larger test groups Test cases may be decomposed into test steps and these test steps into test events This structure is illustrated in figure 2 1 A test event might be the sending or receiving of a Protocol Data Unit (PDU) or possibly the starting of a timer Test steps aid in putting the SUT into a state, suitable for executing a test case and hence, returning it to a stable state once the test case has been executed To impose order within an ATS, test cases sharing a similar test purpose, initial testing state, or final testing state often form test groups

The TTCN ATS can be subdivided into the test suite overview, declarations, constraints, and dynamic behaviour sections The test suite overview provides all the information necessary for the general presentation and understanding of a particular test suite This overview is comprised of four tables, namely, the test suite structure table, the test case index table, the test step index table and the default index table The declarations part provides the definitions of all the components that comprise the ATS, i e abstract service primitives (ASPs), protocol data units (PDUs), timers etc The constraints part provides precise descriptions of all messages exchanged between the tester and the SUT Finally, the dynamic behaviour tables specify test behaviour in terms of messages exchanged The tables in this section are test case Dynamic behaviour tables, test step dynamic behaviour tables, and default dynamic behaviour tables

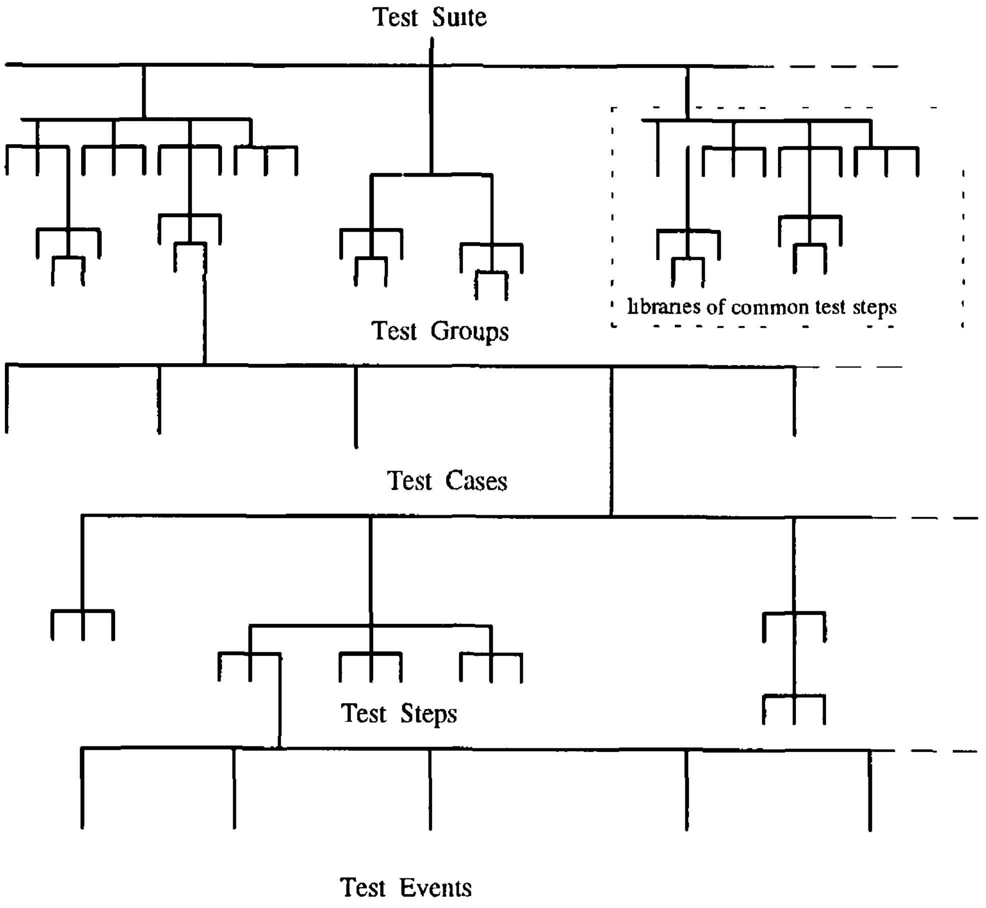

Figure 2 1 Hierarchical structure of an abstract test suite

The formulation of any test case begins with an identification of the test purpose i e what is the purpose of this test? To construct a meaningful test case, a well composed and ordered series of stimuli that will achieve this test purpose are chosen All responses of the system, desirable and undesirable, and their associated verdicts, should be anticipated Possible verdicts are pass, fail, inconclusive and test case error A pass verdict implies that the test purpose has been reached A fail verdict implies that an error has occurred An inconclusive verdict implies, that although no error has occurred, something happened during the test procedure, that prevented the test purpose from being reached Finally, a test case error verdict implies, that one of the following occurred no verdict got assigned, a pass verdict got subsequently assigned to a preliminary fail or inconclusive verdict, or an inconclusive verdict got assigned to a preliminary fail verdict Graphically, the result is a tree-like structure with branching to cope with possible alternative reactions from the testing entities, see figure 2 2 Indentation facilitates the occurrence of test events in a time increasing order Successive test events get indented once from the left, alternative events lie at the same levels of indentation Verdicts get assigned at the furthest points to the right along a particular branch either in the test body or in some adjoining subtree

The test case is generally composed of three phases, namely, the preamble, the test body, and the postamble The purpose of the preamble is to get the SUT into a state from whence the test case can be performed, the test body comprises the bulk of the test case and the verdict assignment, and the postamble returns the SUT to a former stable state, or to one where a subsequent test case may be performed A verification phase may also be present to ensure that the IUT is indeed in the post-transition state A typical transition test is illustrated below in figure 2 3

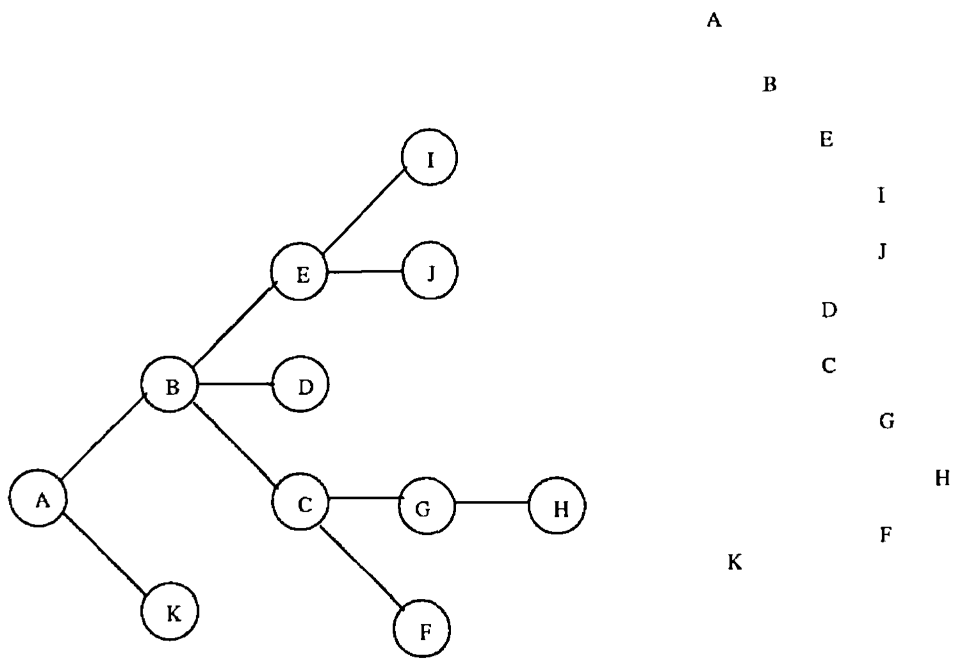

Figure 2 2 Branching in TTCN

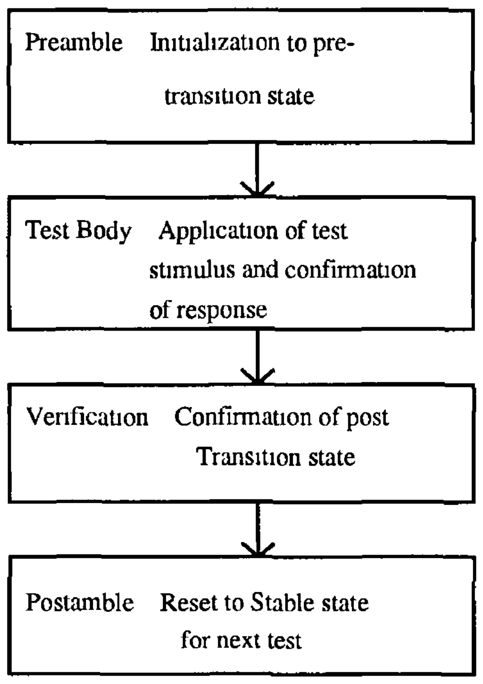
Preamble Initialization to pre-transition state
Test Body Application of test stimulus and confirmation of response
Verification Confirmation of post Transition state
Postamble Reset to Stable state for next test

Figure 2 3 Structure of a transition test case for conformance testing

2 3 TTCN.MP - the machine processable form of TTCN

The second version of TTCN is the machine processable form (TTCN MP), which serves as an exchange format for test suites In this linear form, TTCN is machine independent TTCN MP is defined using syntax productions, which have special keywords as terminal symbols, (all keywords are preceded by the string "$" symbol) These productions replace the fixed lines and boxes characteristic of the graphical format Entries to the boxes of the graphical format are mapped identically into the MP format Since TTCN is intended for information exchange, more uniform and efficient storage, and for electronic transfer, its syntax is necessarily formally defined Using some three hundred BNF (Backus Naur Form) productions, TTCN MP defines the TTCN syntax This BNF is intended to be directly processable by the LALR family of parsers, thus simplifying the task of automatically translating TTCN to other notations and languages The semantics of TTCN are operationally defined, using both natural language and pseudo-code descriptions These descriptions though given in an algorithmic format were not meant as a method of executing TTCN but rather as a means of explaining how the TTCN machine should operate

In summary, the arguments for a second TTCN format are essentially

- to provide a formal syntax for TTCN in BNF,
- to act as a transfer syntax by facilitating the electronic exchange of test suites,
- to ease in the derivation of ETSs from ATSs as test suites are generally complex and voluminous,
- to facilitate further machine processing

Some of the differences between the graphical and machine processable forms of TTCN are

- special tokens in TTCN MP serve as delimiters, where boxes and lines are used in GR

- where explicit indentation is used in MP, indentation is graphically represented in GR,
- page and line continuation and page and line numbering of the GR format have no significance in MP
- the MP format also contains an extra instance of the test suite identifier, thus facilitating test case identification in an automated way

Concluding therefore, both the GR and the MP forms of TTCN are semantically equivalent In other words, if an ATS is specified in the graphical format, in compliance with the international standard, then there exists a unique corresponding machine processable representation, with the same underlying syntax Two representations of an ATS are deemed equivalent, if and only if, they share identical operational semantics It must be stated, however, that the graphical ATS is the standardised test suite and, therefore, in the event of any conflict between the two representations, the graphical format takes precedence

2 4 TTCN editors

The emergence of TTCN as a standardised notation has prompted the development of software support tools to assist in the editing, distribution and translation of abstract test suites Since the tabular specifications cannot be readily developed on conventional editors, purpose-built TTCN editors are required This in itself is one of the major drawbacks of a graphical test notation A schematic form of editing, browsing etc is necessary, therefore, in the development of ATSs using TTCN

Several TTCN editors exist Three of these are CONTEST-TTCN [2], TTCN workbench [3] and ITEX-DE [4] All of these have similar capabilities, such as TTCN MP to TTCN GR translation and vice versa, syntax and semantics checking, and full test suite browsing facilities ITEX-DE the ITEX Development

Environment, was chosen by ISO as one of the standardised TTCN editors Its internal layout is illustrated in figure 2 4 ITEX-DE is a commercial editor, for use in the development and management of ATSs Over and above the features outlined above, the ITEX-DE has plans to include in later releases, both a translation capability, where TTCN could be automatically translated into several executable languages including C and Forth, and a validation facility, whereby TTCN dynamic behaviour could be validated against formal protocol specifications

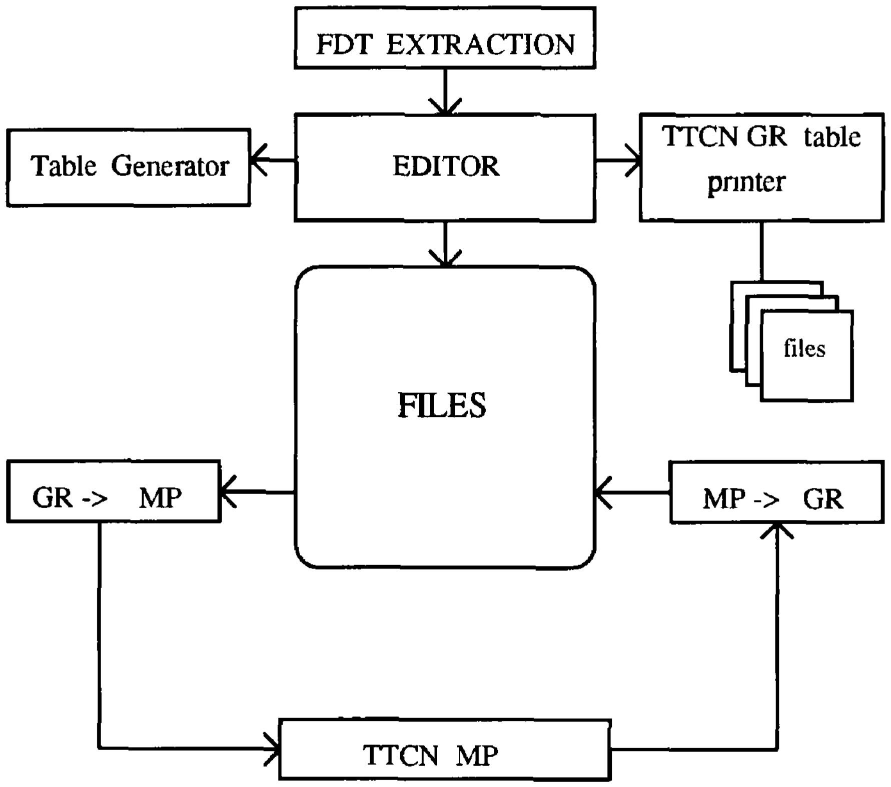

Figure 2 4 Internal structure of the ITEX-DE

2 5 TTCN and Formal Description Techniques (FDTs)

Standard definitions of many protocols are presently given in natural languages Apart from being verbose, these definitions may contain ambiguities and imprecisions Out of this came the need for formal description techniques (FDTs)

ISO and CCITT encourage the specification of communications protocols and services using FDTs Three such languages SDL (Specification and Description Language), LOTOS (Language of Temporal Ordering Specification) and Estelle (Extended State Transition Language) have been standardised ASN 1 (Abstract Syntax Notation One) aimed at protocol data structure definition, and TTCN for the specification of OSI conformance test suites, were also standardised FDTs provide powerful modelling techniques by combining the control and data aspects of systems They were intended for writing formal specifications for the OSI protocols and services, to be used during the protocol development process

The three formal languages SDL, LOTOS and Estelle, and the methodology and framework for conformance testing, developed separately There now exists the need for convergence in these two fields Research is currently going on in the application of FDTs to conformance testing By employing FDTs, to both describe the protocol and the dynamic test sequences, ambiguities in a protocol specification and misunderstandings among protocol developers and test designers may be eliminated, the effort of test designers reduced, and the automatic validation of protocols may be made possible

FDTs already play a central role in the conformance testing process, particularly in the specification of ATSs The specification of ATSs is presently manually derived from a formal description of a protocol i e Q 921 describes layer two of the LAPD protocol using SDLs ISO and CCITT are currently working on longer term objectives to apply formal methods to the testing of protocol implementations, which includes the automatic derivation of ATSs from formal protocol specifications TTCN is, nonetheless, simpler to implement and understand than FDTs and is more suitable for the direct abstract specification of tests by a human test specifier In developing formal approaches to conformance testing, it will be necessary to relate test descriptions in TTCN to formal specifications in FDTs It may be, therefore, necessary to extend TTCN to include some selected FDT capabilities currendy not supported At present the semantics of TTCN are informally expressed, there may now exist the need to formalise them It must be noted, however, that if TTCN is to hold its ground, in the advance of FDTs in the conformance testing field, then TTCN must retain its simplicity, practicality and precision

2 5 1 Automatic generation of abstract test suites using FDTs

Although certain methods exist for automatically developing abstract test suites, most test suites are still developed manually Methodologies and tools exist to automatically generate and execute test sequences, but these tools tend to be localised to one particular stage in the conformance testing process, be it basic inter-connection, capability, behaviour or conformance resolution testing, but never to all four The derivation of tests for complex protocols is cumbersome, since all of the functionality of the protocol must ultimately be checked Consequently, there is a real need for automation

TTCN is the universal language of choice for the specification of test suites since all standardised test suites have been, and currently are being, expressed in TTCN Tools now exist to translate FDTs into the TTCN notation Such tools will be discussed later on in this chapter

2 5.2 Validation of protocol behaviour using FDTs

This section is concerned with the validation of TTCN abstract test suites against their corresponding protocol specifications Since most ATSs are developed manually, one can expect them to contain errors These errors should ideally be detected and removed as early on in the conformance testing process as possible, i e before test case execution In particular, test case verdicts should be consistent with the protocol specifications

A problem exists in the difficulty of comparing a test case written in TTCN to a protocol specification written in some formal notation In order to automate such comparisons, it is necessary that first, the protocol be specified formally and secondly, that the language used to specify the test case i e TTCN, be comparable to the language used in the formal protocol specification For example, if the protocol specification is written in LOTOS then there must exist a means to relate TTCN to LOTOS Two possible approaches exist [5]

(i) The TTCN test case is first translated to an equivalent LOTOS specification (path 2), which may be then compared to the protocol specification (path 3)

(ii) The TTCN test case is executed (path 4) and a comparison is made of the test results with the protocol specification (path 5)

This configuration is illustrated below in figure 2 5

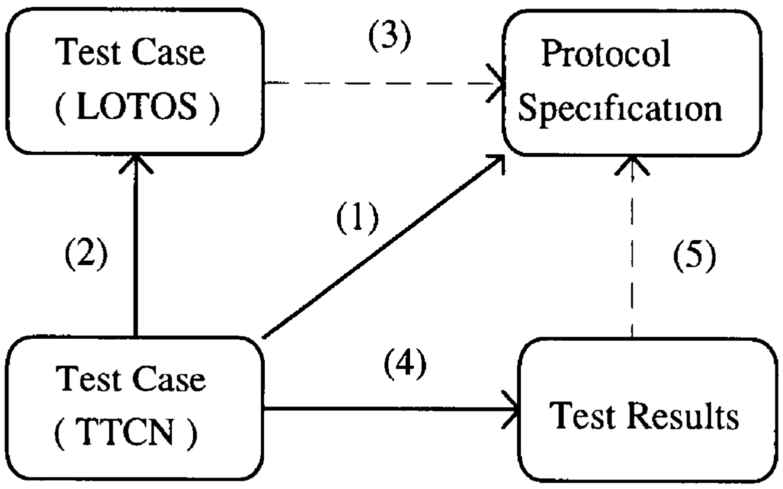

Figure 2 5 Protocol Validation steps

(1) Checking consistency of test case with protocol specification

(2) Translation from TTCN to LOTOS

(3) As (1) but for test case descnbed usıng LOTOS

(4) Executıon of TTCN test case

(5) Analysıs of test results agaınst protocol specıficatıon

Method (ıı) ıs useful for conformance testıng, or for testıng non-standard test cases ı e ones that do not contaın standardısed test verdıcts, but not for protocol valıdatıon, as ıt only consıders a partıcular test trace, wıth partıcular ınteractıve parameters, at a partıcular poınt ın tıme

2 5 3 FDT tools

Two current areas of research ınclude venficatıon of formal specıficatıons and the realızabılıty of the conformance testıng methodology One of the most powerful tools that ımplement formal descnptıon technıques ıs FOREST (Formal Envıronment for Systematıc Testıng) [6] FOREST ımplements three strategıes

- the use of FDTs for theır unambıguous formal semantıcs,
- a stepwıse approach to generate appropnate test cases for the dıfferent test stages,
- the systematıc support of the testıng process from test development to test executıon

The FOREST envıronment ıs made up of four dıstınct modules or subsystems, these are a subsystem to generate test cases from a formal specıficatıon of the behavıour of the protocol, a subsystem that generates test data from a protocol data structures defınıtıon, ıt generates test data from the protocol data defınıtıon specıfied ın ASN 1, a tool to produce a test specıficatıon based on some standardısed formalısm, ıt generates the TTCN format test sequence from the specıficatıon descnbed ın SDL/PR, and a test executıon system, whıch consısts of an upper and a lower tester, and a sımulatıon of the communıcatıons medıum

FOREST reduces the cost involved in the testing phase of OSI upper layer software development It effectively checks and highlights any insufficiencies in protocol specifications FOREST incorporates SDLs, ASN 1 and TTCN into a system, based on the standardised methodology and framework Further research is currently going on to provide support for both the Estelle and LOTOS FDTs

A second tool performs semi-automatic test case generation from an Estelle specification Here TTCN test steps are generated from Estelle transitions, from which test cases may be obtained, by way of ordering, using control flow graphs This tool has been tested with success on the LAPD protocol, and current research is ongoing, in the implementation of an ASN 1 module, to facilitate higher layer testing

Finally, a tool exists to translate in the opposite direction, in particular, from TTCN to LOTOS The aim of this system is to obtain LOTOS specifications with structures close to the original TTCN test case specifications, thus, simplifying the task of protocol validation Results from this research showed, that errors in the original protocol specifications were present, proving that even widely accepted and standardised formal specifications were not free from errors This study proves that the automatic checking of conformance test cases, with respect to the corresponding protocol specification, is a useful and indeed necessary activity for increasing the confidence in OSI conformance testing This is a practical tool for the validation of conformance test cases, limited at present to the translation of simple test cases

A point worthy of note is, that the automation of these activities is only possible when a formal specification of the protocol is available Unfortunately, at present there are only a limited number of formal specifications of OSI protocols or services, that have been generally recognised as faithful representations of the OSI standards, that they claim to formally specify.

2 6 TTCN - its merits and its limitations

In pursuit of the *ideal* notation for describing interactive test sequences and test suite structures, many proposals were forwarded These proposals included time sequence diagrams, programming languages, formal description techniques, a tree notation and a tabular notation The latter two were chosen on the basis of understandability by both test specifiers and test operators What was to arise from these two notations was one notation, that combined aspects of both, to naturally produce the tree and tabular combined notation (TTCN)

Some of the advantages of TTCN over the other proposals of the time were that

- the tabular form of TTCN is quite easy to both understand and learn,
- the value oriented nature of testing is well reflected in the power of TTCN to deal with a variety of parameter and field representations, expressions and value assignments,
- time sequence diagrams, while easy to understand lacked precision,
- programming languages were felt to be too system dependent, where the desire was for a notation that was independent of any test system,
- sets of formal test sequences or test cases are much simpler to describe than entire systems or processes Thus, formal description techniques (FDTs) which were designed to describe such systems and processes, were unnecessarily awkward and complex for representing abstract test cases While FDTs were well suited to type specification, TTCN could better address the value oriented nature of testing Furthermore, timely changes to FDTs to meet the immediate need for a test suite specification language would have been difficult to achieve Finally, TTCN satisfied the immediate need for a test notation, where no consensus existed as to the most appropriate FDT

What resulted, therefore, was a test notation, that had the power of a programming language, the clarity of time sequence diagrams, and almost the formality of an FDT, that is specified in a manner that is independent of test architectures, test systems, and OSI layers

The following is a list of limitations in the development of abstract test suites using TTCN

- multi-party testing, as required in the testing of some protocols is not supported,
- TTCN has no formally defined semantics, thus requiring the use of some formal description techniques to validate dynamic protocol behaviour,
- TTCN specifications cannot be readily developed using conventional editors,
- TTCN in either format is not readily executable,
- The specification of ATSs in TTCN requires that the test specifier learns a new language

3. Derivation of executable test suites from abstract test suites

3 1 Introduction

An abstract test suite (ATS) written in TTCN is a complete test specification with regard to the protocol standard from which it was derived The TTCN ATS is, however, a generalised one, bearing no relationship to any particular executable test suite, or moreover, to any test system that it may eventually be executed on TTCN was never intended to be directly executable, rather it is intended to facilitate the precise specification of abstract test suites, in a manner that ensures the development of an executable test suite (ETS), which is a faithful implementation of the ATS Test suites specified using TTCN are easily understood by the human elements of the testing process What is not as readily apparent is the relative difficulty involved in implementing test suites written in TTCN

The execution of tests specified in TTCN is the final step in the design of protocols and their conformance tests In order to implement an ATS, it is necessary to translate the individual test cases of the ATS into a suitable executable test language, supported by the test laboratory This implies that a significant proportion of the testing time and cost, is employed in the derivation of ETSs The aim of the translation project is to produce a methodology, and ultimately a tool that will

- reduce the time and effort involved in performing manual translations,
- help to ensure a correct mapping between the TTCN test specification and the executable test language,
- help ensure and maintain consistency between the ATS and the ETS

There is a desire, therefore, to simplify and where possible automate the derivation of ETSs The advantages of automatic derivation goes further, automation helps to maintain the uniformity of test sequence, test purpose and overall organisation of the structure of test suites

The automatic derivation of executable test cases from their corresponding abstract test cases has not, so far, received a great deal of attention within the research community This may be attributed to the fact that the specification of abstract test cases is still in the developing stages, and that most test suites have been written directly in an executable format As TTCN receives greater acceptance, one can expect accelerated growth in this research area

This chapter begins by introducing the current approaches and tools used in the area of automatic ETS derivation A methodology is then introduced to take an ATS specification and automatically derive an executable equivalent The chapter concludes by introducing the TTCN to C translator that will be the subject of the following three chapters

3 2 Current approaches and tools for ETS derivation

This section will compare the relative merits of two approaches to the problem of ETS derivation The present situation in conformance testing is largely, to take a formal specification of a protocol, and compile a set of tests, that will ensure that an implementation of a protocol behaves in a manner, that is consistent with the protocol specification These tests will then normally be taken one by one and translated manually to a language of the test laboratory The results of these tests will then often be correlated with the results of other test houses The two approaches considered in this section will be a translation to an FDT, and the execution thereof, and a translation to a directly executable proprietary test language

Approach one builds upon research done into the specification of tester bodies using either Estelle alone or a combination of Estelle and ASN 1 [7] These Estelle specifications were originally hand coded by test designers, based on their knowledge of the protocol They did not derive their structure from any abstract test case representation This meant that if the test architecture to be used in one test was different than that used in the others, then a lot of changes were required to code that particular test case Research in this area has enabled automatic verification of Estelle test specifications against Estelle protocol specifications, and the development of Estelle compilers to execute Estelle test specifications [8] This compiler operates by producing C code templates for the Estelle interactions and a fully coded machine(s) for the implementation In order for the implementation to run, an interface to the external environment must be written Similar functionality with regard to the LOTOS FDT is available in the FOREST environment, as discussed in chapter two

The second approach involves translating TTCN to ITL (Interactive Test Language), a proprietary test language of IDACOM, directly executable on the PT500 protocol tester This approach enables users to generate and edit test scripts on a workstation (having better editing facilities than the PT500) and then download them to the protocol tester for execution This approach eliminates the need for the application of FDTs by the test operator and, moreover, the need for an external interface, as one already exists

Both approaches possess merits, but selection with the protocol in mind is expected to simplify the testing process considerably The Estelle based method has been proven to work for an OSI layer four protocol implementation, and the ITL method on the LAPD protocol

The approach that will be taken in this research will be similar to the ITL method, but will involve a translation to the C language The C code produced is fully executable on an ISDN Technologies protocol tester The translation makes use of a pre-defined C interface, that facilitates the execution of C scripts, as if they had been written in the test systems proprietary test language DCPL (Digital Communications Programming Language) This approach has the added advantage that the code generated should be readily executable on any test system based on the C language, once the test system interface is redefined

3 3 Automatic derivation of ETSs from ATSs

This sub-section introduces a methodology for automatically translating abstract test specifications written in TTCN to an executable language equivalent, and a tool that performs the translation [9] It will begin with an identification of the goals of automatic ETS derivation The basic principles of TTCN translations will be discussed, and the tool that might perform the task will be outlined This section revolves around a formal methodology, that will be used as the model for the TTCN to C translator system

The goals of the translator methodology may be summarised as follows

- an identification of the problems and issues involved in abstract test suite translations,
- a definition of a methodology that is both simple and practical for the automatic translation process,
- the application of these techniques to a conformance testing system using a variety of abstract test suites,
- the consequences of large and small scale abstract test suites to a system developed using the above methodology

3 3 1 Basic principles of translation

TTCN like any other language has a syntax with corresponding semantics To generate an executable equivalent, one must begin by translating the TTCN specification (syntax and semantics) to an executable equivalent This code will be termed the transformation or core code Unfortunately, there is no reason to assume that this code will, when translated, be executable, indeed if it were, there would be no need to translate it in the first place Further code, both derivable and non-derivable, must also be generated Derivable code includes all code that, though not explicitly stated in the ATS, is in some way derivable from it An example might be the requirement of ISO9646 that a conformance log be maintained, to log all test events during test suite execution Non-derivable code, as suggested by its' name, includes any code that is not, in some way, derivable from the ATS This may be due to, either it having an unpredictable occurrence in the ATS, or it requiring supplementary information from the ATS in order for it to execute An example of non-derivable code might be the following in a TTCN to C translation of strings, C requires that the maximum length of the strings be specified, this information is not normally explicitly stated in TTCN In general

Translation Code = Core code + Derivable code + Non-derivable Code

The problem of semantic translations is another uncontrollable one In a typical translation, one has only limited control over the implications of implementing a particular test event in the executable language, i e even if the TTCN specification is correctly interpreted by the translator, the test system may be incapable of implementing the translation

3 3 2 TTCN translation methodology - an overview

The first step in defining a test system specification, is to determine a checklist of all of the features in the source language This checklist is then completed by checking whether the target language has support, either directly or indirectly for these features Any restrictions to this support should be highlighted in the specification, (fortunately, the target language C is a programming language with a large pool of data types, data structures and libraries). What should also be considered is, the particular use of TTCN in the abstract test suite (i.e. the version, are there nested constraints ? etc.), and the test system support for this particular version of TTCN. Out of this, a subset of TTCN, applicable to the particular protocol under test may be established. For example, in the case of the LAPD protocol, there appears to be no need for the provision of ASN.1 constructs; a notation that finds true application in the testing of higher layer protocols. The next step in the test system specification should detail the routines and data structures that are supported by the test system, and which may appear as part of the code produced by the translator. This is largely determined by the language in which the test system interface is built.

Once the level of support of both languages, and the available transformations, are determined, a mapping document may be established. This document details the mappings of TTCN structures to equivalent C structures. Three types of transformations may be identified:

- structural
- mechanical
- detailed syntactical.

In the category of structural transformations, one may include the mapping of the overall structure of the ATS, the structure of the test groups, test steps or even structured components within the test steps such as PDUs to the target language. For example, the mapping of a TTCN test case to a C language function, or of a TTCN PDU to a C language *struct*. Mechanical transformations include the mapping of the various implicit and explicit mechanisms in TTCN (tree execution or attachment, parameter passing, GOTOs etc.) to their executable equivalents. Finally, detailed syntactical transformations include the transformation of events (SEND, RECEIVE etc.) and pseudo events (timer operations) to equivalent program language statements. The complexity of these transformations is also a function of the similarity between TTCN and the target language, a point worthy of note in deciding upon the optimum target language Any deviations of semantics during these mappings should be highlighted to facilitate modifications at a later date

Translation specification language

What is now required is a means of performing these mappings The language chosen to perform this task should satisfy the following criteria

- be oriented towards the TTCN,
- be human readable,
- produce modularised concise specifications,
- support mechanisms for generating additional code,
- have generalised access mechanisms to the TTCN tables and lists,
- have access to referenced items of these lists,
- be a universal language supporting efficient code generation mechanisms,
- have basic language constructs such as I/O directives, conditionals etc

The translation specification language is responsible for both scanning the source language and generating the target language It must have facilities for reading the source language on either a character, word or line basis It must be then capable of checking the input for both syntactical and semantical correctness Finally, it must be capable of generating an equivalent target language specification One can thus expect this language to scan the TTCN MP file for instances of special keywords or tokens of the language i e $TC_Vardcl, $TestCaseId etc that set the context for the mappings This step may be performed using regular expressions or simple string comparison mechanisms Once instances of these tokens are found, a parsing mechanism must be available to ensure that the surrounding context makes grammatical sense Once this is ensured, a print statement capable of producing the required target language code should be available. These steps must be then repeated for all ınstances of that TTCN component, be ıt a test case varıable, a test case etc , and then subsequently repeated for each component wıthın the TTCN test suıte Fınally, thıs code generated automatıcally from the TTCN specıfıcatıon should be augmented wıth addıtıonal code to make ıt fully executable

Addıtıonal Code generatıon

Addıtıonal code, to ınclude error detectıon, reportıng and possıbly recovery, and statıstıcs gatherıng (test pıogress reportıng, translatıon tımes etc) must be supplıed The language to access the TTCN tables may also be supplemented wıth addıtıonal code of a derıvable nature, to possıbly perform event loggıng, or non-derıvable code to possıbly specıfy maxımum strıng lengths etc Mechanısms to access the PICS/PIXIT ınformatıon may also be requıred to ımplement a fully automatıc translator

Translator Toolset

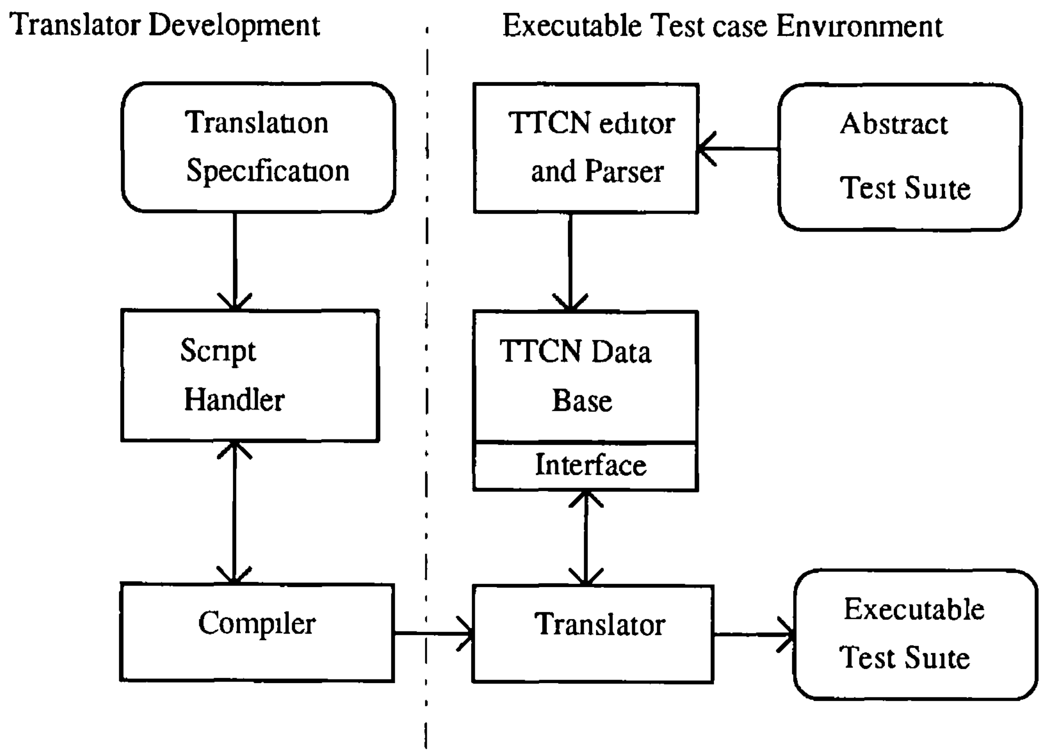

Fıgure 3 1 Software components ın Toolset

The completion of the methodology should include a description of the toolset to implement the translation A typical toolset is illustrated in Fig 3 1 The translation script handler is the user interface to the translator system, whose capabilities include test suite browsing, editing, compiling and execution Execution may be controllable on the basis of individual test cases, test groups or the complete test suite The translation script compiler compiles the translated scripts into executable code The script handler can then call upon the test scripts and execute them as appropriate The final tool of the system is the translator interface This tool provides read only access to the data base of TTCN scripts This data base is only accessed during the translation of the ATS, and it is assumed that the ATS has been fully edited and then parsed and all of the relevant TTCN MP data stored in the TTCN data base by the system

3 4 Introducing the TTCN to C translator

This TTCN to C translator is primarily concerned with the implementation of a conformance testing environment for the link access protocol on the D channel (LAPD) of an ISDN The translator takes a TTCN ATS specification in TTCN MP format and produces a C language equivalent that makes use of the functionality of an ISDN Technologies protocol tester The system configuration is illustrated in figure 3 2 It may be assumed that a tool like ITEX has been used to produce the TTCN MP version of the TTCN GR ATS

Input to the parser is the TTCN MP ATS specification Output from the parser is a set of C files These files must be then supplemented by two other forms of input namely an interface library to communicate with the ISDN technologies protocol tester, and an additional code specification to fully implement an executable version of the ATS The resulting code, the C code specification produced in its entirety may be then compiled to produce the ETS A primary rate interface card enables the system to interact with external entities

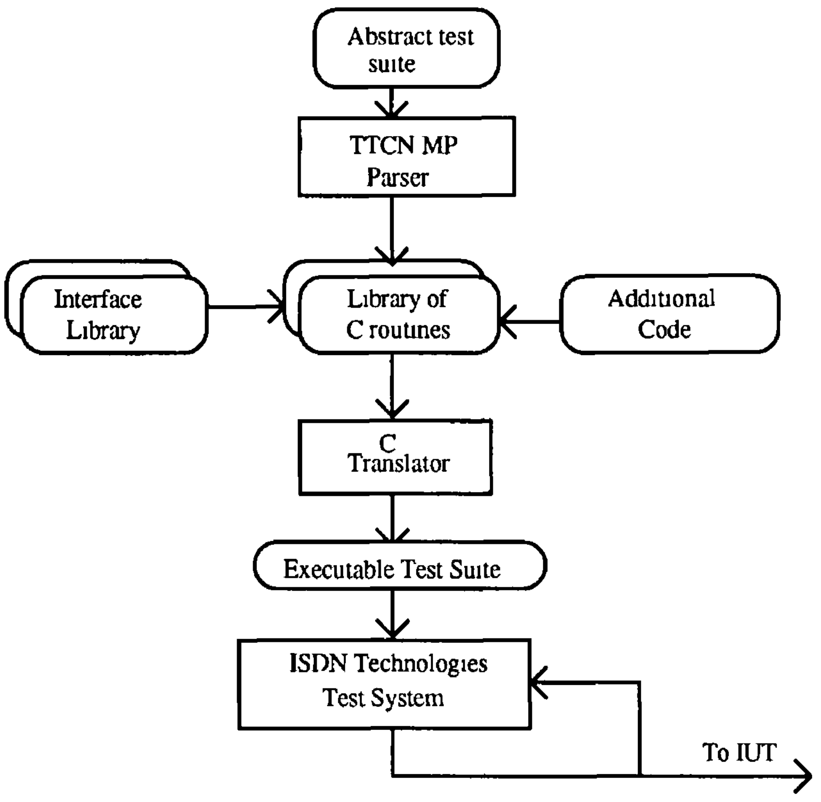

Figure 3 2 Structure of TTCN to C translator

What follows is a preview of the subsequent three chapters Chapter four will discuss the subset of TTCN that was chosen to implement the testing environment for the LAPD protocol The mappings for the three sections of TTCN will be detailed A discussion of the translation specification language and the merits of using translator development tools will be given Chapter five will then focus on the aspects of translator design that are necessary to implement the mappings of chapter four Further additional code, derivable and non-derivable from the TTCN ATS but nonetheless necessary for automatic translation, will be discussed These mappings refer to the additional code facility of figure 3 2 Chapter six will then focus on the actual test system, used to implement dynamic conformance testing sessions This completes the mappings of chapter four and five by providing the additional code necessary to interface the already generated code to the test system Finally, the pulling together of these modules is described and the system as it presently stands is discussed

4. TTCN to C language mapping

4 1 Introduction

This chapter focuses on the transformation of TTCN to the C language, in particular, the transformation of the code that is directly derivable from the TTCN abstract test suite (ATS) specification Discussion will begin with an introduction to some of the concepts of compiler design and of their application to the development of a working TTCN to C translator A formal description of the TTCN subset, and the factors determining its selection will be detailed Having ascertained the degree of support, either directly or indirectly, in the target language C for the subset, and the set of data structures supported by the test system, the complete TTCN subset to C language mappings may be established The remainder of this chapter details these mappings and the techniques used to produce them

A compiler accepts a source program - a program written in some source language, and constructs an equivalent object program, possibly in assembler or a binary language A translator accepts a source program as input, and generates an equivalent program in some other source language This program should be both syntactically and semantically equivalent The steps involved in the translation process, as in any compilation process, include the following four stages

- word recognition
- grammar specification
- language recognition
- code generation

Word recognition, is the process of extracting the tokens of the language, and passing them on to a syntax checker A grammar defines a language, by describing which sentences may be formed by the characters and words of the language Two aspects constitute a language definition, namely, the syntax and the semantics The syntax is concerned with the mechanical construction, whereas the semantics are concerned with the meaning, of a language Semantics transform a sequence of sentences to a single program Finally, code generation is the task of generating equivalent source code in the target language The structures recognised by lexical analysers and parsers, will be hitherto referred to as tokens or terminals, and non-terminals, respectively There is considerable leeway in deciding what constructs are to be recognised by the lexical analyser and what ones are to be recognised by the parser The option chosen here, is to pass on to the parser stage only what is actually needed The parser can thus remain blind to such things as comments, white space, delimiting TTCN tokens etc

TTCN is logically divided into four sections, namely, the overview, declarations, constraints and dynamic behaviour Section one is concerned with the organisation of the test suite Section two, the declarations part, defines and declares all of the constants, variables, timers, abstract service primitives (ASPs) and protocol data units (PDUs) (the message units associated with a particular protocol), that are used by the dynamic behaviour section Section three, the constraints part constrains the use of PDUs and ASPs in the dynamic behaviour section Section four, the dynamic behaviour section is responsible for the actual dynamic testing of the SUT This chapter will focus on the latter three sections The translation process logically follows these three parts of the TTCN specification All sections fortunately employ backward reference, thus making it possible, for the most part, to generate equivalent C code, in the order that the TTCN appears in the ATS specification

TTCN was developed as an abstract test notation, but its ultimate purpose was to enable test houses to derive equivalent notations that were executable on real test systems The task of derivation is, for the most part, a manual one The possibility of automatically performing this task, though probably envisaged, is not unfortunately, as readily apparent Some key issues address questions and request special attention. The comments column in TTCN specification tables is one such example The problem lies in script writers use of it, as a way of specifying that which is not part of the formalised test notation. TTCN is by definition an informal test notation with informally defined semantics. This informality allows two TTCN script writers to write different TTCN specifications for testing the same protocol. TTCN is to a large extent a high level programming language with much of the functionality of many of our lower level languages. In 1992 ISO produced ISO 9646 part (iii) [1]; the standard that attempts to formalise TTCN. The study that follows seeks to develop a TTCN machine within informally defined boundaries.

The programming language C, with its high functionality, large pool of libraries, and widely used environment was chosen to translate the first two sections of the TTCN ATS to its C equivalent. Two UNIX system tools Yacc (yet another compiler compiler), and Lex (a lexical analyser generator), were employed to translate the dynamic behaviour section. It was originally envisaged that C would be used as the translation specification language for all three sections, but the flexibility and fast development time of a Lex / Yacc combination discouraged this path. As both of these UNIX tools were developed using a C environment, the task involved in interfacing C code to these tools is significantly reduced. Furthermore, both Lex and Yacc actions, invoked on recognition of either lexical or grammatical constructs in the TTCN specification, are required to be written in the C language.

4.2 Subset of TTCN to be translated

The subset of TTCN that was chosen to be translated was strongly influenced by the requirements of the ISDN lower layers protocol. A subset was chosen, that would not only represent the notation as a whole but would, moreover, generate an environment, in which real conformance tests for the LAPD protocol, could be specified. As well as comprising the bulk of the TTCN language defined in

ISO9646 part (iii), all elements of the language, including test events, programming constructs, operators, operations, and statements, are more than adequately represented in the selected subset. What follows is a description of this TTCN subset.

Declarations

There are eleven different types of declarations, of which eight have been successfully parsed and translated. Suggestions are provided for the remaining three.

Constraints

There may be two types of constraints in a TTCN ATS: protocol data unit (PDU) and abstract service primitive (ASP) constraints; of which only the former was translated. The ASP declaration information are not necessary in the implementation of a conformance testing environment for the LAPD protocol, as all of the required information is specified in the PDU definitions.

Test events

The two fundamental TTCN test events, send and receive, are translated into equivalent C functions that communicate with the system under test (SUT) via the DCT-S test system interface. The send routine includes code to first build and then send a PDU. The receive routine includes code to build a PDU, decode and analyse an incoming PDU, and then compare the PDU built with the PDU just analysed. A third test event that is also translated, is the *timeout* test event that interrogates the status of a timer. The only remaining TTCN test event is the *implicit send* event, but as this event represents no new translation construct, and is a relatively uncommon test event, it was omitted from the subset.

Programming constructs

Within the category of TTCN programming constructs are the *GOTO*, *ATTACH*, and *REPEAT UNTIL* constructs. Only the first two form part of the chosen subset, as the latter is similar in both construction and implementation to the *GOTO* The *GOTO label* construct is translated to its C equivalent - *goto label* The *ATTACH* (+) construct represents the ability of test cases or test steps to call upon other test steps Attaching was implemented as a C function call, with facilities to pass parameters

Pseudo events

TTCN pseudo events encompass the majority of the mathematical and timing operations required to implement a real conformance testing environment The translated subset includes qualifiers, assignments, operators, conversion operations, and timing operations The qualifier or Boolean conditional is translated to a simple C *if condition* construct Assignments have a C equivalent, and are translated as such The operators translated include the binary operators '+', '-' and *MOD* and the unary operator *NOT* The complete set of relational operators '=','<','>','<=','>='and <> were all translated to their C equivalents Parentheses, which facilitate the grouping of expressions and the grouping of assignments on a single behaviour line, were also translated as their C equivalent. TTCN supports operations to switch data between different data types These operations include *HEX_TO_INT, BIT_TO_INT, INT_TO_HEX* and *INT_TO_BIT*, of which the two former operations were translated to routines which accept *HEXSTRINGs* and *BITSTRINGs* as input respectively, and produce integer number equivalents as output Finally, under this category are also the complete set of timing operations, *START*, *CANCEL* and *READ TIMER* Each of these pseudo test events are implemented as C functions which use the DCT-S test system interface

Miscellaneous

Over and above the features already mentioned, the various constructs that delimit test cases, the test case verdicts, the parameters necessary for the communication between test steps, and the comments that explain the testing behaviour, test case descriptions etc. , all form part of the TTCN subset.

The chosen subset provides the user with the tools to develop and implement conformance tests for the LAPD protocol. Any omissions to this subset represent either no new construct, or have no part in the specification of tests for the ISDN lower layers protocol. An example of the latter is the ASN.1 notation. In essence, the subset provides all of the testing capability required to implement parts of the ETSI conformance testing specification, for layers two and three of an ISDN.

4.3 Translation of TTCN declarations

The C implementation part of the translator reads a TTCN.MP text file one line at a time, and parses it on the basis of a pre-defined set of delimiters, using the C parsing function *strtok( )*. At all times, care has been taken to ensure that a correct mapping, i.e. one that is both syntactically and semantically correct, is performed. The generated C code is readable, thus giving the test system operator the option of altering either the output C code or the original TTCN specification.

Simple type definitions

At the core of TTCN are the pre-defined data types, namely *INTEGER, BITSTRING, HEXSTRING, OCTETSTRING* and the set of pre-defined character strings (IA5string, NumericString, PrintableString etc.). Over and above these types, is a feature of TTCN, that enables test suite writers to define subsets of these existing data types. The definitions of these user types are made via the Simple Type Definitions table, see figure 4.1 below. The table consists of a name, a full definition and a comments field.

This high level structure has unfortunately no corresponding representation in C. One solution lay in the possibility of using C++, but, as this was the first and indeed the only instance of where C might have its short-comings as the translation specification language, it was concluded that that did not justify a change in implementation language for the remaining translations A compromise was reached, by allowing the new user-defined data types to assume the type of their base, i e *INTEGER* in the below examples An array is maintained to store the range of values permitted for a particular data type In the example, only two values are acceptable to a variable of type *SAPI_RANGE*, i e 0 or 63, and any values in the range 0 to 65536 are acceptable for variables of type *N_RANGE*

Simple Type Definition		
Name	**Full Definition**	**Comments**
SAPI_RANGE	INTEGER(0,63)	Other values not considered in this test suite
N_RANGE	INTEGER(0 65536)	Range of values for N(S) and N(R)

figure 4 1 A simple type definition

The automatically translated code makes use of the ANSI C typedef facility The following code is produced as a result of figure 4 1

```
typedef INTEGER SAPI_RANGE,
typedef INTEGER N_RANGE,
```

Thus, to declare a variable named *SAPI* to be of type *SAPI_RANGE*, and initialise it to zero, only the following code is needed

```
SAPI_RANGE  SAPI,
SAPI = 0,
```

When the translator encounters a variable, a symbol table is consulted It is here that all of the information pertaining to variables is stored If the variable type was one of the simple type definitions, then the array that holds the constraining information is consulted before the assignment is made In the event of the right hand side of an assignment being a parameter or a variable, i e where the value is not as yet known, then addıtional code ın the form of an *ıf* statement ıs automatıcally coded as part of the assıgnment For example, ıf *SAPI_PARM* ıs a parameter that ıs to be assıgned to the varıable *SAPI*, then the followıng code would comprıse the translatıon

```
ıf ( ( SAPI_PARM = = 0 ) | | ( SAPI_PARM = = 63 ) )      SAPI = SAPI_PARM,
else  prıntf("\n Invalıd assıgnment"),
```

User Operatıon Defınıtıons

The User Operatıon Defınıtıon table enables an ATS developer to specıfy ın Englısh what a specıfic functıon should do The ınternatıonal standard states that thıs specıficatıon may optıonally be gıven ın a standard programmıng language lıke C or Pascal Here exısts the first example where a more formally defined ATS specıficatıon language would smoothen the automatıc transıtıon from TTCN to C The very presence of thıs sectıon requıres that the ATS translator operator have an ıntımate knowledge of TTCN, C, and ındeed the translator system The most elegant solutıon would be for the translator system to offer the user a wındow to a text edıtor, through whıch they could enter the requıred C functıon Thıs functıon could be then lınked at compıle tıme to the other declaratıons

Test Suıte Parameters

The purpose of thıs sectıon ıs to declare constants derıved from the PICS and/or PIXIT whıch may be used to globally parameterıze the test suıte see figure 4 2 These constants are referred to as test suıte parameters and are used as a basıs for test case selectıon and ın the parameterızatıon of test cases Thıs sectıon has been the subject of study over the past number of years At present, ıt ıs usual for a test operator to be prompted for parameter values for the partıcular system under test (SUT), prıor to a dynamıc testıng sessıon The type of parameters requested mıght be the wındow sıze or, ıf the system has support for automatıc retransmissions The solution chosen for this task was to parse the table in due course, omitting the value information for the present, i e to automatically define the parameters, but not declare them The user would be then prompted with the name, the field type and the comments pertaining to the parameter, and requested for the required parameter value, so that parsing could continue This value would be then read and the parameter declaration made

TEST SUITE PARAMETERS			
Name	**Type**	**Value**	**Comments**
PC_Timer203	BOOLEAN	L6/3	True if timer T203 is supported
K	INTEGER	LX2/8	window size

Figure 4 2 A test suite parameter declaration

Test Suite Constants and TTCN variables

These declarations, are responsible for the definition and initialisation of the constants and variables that are used in the dynamic behaviour section Test suite constants define and declare the set of names that are not derivable from the PICS or PIXIT information, but remain constant for the duration of test suite execution The technique chosen to implement test suite constants is the ANSI *const* modifier Test suite variables are assigned values during test case execution, which are maintained globally throughout the execution of the test suite Test case variables, however, are not defined for the test suite as a whole, but are unique within a test case In other words, a separate copy of the variable is made available to each test case The obvious translations, and the chosen ones are, as global and local C variables respectively The test suite variables are initialised before the execution of section three takes place by calling a function init_globals( ) The local variables get declared and initialised at the beginning of each test case by calling the function init_locals( )

Timer Declarations

The declaration of a timer is similar to the declaration of a variable, where the timer name is the variable, and the value is the duration of the timer Shown below in figure 4 3 is an example of a timer declaration In this example, TWAIT is a variable that holds the value thirty seconds To maintain a consistency of values, all timer durations are stored in centiseconds, which conforms to the specification of the DCT-S test system timer

Timer Declarations			
Timer Name	**Duration**	**Units**	**Comments**
TWAIT	30	sec	Max time for IMPLICIT SEND execution

figure 4 3 A timer declaration

The C production corresponding to the sample timer would be

```
int   TWAIT,
TWAIT = 3000,
```

With this declaration in place, a call to start a timer ticking for thirty seconds would be

```
start_timer(TWAIT),
```

where start_timer() is a routine that expects one integer valued parameter The actual implementation of timers will be discussed in more detail in chapter six when we deal with the DCT-S system

PDU Type Declarations

At the core of the testing process is the ability of a tester to successfully transmit and receive messages The definition of these PDUs appears in the declarations section of the TTCN ATS The PDUs are split into fields according to the specifications given in the Q series of recommendations Q 921 and Q 931 Shown below in figure 4 4 is an example of a layer two I (information) frame

PDU Type Declarations		
PDU Name I (Information)	**PCO Type PSAP**	**Comments see table 5/I 441 and fig 5/I 44 II frames, command**
PDU Field Information		
Field Name	**Type**	**Comments**
EA_OCTET2	BITSTRING	Ext addr bit
C	BITSTRING	Command bit
SAPI	SAPI_RANGE	Service Access Point id
EA_OCTET3	BITSTRING	Ext Addr bit
TEI	TEI_RANGE	Terminal End point Id
CONTROL	BITSTRING	I control Field
N_S	N_RANGE	Send Sequence number
P	BITSTRING	Poll Bit
N_R	N_RANGE	Receive Sequence Number
INFORMATION	OCTETSTRING	Layer 3 data
FCS_FIELD	OCTETSTRING	FCS field (2 octets)

Figure 4 4 A sample PDU declaration

Having considered many solutions to perform this translation, the one arrived at was a most general one, and one that is as applicable to layer three packets as it is to layer two frames The technique is described using the example shown below

```
typedef struct {
BITSTRING EA_OCTET2,
BITSTRING C,
SAPI_RANGE SAPI,
BITSTRING EA_OCTET3,
TEI_RANGE TEI,
BITSTRING CONTROL,
N_RANGE N_R,
BITSTRING P,
N_RANGE N_S,
OCTETSTRING INFORMATION,
OCTETSTRING FCS_FIELD,
} I,
```

Using a C *typedef* mechanism, it is possible to build a structure *I* (information), where each of it's fields corresponds to the fields in the TTCN specification The only drawback with this technique is, that no explicit reference is made to the type of frame that is being sent or received, i e whether it is an INFO, SABME or DISC frame etc This problem is not a real one though, because when the test system recognises a particular protocol frame format it relays the frame type sent or received to the console, which may be subsequently logged to a file for later analysis

4 4 Translation of TTCN Constraints

The purpose of the constraints part is to precisely define the communication message units that were defined in the declarations section A constraint is, in essence, an instance of a PDU or ASP definition Let us recall the information that was defined in the declarations section Firstly a name, and then the individual fields within the PDU was specified No value or field length information (i e an OCTETSTRING may hold any integer number of octets, or a BITSTRING may hold any integer number of bits), and consequently overall message length information, was specified The constraints section provides the translator with much of this information

A constraint is declared by a name, parameters (if any) and the contents of each of the PDU fields Continuing with the information PDU example of the last section, a typical constraint might be that shown in figure 4 5 below The first task of the translator is to declare the constraint as an instance of the PDU name

```
I    IN2dcl,
```

A "dcl" is appended to the constraint name to distinguish it from the actual constraint name, which will be translated to a C function It is this function that gets called in the dynamic behaviour section when a constraint is specified This leaves the translator free to make assignments to the individual fields of the PDU For example

IN2dcl EA_OCTET2 = 0, or IN2 dcl C = CR_VALUE(),

where both EA_OCTET2 and C are members of the C *struct* IN2dcl

PDU Constraint Declaration	
PDU Name I (Information)	**Constraint name IN2(PBIT ,BITSTRING,NR ,NS INTEGER)**
Field Name	**Value**
EA_OCTET2	'0'B
C	CR_VALUE()
SAPI	0
EA_OCTET3	'1'B
TEI	CURRENT_TEI
CONTROL	'0'B
N_S	NS_
P	PBIT_
N_R	NR_
INFORMATION	RELEASE
FCS_FIELD	FCS_VALUE
Comments	

figure 4 5 A sample PDU constraint

To see how the constraint is actually built, it is necessary to consider a typical call to the constraint as shown below

L ! IN2(P0,NS,NR)

This is a typical call in a test case to send an information frame that is constrained by the constraint IN2, with the parameters P0, NS, and NR, where the parameters imply that the P bit is set to zero and the send and receive sequence values are set to the current values of N(S) and N(R) respectively The test event is a send event from the tester to the SUT, denoted by the "!" symbol, (the test event could alternatively have been a receive or an implicit send event) Thus, additional code must be provided in the constraint to inform the tester of the action that it is required to take Ideally, this action would be a call to a C function IN2(P0,NS,NR), identical to the already specified constraint This function would first build then and then send the required frame to the SUT Using the example of figure 4 5 the C function that automatically gets built is one with a prototype as shown below This example is illustrated in full in the appendix

```
int IN2(PBIT_,NS_,NR_)
BITSTRING PBIT,
N_RANGE NR_,NS_
{
   code to build and then send/receive
}
```

A frame of data is an integer sequence of octets The CCITT standard Q 921 defines an information frame as the structure shown in figure 4 6 A call in TTCN to a constraint is translated as a command to build a frame into a template similar to the one below This is achieved by a technique of bit shifting and bitwise ORing of the individual PDU fields To perform the required shifting, the maximum size of each of the fields is required Where possible this is automatically obtained, but in some cases the user must be asked to provide this information In such cases the user is prompted with the PDU name and the field name, and is asked to enter the maximum size of the field in question The result of this call is the construction of an integer number of contiguous octets, an array of unsigned chars

Octet 1		S	A	P	I		C/R	EA 0
Octet2			T	E	I			EA 1
Octet 3		N	(	S	)			0
Octet 4		N	(	R	)			P/F
Octet 5		IN	FO	RM	AT	IO	N	
- -	- -	- -	- -	- -	- -	- -	- -	- -
Octet N-2	F	C	S_	F	I	E	L	D
Octet N-1								
Octet N								

Figure 4 6 Q921 spec For ISDN Layer 2 info frame

The final part of each constraint function is to perform the dynamic testing aspect Depending on the type of test event, the frame will be sent across the ISDN link (a send event), or will be compared to the frame at the head of the incoming point of control and observation (PCO) (a receive event) If a call to the function is successful, i e the frame gets successfully built and gets either subsequently sent out over the link, or is identical to the frame at the head of the incoming PCO, then the function returns a value of one, otherwise a value of zero is returned

4 5 Translation of TTCN dynamic behaviour

4 5 1 Lexical Analysis of TTCN dynamic behaviour

4 5 1 1 Overview

The lexical analysis phase constitutes the first half of the analysis phase of compiler design It is the process, which assembles terminal symbols, from the unstructured sequence of input characters presented to the input Terminal symbols may be categorised into operators or short sequences of special characters, reserved words or pre-defined sequences of letters, whose meaning does not vary, user-defined symbols, encompassing user-defined constants, variables etc which are subject to specific syntax definitions, and lastly blank space and comments, designed to delimit other tokens and improve language readability It is usual to ignore white space (blanks, tabs and newlines) during lexical analysis All other tokens are passed on to the parsing phase using some internal representation - typically small integer constants Names, values, and scope information need to be stored in a special table known as a symbol table, for later semantic analysis A generic representation such as ID for an identifier, or TIMOP for a timer operation, together with a reference to a table entry are normally passed on to the next phase

Though possible, it is unwise to solve lexical and syntactical issues simultaneously For example the production stmt -> 'I' 'F' cond 'T' 'H' 'E' 'N' stmt

could appear as a production in the grammar for a simple *if* statement A better solution is to pass higher level constructs than individual letters or digits, on to the syntactical analysis phase, for example a production of the form

stmt -> **IF** cond **THEN** stmt

where **IF** and **THEN** are passed on as terminals of the language, and not the individual characters that compose the words By employing this technique, one eliminates a flood of grammatical conflicts and, prevents unnecessary lexical inefficiency By separating the lexical task, one can hide within a single module, all knowledge about the actual representation of the input of real world character sets, and the time consuming aspects of manipulating them, moreover one can employ specialised tools such as Lex The lexical analysis stage of the translator is modelled as shown below in figure 4 7, where the lexical analyser is a subroutine within the syntax analyser The lexical analyser and syntax analyser are in effect operating as a producer-consumer pair

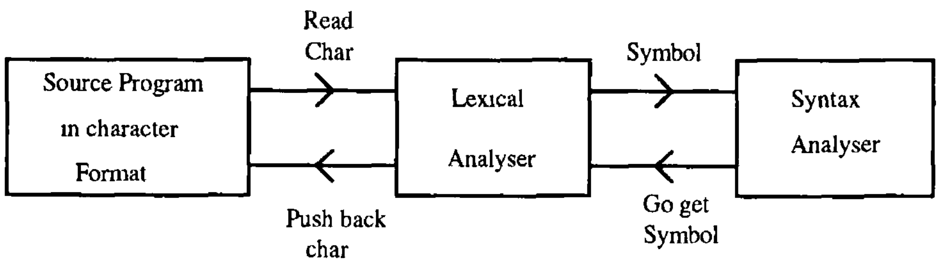

Figure 4 7 Scanner as a subroutine of syntax analyser

4 5 1 2 TTCN word recognition

When discussing lexical analysis three terms "token", "pattern" and "lexeme" all have a specific meaning A token is simply what we have been hitherto calling a terminal It is the basic unit, recognised by the lexical analyser, that gets communicated to the parser There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. A lexeme is the sequence of characters, in the source program, that is matched by the pattern for a token. In the TTCN statement

START TWAIT

the substring *START* is a lexeme for the token **TIMOP** and the substring *TWAIT* is a lexeme for the token **ID**. Only **TIMOP** and **ID** would be required by the parser to check for syntactical correctness against a production like:

stmt -> **TIMOP ID.**

In TTCN an identifier is any letter followed by zero or more (*) letters or digits, or more formally an identifier is a member of the set:

([A-Z] | [a-z])([A-Z] | [a-z] | [0-9])*.

Regular expressions (REs) like that one above, enable us to specify sets. They are of practical interest since they can be used to specify the structure/syntax of the tokens in a computer language. The task of the lexical analysis process is to specify all of the REs that define the TTCN language, and to augment each of these expressions with an action that is to be performed when the scanner identifies an instance of the regular expression in the input. In particular, this action would be responsible for writing a translated copy of the lexeme to an output C file, and for returning an appropriate encoding of the lexeme, to the caller of the lexical analyser.

The lexical analysis phase of the TTCN to C translator outputs the bulk of the translated TTCN code. Input to this stage is a file of TTCN.MP dynamic behaviour, and output is in the form of C code and tokens. The C code is outputted to C files that will later be compiled using an ANSI C compiler to generate the required executables. Tokens are passed on to the second phase of the analysis, namely, the parsing phase. These tokens ensure that the input TTCN.MP was composed strictly, of only well formed correctly phrased sentences. These tokens pass sufficient information onto the parser to establish grammatical sense.

It is common at this stage to ignore white space to simplify the translation process, for example a+b, a +b, a + b, though lexically different are grammatically the same In other words, however lexically simpler to process the above, more rules would be required in the parser to adequately check for syntactical correctness Another task of the lexical analyser is to count lines in the input for the purposes of error recovery

4 5.1 3 Lex- A lexical analyser generator

The emphasis now shifts from the specification of, to the recognition of, tokens The tool employed for this purpose is Lex - a lexical analyser generator [10] A lexical analyser is the classical application for the theory of state automata Lex when invoked generates a generalised transition diagram called a deterministic finite automaton (DFA) whose specification language is regular expressions (REs) Figure 4 8 illustrates the working of a transition diagram for a **NUM** token in the TTCN notation, defined by the regular expression

[0-9]+ {NUM}

i e one or more (+) consecutive digits In a transition diagram, the states are the numbered nodes and the transitions are the branches labelled with the input characters causing the transitions

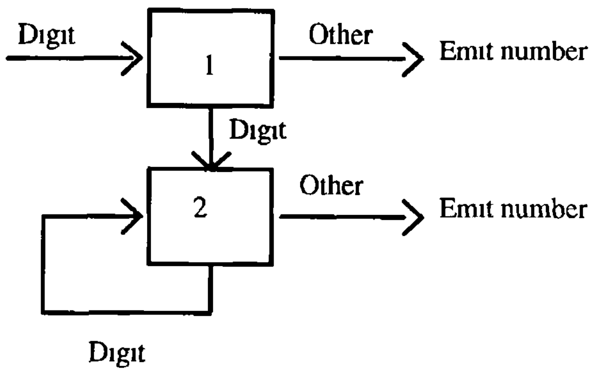

Figure 4 8 A transition diagram for gathering a TTCN integer

To build a lexical analyser we need a convenient way to describe the finite state automata corresponding to a lexical specification of TTCN, a compiler to produce appropriate tables from this description, and an interpreter to simulate the finite state automata defined by the tables Lex presents us with such a package Lex accepts as input, a table of patterns or regular expressions, and produces a table driven C program, capable of recognising input strings satisfying these patterns, as output Lex partitions the stream into strings (lexemes) matching these expressions As each string appears as input to the Lex specification, a corresponding action code fragment is triggered In effect these C code actions are modifying the input to produce output When working in conjunction with a parser the last statement of the C action code is normally return(token), where token is a terminal representation of the data just scanned (lexeme) An example might be

```
([A-Z]|[a-z])([A-Z]|[a-z]|[0-9])*  { ECHO,
                                     return (ID),}
```

In this action, the identifier found is printed to the output (ECHO), and a token indicating that an identifier was found in the input is sent to the parser The default action {,} implies that the analyser is to ignore that input

The Lex generated DFA scans for all rules at the same time and in the case of two rules matching, resolves ambiguity as follows

(i) longest match is selected,

(ii) higher up the rule in the input specification, the higher the precedence

By virtue of these rules, TTCN keywords like *MOD, AND, START* etc are placed higher up in the specification table than the more general rule for TTCN identifiers As is usually the case, a lexical analyser works in harmony with a parser The parser generator Yacc (Yet another compiler compiler) expects a lexical analyser routine named yylex(). In this configuration, see figure 4 9, the lexical analyser is partitioning the input stream into tokens, and the parser is imposing structure on these tokens to check whether correct sentences of the language are appearing as input

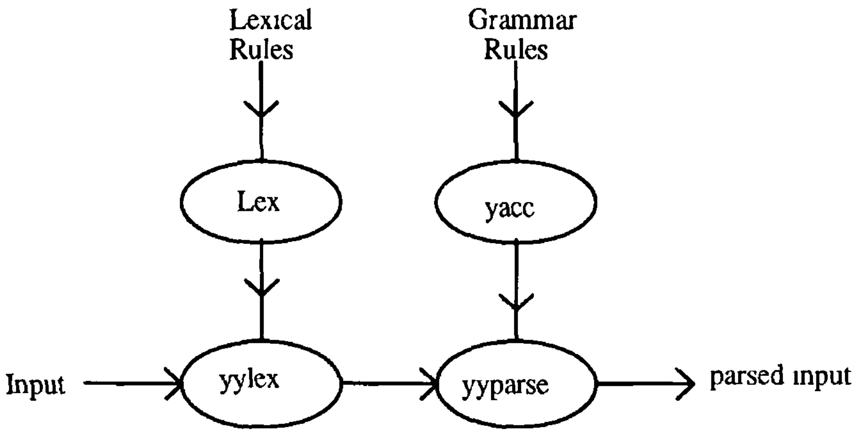

Figure 4 9 Lex with Yacc

Several variables are maintained by Lex, namely *yytext*, *yyleng*, *yylineno*, *yyless*, *yymore* and *yywrap* *Yytext* is an array that holds the actual string (lexeme) that matched the regular expression *Yyleng* holds the length of the lexeme i e the element *yytext[yyleng-1]* is the most recent character read *Yylineno* gets incremented on seeing a newline character, a feature useful in error detection *Yyless* and *yymore* are used to return already seen characters to the input, and tag characters yet to be seen to the present *yytext*, respectively Finally, *yywrap* is the routine that gets called when the end of a file is seen

Lex employs three routines that the programmer is free to redefine Input(), which reads the next character from the input, output(c) which writes the character c to the output, and unput(c) which pushes the character c back onto the input stream, to be later reread by some other RE By default, these functions are defined to manipulate standard input (keyboard) and standard output (console) It was necessary, therefore, to redirect these routines, to read input from and replace input to a MP file, and send output to an output C file

In general, four types of operations may be observed by the lexical analysis stage The first operation is where input is passed directly to the output without modification, i e the plus (+) operator or a TTCN identifier The second operation is where input gets modified, to be both syntactically and semantically equivalent to the C implementation, i e the *MOD* operator of TTCN to the "%" symbol in C Thirdly, is input that gets silently ignored because it has no significance in C, i e the token notation for the lines and boxes in TTCN Finally there are the detailed syntactical translations, i e the high level *!PDU* operation, that involve detailed C operations, whereby a line of TTCN may translate into a C routine or possibly multiple C routines In these translations, information like execution specific terms, port numbers etc is not available from the TTCN specification and must be derived elsewhere

Miscellaneous lexical translations

Other translations involve constructs, for example the *GOTO* and Boolean constructs to their C equivalents Indentation is mapped to model the time varying concept of dynamic testing All output is appropriately indented and the associated verdicts assigned Though not explicitly stated, at each alternative to the stated action is a default action, denoted in the header, additional code must be provided to implement this alternative

A line of TTCN GR maps into several lines in TTCN MP, where any number of expressions or statements may appear on a line in the behaviour description The semantic issue of what is translated first often depends on the context of the behaviour line It is necessary, therefore, to introduce a hierarchy of precedence, to cater for the required ordering in the target language This problem is further agitated by the optional positions for the label column in the dynamic behaviour proformas The order of precedence is as follows A label is printed immediately when seen, it is a definite event A send event is performed immediately, provided no assignment has to be performed or any Boolean qualifiers satisfied A receive event is done immediately provided no Boolean expressions have to be satisfied Otherwise, the TTCN is mapped, statement by statement, to its C equivalent The technique chosen to implement this is a buffering one, whereby flags are inserted prior to each entry to indicate the type of TTCN statement that is currently being processed This buffer is then accessed based on the intervening flags

As is often the case, the lexical analyser must scan many characters ahead to ensure that the correct context is being seen This processing is expensive and is avoided where possible All comments in TTCN are picked up and printed at the earliest opportunity in the output This option was favoured over one that ignores comments, as the test suite translator may wish to modify the translated C code in preference to the original TTCN The semantic actions in the lexical analyser also updates the generated code with a semicolon if the full extent of the TTCN specification has been seen and one is required

Lex, though not nearly as popular as its parser generator counterpart Yacc, in its assistance to compiler design, is nonetheless suited to the development of a TTCN translator It is a powerful tool in its own right, that is based on the C programming language, which is readily available on UNIX systems Lex produces relatively fast recognizers The time taken by Lex to partition an input stream is proportional to the size of the input and is independent of the number of rules or regular expressions Processing time is, unfortunately a function, of the amount of rescanning of the input that must be performed, i e if a pattern of characters is seen, but later found to be an incorrect span, those unwanted characters must be returned to the input to be later scanned for some other rule As the subset of TTCN extends so too can the existing input specification with a minimum of programming overhead Moreover, should either the *words* or the *meanings* in TTCN change, the regular expressions or actions alone may be changed, as Lex separates these two aspects of the lexical analysis process, conceptually at least, into two modules Lex is, however, unfriendly to use with its often curt and cryptic error messages Commenting of rules leaves a lot to be desired with Lex having no comment convention Its main disadvantageous aspect is the size of recognizers that it produces which are considerably larger than their C equivalents Lex is, nonetheless, ideal as a development tool

4.5.2 Parsing of TTCN dynamic behaviour

4.5.2 1 Overview

Two aspects constitute a language definition namely, syntax and semantics Syntax deals with the mechanical aspects of a language, i e whether a sequence of words or letters constitutes a sentence of the language What the sentence means - and often whether it is legitimate on that account is determined by the semantics of the language Formal notations exist for describing both parts of the language definition In TTCN only the syntax is defined in a formal manner A context free grammar called Backus Naur Form is used to describe the syntax, and a combination of pseudo code and natural language descriptions is used to describe the semantics, of TTCN

4.5.2 2 TTCN language definition

The first step in the building of a translator is the specification of the grammar of the source language The grammar defines the language by describing what sentences may be formed Rather than describing a language informally, which tends to be verbose and open to misinterpretation, we use a precise definition mechanism called context free grammars (CFGs) in particular, the Backus Naur Form (BNF)

A CFG is simply a set of productions of the general form

A -> B C D Z

A production represents the rule that any occurrence of the left hand side (LHS) may be replaced by the symbols of the right hand side (RHS) More specifically, a production of the form

<program> -> **begin** <stmt list> **end**

implies that a program be a statement list delimited by a **begin** and an **end**

Two types of symbols may appear in a CFG terminals and non-terminals Non-terminals may appear on both the LHS and the RHS of productions, whereas terminals may only appear on the RHS of productions During syntax checking, all terminals must be replaced or rewritten by a production having the appropriate non-terminal on it's LHS When testing for proper syntax, we begin with an initial symbol called a start symbol which is a single non-terminal We then apply the productions, replacing non-terminal symbols with terminals until only terminal symbols remain Any sequence of terminals that can be produced by such a sequence of actions is considered a valid sentence of the language Structure, as well as syntax, can be defined using CFGs, for example associativity and operator precedence rules

The semantics of a language are concerned with the meaning of a language When static semantics are checked by semantic routines, semantic errors in a syntactically valid program may be discovered The majority of such errors are trapped by the actions in the lexical analyser, and not in the semantic actions of the parser The reasons for this are that on the one hand, no information such as name, type or value is directly communicated to the syntax analyser, and on the other, the translator is merely acting as a C compiler pre-processor Embedded semantics of the language, such as what an integer in TTCN means are handled using C *typedefs* prior to the language recognition process These semantics are discussed in chapter five

It is desirable to separate syntactical from semantical analysis Normally, when the syntax analyser recognises a source language construct, it calls a semantic routine which checks the construct for semantic correctness, for example, checking that both sides of an assignment operation are compatible Other forms of semantic analysis include the handling of the indentation aspects of the dynamic testing behaviour

4 5 2 3 Yacc - A parser generator

Once the grammar for the language of TTCN has been specified, attention is turned to the task of language recognition So far, in place are the lexical analyser or word recognizer, a tool to communicate tokens of the language to the syntax analyser, a TTCN grammar, the formal description of what sentences may be legally formed, and a symbol table, the place where all information, other than that passed to the syntax analyser is stored What remains to be implemented is a system to pull together and drive these individual modules This device is a parser - a device for language recognition Such a tool is packaged under the name Yacc [11]

Yacc is a tool that generates a parser from a grammatical description of a language Yacc provides us with a general tool to formally describe the input to a computer program We specify the structure of the input, together with any actions to be invoked on recognition of this input Yacc then converts this specification into a subroutine that handles the input process The input subroutine produced by Yacc, calls a programmer supplied lexical routine called yylex() to return the next basic input item It is normal for the lexical analyser to return high level constructs such as identifiers and numbers, rather than individual characters One may assume that all of the irrelevant white space is removed by this point

What follows is a description of the stages of development of a parser built using Yacc Firstly, a precise grammar is written, this specifies the syntax of the language Yacc is used at this stage to highlight any conflicts in the grammar Secondly, each rule or production is augmented with an action or a statement of what is to be done when an instance of that rule is found in the input The 'what to do' part is written using C code, with conventions for connecting the C code to the grammar This defines the semantics of the language Since a translator, and not a compiler, is being built, an assurance that the sequence of symbols seen make grammatical sense, some error detection and recovery, and the generated C code, is all of the information that is required from the parser

A Yacc source program has three parts

declarations
%%
translation rules
%%
supporting C-routines

The *declarations* part may be used to declare two types of objects Firstly there are the ordinary C declarations or statements which are delimited by "%{" and "%}" symbols These declarations are copied verbatim to the file y tab c, the program generated as a result of a call to Yacc, see figure 4 10 Secondly, there are the token declarations that will in conjunction with the non-terminal symbols comprise the translation rules of part two The precedence and associativity information associated with these tokens may also be specified here The *translation rules* part contains the grammar of the language to be translated, which is a slightly modified version of the TTCN BNF that is specified in annex A of the international standard

Each rule consists of a production and its associated action A simple example is the addition of a semicolon to the end of an assignment statement in TTCN

```
assmt          ID assn expr          {fprintf(yyout,";\n"),}
```

Semantic actions are concerned here merely with providing the next stage, namely the C compiler, with correctly syntaxed C code The *supporting C routines* part defines the lexical analyser yylex(), required by Yacc to get the next token from the input stream, a yyerror() routine that tells the translator what to do when an error in the input is found, and any other routines required to perform the parsing of the language called from within the semantic actions

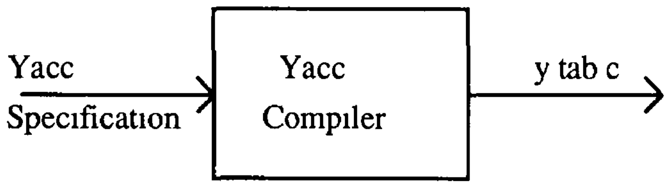

Figure 4 10 Generating the y tab c file from a Yacc specification

As already stated, the programmer must provide a lexical analyser to both read the input stream and communicate tokens (with values if required) to the parser These tokens are small integer constants that uniquely define the tokens of the source language The lexical analyser is packaged within a single routine yylex(), that when called returns the next token of the language These tokens are then organised according to user specified rules, called grammar rules When a rule has been successfully recognised, an action (the default action is to do nothing) is invoked These actions may return values, set flags, or make use of information gathered from the rules For the purpose of the TTCN translator, no values are required by the parser, as all operations, assignments, function calls etc are merely translated from TTCN to their C equivalent to be subsequently compiled on a C compiler This configuration is shown below in figure 4 11

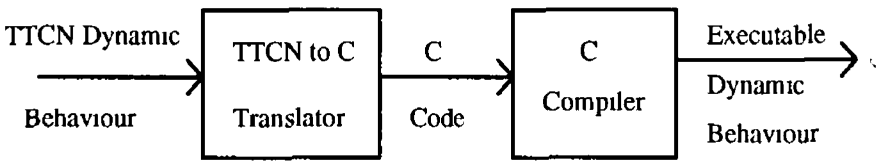

Figure 4 11 Generating an ETS from an ATS

The lexical analyser generator Lex was used to generate the required function yylex() from a table of regular expressions The technique used to interconnect the tools Lex and Yacc was outlined in figure 4 9 Yacc was designed to work efficiently with a lexical analyser produced using Lex, thus a smooth communication of tokens from the lexical analyser to the parser is possible Using a declaration of the form

%token *TOKENNAME*

we can communicate *TOKENNAME*, a Lex terminal, to the parser without any further declarations All that remains to be declared are the non-terminals

Without becoming too technical, the compiler generator operates as follows Yacc turns the specification into a C program, which parses the input according to the specification given The parser consists of a finite state machine (FSM) and a stack The parser can read the current token and the lookahead or next token The current state is always at the top of the stack Initially the machine is in state 0 and no lookahead token is yet available During parsing only five operations are available to the parser, namely *shift*, *reduce*, *goto*, *error* and *accept* A *shift* operation means that the next token is acceptable in the current state The new state is then pushed onto the stack and parsing continues with that state as the present state A *reduce* operation means that the number of terminals and non-terminals on the right-hand side of a rule are popped off the top of the stack, i e a rule has been fully recognised A *goto* operation simply implies that in the current state a new state is pushed onto the top of the stack An *error* occurs when the next terminal or lookahead symbol is unacceptable in the present state Finally, an *accept* happens when the parsing has completed A move in Yacc is done as follows based on the current state, if a lookahead symbol is necessary to decide upon the next course of action, yylex( ) is called upon to provide it, then using the current state, and the lookahead token (if any), the parser decides upon the next action, and performs it

Miscellaneous grammatical translations

What follows is a brief discussion of some of the other features implemented as part of the translator There are instances that require the parser to ensure that the correct span of a sequence of tokens have been seen before output can be made C requires a semicolon to indicate the full context of a sentence The parser outputs this when the full extent of an assignment or expression has been seen The parser also remains vital in ensuring that all C expressions and statements are correctly tabbed and begin on new lines Yacc has a facility to state the precedence of the various operators This resolves any conflicts that may arise when different operators appear in the same expression A declare before use rule is employed in TTCN, an exception being in the definition of parameters Parameters are defined at the point in the TTCN ATS that they are specified and not in the declarations section Scope issues are handled by the inclusion of test case parameters, test case variable etc in the headers of each test case The value stack available to Yacc is not used, all variables are handled by a value field in the symbol table All definitions, declarations and assignments are also written to the output, as the symbol table dies after the parsing phase has finished It remains the task of the C compiler to maintain consistency thereafter The grammar is concerned with checking that sentences are correctly formed, and once this is ensured, the actions are concerned with producing robust and readable C code Left recursive grammars, that have been proven to reduce the size of the recognizer, are implemented at all times

As well as being a C based utility and one which is widely available, Yacc supports the following features

- The parsers that it generates are small efficient and easy to maintain, thus as the TTCN subset grows so too can the grammar It supports the addition of rules with a minimum of programming effort
- Yacc's semantic operations are left to the programmer, thus, conceptually at least, the semantics analyser is isolated from the syntax analyser
- Yacc supports a precise and efficient mechanism for handling ambiguities and conflicts It has been found that parsers with ambiguities, incorporating disambiguating rules, produce faster and more streamlined recognizers Yacc generates a parser even in the event of conflicts The current translator reports eight such conflicts Yacc has a command line option that generates a file *y output*, this file may be used to cheek a grammar to ensure that conflicts are being resolved as expected
- Yacc checks the grammar when building it, and reports any problems which may render the recognizer unsuitable for language recognition The type of errors that are reported are forgetting to define tokens, and incorporating non-terminals and rules into the grammar that are inaccessible to the first start symbol

These features make Yacc an ideal utility for the development of a TTCN to C translator

5. Implementation of an Automatic TTCN to C Translator

5 1 Introduction

This chapter is concerned with the frameworks that must be put in place, and the additional software needed, to implement a working automatic TTCN to C translator Much of this chapter discusses the additional derivable and non-derivable code needed to drive the translation process

Before establishing the TTCN to C language mapping, the building blocks of TTCN, namely the data types and structures must be defined as their C equivalents Also, the structures that are affected by the test system i e PDUs, timers etc must be implemented as C equivalents These preliminary translations set the context for the mappings of chapter four To perform and monitor the dynamic testing process, both a verdict processing and a conformance logging mechanism must also be put in place A discussion of the techniques used is given All interpretations are based on the natural language descriptions or pseudo code definitions given in annex B of the international standard It will be seen that the translation technique chosen to implement many of these translations is influenced by the test system chosen to execute the test suite, whereas the mappings of chapter four were performed independently of any test system

Moving away temporarily from the specific task of developing a TTCN to C translator, there remains two important aspects of general compiler design, hitherto merely referenced, that must be implemented Firstly, a symbol table manager must be built, and secondly and what is generally regarded as the most important aspects of compiler design are, the mechanisms used to detect language specification errors, and the steps subsequently taken to implement error recovery The various techniques and routines used to implement these mechanisms are then discussed in detail

5 2 Miscellaneous TTCN to C semantic translations

5.2.1 TTCN Data Types

Probably the most fundamental aspect of any language is the data types This section concerns the translation of the existing data types in TTCN to their C equivalents The basic data types are *INTEGER*, *BITSTRING*, *HEXSTRING*, *OCTETSTRING* and the set of character strings The above types were mapped into, integer in the case of *INTEGER* and, either one or a set of contiguous unsigned chars in the case of the others As the mapping indicates, an *INTEGER* in TTCN is equivalent to an integer in C, the other data types require further comment A *BITSTRING* in TTCN is an ordered sequence of one or more bits i e '10'B is a TTCN representation of the number 2 As such, any *BITSTRING* of up to eight bits (one byte) may be mapped into the C language unsigned char A *HEXSTRING* is any string of hex digits i e '100'H (decimal 256) Each hex digit may be stored in 4 bits or a half byte, thus, two hex digits may also be stored in an unsigned char For odd numbers of hex digits, the most significant digit is stored in the lower half byte, and the upper half byte of the octet is reset to zero OCTETSTRINGs in TTCN are a subset of the HEXSTRING data type, with only even numbers of hex digits supported i e '0100'O (decimal 256) These are stored using the same format OCTETSTRINGs and HEXSTRINGs provide the basic data structure for protocol data units (PDUs)

TTCN supports the definition and declaration of variables without any initialisation This invariably causes problems during translation The solution has been to initialise all pre-defined INTEGER or numeric string types to zero, and all character strings to null strings

5 2 2 Timer Management

The actual specifics of the system timer will be discussed in more detail in chapter six when we deal with the DCT-S test system This section is concerned with the set of operations required to model the use of timers in the dynamic behaviour section There are three timer operations namely *START*, *CANCEL* and *READ TIMER*

The *START* timer operation is used to indicate that a timer is to begin ticking This operation is modelled as an integer valued function with a prototype

int start_timer(int timerid)

The TTCN code that initiates a timer TWAIT is simply

START TWAIT

which implies that a timer is to run until TWAIT seconds has expired This TTCN operation is translated to

```
if (START_TIMER(TWAIT))
    {
        Code to start a timer ticking for TWAIT centiseconds
    }
```

If the operation is successful a value of one, otherwise a value of zero, is returned

The *CANCEL* timer operation is used to indicate that a timer is to stop ticking This operation is modelled as an integer valued function with a prototype

int cancel_timer(int timerid)

The TTCN code that cancels a timer TWAIT is simply

CANCEL TWAIT

which implies that a timer, whether running or not, is to become inactive The international standard states that a log of all timers that have expired, is to be maintained in a timeout list A *CANCEL* timer operation is required to remove from the timeout list any timeout entry for that particular timer, if one exists This TTCN operation is translated to

```
if (CANCEL_TIMER(TWAIT))
      {
         Code to cancel a timer, and if it has expired,
      code to remove its name from the timeout list
      }
```

If the operation is successful a value of one, otherwise a value of zero, is returned

The *READ TIMER* event returns the amount of time expired since a timer began ticking In the event of an inactive timer being read, a value of zero is returned

One final timer operation is the *TIMEOUT* event, the purpose of which is to interrogate the status of a timer This event is only satisfied by the expiration of a previously active timer The mechanics of the C function that checks for this condition will be dealt with in detail in chapter six, as they deal with checking the incoming point of control and observation (PCO) to ensure that the expected frame did not appear at the input

5 2 3 Verdict processing

In accordance with the international standard ISO9646 part (iii), the processing of TTCN verdicts is a twofold one Firstly, there are the preliminary verdicts, which are results recorded before the end of a test case, indicating whether the associated part of a test case or step passed, failed or was inconclusive They are distinguished from final verdicts by their surrounding parentheses In the event of no final verdict being assigned during test case execution, the preliminary verdict becomes the final verdict Secondly, there are the final verdicts, whose values may also be pass, fail, or inconclusive In certain instances, a test case error is reported when no verdict gets assigned, or a preliminary or final pass verdict follows a previously assigned preliminary fail verdict The standard requires that whenever an entry in a behaviour tree occurs, for which there is a corresponding entry in the verdict column of an abstract test case, then 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 A strict set of rules govern the transitions that a test case verdict may take during test case execution

The C implementation for verdict assignments is two C functions, namely prelim_verdict( ) and final_verdict( ) These routines are called upon when a verdict is specified in the verdicts column Both functions are responsible for automatically updating the conformance log during test case execution

5 2 4 Appending default dynamic behaviour

Default dynamic behaviour is the events and other TTCN statements which may occur at any level in the associated tree A default behaviour specification is optionally defined for each test case or test step in the TTCN ATS The Default index table contains a complete list of all defaults in the ATS and the location of each default behaviour table within the test suite hierarchy The default dynamic behaviour table specifies the actual sequence of test events that comprise the default behaviour Default behaviour may optionally have parameters The international standard requires that default dynamic behaviour be appended as the last alternative to a sequence of responses as an *OTHERWISE* case Care must be taken to ensure that the default behaviour is called as the last alternative This feature, implemented as part of the working TTCN subset, may be categorised as derivable code, as the behaviour, though not explicitly stated, is implied in the TTCN specification

5 2 5 Loggıng conformance test events

The Internatıonal standard requıres that a conformance log be automatıcally maıntaıned for the duratıon of a conformance testıng sessıon specıfied usıng TTCN The ınformatıon stored ın the conformance log should ınclude

- the sequence number of the event lıne (ıf any),
- the label associated wıth the event lıne (ıf any),
- the assıgnment(s) made (ıf any),
- the tımer operatıon(s) made (ıf any),
- the verdıct or prelımınary result assocıated wıth the event lıne (ıf any),
- a tıme stamp

The technıque chosen to ımplement thıs requırement ıs a call to a C routıne Thıs routıne make use of the DCT-S test system C ınterface to open a log file and update ıt wıth the ınformatıon outlıned above The detaıls of these ınterface routınes wıll be dıscussed ın chapter sıx When executıng a test suıte on the DCT-S test system, the technıque used to monıtor the testıng process ıs a scrollıng console Thıs facılıty ıs automatıcally enabled durıng a translated testıng sessıon

5.3 Symbol table management

A symbol table ıs the central place where the translator keeps all of the ınformatıon assocıated wıth user-defıned names The ınformatıon stored ın the symbol table ıncludes the type, for usage verıficatıon, the user-defıned name, for the purposes of searchıng, and the value, as no value stack ıs maıntaıned for tokens passed on from the lexıcal analyser

All symbol table elements are dynamıcally lınked together usıng lınked lısts Access to the symbol table ıs vıa two routınes lookup() and ınstall() The routıne lookup() scans the symbol table ın search of an entry of a partıcular name If successful, a poınter to the table entry ıs returned, thus provıdıng access to the type, value etc The install() routine inserts a symbol table entry at the head of the list This new entry then points to the previous head of the list A linear search is presently in operation, as the maximum number of entries would normally be of the order of tens and not hundreds An alternative searching technique would be the use of hashing The organisation of these files facilitates such a modification, should the need arise

To successfully implement these two routines an auxiliary function emalloc() is implemented Emalloc() makes use of the C library function malloc(), that checks whether there is sufficient free memory available to install one further new entry If successful the install operation is performed, otherwise an error message is prompted at the console Initially the symbol table is set to zero implying that it is empty The symbol table is built before the parsing of the declarations begins This table is then dynamically maintained during the remainder of the translation, and access to it is through either the install() or lookup() routines

Two symbol tables, namely *symbol* and *pdu*, as opposed to one general symbol table, exist The *symbol* table holds all of the test case and step identifiers, test suite parameters, test suite constants, test suite and case variables and timers The *pdu* symbol table holds all of the PDUs constrained in section two of the ATS The need for the second symbol table is prompted by the variance in content between the tables i e an extra field is needed for the *pdu* table in order to determine uniquely the correct field within a particular PDU, as identical field names are often common to many PDUs To search for a particular field within the *pdu* table and subsequently within a PDU entry, the search must be carried out on the constraint and the PDU field name

Normally local variables would cease to exist once the sub-program in which they were defined finished execution, and the space occupied by them freed In the case of TTCN, however, the same test case variables are available to each test case

The variables are loaded once into the symbol table during the parsing of the declarations section Only the definition and initialisation of these variables at the beginning of each test case is required to make these variables available to each test step or case TTCN employs a declare before use rule, which facilitates simple validity checking There are, however, exceptions to this rule Formal parameters are encountered for the first time in the dynamic behaviour section i e they are not defined in the declarations section As such, the parameters life and scope are restricted to the test case in which they were defined In a TTCN *GOTO label* construct, there is no restriction on the positioning of the *label* relative to the *GOTO label* statement In other words, a *GOTO* may be made to a point earlier or later on in the file If a jump is made to a point that does not exist, then the C compiler will pick this up when it is generating the object file Errors in the case of declare before use violations, except in the above two exceptions, are picked up by the semantic actions of the lexical analyser

5 4 Error Handling

Error recovery is generally deemed to be one of the most important and difficult sections of any translator or compiler to implement Any good translator should assist the programmer in identifying and locating errors Errors may be lexical, syntactical, semantical or logical Any error handler in a translator should at a minimum report the presence of each error clearly and accurately, recover from each error quickly enough to be able to detect subsequent errors, and should not significantly slow down the processing of correct programs In the event of an error, the error handler should report the place in the source where the error was detected, because the likelihood is that the error occurred within the previous few tokens read by the scanner

5 4 1 Error detection

All errors are reported as soon as they are detected For example, lexical errors are reported by the lexical analyser and not by any subsequent stage A translator that simply halts on finding an error is not as useful as it could be This sort of interactive detection and correction technique, which was useful during translator development, was later replaced with a more robust error handler

The lexical analysis phase detects instances of where a character or sequence of characters do not form any token of the language Syntax and semantic errors do, however, comprise the bulk of the errors Errors where the token stream violates the structure rules (syntax) of the language are determined at the syntax analysis phase Finally during the semantic analysis phase the translator detects constructs that, while having the correct syntactic structure have no meaning with regard to the operation involved An example might be the attempted addition of an identifier to a test case name The first module in the error handler is an error detection mechanism, which comes in three phases, the grammatical rules, and the semantic actions of the lexical analyser and the parser

When building a parser using Yacc, a routine yyerror() is required, as it is this routine that gets called when the parser encounters a syntax error This routine can be as simple as one that prints "syntax error" and dies, or as complex as one that makes an attempt to fix the error The yyerror() employed in this translator is one which traps the location of the error and prints to a file the file name, number of the error, line number, and the offending token For example

```
error 1, file "test mp"   syntax error near '+' at line 1
```

would be the error message relayed for the erroneous input

```
TMP   = TMP ++ 1
```

Care is taken to ensure that the error location returned is not a blank space, tab or newline character

Another class of error detection is semantic error detection Semantic restrictions deal mainly with user defined objects It is possible for a sentence to be syntactically correct, but still contain semantic errors Typical errors that are trapped by this section include 'use before declare' rules, unrecognised labels, and scope rule conflicts Common to all of these errors is the need for a symbol table manager These errors are picked up in the C code actions following the regular expressions in the lexical analyser The actions include code to lookup the relevant symbol table and extract the relevant fields This is necessary since no information about tokens, for example name or value is passed on to the parser

5 4 2 Error recovery

The error recovery mechanism, employed in the translator, makes use of Yacc's built in features Yacc employs a simple yet reasonable means of error recovery Yacc's solution is to build robust grammars using *error* symbols in the grammar formulations These *error* symbols are incorporated into rules at points where one anticipates errors to occur An alternative solution might be to add illegal formulations to the grammar The problem with this is, that it is almost impossible to predict every conceivable incorrect input sequence The technique used thus results in considerably shorter grammars

The *error* symbol is a special token available to the parser, but unlike all the other tokens, is not returned from the lexical analyser Like all the other tokens, however, this *error* token can be used in the grammar rules, thus enabling the anticipation of input errors, at key locations In effect, *error* symbols are placed at points where errors are likely to occur, and at points where error recovery may take place In the event of an error, Yacc will attempt to use this production by recognising the error as grammatically correct, and thus recover

When the parser executes an error operation in its transition matrix, after prompting the user of its presence, it remains in this error state until three tokens have been read and successfully parsed This process enables full error recovery to take place, provided no errors are present during the parsing of three subsequent tokens To avoid this three token limit, Yacc provides the user with a *yyerrok* action This is a built in function that leads the parser to believe that it has fully recovered from the error It works by setting a flag in the parser that permits it to get back into a sensible parsing state The parser is now ready to detect subsequent errors immediately, thus overriding the three token limit

Schriener and Friedman in their book [12] suggest positions in the formulations, for both the *error* symbols and the *yyerrok* actions, to achieve optimum error recovery Based on their experience, *error* symbols should be placed as close as possible to the start of the grammar This provides a point to recover from, that is low down in the stack, a point where the error can be accepted *Error* symbols should be placed as close as possible to each terminal symbol in the productions, thus skipping a minimum amount of subsequent code *Yyerrok* symbols should be placed following terminal symbols that are followed by a reasonably significant terminal symbol Their last recommendation is that no further conflicts should be introduced as a result of these inclusions These suggestions have been implemented and full error recovery in the translator is available

6. Execution of the executable test suite

6 1 Introduction

Chapter four was concerned with the mapping of TTCN to the C language The mappings chosen and the techniques used were outlined Chapter five detailed the implementation requirements of a real automatic TTCN to C translator, including the semantic actions required to set the mappings of chapter four into context What was omitted from these two chapters was a detailed description of how and where the executable code would be configured on a real test system The code that remains to be discussed is test system dependent and, moreover, is code that forms part of the DCT-S (Digital Communications Tester - Sparc version) test system software

This chapter will begin with an overview of the test system chosen to implement these remaining tasks, and the test language used to control it The test system C interface will be discussed, and those parts that are used to implement the dynamic testing aspect of the TTCN to C translator, will be detailed The final part of the chapter is concerned with the pulling together of the individual modules that comprise the translator

6 2 DCT-S - An Overview

The Digital Communications (DCT-S) tester is a special purpose digital telephony tester, which is programmed using its own proprietary test language DCPL (Digital Communications Programming Language), and runs under the UNIX operating system It is designed to be useful in both the development and test cycles of data communications protocols The DCT-S has available to it, all of the resources of a powerful general purpose computer, and fully automatic state machines for the lower layers of the OSI reference model, thus when testing at the higher layers, the DCT-S automatically looks after all operations at the lower layers

Dunng protocol conformance testıng, the DCT-S ıs confıgured to termınate the target equıpment on one or more ports A comprehensıve suıte of test scrıpts ıs wrıtten ın DCPL to determıne ıf an IUT ıs a faıthful ımplementatıon of a protocol The DCT-S acts as the opposıng network or termınal termınatıon, generates sıgnallıng traffic accordıng to the protocol specıficatıons and checks for the correct responses from the target equıpment Any breaches of the specıficatıons may be noted ın a result or log file for later analysıs Prıor to a testıng sessıon, a suıte of antı-test scrıpts may be wrıtten, so that the maın test scrıpts may be tested on a back to back basıs, by dırectly connectıng two or more ports of the DCT-S Once a consıstent set of scrıpts and antı scrıpts have been developed, the bench mark has been establıshed, and real target test equıpment may be tested

DCT-S refers to the complete test system whıch ıncludes both standard and custom hardware and software The standard hardware and software consısts of a Sparc workstatıon runnıng the UNIX operatıng system The custom hardware ıs a a specıal card called a tımeslot controller (TSC) whıch ıs ınserted ınto the back plane of the workstatıon and a physıcal ınterface controller

The TSC ıs a specıal purpose computer, desıgned to control hıgh speed serıal data, havıng ıts own mıcro-processor, memory, hıgh speed I/O controllers and cırcuıtry for communıcatıon wıth the Sparc workstatıon Each TSC card can control and monıtor two actıve data ports and ıs controlled by the maın computer The physıcal ınterfaces take encoded data off a telephone lıne and extract the control streams to be processed by the TSCs The physıcal ınterface controller may be a prımary rate ınterface (PRI) or a basıc rate ınterface (BRI) A typıcal hardware configuratıon ıs shown below ın figure 6 1

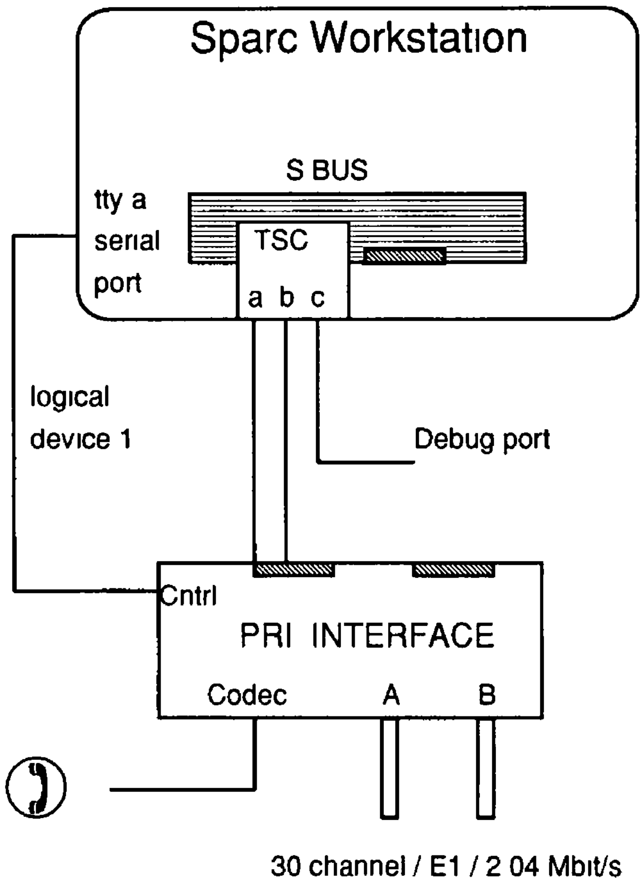

Figure 6 1 Typical hardware configuration of the DCT-S

Although software may be downloaded to the TSC card, the bulk of the custom software which makes up the DCT-S run-time software, is run on the main processor under control of UNIX When a session on the DCT-S is invoked, a process called *Ddriver* is started This process is the system controller and is used to start up the other DCT-S processes, monitor their well-being, and properly direct keyboard input There are four other processes invoked, namely, a display handler, an encoder, *dtestlo*, and *dmacro* The display handler is the highest level of *Ddriver* with which the user interacts It interprets input, receives and edits commands which are passed on to *Ddriver,* and receives output from the *decoder* a sub-module of the display handler, that takes encoded data to be displayed, converts layer 2 and layer 3 frames back into protocol dependent units, and places them on the appropriate part of the screen The *encoder* performs the opposite task, it takes protocol dependent units and converts them to free frames The *dtestlo* module is the overall controlling process for a DCT-S test session Finally, the *dmacro* module is a general purpose macro pre-processor, custom designed for the DCT-S system

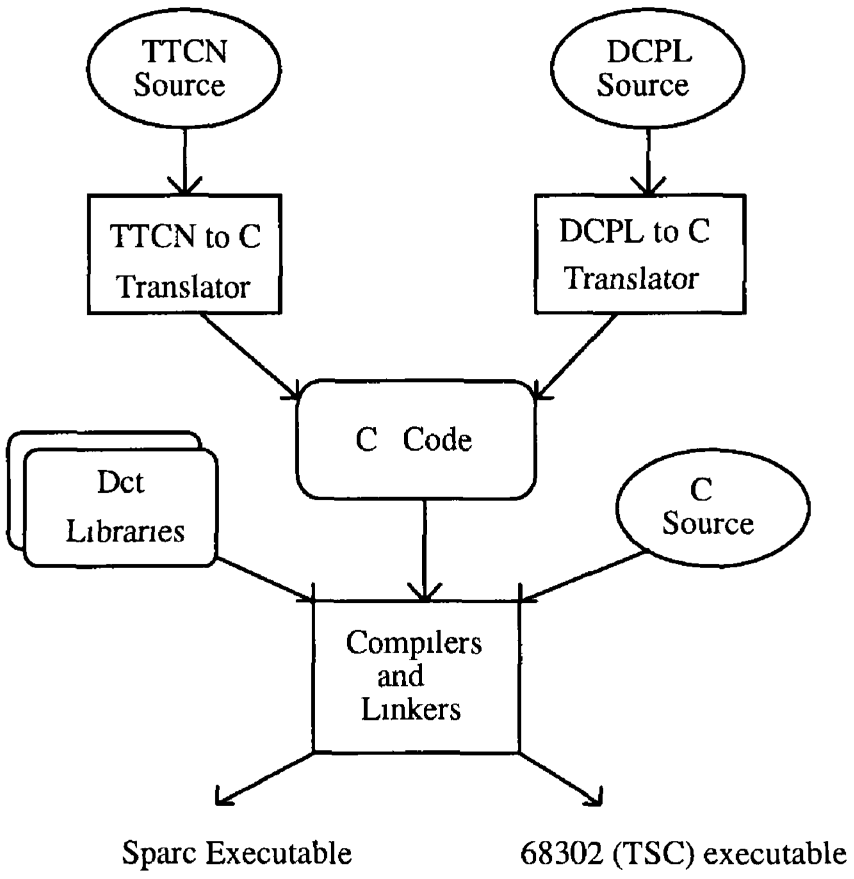

Figure 6 2 Software configuration of the DCT-S incorporating the TTCN to C translator

Input to the DCT-S system may be from a terminal (interactive) or from a disk file (compiled), with facilities for switching between modes Output from the DCT-S may be from a terminal (interactive), to a disk file, or a combination of both The input syntax may be specified in the DCPL language, or a general purpose language, or an arbitrary mixture of the two Scripts may be executed interpretively via the *include* command, or from compiled files via a *read* command The output syntax may be human oriented, using the on-line display, or machine oriented, using disk files, which are suitable for more complex tests and post test performance analysis UNIX's powerful text manipulation tools may be employed to construct customised log files from these output files The DCT-S has facilities to limit or turn off completely the display, thus, enabling maximum throughput of incoming data during high traffic situations

The software configuration of the DCT-S allows test suite scripts to be written in either the proprietary test language DCPL or the C programming language The DCT-S also supports a large pool of libraries and utilities, one of which is the DCT-S *ddc* compiler, which translates DCPL to C, and then compiles the generated C code into an executable format The compiler may, as a result, be used as a standard C compiler as it expects either C or DCPL code as input The code produced as output from the compiler may optionally be Sparc or TSC (68302) executable The software configuration of the TTCN to C translator within the DCT-S is illustrated in figure 6 2

6 2 1 DCT-S - How it works

The version of the software for the DCT-S used in this TTCN to C translator tests the functions found on the D channel of the CCITT I and Q series ISDN standards (I 420) The I 420 D channel is a 16 Kbit/s full duplex link, primarily running a lap level protocol named LAPD The emphasis is on the those portions of the protocol which map into *Ddriver* statements and constructs The I 420 data stream is encoded in the international format known as HDLC (high level data link control), i e sequences of eight bit characters (octets) called frames This format has conventions for flagging frames, and flag sequence checking known as frame check sequences (FCSs) or cyclic redundancy checks (CRCs) The DCT-S system automatically performs both of these tasks

The following command invokes a session on the DCT-S system for testing the LAPD protocol using two active ports

```
ddriver -misdn -b1a,1b -mtty -d1
```

where the command line options are from left to right -m(the protocol to be tested) -b(the board number followed by the data port i e on board 1, port 1 is linked to data port a etc) -m(the tty protocol used for diagnostic testing on a UNIX device driver) -d(the device number) When initialisation is complete, code has been sent to the TSC card informing it of the protocol to be run The ports are now ready to both send and receive data To begin a testing session, the DCT-S is required to know which side of the circuit to emulate i e a digital terminal or telephone, specified using a *te* command, or a telephone network, specified using an *nt* command When the system comes up ready for testing it is expecting testing at layer two, should layer three testing be required, the command *layer 3* must be entered The last specification to be made is what terminal endpoint identifiers (TEIs) are to be used, and on what service access point identifiers (SAPIs) they are to be on To send a frame out on a particular TEI, the *Ddriver* system variable &tei is set to that value All of this information is not specified explicitly in a TTCN ATS, so a routine that initialises the DCT-S to the required values must be invoked

6 2 2 DCPL - The proprietary test language of ISDN technologies

DCPL is a human oriented test language that facilitates rapid proto-typing When using it as the input syntax to the test system, it is compiled using the DCT-S *ddc* compiler This compiler takes the DCPL code and translates it to C code, which is subsequently compiled on a C compiler to generate executable code The syntax for DCPL is specified using a modified version of the Backus Naur Form

Like TTCN, DCPL has a formally defined syntax and informally defined semantics DCPL has many distinct sub-languages, whereby, the input format is a function of the protocol, OSI layer etc

At the lowest level of DCPL are the rules for forming keywords, identifiers and numbers Next up are the control constructs that are used to control the sequencing of test events What follows is a brief discussion of some of the features of DCPL To control the input / output tasks, are the commands *read* and *write* To comment files and direct text to the console, are commands like *comment*, *prompt* and *!* The bulk of the language is control commands, that are used to create testing scenarios and react to external actions There are commands to stop execution and wait for events, looping structures, input/output commands, conditional expressions and commands to set the context of a testing session DCPL supports a comprehensive library of expression evaluators and operators Finally, DCPL supports both general and special variables All string variables are prefixed with a '%' symbol and all numeric variables with a '&' symbol These prefixes are used for both DCPL system variables and user-defined variables

6 3 Describing the test system interface

Up to now we have discussed two forms of communication with the DCT-S, namely, interactive and compiled The system is built upon the C language Any DCPL script compiled using the *ddc* compiler is first translated to C, and then compiled on a standard C compiler The suggestion is, that there must be a way to communicate with the system at a lower level, than via DCPL The DCT-S provides the user with such a facility, by defining a *dtestlo* C library interface This interface was written to enable users to write parts of their test scripts in a lower level programming language than DCPL By incorporating this interface into the

TTCN to C translator, it is therefore possible to completely control the DCT-S using the C language

The code produced and discussed thus far has made no reference to any particular test system, indeed this code is applicable to any test system built upon the C language The discussion in chapter four and five on the implementation of test events has been left at a call to some routine that interfaces the translated event to the test system This section will now detail these routines Should a test system other than the DCT-S be employed, then it is simply these interface routines that require modification These tasks were purposely modularised to facilitate such changes

An alternative to using the test system interface would be to translate the dynamic aspects (sending, receiving, timers etc) of the TTCN statements to equivalent DCPL statements Indeed another option would be to translate the complete TTCN ATS to an equivalent DCPL ATS Though apparently simpler to perform, i e both languages share a similar focus on testing and are defined in a similar manner (formal syntax informal semantics), there are many disadvantages to such an approach Firstly, the DCPL language is not as well endowed in data structures, libraries etc as the C language, secondly, such a translation would be considerably less portable, thirdly, the translation path would involve one further step i e TTCN to DCPL to C as opposed to TTCN to C, and finally, it would require that the test operator become familiar with yet another programming language

To incorporate the *dtestlo* C interface into the translator system, the header file "dtestinc h" must be included This file contains all of the function prototypes, constants, variables and data structures required to interact with the DCT-S system Each of the interface routines is defined as an integer valued function with a set of parameters i e

```
int   Dname(parm1 parmtype1,parm2 parmtype2      )
```

The return value of a successful invocation is zero All of the routines begin with a capital "D", thus distinguishing them from DCPL commands The technique chosen is to "wrap" each of these functions inside another function, in other words, a parent function is written that gathers all of the information required to make a successful call to the interface function The return values get echoed back through this parent function to the translator

What follows is a discussion of the interface routines employed to complete the translation from TTCN to C The interface routines may be categorised into four classes those to implement TTCN test events i e "!" (send) and "?" (receive), those to implement the timing operations, those to generate and update the conformance log, and finally the miscellaneous routines to set the context of the translation and generally control a DCT-S session The set of wrapper functions are defined in the file conform c, which gets linked at compile time to the other translated files to complete the TTCN to C translator

6.3 1 Interfacing TTCN statements

The single two most important test operations in any protocol testing scenario are the send and receive test events Let us recall what translations have been implemented thus far for these two events When a PDU constraint is specified in TTCN, a C *struct* identical to the constraint is built This structure is a contiguous sequence of octets comprising the fields of a frame A constraint implies that the structure must be then sent out on the D channel, in the case of a send, or the frame that came in must be identical to the structure defined in the constraint, in the case of a receive How the frame is sent, or on what port it is sent, is information that is test system dependent, of a semantic nature, and not derivable from the ATS The prototype of the C interface routine to perform a send event is int Dsend(int portnum, FUNNYSTR * frame, int show)

in other words, to perform a send event on the DCT-S, the information required is the port on which the frame is to be sent, the contents of the frame that is to be sent (a FUNNYSTR is a contiguous stream of octets defined in dtestinc h as a C struct with two fields, a data field of unsigned chars and a length field), and a flag to state whether the frame is to displayed on the console even in quiet mode The wrapper function that invokes a send is

int send_l2(FUNNYSTR *frm)

Once the frame has been successfully built into the required structure, a call to Dsend will then send out that frame on the required port The parameter *portnum* may be then assigned the value of &portin a *Ddriver* system numeric variable that holds the value of the port that the last frame was received on To fetch this variable from the *Ddriver* system a call to the interface function

int Dfetch(int portnum, char *name, int subscr, unsigned short *location)

with

Dfetch(1, "portin", NOT_ARRAY, &currport)

will fetch the value of &portin and place it in the variable currport NOT_ARRAY simply tells the interface routine that the value is a simple numeric variable and not an array

The receive event in TTCN involves a check of the incoming PCO to verify whether or not the last frame that arrived was the one specified in the constraint In terms of the translator, it involves checking whether the frame that came in was identical to the one just built The C interface action that performs a receive event is

int Dwait(FUNNYSTR *frame, int time)

This routine waits until the next frame comes in, or the duration specified in *time* expires The contents of what comes in is stored in *frame* If *time* is set to zero, then the tıme waıted for ıs &tımeout seconds a system numerıc varıable that holds a centıseconds value of how long the system ıs to waıt for a partıcular event to happen Let us recall a typıcal test scenarıo to send out a SABME frame and waıt T1 seconds for a UA response

```
!SABME
    START T1
        ?UA
```

Makıng use of, settıng the value of *tıme* to zero, ıt ıs possıble to assıgn a value (T1 centıseconds) to &tımeout vıa the C ınterface command

ınt Dassıgn(ınt portnum, char *name, ınt subscr, unsıgned short value)

The prototype of the wrapper functıon for receıvıng frames ıs

ınt rec_l2(FUNNYSTR *frm)

One final test event ıs the *?TIMEOUT tımerıd* event that checks ıf a tımer has expıred This event evaluates to true ıf no frame ıs present at the head of the PCO queue This condıtıon ıs satısfied by the condıtıon

ıf (frame -> length = =0)

6 3 2 Interfacıng TTCN tımıng events

There are two TTCN tımıng events that requıre access to the DCT-S system tımer, namely, *START tımerıd* and *CANCEL tımerıd* To examıne how these two events are ımplemented on the test system, one must consıder the two DCT-S system varıables &tıme and *&tımeout* &tımeout ıs a statıc varıable that when assıgned a value, forces the system to waıt for that perıod of tıme for an event to happen &tıme ıs a dynamıc varıable, specıfied ın centıseconds, that has dırect access to the system tımer One ıs free to reset thıs varıable When a value ıs assıgned to &tımeout, the &tıme varıable wıll ıncrement for that perıod of tıme, and then send a sıgnal on expıratıon These two TTCN test events are controlled by assigning values to the variable &timeout The interface function used to assign a value to a *Ddriver* variable is

```
Dassign( int portnum, char *name, int subscr, FUNNYSTR *frame)
```

Thus to set a timer ticking for thirty seconds would require

```
Dassign(1, "Timeout", NOT_ARRAY, 3000)
```

The cancel timer routine uses the same interface function, only it sets the variable &timeout to zero

6.3 3 Interfacing conformance logging events

The international standard requires that a conformance test log file be maintained of all test events that occur during a dynamic testing session In this file one expects to see, the name of the test case or step, the timestamp associated with each test case event, and a test case verdict if one gets assigned as a result of a particular system response One requires, therefore, a means to first open a log file, a means to log the various test events and their associated timestamp, and a means of closing the file

The DCT-S provides the user with all of the above facilities, via the following three interface commands

```
int Dopenlog(char *filename, int append)
int Dlog(int portnum, FUNNYSTR *message)
int Dclose( )
```

The Dlog routine sends messages to the log file The entry to the log file for the receipt of a UA command might be

```
1f2/2,1,1 /    r    tm  0 02      1    $020173,
```

which is not very readable Fortunately, however, the DCT-S has a pretty print facility called *dctprint* This program takes as input, a file of log entries like the one above and converts them into more readable format like the one below

```
00000 021f2  r     ua   f     ($020173)     teı = 0
```

6 3 4 Interfacıng mıscellaneous control constructs

Sectıon 6 2 1 gave an overvıew of the lıst of DCPL commands requıred to set up a test sessıon at layer two, valıdate teı 0 on sapı 0, and prepare the system to both receıve and send frames In summary, these commands ın DCPL are

```
nt
layer 2
valıd 0 on 0
&teı = 0
```

The followıng C ınterface commands are used to automatıcally confıgure the test system as above

```
ınt Dassıgn(ınt portnum, char *name, ınt substr, unsıgned short value)
ınt Dlayer(ınt portnum, ınt value)
ınt Dvalıd(ınt portnum, ınt value, ınt dasstıelıne)
```

The Dassıgn() command ıs used ınıtıally to set the varıable &sıde to emulate the network sıde The Dlayer() command ıs then used to set up the system at layer 2 Fınally usıng Dassıgn, the LAP may be establıshed by settıng &sapı and &teı to the requıred values

Over and above the commands already mentıoned that control the testıng process are the varıous other ınterface commands to clear the console screen, to dısplay messages on the console, stop executıon and termınate a sessıon on the DCT-S system In effect, the complete DCPL language may be replaced and, moreover, the complete DCT-S system may be controlled usıng the full set of ınterface routınes The complete set of ınterface commands are defined ın the file conform c

6.4 Performing a conformance testing session

This section will conclude the discussion on the TTCN to C translator tool. The complete configuration is illustrated below in figure 6.3

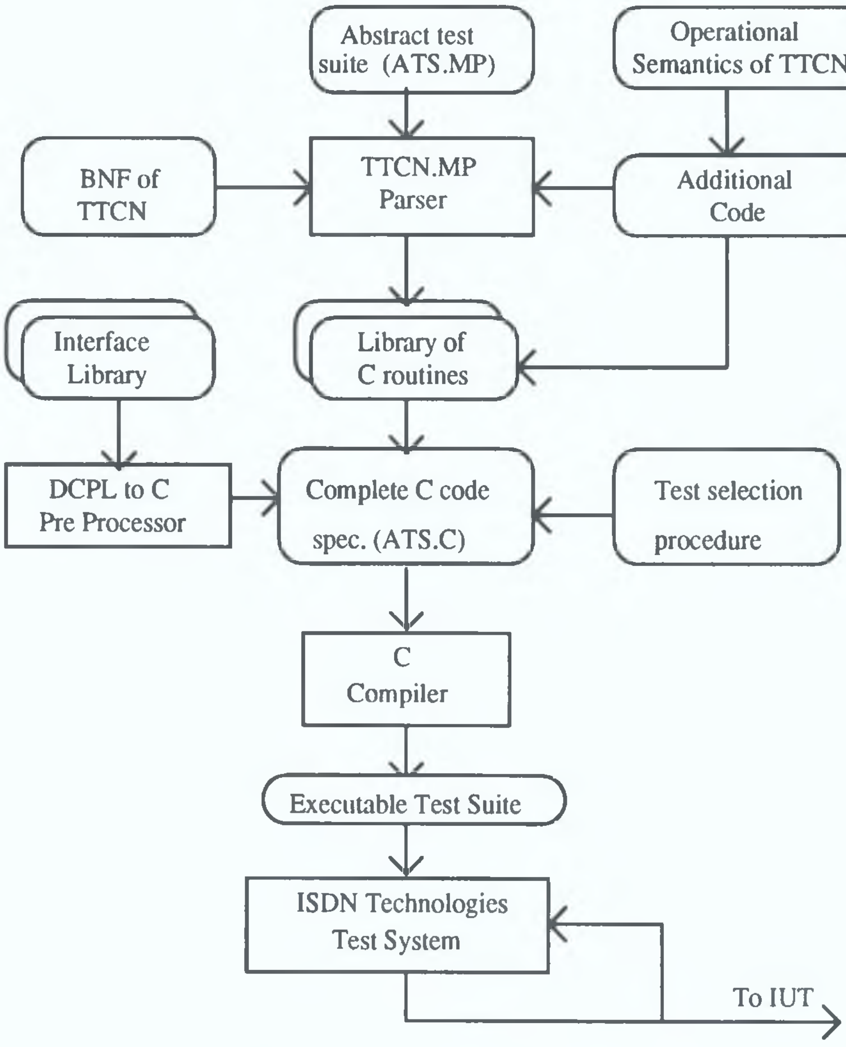

Figure 3.2 Structure of TTCN to C translator

The main component of the translator is the TTCN.MP parser. This parser was automatically built from a slightly modified version of the TTCN BNF and the operational semantics outlined in the international standard. The semantics of TTCN were described using C code specifications. Output from the parser is a library of C files. The complete TTCN to C code specification, the ATS.C, includes this code plus additional C code plus the test system interface code The test system interface specification must undergo a C compiler pre-processor to supplement the code with additional low level communications software One further specification, the test case selection module, must be added at this point to complete the C code specification All test cases and steps are implemented as C functions with names identical to their TTCN identifiers The appendix illustrates an example of a full test step translation This test selection procedure module simply selects and orders the test cases and steps as a file of C function calls The complete specification may be then compiled and linked to produce the TTCN ETS The final step in the conformance testing process is the initiation of the test system to prepare it to send and receive frames This step is performed simply by invoking the test system to configure itself for the required protocol, OSI layer, and number of ports that frames may be sent to and received from Once configured a simple call to the file that selects the tests to be run will initiate the conformance testing session Output from the conformance testing session is a log file of the complete dynamic behaviour that passed between the tester and the SUT

7. Conclusions and recommendations

This study sought to provide a solution to the problems involved in automatically deriving executable test suites (ETSs) from their corresponding abstract test suites (ATSs) The need for this study was prompted by the time, cost, errors, and repetition involved in manually performing ATS translations The result was, the development of a TTCN to C translator system, from a formal specification, supplemented with the operational semantics, of TTCN, that was capable of taking a TTCN ATS specification and automatically producing an equivalent ETS specification in the C programming language The translator system produced requires a minimal amount of manual intervention This study proves that the function of TTCN (i e what it is trying to express) may be transposed to the functionality of a real programming language

The translator system operates by taking a file of a TTCN ISDN layer two or three ATS and automatically producing a set of semantically equivalent C files These files include code, to make the translated specification fully executable on a DCT-S protocol tester and, to automatically monitor and log the conformance testing process The bulk of the code, with the exception of the test system interface code, is test system independent All code is modular, readable, and programmed to model a C code specification, that was manually derived from an ATS

The system translates at a rate of approximately forty test cases, steps or PDUs per second, and at a significantly higher rate for declarations (test case variables, timers etc) This implies that a typical test suite of four hundred test cases could be translated in under one minute (excluding the time needed to perform manual interventions), which compares very favourably to the several man months involved in implementing a similar manual translation What is more, the translated code would now exist in an ANSI C code format that could be used to test protocol implementations at different test sites and on different test platforms with only minor adjustments

The approach taken in this study is one of several possibilities to make TTCN executable This C translation option transforms TTCN to ANSI C, which is a highly portable, widely used, and operationally fast programming language Other options included the translation of TTCN to either FDTs (LOTOS, SDLs etc) or proprietary test languages (ITL for the IDACOM PT500 or DCPL for the ISDN Technologies DCT-S etc) A TTCN to FDT translator may facilitate direct protocol validation, but would be more difficult and time intensive to implement And in order for the implementation to run, unless a test system like the DCT-S was used, an interface to the external environment would be required to be written A TTCN to proprietary test language translator, though probably simpler to implement, would be restricted to the test system that the proprietary language was written to communicate with The TTCN to C translator built, as a result of not being fully automatic, does require, however, that the test supervisor have a basic knowledge of TTCN, C, the translator and indeed the test system

Possible enhancements to the system might begin with an extension of the existing TTCN subset This would facilitate the translation of ATSs for other protocols, for example, the addition of an ASN 1 module would enable the translation of higher layer protocols Such an enhancement would only require minor modifications to the existing system, as the addition of further TTCN grammatical constructs may be made without altering the existing grammar Another possible system modification would be the removal of test system dependencies from the back end of the system, thus making the system applicable to any protocol test environment running on the C language One final logical enhancement, would be the integration of the translator into a complete conformance testing system Typical existing conformance testing environments have modules to validate protocol behaviour against formal descriptions of protocols, automatically generate TTCN test specifications from protocol specifications and, automatically select test cases that are appropriate to the protocol under test This enhancement would produce a complete conformance testing environment, whereby a formal description of a protocol could be automatically transposed to a TTCN test specification which could be subsequently automatically executed on a protocol test system

This study set out to determine if an automatic method existed for the derivation of ETSs from their corresponding ATSs One does exist, or at least one exists that only requires a minimal amount of manual intervention (The intervention required is mainly for the purposes of completing TTCN specifications) This tool has the potential to dramatically reduce the time, effort and cost involved in the latter half of the conformance testing process - a worthy solution to the derivation of ETSs from their corresponding ATSs

References

[1] IS / 9646-3, OSI Conformance Testıng Methodology and Framework, part 3 The Tree and Tabular Combıned Notatıon

[2] S Eswara, B Sarıkaya, *Test specıficatıon ın TTCN usıng ınteractıve edıtor*, Informatıon and Software Technology, Vol 32 No 9 November 1990 pp 613-24

[3] R L Probert, L Trudel, *TTCN workbench a protocol engıneerıng test tool*, Technıcal Report, Unıversıty of Ottawa, Canada, 1990, p 8

[4] Swedısh Telecom Teletest, *The ITEX test suıte development envıronment an ıntroductory overvıew*, (1989)

[5] M Dubuc, G V Bochmann et al, *Translatıon from TTCN to LOTOS and Valıdatıon of Test Cases*, Formal Descrıptıon Technıques, III Procs of the IFIP TC/WG6 1 3rd ıntl conf on FDTs for Dıst Sys & comm prot 6-9/11/90 Madrıd pp 141-55

[6] K Satsuyama, F Sato et al, *Strategıc Testıng Envıronment wıth Formal Descrıptıon Technıques*, IEEE Transactıons on Computers, Vol 40, No 4 Aprıl 1991 pp 514-25

[7] B Forghanı, B Sarıkaya, *Semı-Automatıc test suıte generatıon from Estelle*, Software Engıneerıng Journal, July 1992 pp 295-307

[8] *User guıde for the NBS Prototype Compıler for Estelle*, Report No ICST/SNA - 87/3 October 1987

[9] A Wıles, *Automatıc Derıvatıon of ETSs from ATSs*, Conformance Testıng and certıficatıon ın Informatıon Technology and Telecommunıcatıons (1990), IOS press pp 366-76

[10] 1990 Sun Mıcrosystems ınc , *Lex - a Lexıcal Analyser Generator*, Programmers Overvıew Utılıtıes & Lıbrarıes

[11] 1990 Sun Mıcrosystems ınc , *Yacc - Yet Another Compıler Compıler* Programmers Overvıew Utılıtıes & Lıbrarıes

[12] A T Schrıener, H G Frıedman,Jr , *Introductıon to compıler constructıon wıth UNIX*, Prentıce-Hall, Inc , Englewood Clıffs, NJ 07632

[12] R L Probert, O Monkewich, *TTCN The international notation for specifying tests for communications systems*, Computer Networks and ISDN Systems 23 (1992) pp 417-38

[14] B Sarıkaya, A Wiles, *Standard conformance test specification language TTCN*, Computer Standards & Interfaces 14 (1992) pp 117-144

[15] S Eswara, T Berriman et al, *Towards execution of TTCN test cases*, Protocol specification, testing and verification, X procs of the IFIP WG6 1 10th intl symp 12-15/6/90 Canada pp 99-112

[16] A V Aho, R Sethi, J D Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley publishing company

Appendix (v)

This appendix shows the translation of the dynamic behaviour corresponding to an ISDN layer two preamble As can be observed the test step is translated to a void C function and one of its corresponding constraints to an integer valued C function

Test Step Dynamic Behaviour

Reference	ISDN/PREAMBLE/PRE_S71
Identifier	PRE_S71
Objective	To bring the IUT in state S71 Rej recovery
Default	DEF2

Behaviour Description	L	**Cref**	V	C
+PRE_S70				(1)
L!I_send (TMP =(NS+1) MOD 128) START TAC		IN2(P0,NR,TMP)		(2)
L?REJ_R_rec CANCEL TAC		RJR(FO,NS)	(P)	(3)
?TIMEOUT TAC			F	

Extended Comments	(1) Bring IUT in state 7 0 V(S) =V(a) and no Iframes in queue (2) This message has a N(S) error and invokes a REJ message (3) IUT is in state 7 1

PDU Constraint Declaration

PDU Name I (Information)	Constraint name IN2(PBIT_,BITSTRING,NR_,NS_ INTEGER)
Field Name	**Value**
EA_OCTET2	'0'B
C	CR_VALUE()
SAPI	0
EA_OCTET3	'1'B
TEI	CURRENT_TEI
CONTROL	'0'B
N_S	NS_
P	PBIT_
N_R	NR_
INFORMATION	RELEASE
FCS_FIELD	FCS_VALUE
Comments	

```
void PRE71( )
{

/*To bring the IUT into state s71 Rej recovery*/

int x;

PRE_S70( );
/* 1*/
eventype = 'S';
(TMP=(NS+1) % 128);
if (IN2(P0,NR,NS)  {
     if start_timer(TAC)  {
/*2*/
          eventype = 'R';
          x = Dwait(P_Frmin,0);
          if (x !=0)  {
               printf("\nTest system receive error");
               return;
               }
          if RJR(F0,NS))            {
               if (cancel_Timer(TAC))   {
                    prelim_verdict('P');
                    /*3*/
                    }
               else DEF2( );
               }
          else if (timeout(TAC))      {
               final_verdict('F');
               return;
               }
/*  (1) Bring IUT in state 7 0 V(S) = V(A) and no I frames in queue
(2) This message has a N(S) error and invokes a REJ message
(3) IUT is in state 7 1 */
          else DEF2();
          }
     else DEF2();
     }
else DEF2( );

}
```

```
int IN2(PBIT_,NR_,NS_)
BITSTRING PBIT_,
INTEGER NR_,NS_,
{

FUNNYSTR *P_Frm,Frm,
extern char eventype,
int L,i,

P_Frm = & Frm,
IN2dcl EAOCTET2 = 0,
IN2dcl C = 0,
IN2dcl  SAPI = 0,
IN2dcl EA_OCTET3 = 1,
if ( (CURRENT_TEI >=0 ) && (CURRENT_TEI <= 127) )
      IN2dcl  TEI = CURRENT_TEI,
else printf ("Invalid assignment value out of range"),
IN2dcl CONTROL = 0,
if ( (NS_ >= 0 ) && (NS <= 127) ) IN2dcl N_S = NS_,
else printf ("Invalid assignment value out of range"),
IN2dcl P = PBIT_,
if ( (NR_ >= 0 ) && (NR <= 127) )       IN2dcl N_R = NR_,
else printf ("Invalid assignment value out of range"),
L= RELEASE length,
for (i=0, i<L,i++) IN2dcl INFORMATION DATA[i] = RELEASE data[i],
IN2dcl FCS_FIELD data[0] = 255,
IN2dcl FCS_FIELD data[1] = 255,
P-Frm->data[0] = (IN2dcl EA_OCTET2 << 0) | (IN2dcl C << 1) | IN2dcl SAPI << 2) ,
P-Frm->data[1] = (IN2dcl EA_OCTET3 << 0) | (IN2dcl SAPI << 1),
P-Frm->data[2] = (IN2dcl CONTROL << 0) | IN2dcl N_S << 1), ,
P-Frm->data[3] = (IN2dcl P << 0) | (IN2dcl C N_R << 1) ,
P-Frm->data[4] = (IN2dcl INFORMATION data[0]),
P-Frm->data[5] = (IN2dcl INFORMATION data[1]),
P_Frm->data[6] = (IN2dcl FCS_FIELD data[0]),
P_Frm->data[7] = (IN2dcl FCS_FIELD data[1]),
P_Frm->length = 8,


if eventype == 'S')    {
      if (send_l2(P_Frm)) return 1,
      }
else if eventype == 'R')    {
      if rec_l2(P_Frm)) return 1,
      }
else return 0,
return 0,

}
```