Computer Standards & Interfaces 14 (1992) 117-144
North-Holland

Standard conformance test specification language TTCN

Behcet Sarikaya [a] and A. Wiles [b]

[a] *Department of Computer and Information Sciences, Bilkent University, Bilkent, Ankara, Turkey 06533*
[b] *TeleTest Box 1218, 751 42 Uppsala, Sweden*

Abstract

Sarikaya, B. and A. Wiles, Standard conformance test specification language TTCN, Computer Standards & Interfaces 14 (1992) 117-144.

The International Standards Organization (ISO) has defined a protocol test language called TTCN (Tree and Tabular Combined Notation) to specify abstract test suites for Open Systems Interconnection (OSI) protocols. TTCN combines a tree notation for dynamic behaviour description with a tabular representation of various language constructs. TTCN allows tabular constraints to enforce values on the Abstract Service Primitive (ASP) or Protocol Data Unit (PDU) parameters. For application layer protocols, Abstract Syntax Notation One (ASN.1) constraints are used. Dynamic behaviour description in TTCN is shown to address many important aspects of conformance testing such as modularity support in terms of test cases, steps and default behaviour tables and sophisticated timer management. TTCN has a machine processable form called TTCN-MP that defines all the TTCN syntax using BNF. Semantics of the tests specified in TTCN is operationally defined rendering TTCN almost a formal notation.

Keywords. Specification languages; distribution systems; conformance testing; communication protocols; test specification; test suite; tree and tabular combined notation.

1. Introduction

Distributed processing and distributed systems imply that the entities in different systems need to communicate with each other, 'speaking the same language'. The information transferred must conform to some mutually acceptable set of rules of syntax and semantics. This is where the ISO (International Standards Organization) comes in with its OSI (Open Systems Interconnection) model. The model [1] is divided into seven layers, each performing a function to achieve open communication. The concept of testing various implementations of these layers for conformance to the relevant ISO standards is a logical step in the development chain. ISO has developed a standardized methodology and framework for all aspects of conformance testing of OSI protocol implementations. Conformance testing is not concerned with the performance or efficiency aspects of an implementation. The stated purpose is to decide whether the implementation under test (called IUT) adheres to the relevant standard in all instances of communication, both inter-layer and inter-system. ISO conformance testing standard [2] is a six part document. The first defines conformance in the OSI context, the second part defines the abstract test suite specification and the third part defines the TTCN. The other parts are on test realization and test laboratory operations.

We describe the International Standard TTCN hereafter called TTCN. Conformance test suites such as the X.25 test suite [7] have been specified in earlier versions. Test suites are complex, and voluminous, for example the X.25 test suite contains over 500 tests. It is important to be able to electronically treat TTCN which is a tabular language. TTCN's linear form, called TTCN-MP, is designed to facilitate electronic exchange of test suites since it is a machine independent internal form of the TTCN tables.

0920-5489/92/$05.00 © 1992 - Elsevier Science Publishers B.V. All rights reserved

Thus it is also important to support TTCN-MP and its conversion to TTCN-GR. Several TTCN editors have been designed to facilitate interactive editing of TTCN test suites.

Below, in Section 2, we start the discussion by first introducing the Abstract Syntax Notation One (ASN.1) which is an integral part of TTCN. Section 3 introduces TTCN and declaration facilities of TTCN, Section 4 is on constraint declarations, Section 5 describes how the dynamic behaviour is specified, Section 6 develops a test case and discusses its semantics, Section 7 discusses the machine processable form of TTCN, Section 8 is on test realization from the abstract test suites and the test suite overview tables, and Section 9 concludes the paper.

2. ASN.1

In this section we introduce the standard ASN.1 [5], an integral part of TTCN. Abstract syntax notation one (ASN.1) is a language for defining types and values. Also a set of encoding/decoding rules, called basic encoding rules (BER) is defined for ASN.1 defined types and values [6]. ASN.1 provides several built-in types such as integers, reals, booleans, bit and octetstrings.

Structured types can also be defined: SEQUENCE and SET for a group of elements (RECORD of Pascal), CHOICE for a type of alternatives (variant record of Pascal) with a TAG defining a code for distinguishing the alternatives, SEQUENCE OF and SET OF for a sequence of identical types (array of Pascal).

Types are defined using the following notation:

typereference ::= Type

where *typereference* is the name of the variable which should start with a capital letter and *Type* is a built-in or a constructed type.

Values are defined using the following notation:

valuereference Type ::= Value

where *valuereference* is the name of the variable to which a value *Value* of type *Type* is assigned.

The name of the item referred to as *identifier* consists of one of more letters, digits, and hyphens. The initial character shall be a *lower-case* letter. The item *valuereference* is defined exactly as an *identifier*.

Every type in ASN.1 has a tag. A tag is specified by giving its class and the number within the class. The class is one of **UNIVERSAL, APPLICATION, PRIVATE**, and **CONTEXT-SPECIFIC**. The number is a non-negative integer, specified in decimal notation. All built-in types are of universal class and they carry a unique class number.

2.1. Built-in simple types

ASN.1 includes several built-in simple types such as BOOLEAN, INTEGER, ENUMERATED, BIT STRING, OCTET STRING and OBJECT IDENTIFIER. Type definition, corresponding value definition notation and class numbers for these types are given in *Table 1*.

Table 1
Built-in simple types

Type definition	Value definition	Class number
BooleanType::= BOOLEAN	BooleanValue::= TRUE | FALSE	1
InterType::= INTEGER |INTEGER {NamedNumberList}	IntegerValue::= SignedNumber | identifier	2
EnumeratedType::= ENUMERATED {Enumeration]	EnumeratedValue::= SignedNumber |identifier	10
BitStringType::= BIT STRING |BIT STRING {NamedBitList}	BitStringValue::= bstring |hstring | {identifierList} | { }	3
OctetStringType::= OCTET STRING	OctetStringValue::= bstring |hstring	4
ObjectIdentifierType::= OBJECT IDENTIFIER	ObjectIdentifierValue::= {ObjIdComponentList}	6
NullType::= NULL	NullValue::= NULL	5

Example of an (enumerated) integer type definition for *ABRTSource* is:

```
ABRTSource ::= INTEGER
  { requestingACPM (0),
    requestor (1) }
```

An example value definition for *ABRTSource* above is:

```
defaultABRTSource ABRTSource ::= requestor
```

An example bitstring type definition for *PSREQ* is:

```
PSREQ ::= BIT STRING { context-management(0),
                       restoration(1) }
```

and a bitstring value:

```
defaultPSREQ PSREQ ::= '01'B
```

As an example we define the variable *Protocol* whose type is object identifier as:

```
Protocol ::= OBJECT IDENTIFIER
```

Now a value *ftam* for the above type can be defined as:

```
ftam Protocol ::= {iso standard 8571}
```

The same value in numbers can be defined as:

```
ftam Protocol ::= {1 0 8571}
```

2.2. Structured types

Sequence type is similar to record type in programming languages. A sequence type is defined in *Table 2* where SEQUENCE { } is used to define an empty sequence, *ElementTypeList* is a list of *ElementTypes* defined as:

ElementType ::= NamedType | NamedType **OPTIONAL** |
NamedType **DEFAULT** Value |
COMPONENTS OF Type

where OPTIONAL indicates a type whose value can be optionally included, DEFAULT gives a default value which can be optionally included in the value. *COMPONENTS OF* is used for inclusion at this point of all *ElementTypes* appearing in the referenced type which must be a *SEQUENCE* or a *SET* type.

A typical use of sequence type is to define Protocol Data Units (PDUs), such as in:

```
ABRTapdu ::= [ APPLICATION 4 ] IMPLICIT SEQUENCE
  { abortSource [0] IMPLICIT ABRTSource,
    userInformation [1] AssociationData OPTIONAL }
```

Table 2
Structured types

Type definition	Value definition	Class number
SequenceType::= SEQUENCE{ ElementTypeList} | SEQUENCE { }	SequenceValue::= {ElementValueList} | { }	16
SequenceOfType::= SEQUENCE OF Type	SequenceOfValue::= {ValueList} | { }	16
SetType::= SET{ElementTypeList} | SET { }	SetValue::= {ElementValueList} | { }	17
SetOfType::= SET OF Type	SetOfValue::= {ValueList} | { }	17
ChoiceType::= CHOICE {AlternativeTypeList}	ChoiceValue::= NamedValue	Tag of type chosen
TaggedType::= Tag Type | Tag IMPLICIT Type | Tag EXPLICIT Type	Tag::= [Class ClassNumber]	N/A

which defines the abort PDU called *ABRTapdu* of the ACSE protocol [4]. This type has a tag which is of application class, number 4. The field *AssociationData* is defined to be of **EXTERNAL** type. The type *abortSource* is of context-specific class, number 0 whose type is *ABRTSource* defined above.

The type sequence-of is used to define sequences of a single type similar to arrays. The notation used to define this type is given in *Table 2*.

The types **SET** and **SET OF** are defined exactly the same manner as **SEQUENCE** and **SEQUENCE OF** respectively (see *Table 2*). The difference is that the order of elements on set and set-of values is irrelevant.

Let us first define *Presentation_Context_Definition_List* as an example for sequence and sequence-of types:

```
Presentation_Context_Definition_List ::= SEQUENCE OF
   SEQUENCE { context_id INTEGER,
              abstract_syntax OBJECT IDENTIFIER }
```

We next define a value for the above sequence-of-type:

```
Context_def_list Presentation_Context_Definition_List
          { Presentation_context Context_def_FTAM,
            Presentation_context Context_def_ACSE }
```

The value *Context_def_FTAM* is defined as:

```
Context_def_FTAM Presentation_context {
     Presentation_context_identifier TSP_Pres_Context_id_FTAM,
     Abstract_syntax_name {1,0,8571,2,1} }
```

where *TSP_Pres_Context_id_FTAM* is an integer constant and *{1,0,8571,2,1}* is the object identifier value that represents the abstract syntax name (the ASN.1 definitions of FTAM protocol) for FTAM.

ASN.1 choice type is similar to Pascal variant records and is defined in *Table 2* where *AlternativeTypeList* is a list of *NamedTypes*. In order to unambiguously differentiate each alternative the tags of all types defined in the *AlternativeTypeList* shall differ from those of other types. Also the *identifier* in all *NamedType* sequences shall be distinct.

An example choice type definition is the (FTAM) PDU type defined as:

```
PDU ::= CHOICE {
     InitializePDU,
     FilePDU,
     BulkdataPDU }
```

and a choice value is:

```
ftamPDU PDU ::= FilePDU
```

A tagged type is a new type which is isomorphic with an old type, but which has a different tag. It is defined in *Table 2* where *Class* is either **UNIVERSAL, APPLICATION, PRIVATE**, or empty if **CONTEXT-SPECIFIC**. *ClassNumber is a decimal number*.

3. TTCN overview and declarations

A conformance test suite in TTCN consists of a number of test cases which test the implementations for conformance. Tests are hierarchically organized into test groups each consisting of one or more test cases. Test cases are specified using the tree notation and are made up of test steps. The Tree and Tabular Combined Notation (TTCN) specifies a test suite in four parts:

- test suite overview,
- declarations,
- constraints,
- dynamic behaviour.

We will discuss first the declarations followed by the constraints and the dynamic behaviour. Finally test suite overview will be discussed together with test realization.

Table 3
Tabular type definitions

Simple Type Definitions		
Type Name	Type Definition	Comments
Transport _ classes	INTEGER(0,1,2,3,4)	Classes of the transport protocol
String5	IA5STRING[5]	String of maximum length 5

3.1. Declarations

Different type declarations and value declarations must be done before specifying the dynamic behaviour. Type declarations will be discussed in this section and value declarations, or constraints are discussed in the next section.

3.1.1. Types

TTCN supports a number of basic types such as INTEGER, BOOLEAN, BITSTRING, HEXSTRING and OCTETSTRING and all the character string types of ASN.1. Types can be defined in tabular form using Simple Type Definitions proforma which contains Type Name, Type Definition and Comments columns. For example the type *Transport _Classes* as an enumeration type and *String5* as an IA5String can be defined as shown in *Table 3*.

As we can see from *Table 3* base type such as INTEGER is followed by the type restriction which can take one of the following forms:

1. a list of distinguished values of the base type; these values comprise the new type. *Transport _classes* is defined with a list of distinguished values.
2. a specification of a range of values of type INTEGER such as:
   seq _numbers INTEGER(0..127) or
   positive _numbers INTEGER(1..INFINITY) to represent all positive INTEGER numbers.
3. a specification of the maximum length of a predefined string type. *String5* in *Table 3* is defined with a maximum length.

Any type definitions using ASN.1 language is also accepted in TTCN. In this case ASN.1 Type Definition table is used. As an example the ASN.1 type *P_address* is defined to be of SEQUENCE type in *Table 4*.

Structured tabular types are declared using Structured Type Definition table. As an example *T_AddressInfo* is defined as a structured type in *Table 5*.

3.1.2. TTCN operators and operations

Commonly used arithmetic operators of ‘+’, ‘−’, ‘ * ’, ‘/’, and MOD are predefined operators in TTCN. The predefined relational operators are ‘ = ’, ‘ < ’, ‘ > ’, ‘⟨⟩, 1 > = ’, ‘ < = ’ and Boolean operators are NOT, AND and OR.

Table 4
ASN.1 type definition table

ASN.1 Type Definition
Type Name: P _ address Comments: Presentation address
Type Definition
SEQUENCE { Presentation _ selector OCTET STRING, Session _ selector OCTET STRING, Transport _ selector OCTET _ STRING, Network _ Service _ Access _ Point OCTET STRING }

Table 5
Structured type definition

Structured Type Definition		
Type Name: Comments:	T_Address_info Transport address	
Element Name	Type Definition	Comments
Source Destination	BITSTRING [4] BITSTRING [4]	Length is 4 bits Length is 4 bits

For type conversions a number of predefined operations is supported:
1. HEX_TO_INT(hexvalue:HEXSTRING) –)INTEGER
   which converts a single HEXSTRING to a single INTEGER value.
2. BIT_TO_INT(bitvalue:BITSTRING) –)INTEGER
   which converts a single BITSTRING value to a single INTEGER value.

Similarly INT_TO_HEX and INT_TO_BIT are defined.

Other predefined operations are IS_PRESENT, NUMBER_OF_ELEMENTS, IS_CHOSEN and LENGTH_OF.
1. IS_PRESENT(DataObjectReference) –)BOOLEAN
   this operation can be used to check the presence of an OPTIONAL or a DEFAULT field in the actual instance of the data object defined to have ASN.1 SET or SEQUENCE type. As an example assume *received_PDU* is of *ABRTapdu* type defined in Section 2.2 above. Then the operation call
   IS_PRESENT(received_PDU.userInformation)
   evaluates to TRUE if userInformation (an OPTIONAL field) in the actual instance of *received_PDU* is present.
2. NUMBER_OF_ELEMENTS(DataObject_Reference) –)INTEGER
   this operation returns the actual number of elements of a data object that is of type ASN.1 SEQUENCE OF or SET OF.
3. IS_CHOSEN(DataObject_Reference) –)BOOLEAN
   this operation returns TRUE if the data object reference specifies the variant of the CHOICE type that is actually selected for a given data object. As an example assume *received_PDU* is of (FTAM) PDU type defined above. Then the operation call
   IS_CHOSEN(received_PDU.field2)
   returns TRUE if the actual instance of *received_PDU* is of the type *FilePDU*.
4. LENGTH_OF(DataObjectReference) –)INTEGER
   returns the actual length of a data object that is of type BITSTRING, HEXSTRING, OCTET-STRING or CharacterString. For example:
   LENGTH_OF('010'B) returns 3,
   LENGTH_OF('F3'H) returns 2,
   LENGTH_OF('F2'O) returns 1 (octet),
   LENGTH_OF("EXAMPLE") returns 7.

For user defined operations Test Suite Operation Definition table must be used. As an example *diffOBJECT* to return an OBJECT IDENTIFIER is defined in *Table 6*. In this table the operation is informally defined. More precise definitions in a programming language such as C can also be given.

3.1.3. Test suite parameters

Abstract test suites must be developed based on the protocol standard and they must contain test cases to correspond to different categories, i.e. basic interconnection, capability tests, behaviour tests, etc. Some features of protocols are optional, not every IUT is supposed to implement every feature. Because of these, the IUTs are asked to declare their capabilities in a standard form called Protocol Implementation Conformance Statement (PICS). PICS proformas are standardized for every protocol.

Table 6
A test suite operation definition

Test Suite Operation Definition	
Operation Name:	diffOBJECT(obj:OBJECT IDENTIFIER)
Result Type:	OBJECT IDENTIFIER
Comments:	
Description	
diffOBJECT(obj) returns an OBJECT IDENTIFIER different from the input obj. For example: diffOBJECT({1,3,9999,1,7}) = {2,3,9999,1,7}	

Table 7
Test suite parameters

Test Suite Parameter Declarations			
Parameter Name	Type	PICS/PIXIT Ref	Comments
TSP_Pres_Context_id_FTAM	INTEGER	PIXIT question yy	pres. context
TSP_pres_address_tester	P_ADDRESS	PIXIT question xx	TESTER
TSP_FX	BOOLEAN	PICS question FX1	

Some extra information such as the addresses to use in order to apply the tests are declared using Protocol Implementation Extra Information for Testing (PIXIT) proforma.

Test suite parameters are constants derived from the PICS and/or PIXIT which globally parameterize the test suite. These constants are declared using Test Suite Parameter Declarations table. *Table 7* shows the test suite parameter declarations for the Association Control Service Element (ACSE) test suite [9].

3.1.4. Test case selection expressions

Test case selection expressions declare the Boolean expressions to be used in the test case selection process. The Boolean expressions defined are given a name and these names are used in test suite overview tables to indicate the conditions imposed on executing a test case. In *Table 8* two test case selection expression declaration examples are given.

3.1.5. Test suite constants

Test suite constants are a set of names for values not derived from the PICS or PIXIT that will be constant throughout the test suite. They are declared using test suite constant declarations table. Some of the test suite constants from the transport test suite is given as an example in *Table 9*.

3.1.6. Variables

There are two types of variables in TTCN: test suite and test case variables, defined using test suite variables/ test case variables tables, respectively. Test suite variables are globally defined for the test

Table 8
Test case selection expressions

Test Case Selection Expression Definitions		
Expression Name	Selection Expression	Comments
SEL1	TSP_FX	Feature X supported
SEL2	TSP_FX_VAL = 0	Accept Feature X Val = 0

Table 9
Test suite constants

Test Suite Constant Declarations			
Constant Name	Type	Value	Comments
CR_code CC_code	INTEGER INTEGER	14 13	

Table 10
Test suite variables

Test Suite Variable Declarations			
Variable Name	Type	Value	Comments
state	IA5STRING	'idle'	used to pass the final state of the previous test case to determine which preamble to use

Table 11
Test case variables

Test Case Variable Declarations			
Variable Name	Type	Value	Comments
SESS_CON_ID_IUT	SSCONID	{'ref1', 'ref2', 'ref3'}	used to store the value of the session connection identifier for an established connection. It will be used while reestablishing the connection

suite, i.e. they retain their values throughout the test suite. An example suite variable declaration is shown in *Table 10*.

A test suite may define a set of variables, test case variables which are declared globally to the test suite but whose scope is defined to be local to the test case. An example test case variable from the ACSE test suite is shown in *Table 11*. The type of this variable *SSCONID* is an ASN.1 SEQUENCE:

SEQUENCE { SS_ user_ref OCTET STRING, Common_ref OCTET STRING,
Additional_ref OCTET STRING }

3.1.7. PCO declarations

In the PCO Type Declaration table, the set of points of control and observation (PCOs) to be used in the test suite are declared. For each PCO used, its name, its type identifying the layer boundary where the PCO is located and its role (either UT for Upper Tester or LT for Lower Tester) must be provided. As an example:

```
L  TSAP  LT
```

defines the PCO L of the transport service access point at the lower tester.

The number of PCOs in a test suite depends on the test architecture used: one for the Remote and Coordinated test architectures (see *Table 12*), and two for Distributed architecture. The PCO model in

Table 12
Points of control and observation

PCO Declarations			
PCO Name	PCO Type	Role	Comments
L	PSAP	LT	Presentation service access point at the lower tester

Table 13
Timers

Timer Declarations			
Timer Name	Duration	Unit	Comments
wait	15	sec	General purpose wait

TTCN is based on two First In First Out (FIFO) queues, one for control (stimulus) and one for observation.

3.1.8. Timers

Timers used in the test suite are declared in the Timer Declarations table which contains Timer Name, Duration, Units and Comments columns. For example a general purpose timer called wait can be declared to be of 15 seconds in *Table 13*. Time units can be ps (picoseconds), ns (nanoseconds), us (microseconds), ms (milliseconds), sec (seconds), min (minutes).

3.1.9. ASP type declarations

Abstract Service Primitives are messages exchanged between consecutive layers. In TTCN, ASPs can be declared either using ASP Type Definition proforma or ASN.1 ASP Type Definition proforma. Tabular ASP type definitions are done by making a separate proforma for each ASP declared in the service standard. An example tabular ASP definition is shown in *Table 14*.

Parameters may be of a type of arbitrarily complex structure. If a parameter is to be structured as a PDU then the type is specified either as a PDU identifier to indicate that in the constraint for the ASP this parameter may be chained to a PDU constraint of a specific PDU type; or as PDU metatype to indicate that in the constraint for the ASP this parameter may be chained to a PDU constraint of any PDU type.

Structured types can be used in ASP declarations. This leads to a multi-level substructure of parameters if a parameter name is given to be of a substructure type. For example *CONreq* defined in *Table 14* has a multi-level substructure due to *T_Address_info* defined above as a structured type.

If a macro symbol ⟨ − is used instead of a parameter name then this is equivalent to a macro call which expands into a list of parameters without introducing an additional level of substructure. For example we can define an ASP *CONreq*2 as shown in *Table 15*. In this case *CONreq*2 contains the fields of *Source, Destination, T_Class* and *UserData*.

ASPs can be declared in ASN.1 using ASN.1 ASP Type Definition proforma. This declaration might make reference to some parameters that are declared again in ASN.1 using ASN.1 Type Declaration proforma.

Table 14
ASP type definition proforma

ASP Type Definition		
ASP name: CONreq(T_CONNECTrequest) PCO type: TSAP Comments: Transport connect request service primitive		
Parameter Name	Parameter Type	Comments
Cda (Called Address)	T_Address_info	of upper tester
Cga (Calling Address)	T_Address_info	of lower tester
Qos (Quality of Service)	QOS	should ensure class 0 is used

Table 15
Macro symbol in ASP definition

ASP Type Definition		
ASP Name:	CONreq2(T_CONNECTrequest)	
PCO Type:	TSAP	
Comments:	Transport connect request service primitive	
Parameter Name	**Parameter Type**	**Comments**
<-	T_Address_info	
T_Class	INTEGER	
UserData	PDU	

3.1.10. PDU type declarations

Protocol Data Units are messages exchanged between peer entities. The declaration of PDUs is similar to that of ASPs. PDU Type Definition proforma is used to declare various fields of PDUs. All ASP parameters and PDU fields defined in TTCN are assumed to be optional by default. Optionality of parameters/ fields must be explicitly specified if ASN.1 is used to define ASP/PDUs.

An example PDU Type Definition table from transport test suite for CR PDU is shown in *Table 16*. Some fields of CR like *PDU_CODE* and *CDT* are of 4-bits long. In the table they are defined as integers and the details of encoding/ decoding is left to the implementation.

The type for some fields could be PDU (of a higher level protocol) to indicate that in the constraint for the PDU this parameter may be chained to a PDU constraints identifier of any PDU type. The type could be a PDU identifier to indicate that in the constraint for the PDU this parameter may be chained to a PDU constraints identifier of a specific PDU type.

Multi-level substructure of fields can be declared if a field name is given in the declaration referencing a structured type. Instead if the macro symbol is used to expand directly to a list of fields then no additional level of substructure is introduced.

Similarly, application layer PDUs can be declared using ASN.1 PDU Type Definition proforma in ASN.1. An example ASN.1 PDU type definition from the ACSE test suite is shown in *Table 17*.

When declaring ASP parameters or PDU fields of string types length of the parameter/ field can also be declared in two forms:

1. field_type[Length]
   restricting the length of that parameter or field to exactly *Length*, or

Table 16
PDU type definition proforma

PDU Type Definition		
PDU Name:	CR (CONNECTIONrequest)	
PCO Type:	NSAP	
Comments:	Transport connect request protocol data unit	
Field Name	**Field Type**	**Comments**
LI_f	INTEGER(1...254)	
PDU_CODE	INTEGER	CR_CODE is 14
CDT	INTEGER(0..15)	
DST_REF	INTEGER(0..65535)	fixed to zero
SRC_REF	INTEGER(1..65535)	
CLASS	INTEGER	4 bits
OPTIONS	INTEGER	4 bits
calling_TSAP_ID	IA5STRING	
called_TSAP_ID	IA5STRING	
TPDU_size	INTEGER(7..13)	
userdata	IA5STRING	

Table 17
ASN.1 PDU type definition

ASN.1 Type Definition	
PDU Name:	PARP
PCO Type:	SSAP
Comments:	Presentation PDU
Type Definition	
SEQUENCE { provider_ reason [0] IMPLICIT Abort_ reason OPTIONAL, event [1] IMPLICIT Event_ identifier OPTIONAL }	

2. field_type[min_len TO max_len] or field_type[min_len .. max_len]
   specifies the minimum length *min_len* and a maximum length *max_len*.

The units of lengths for BITSTRING is bits, for HEXSTRING is hex digits, for OCTETSTRING octets and for CharacterString is characters.

4. Constraints

It is necessary to specify values of ASP parameters, PDU fields and structured elements in detail. These values are specified in the form of constraint tables. TTCN allows two types of constraints: Tabular constraints or ASN.1 constraints.

Constraints are used in SEND and RECEIVE events. If a constraint is used in a SEND event, it shall contain specific values for each and every ASP parameter or PDU field. If a constraint is used in a RECEIVE event, the test suite specifier may use special matching symbols where it is not possible to specify specific values. These special symbols can be used to replace values of single ASP parameters or PDU fields, groups of ASP parameters or PDU fields or even the entire contents of ASPs or PDUs.

Neither test suite variables nor test case variables can be used in the constraints. They can only be passed as actual parameters.

Constraints may be parameterized. In this case the constraint name is followed by a parenthesized formal parameter list and the parameterized ASP parameters or PDU fields shall have these parameters as values. Each formal parameter name is followed by a colon (:) and the name of the parameter's type. If more than one parameter is used the parameter name-type pairs are separated from each other by semicolons. Actual parameters could be literal values, test suite parameters, test suite constants, test suite variables, test case variables and other constraints.

4.1. Constraint chaining

Constraints may be chained by referencing a constraint as the value of a parameter or field in another constraint. For example, the value of the Data parameter of an NDATAreq ASP could be a reference to a CR PDU constraint, i.e. the transport PDU is chained to the network ASP.

Constraint chaining could be in one of two ways:

1. static chaining, where an ASP parameter value or PDU field value in a constraint is an explicit reference to another constraint; or
2. dynamic chaining, where an ASP parameter value or PDU field value in a constraint is a formal parameter of the constraint. In this case the actual parameter in the dynamic behaviour table defines the dynamic value.

4.2. Constraint matching mechanisms

Values of all types can be used in constraints. Literal values, test suite parameters/ constants, formal parameters and test suite operations are all valid as actual values in a constraint. To facilitate static chaining a (parameterized) constraints reference is also allowed as a parameter or field value. A constraint ASP parameter or PDU field shall match the corresponding received ASP parameter or PDU field if the received ASP parameter or PDU field has exactly the same value to which the expression in the constraint evaluates.

4.2.1. Constraints for SEND events

When a constraint is referenced for a SEND event, it should provide a specific value for each and every ASP parameter and PDU field. TTCN also allows assigning values explicitly in the SEND event line.

If an ASP parameter or PDU field is declared optional, then the Omit symbol (−) can be used to indicate that no value will be sent for the parameter/ field. AnyValue symbol (?) may be used to indicate that no value is specified in the constraint.

4.2.2. Constraints for RECEIVE events

Complement is a special symbol for matching that can be used on values of all types. *Complement* is denoted by the keyword COMPLEMENT followed by a list of constraint values. For a value of type INTEGER:

COMPLEMENT(5)

shall match the corresponding ASP parameter or PDU field if the received ASP parameter or PDU field does not match any of the values listed in the value list, i.e., is not equal to 5 for the above example.

Omit is a special symbol for matching that can be used on all values of all types, provided that the ASP parameter or PDU field is declared as optional. *Omit* is denoted by '−' in tabular constraints and OMIT in ASN.1 constraints. In ASN.1 constraints it is also possible to simply leave out an OPTIONAL ASP parameter or PDU field instead of using OMIT explicitly.

AnyValue is also a special symbol for matching on values of all types. In both tabular and ASN.1 constraints *AnyValue* is denoted by '?'. *AnyValue* shall match if, and only if, the received ASP parameter or PDU field evaluates to a single element of the specified type.

AnyOrOmit is a special symbol for matching on values of all types, provided that the ASP parameter or PDU field is declared as optional. In both tabular and ASN.1 constraints it is denoted by ' * '. The matching mechanism provided is equivalent to those of *Omit* or *AnyValue*.

ValueList can be used for values of all types. In both tabular and ASN.1 constraints *ValueLists* are denoted by a parenthesized list of values separated by commas. A constraint ASP parameter or PDU field that uses a *ValueList* shall match the corresponding incoming ASP parameter or PDU field if the incoming ASP parameter or PDU field value is equal to one of the values in the *ValueList*. For a value of type:

```
CHOICE {a INTEGER
        b BOOLEAN }
(a 2, b TRUE)
```

will match if the values 2 for a and TRUE for b are received.

Ranges can only be used on values of INTEGER types. In both tabular and ASN.1 constraints *range* is denoted by two boundary values, separated by '..' or TO, enclosed by parentheses.

Range matches if the incoming ASP parameter or PDU field value is equal to one of the values in the *Range*. As an example for a value of type INTEGER:

(1 .. 6)

matches if the value is between 1 to 6.

SuperSet denoted by *SUPERSET* and *SubSet* denoted by *SUBSET* are only used in ASN.1 constraints for SET OF types and they match with at least all (possibly more) or all (possibly less) elements, respectively.

4.2.2.1. Inside values. *AnyOne* denoted by '?' is used for matching within values of string types, SEQUENCE OF and SET OF. Inside a string, SEQUENCE OF or SET OF a '?' in place of a single element means that any single element will be accepted.

AnyOrNone denoted by ' * ' is also used within values of string types, SEQUENCE OF or SET OF. Inside a string, SEQUENCE OF or SET OF a ' * ' in place of a single element means that either none, or any number of consecutive elements will be accepted. It will match the longest sequence of elements possible.

4.2.2.2. Attributes of values *IfPresent* denoted by IF_PRESENT can be used as an attribute of all matching mechanisms provided that the type is declared as optional. *Length* denoted by a positive integer expression, enclosed in square brackets can only be used as an attribute of the following mechanisms: Specific value, Complement, Omit, AnyValue, AnyOrOmit, AnyOne, AnyOrNone and Permutation (to match values of SEQUENCE OF type in any permutation).

For a type of IA5STRING the constraint:

'ab * ab'[13]

matches if the string has at least 13 characters starting and ending with 'ab'. For a type of IA5STRING OPTIONAL the constraint:

'abcdef'IF_PRESENT

matches either if the string is equal to 'abcdef' or if this field is absent.

4.3. Tabular constraints

Tabular constraints can be on PDUs and ASPs.

4.3.1. PDU constraints

For every PDU type declaration at least one *base constraint* must be defined. In a base constraint, a set of base or default values for every field defined is specified in a horizontal manner. *Table 18* contains the base constraint proforma for the CR PDU defined in *Table 16*.

Table 18
A constraint on CR PDU

PDU Constraint Declaration		
Constraint Name: CR1 PDU Type: CR Derivation Path: Comments: Transport connect request protocol data unit base constraint		
Field Name	**Field Value**	**Comments**
LI_f	(4..INFINITY)	greater than 3
PDU_CODE	14	CR PDU
CDT	15	credit value
DST_REF	0	must be zero
SRC_REF	myref	a test suite constant
CLASS	0	class zero
OPTIONS	0	no expedited data
calling_TSAP_ID	id1	a test suite constant
called_TSAP_ID	id2	a test suite constant
TPDU_size	7	minimum accepted, i.e. 128
userdata	'testing,testing'	arbitrary text

Table 19
Constraint inheritance

PDU Constraint Declaration		
Constraint Name:	CR2	
PDU Type:	CR	
Derivation Path:	CR1.	
Comments:	CR PDU constraint inheritance	
Field Name	Field Value	Comments
CDT	1	In the base CR1 this value is 15

PDU Constraint Declaration		
Constraint Name:	CR2	
PDU Type:	CR	
Derivation Path:	CR1.	
Comments:	CR PDU constraint inheritance	
Field Name	Field Value	Comments
TPDU_size	?	Accept any legal value

More specific values of PDU parameters are defined in *modified constraints*. Any fields not specified in the modified constraint will be inherited from the values specified in the base constraint. The name of the modified constraint is a unique identifier. The name of the base constraint which is to be modified is indicated in the *derivation path* entry in the constraint header. This entry must be left blank for a base constraint. A modified constraint can itself be modified. In such a case the *derivation path* indicates the concatenation of the names of the base and previously modified constraints, separated by dots ('.'). A dot should follow the last modified constraint name. As an example we modify the base constraint CR1 to obtain CR2 shown in *Table 19* to change the credit value *CDT* to one and furthermore modify CR1 to obtain CR3 to accept any value for *TPDU_size* to be used in a RECEIVE event.

Parameterized constraints specify the constraint values in a parameterized form. The constraint name in this case contains the list of formal parameters with their types, such as in:

Table 20
ASP constraint declaration

ASP Constraint Declaration		
Constraint Name:	P_CONreqbase(aarq:AARQ)	
ASP Type:	P_CONNECT_request	
Derivation Path:		
Comments:	ASP constraint example	
Parameter Name	Parameter Value	Comments
Calling_presentation_address	TSP_pres_address_tester	test suite parameter
Called_presentation_address	TSP_pres_address_IUT	test suite parameter
Presentation_context_definition_list	Context_def_list_2	
Default_context_name	–	
Quality_of_service	TSP_QOS	
Presentation_requirements	–	
Mode	Mode–normal	
Session_requirements	FU_duplex_only	
Initial_synchronization_point_serial_number	–	
Initial_assignment_of_tokens	Initiator_side	
Session_connection_identifier	–	
User_data	aarq	

Table 21
Structured type constraint declaration

Structured Type Constraint Declaration		
Constraint Name: Cc1 Structured Type: T_Addressinfo Derivation Path: Comments: Structured type constraint example		
Element Name	Element Value	Comments
Source Destination	TS_PAR1 TS_PAR2	test suite parameter test suite parameter

C0(P1:INTEGER,P2:BOOLEAN).
Both base and modified constraints can be parameterized.

If the ASP or PDU declaration refers to a structured type as a substructure of a parameter or field then a *Structured Type Constraint Declaration* should be made and the name should be referenced at the value column. Structured constraint tables are very similar to PDU constraint tables, as an example see *Table 21.*

If the ASP or PDU declaration refers to a parameter or field specified as being of PDU metatype then in a corresponding constraint the value for that parameter or field shall be specified as the name of a PDU constraint, or formal parameter.

4.3.2. ASP constraints

ASP constraints are similar to PDU constraints. As an example the ASP constraint *P_CONreqbase* of the ACSE test suite is shown in *Table 20.* This is a parameterized ASP base constraint. The constraint value for the ASP parameter *User_data* defined to be of metatype PDU is a formal parameter *aarq* to be of the type *AARQ* PDU of the ACSE protocol.

4.4. ASN.1 constraints

Constraints on ASPs and PDUs can be declared using ASN.1 constraints. ASN.1 value definitions discussed in Section 2 is extended to allow use of matching mechanisms of TTCN.

Table 22
ASN.1 PDU constraint

ASN.1 PDU Constraint Declaration
Constraint Name: AARQbase_S(extconrq : SEQUENCE OF EXTERNAL) PDU Type: AARQ Derivation Path: Comments: Base constraint for AARQ for SEND events

Constraint value
{
protocol_version Version001,
application_context_name Application_context_name_FTAM,
called_AE_qualifier TSP_AE_qualifier_IUT,
calling_AP_title TSP_AP_title_tester,
calling_AE_qualifier TSP_AE_qualifier_tester,
user_information extconrq
}

Table 23
Modified ASN.1 PDU constraint

ASN.1 PDU Constraint Declaration	
Constraint Name:	AARQmod(extconrq: SEQUENCE OF EXTERNAL)
PDU Type:	AARQ
Derivation Path:	AARQbase_S.
Comments:	Modified constraint for AARQ for SEND event
Constraint Value	

{
REPLACE protocol_version BY Version002
OMIT called_AP_title
OMIT calling_AP_title
OMIT calling_AE_qualifier
user_information extconrq
}	

4.4.1. PDU constraints

PDU base constraints can be defined in ASN.1. *Table 22* is an example ASN.1 PDU constraint for SEND events for *AARQ* PDU of the ACSE test suite. This constraint is also parameterized with the formal parameter *extconrq* providing constraint value for *user information* field of the PDU.

Table 24
ASN.1 ASP constraint declaration

ASN.1 ASP Constraint Declaration	
Constraint Name:	N_DATAreq_base
ASP Type:	N_DATArequest
Derivation Path:	
Comments:	Base constraint for N_DATArequest for SEND events
Constraint Value	

{
callingNetworkAddress TS_PAR_3,
calledNetworkAddress TS_PAR_4,
connectionIdentifier 'ABCDEF'H,
data TCON_Class_4_1
}	

Table 25
ASN.1 type constraint

ASN.1 Type Constraint Declaration	
Constraint Name:	P_addr_cons
Structured Type:	P_address
Derivation Path:	
Comments:	Presentation address constraint
Constraint Value	
SEQUENCE { presentation_selector TSP_1, session_selector TSP_2, transport_selector ?, network_Service_Access_Point ?}	

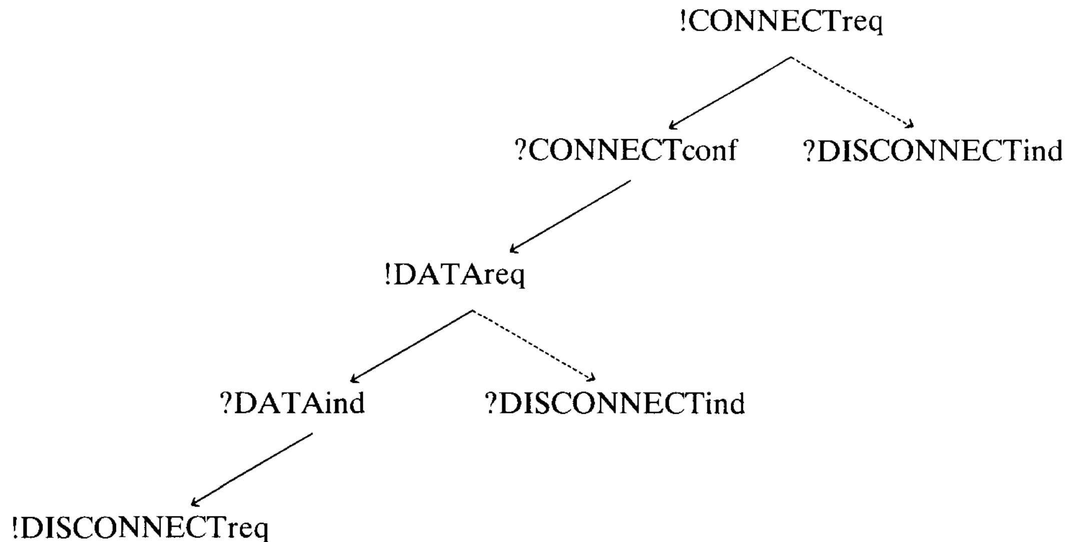

Fig. 1. A test tree.

PDU base constraints can be modified to create a new constraint by using the REPLACE/OMIT mechanism. As an example we can modify the base constraint *AARQbase_S* by omitting some of the optional fields and by replacing *protocol_version*. The resulting constraint is shown in *Table 23*.

4.4.2. ASP constraints

If ASPs are defined in ASN.1 their constraints should then be declared in ASN.1 ASP constraints. An example base constraint declaration for *N_DATArequest* ASP is shown in *Table 24*. The parameter *data* defined to be of PDU type *TPDUS* is assigned the static PDU constraint value *TCON_Class_4_1*.

4.4.3. ASN.1 type constraints

Both ASN.1 ASP and PDU constraints can be structured by using references to ASN.1 type constraints for values of complex fields. ASN.1 type constraint tables are similar to ASP constraint tables. An example type constraint for *P_address* type defined above is shown in *Table 25*.

5. Dynamic behaviour

The dynamic behaviour of test suites are declared in this section. Test case, test step and default behaviour tables comprise the dynamic behaviour.

In the dynamic behaviour specification TTCN uses the tree notation. For example, suppose the following events can occur during a test whose purpose is to establish a connection, exchange some data and then disconnect (see *Fig. 1*).

The tree notation expresses this sequence of events as:

```
TREE[L]
  L! CONNECTrequest
    L ? CONNECTconfirm
      L ! DATAreq
        L ? DATAind
          L ! DISCONNECTreq
        L ? DISCONNECTind
    L ? DISCONNECTind
```

Table 26
Test case table

Test Case Dynamic Behaviour					
Test Case Name:	EX1				
Reference:	TTCN_EXAMPLES				
Purpose:	To illustrate use of labels, GOTO and REPEAT				
Default:					
Comments					
Nr	Label	Behaviour description	Cref	V	Comments
1		GOTOTREE			
2	LA	!A	A1		
3	LB	?B	B1		
4	LB2	+ B-tree			
5	LC	?C	C1		
6	LD	[D = 1]			
7		→ LA			
8	LE	[E = 1]			
9	LF	!F	F1	F	
10		REPEATTREE			
11		(FLAG := FALSE)			
12		!A			
13		REPEAT STEP1(FLAG) UNTIL [FLAG]	A1		
14	à	!E	E1	P	
15		STEP1(F: BOOLEAN)			
16		?B (F := TRUE)			
17		?OTHERWISE	B1		

Here, L stands for the PCO at which the lower tester exercises the test. The symbols ? and ! stand for receive and send respectively. So, L! CONNECTrequest means that the tester transmits the CONNECTrequest primitive at the PCO L at this point in the test. TREE[L] is the identifier for this behaviour tree and L stands for the formal PCO used.

As can be seen from the proforma for dynamic behaviour (*Table 26*), there are six fields:

- line number (Nr),
- label (L),
- behaviour description,
- constraints reference (Cref),
- verdict (V),
- comment (C).

The *behaviour description* column contains the specification of all the possible combinations of events which may occur. There is a provision called *attach* to use library test steps. Other test events are the sending or receiving of ASPs, timer events and so-called pseudo-events, which are Boolean expressions or assignments. Goto and repeat statements are provided to specify loops and related structures.

The line number column is used to optionally indicate line numbers. This is useful especially when the behaviour lines are too long when printed. When this column is not used the line continuation can be indicated by a hash ('#') symbol located at the leftmost position within the behaviour column. The comment column is used to clarify and explain the various test events.

5.1. Label and constraints reference columns

The *label* column gives the labels for the Goto statements used in the *behaviour description* column. The *constraints reference* column is used to specify a particular value of a data type (ex. PDU) or event

which is used to be sent or received in the corresponding event. For example:

```
Event              Cref
L?CONNECTconfirm    CC1
```

The above line says that the event matches if at the PCO L, the tester receives a CONNECTconfirm if the parameter values match the constraint CC1. CONNECTconfirm must be defined in the declarations part as ASP type declaration, and CC1 should be declared as an ASP constraint in the constraints. ASP/PDU type definitions define the template in which the received event is decoded. The template disappears at the end of the event line, therefore should an access to the primitive parameters needed, they must be stored in global variables.

5.2. Verdicts column

This column is used to assign verdicts to those events at the tip of a branch of the test tree. There are two types of verdicts: preliminary and final. Preliminary results are used to indicate if part of the test purpose has been achieved, to be used in assigning final verdicts. The preliminary result can be stored in a predefined global variable called 'R'. Possible preliminary results are Pass, Fail and Inconclusive. The final verdict specifies the actual verdict to be assigned to a test case and should be computed in a manner that is consistent with the value of 'R'.

5.3. Behavior description

Event lines in TTCN are structures as in the following:

- event ::= Send | Implicit Send | Receive | Otherwise | Timeout,
- statement ::= Goto | Attach | Repeat,
- pseudo event ::= Boolean Expression | Arithmetic Expression | Timer Operation,

Test events can be accompanied by Boolean expressions, assignments and timer operations. Boolean expressions, assignments and timer operations can also stand alone constituting pseudo events.

The *otherwise* statement is used to specify the reception of unforseen or don't care events. Otherwise is used to specify that the tester will accept any event which has not been given previously as an alternative.

The *timeout* event specifies the action to be taken on the expiration of a given or default timer.

The *goto* statement is used to unconditionally jump to another part of a test tree which is labeled using the second column of the test case/ step proforma. The referenced event must be the first set of alternatives, if any.

The *attach* statement is used to modularize test trees by acting as procedure calls. The attach statement which is symbolized by '+' is used to specify that a particular test step is to be attached at that point of the tree. The test step referenced may be either a locally defined one or a library test step.

The *repeat* statement is used to specify iteration over a test step a required number of times, the minimum being once. Boolean guards may be used to come out of the loop. Since REPEAT will always succeed at least once, it is redundant to place any alternatives to a REPEAT statement.

A *pseudo event* consists of boolean, arithmetic expressions and timer operations. These may be specified on a line by themselves or may follow regular events. Boolean expressions are used as guards for events. Arithmetic expressions are performed only if the event which preceds the expression can occur and any boolean expression specified is satisfied. For example in:

L!NDATAReq [X > 1] (a := 2)

NDATAReq is sent and the variable 'a' is set to 2 only if X is greater than 1.

Timer operations consist of Start, Cancel and Readtimer statements. Start operation (re)starts the specified timer, Cancel deactivates a timer and Readtimer stores the current value of a timer into a global variable.

5.3.1. Send / receive

In send/ receive events the PCO name can be omitted if there is only one PCO in the test suite. Identifiers of ASP parameters and PDU fields associated with SEND and RECEIVE can only be used to reference ASP parameter and PDU field values on the statement line itself.

For send events relevant ASP parameters and PDU fields can be set on the SEND line, as in:

!PDU (PDU.FIELD := 3).

In case of receive events the whole ASP or PDU or a relevant part of it can be assigned to variables on the event line. These variables may then be referenced in subsequent lines. For example:

?PDU (VAR := PDU.FIELD).

5.3.2. Implicit send

In the Remote Test Architecture [2] it is necessary to have a means of specifying that the IUT should be made to initiate a particular PDU or ASP. Implicit send is used for this purpose. For example to specify that the IUT should initiate sending a CR PDU of the transport protocol we use:

⟨IUT!CR⟩.

Since implicit send is always considered to be successful, any alternatives at the same level and coded after are unreachable. No verdict can be coded on an implicit send event.

5.3.3. OTHERWISE

The predefined event OTHERWISE is for dealing with unforeseen test events. The syntax is as follows:

[pco-id] ? **OTHERWISE** verdict

If a tree uses multiple PCOs then the OTHERWISE must be preceded by a PCO identifier. OTHERWISE is used to denote that the appropriate tester accepts any incoming event which has not previously matched one of the alternatives to the OTHERWISE. Due to the significance of ordering of alternatives in tree notation, incoming events which are alternatives following an unconditional OTHERWISE on the same PCO will never be matched.

OTHERWISE can be used in conjunction with Boolean expressions and assignments. If a Boolean expression is used, this Boolean becomes an additional condition for accepting any incoming event. If an assignment is used, the assignment will take place only if all conditions for matching the OTHERWISE are satisfied. For example consider the following TTCN tree:

partial_tree(pco1:XSAP;pco2:YSAP)	
pco1?A	PASS
pco2?B [X = 2]	INCONC
pco1?C	PASS
pco2?OTHERWISE [X⟨⟩2](Reason := "X not equal 2")	FAIL
pco2?OTHERWISE (Reason := "X equals 2 but event not B")	FAIL

Assuming that no event is received at *pco1*, reception of event B at *pco2* when X = 2 gives an inconclusive verdict. Reception of any other event at *pco2* when *X*⟨⟩2 results in a fail verdict and assigns a value of *'X not equal to 2'* to the *CharacterString* variable *Reason*. The final OTHERWISE will match if an event is received at *pco2* that satisfies neither of these scenarios.

5.3.4. TIMEOUT

The TIMEOUT event allows expiration of a timer, or of all timers, to be checked in a test case. Timeout for the timer T is indicated as:

?TIMEOUT T

If the timer identifier (T) is omitted, then the TIMEOUT applies to any timer which has expired. When a TIMEOUT event is processed, if a timer name is indicated, the timeout list is searched, and if there is a timeout event matching the timer name, that event is removed from the list, and the TIMEOUT event succeeds.

5.4. Assignments and Boolean expressions

Both assignments and Boolean expressions may contain explicit values. Test suite parameters, constants, test suite and test case variables, formal parameters of a test step, default or local tree can be referenced. On event lines, ASPs and PDUs can be referenced.

To reference ASN.1 defined data objects TTCN requires a dot notation to be used. For example assume the following type definition:

```
example_type ::= SEQUENCE {field1 INTEGER,
                           field2 BOOLEAN
                           OCTETSTRING }
```

A variable, *var1* should be defined to be of *example_type*. Then in the assignments or Boolean expressions we could write:

```
var1.field1 -- to refer to the first INTEGER field
var1.(3) -- to refer to the third unnamed field
```

For PDU references assume *XY_PDUtype* defined in ASN.1 as:

```
XYPDUtype ::= SEQUENCE{ .
               user_data OCTETSTRING,
                     .
                    }
```

Then we could write on an event line:

```
L?XY_PDU (buffer ::= XY_PDUtype.user_data)
```

References to data objects defined using tables are done in a similar way by a reference to the parameter, field or element identifier followed by a dot and the identifier of the item within that substructure.

5.5. Timer management

A set of operations are used to model timer management. There are three predefined timer operations: START, CANCEL and READ TIMER. These operations can appear in combination with events or standalone pseudo-events.

The START operation is used to indicate that a timer should start running. An optional timer value parameter in parentheses can follow the timer name. In this case the default value specified in the timer declarations is overriden to assign an expiration time. As an example consider the following behaviour description and verdict columns:

```
U!TConReq START retransmission_timer
 ?TIMEOUT retransmission_timer INCONC
 U? TCONConf                   PASS
 U? TDISInd                    INCONC
 U? OTHERWISE                  FAIL
```

The test is terminated with an inconclusive verdict if the timer is expired or if a disconnection request is received from the service provider. The test is concluded with a pass verdict if a connection confirmation is received and with a fail verdict in any other cases. Another example is:

START T2(V2)

which starts a timer T2 with a default integer value V2.

The CANCEL operation is used to stop a running timer, such as in:

CANCEL T1

which cancels the timer T1, i.e., if a TIMEOUT event for T1 is in the timeout list, that event is removed from the list.

The READ TIMER operation is used to retrieve the time that has passed since the specified timer was started and to store it into the specified test suite or test case variable of type INTEGER. For example:

READ TIMER T1 (V1)

reads the amount of time which has passed since starting the timer into the test suite variable V1.

5.6. Tree attachment

Trees can be attached to other trees by using the ATTACH construct, which has the syntax:
+ tree-identifier [par-list]
A tree-identifier is the name of one of the trees in the current behaviour description; this is attachment of a local tree. Tree-identifier can be a test step identifier denoting the attachment of a test step that resides in the test step library.

The par-list defines the formal parameters. On encountering a tree attachment, the execution replaces formal parameters by actual ones (which may be PCO names, variable names, values or parameter names). Constraints may be passed as parameters to test steps. If the constraint has a formal parameter list then the constraint shall be passed together with its actual parameter list. As an example assume a subtree STEP:

```
STEP(PAR:A_PDU)
    .
    !A_PDU PAR
    .
```

Now this tree can be attached from another tree TOP_TREE:

```
TOP_TREE
    .
    +STEP(C1(3))
    .
```

assuming that the constraint C1 has a single formal parameter of type INTEGER.

5.7. GOTO and REPEAT

A GOTO to a label may be specified within a behaviour tree provided that the label is associated with the first of a set of alternatives, one of which is an ancestor node of the point from which the GOTO is to be made. A GOTO is specified by placing an arrow ($\rightarrow$) or the keyword GOTO, followed by the name of the label, on a statement line on its own in the behaviour tree, at the appropriate level of indentation. A test case with an example GOTO is shown in *Table 26*.

The REPEAT construct describes a mechanism for iterating a test step a number of times. The syntax of this construct is:
REPEAT tree-reference (par-list) UNTIL [Boolean-exp]
where the tree-reference is a reference to either a local tree or a test step defined in the test step library. Par-list is an optional parameter list of actual parameters and Boolean-exp is a Boolean expression used to control the iteration.

The REPEAT construct has the following meaning: first the tree is executed. Then the Boolean expression is evaluated. If it evaluates to TRUE, execution of the REPEAT construct is completed. If not, the tree is executed again, followed by the evaluation of the Boolean expression, and this process is repeated until the Boolean expression evaluates to TRUE. A test case with an example REPEAT construct is shown in *Table 26*.

5.8. Referencing constraints in dynamic behavior tables

Constraint names are placed in the constraint reference column of test case/ step or default behaviour tables along with the name of the ASP/ PDU to which it applies. A constraint reference takes the form of:
cons-name(par-list)
where the cons-name is the name of the reference which must be defined in one of the constraint tables, and par-list is an optional (possibly nested) parameter list defining the actual parameter values to the formal parameter list.

Table 27
A TP2 test case

Test Case Dynamic Behaviour				
Test Case Name:	ABCT2UMA00			
Group:	TP2/Valid/DataTransfer			
Purpose:	Send expedited data and receive normal data			
Default:	Def1			
Comments:				
Label	Behaviour description	CRef	V	Comments
	+Preamble (result := open) +Send_ed(16) +Rec_data(1,250) [result = open] +Give_counts +Postamble [result = fail] +Postamble			 Send Exp. Data to UT Receive Normal Data from UT Check UT gotten data

A constraint reference with a parameter list is:
NSAP?NDATA_req D1(P1,CR1(P2))
where D1 is a constraint on N_DATAreq with two parameters (actual parameters P1 and CR1), and CR1 is a constraint with one parameter (actual parameter P2).

6. Example test case

A valid behaviour test case for the data transfer capability of the transport protocol class 2 is shown in *Table 27*. This test case is taken from the ATS developed within the CTS-WAN project [10]. The architecture used is the coordinated single layer (CS) which assumes a single PCO, L, and a test management protocol to communicate with the upper tester on the IUT side.

The purpose of the test case is to send 16 octets of expedited data and receive 250 octets of normal data from the IUT. The test management protocol that is part of this ATS is used to instruct the upper tester to introduce the normal data. This part of the test case is omitted from *Table 27*. The subtrees *Preamble* and *Postamble* establish/ disconnect the transport connection required. These subtrees are not shown.

The subtrees *Send_ed* and *Rec_data* are shown in *Tables 28* and *29* as test step trees.

According to TTCN semantics, the execution of the test case *ABCT2UMA00* will take place in two steps:

1. Obtain the test case in pure tree form – static semantics,
2. Execute this tree – snapshot semantics.

In order to get the test case in pure tree form the behaviour specification is scanned and all tree attachments are removed by body replacements of the trees attached, starting with the subtree *Preamble*. Next the subtree *Send_ed* is attached at level 3. In this attachment the formal parameter *param* is replaced by the actual parameter 16. Step 1 terminates after the last subtree *Postamble* is attached. Finally, default tree *Def1* is attached to every level.

In Step 2 the pure tree obtained from Step 1 is interpreted considering the PCOs as queues and the timeout list. Interpretation starts at level 1. The behaviour lines in this level are evaluated in sequence, until a successful match is found. With a match the level is changed and a new set of alternatives are considered. Test case interpretation stops at the leaf nodes in the tree by resetting the values of all test case variables and all timers.

Table 28
Send_ed test step

Test Step Dynamic Behaviour				
Test Step Name:	Send_ed			
Group:	TP2/Valid/DataTransfer/Subtrees			
Objective:	Send expedited data of param size			
Default:				
Comments:				
Label	**Behaviour description**	**CRef**	**V**	**Comments**
	Send_ed(param)			
	L!ED Start B	ED1(param)		Send param octets of expedited data
				Start Timer B
	L?EA	EA1		Receive EA
	Cancel B		(P)	
	L?OTHERWISE			
	?TIMEOUT B			
	(result := fail)			IUT not responding
	+Postamble		(F)	

Suppose the interpretation of the test case *ABCT2UMA00* has reached to the first event line in the test step *Send_ed*. Since this is a send event and there is no Boolean guard specified a successful match is found. Corresponding action is to perform the sending by first creating a SendObject and then placing it into the output queue at the PCO *L*. The SendObject in this case is the ASP *TEXTDATAreq* containing an *ED* PDU formed with values obtained from the constraint *ED1*, i.e. containing 16 bytes of data.

Similarly, the receive event is executed if the subsequent event in the queue matches the constraints in the receive event. Any mismatch would cause a match with the OTHERWISE event at the same indentation.

Table 29
Rec_data test step

Test Step Dynamic Behaviour				
Test Step Name:	Rec_data			
Group:	TP2/Valid/DataTransfer/Subtrees			
Objective:	To receive "tsdu" TSDUs of "octet" octets			
Default:	None			
Comments				
Label	**Behaviour Description**	**CRef**	**V**	**Comments**
	Rec_data(tsdu, octet)			
	(rec_tsdu := 0)			No tsduss received so far
T1	[rec_tsdu = tsdu]			All tsdus received?
	(result := open)		(P)	If yes, quit
	[rec_tsdu⟨tsdu]⟩			
T2	Start B			
	L?DT	DT52		Not end of TSDU
	Cancel B			
	L!AK → T2	AK51		
	L?DT	DT51		DT with end of TSDU
	Cancel B			
	L!AK	AK51		
	(rec_tsdu := rec_tsdu + 1) → T1			
	L?OTHERWISE			
	?TIMEOUT B		(F)	IUT not responding
	(Result := fail)			

```
$Begin_TTCN_ASP_TypeDef
$ASP_Id CONreq (T_CONNECTrequest)
$PCO_Type TSAP
$ASP_ParDcls
$ASP_ParDcl
$ASP_ParId CdA (Called Address)
$ASP_ParType T_Address_info
$End_ASP_ParDcl
$ASP_ParDcl
$ASP_ParId CgA (Calling Address)
$ASP_ParType T_Address_info
$End_ASP_ParDcl
$ASP_ParDcl
$ASP_ParId Qos (Quality of Service)
$ASP_ParType QOs
$End_ASP_ParDcl
$End_ASP_ParDcls
$End_TTCN_ASP_TypeDef
```

Fig. 2. Example MP form.

7. TTCN-MP

Machine processable form of TTCN, TTCN-MP is a linear form of the TTCN-GR (graphic) that is discussed above with the difference that tokens serve as delimiters between fields instead of tables. In the MP form, a definite order in which the various parts of TTCN must appear is given. Another difference in the MP form is that, in the dynamic behaviour, indentation is explicitly denoted. For example,

```
L!ASP1
   L?ASP2
```

would be represented in TTCN-MP as:

```
$BehaviourLine $Line[0] L!ASP1 $End_BehaviourLine
$BehaviourLine $Line[1] L?ASP2 $End_BehaviourLine
```

Complete tables defined in TTCN.GR are represented in TTCN.MP by productions of the kind:

$Begin_KEYWORD ... ...$End_KEYWORD

Both sets of lines of a table and sets of fields are represented by productions of the kind:

$KEYWORD ... ...$End_KEYWORD

Individual fields in a line are represented by:

$KEYWORD ... ...

An example MP form of the ASP definition given in *Table 14* is given in *Fig. 2.*

8. Test realization

Test realization is based on the abstract test suite (ATS), PICS and PIXIT. PICS proforma is basically used to select tests from the ATS and PIXIT is used to parameterize the selected tests.

The abstract test suite becomes a Selected ATS after the selection mechanism and a Parameterized ATS after the parameterization, and its executable form is called a Parameterized Executable Test Suite (PETS). This process is shown in *Fig. 3.*

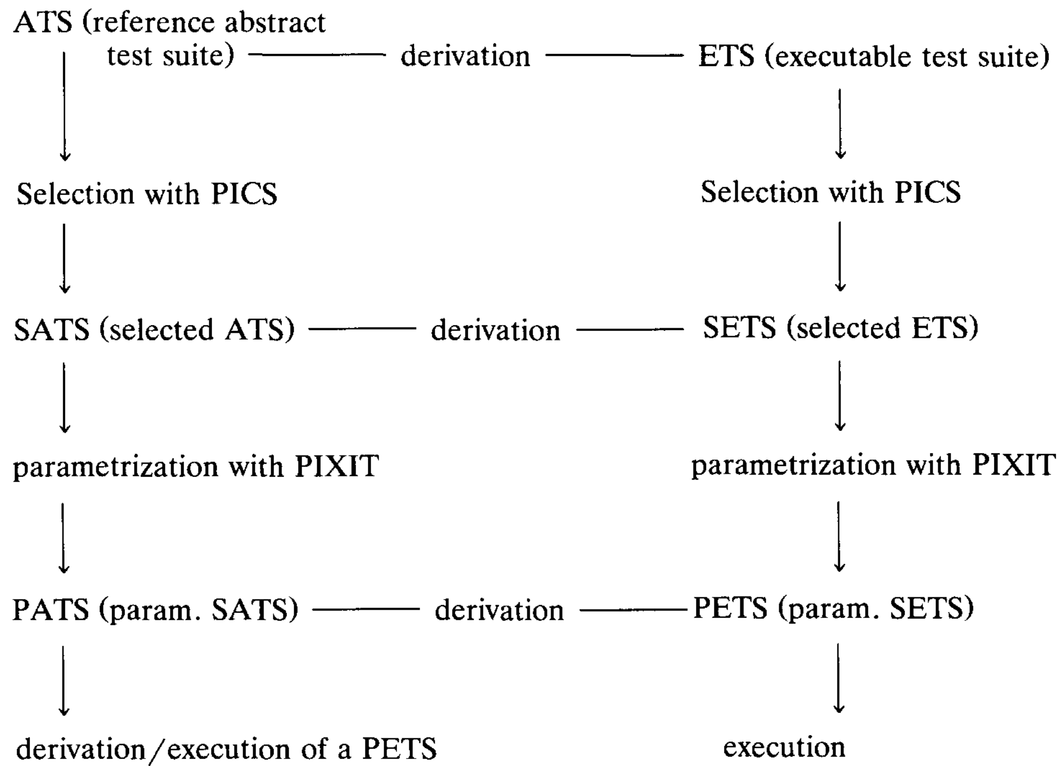

Fig. 3. The PETS derivation process.

8.1. Suite overview

The test suite overview table is for describing structure of the ATS and for providing an index of its test cases and steps/ defaults. Test suite structure table describes grouping and objectives of each group in an ATS. For example the test suite structure of the test suite called ATS_2 is shown in *Table 30*. The selection expressions *SELEXP_100* and *SELEXP_101* are evaluated to determine if test cases in the groups *ABCT2BAS00* and *ABCT2BVE00*, respectively can be executed. These expressions are defined in test case selection expression definition tables described in Section 3.

The test suite structure proforma must be followed by test case/ step and default index tables. An example test case index table is shown in *Table 31*. Selection reference column may contain a selection

Table 30
Test suite structure proforma

Test Suite Structure			
Suite Name: ATS_2 Standards Ref: IS 8072/8073 PICS Ref: IS 8073/AM3 PIXIT Ref: Test Method(s): Coordinated single layer Comments: Transport abstract test suite for class 2			
Test Group Reference	Selection Ref	Test Group Objective	Page Nr
ABCT2BAS00	SELEXP_100	Basic Interconnection	20
ABCT2BVE00	SELEXP_101	Valid Behaviour	38

Table 31
Test case index table

Test Case Index				
Test Group Reference	Test Case Id	Selection Ref	Description	Page Nr
ABCT2BVE00	ABCT2URA00	Sel_ura00	Connection Establishment Test Group	39
ABCT2BVE00	ABCT2CRA00	Sel_cra00	idem	40

expression which should evaluate to true for the test case to be executed provided that the group selection expression specified in test suite structure table also evaluates to true. Selection expressions should be defined in test case selection expression definitions table. Test step and default index tables are similar, but they do not contain the selection reference column.

9. Conclusions

We described a tabular language designed for specifying conformance test suites. TTCN divides a test specification into four parts. Test suite overview part is for documentation, declarations part is for declaring types as well as templates for the PDUs and ASPs. Dynamic behaviour part specifies dynamic behaviour of the tests using a tree notation. This part modularizes the test specification into test cases which are made up of one or more test steps. Test cases/ steps can have default behaviour. Constraints on the parameter values of the events to be sent/ received are declared in the constraints declaration part.

Despite its unusual appearance, TTCN should be treated as a high-level test specification language. As such it has potential to become a formal technique to study various test architectures, conversion of the tests designed for one architecture to another. Those test suites that are manually derived have to be validated for correctness. In this case some sort of comparison of the TTCN dynamic behaviours with the behaviours derived from the formal specification of the protocol is needed.

Conversion of TTCN specifications to an executable form is being investigated. It is desirable to obtain executable test suites from abstract test suites, i.e. executable TTCN, such as in the case of formal specification techniques. It would also be interesting to investigate how test suites in TTCN form could be semi-automatically obtained from FDTs such as SDL, Estelle and LOTOS.

TTCN extensions [8] such as parallel trees are presently being progressed in the standardization committees. How to treat spontaneous outputs from the implementations, nondeterminism in ASN.1 encoding/ decoding are among the issues being investigated.

References

[1] ISO, Data Processing – Open Systems Interconnection – Basic Reference Model, ISO International Standard 7498, 1984.
[2] ISO, Conformance Testing Methodology and Framework, IS 9646 Parts 1 and 2, February 1989.
[3] ISO, Conformance Testing Methodology and Framework, IS 9646 Part 3 : The Tree and Tabular Combined Notation (TTCN), September 1991.
[4] ISO, Protocol Specification for the Association Control Service Element, IS 8650, 1988.
[5] ISO, Information processing – Open Systems Interconnection – Specification of Abstract Syntax Notation One, (ASN.1), IS 8824, 1987.
[6] ISO, Abstract Syntax Notation One-Encoding Rules, IS 8825, 1987.
[7] ISO, X.25 – DTE Conformance Testing : Data Link Layer Test Suite, DIS 8882 Part 2, 1990.
[8] ISO, Working Draft Amendment to ISO/IEC 9646-3: TTCN Extensions, ISO/IEC JTC1/ SC21 9646-3 PDAM1, June 1991, 18p.
[9] PTT-NL, ACSE Abstract Test Suite (version 2.00), PTT Netherlands, January 1991.
[10] CTS-WAN, ISO/OSI Transport Class 2 CTS-WAN Abstract Test Suite (version 2.1), National Computing Centre Ltd., UK, October 1988.

Behcet Sarikaya received his B.S.E.E. degree (honors) from the Middle East Technical University (METU), Ankara, Turkey in 1973, M.Sc. degree in Computer Science from METU in 1976, and Ph.D. degree in Computer Science from McGill University, Montreal, Canada, in 1984.

He worked in the Universities of Sherbrooke and Concordia as Assistant Professor. He is presently working in the Department of Computer and Information Sciences, Bilkent University, Ankara. He has published over 40 papers in protocol engineering and related areas. He has been co-chairman of the conference IFIP PSTV VI held in Montreal in 1986. He served in the program committees of all three protocol conferences. His current research interests lie in all aspects of conformance testing and high-speed networks. He is a senior member of IEEE.

He is actively involved in OSI standardization activities. He is an active member of the joint CCITT SG X Question 10, ISO IEC JTC1/SC21 Project 54 on Formal Methods in Conformance Testing. This committee aims at developing a standard that will relate the Formal Description Techniques to the Conformance Testing Methodology and Framework.

Anthony Wiles has been actively involved in the national and international development of the ISO conformance testing standard (ISO/IEC 9646) since 1984. His participation in the ISO/IEC JTC1/SC21/WG1 standards committees includes being editor of the TTCN (ISO/IEC 9646-3). He also actively participates in CCITT working groups on conformance and is a project rapporteur within ETSI ATM (Advanced Testing Methods).

He graduated from Uppsala University, Sweden with a BSc. in Technical Physics and an MSc. in Computer Science. His work has included the implementation of several OSI protocols (including Transport and FTAM) and management of one of the first X.25 networks in Sweden. He currently specializes in all aspects of protocol validation and testing, and is project manager and conformance expert at Swedish Telecom's test laboratory, TeleTest, in Stockholm. He previously worked in the protocol communications department at the Swedish Institute of Computer Science (SICS).