Chapter 22 *TTCN Access*

This chapter describes TTCN Access. It is intended to be read by developers of executable test suites, translator developers and test result analyzers. It requires some basic programming knowledge in C++ as TTCN Access is a C++ programming tool.

Introduction to TTCN Access

A TTCN test suite can be looked upon as a formal description of a collection of test sequences where each test sequence involves signals and values. The abstract test suite contains formal definitions of these signals and values as well as a structural description of each test sequence. A common formal notation used to describe these test sequences, as well as all the other items in an abstract test suite, is TTCN.

TTCN Access is a C++ application programmers interface towards an arbitrary abstract test suite written in TTCN and incorporated in the TTCN suite. TTCN Access reveals the content of the test suite in a high level abstraction and allows various users to access the content in a read-only manner.

TTCN Access is a platform for writing applications related to an abstract test suite such as:

- Executable test suites
- Interpreters
- Encoders and decoders
- Analyzers
- Reporters

By using TTCN Access, a variety of applications can be implemented that will ease the maintenance of abstract test suites and the development of executable test suites. TTCN Access is an easy-to-use application programmers interface that provides the required functionality for applications in these areas.

Terms Used in This Document:

- ISO/IEC 9646-3 : 1991 is called *the TTCN standard*.
- ISO/IEC 8824 : 1990 is called *the ASN.1 standard*.
- Backus-Naur Format is called *BNF*.

General Concepts

The easiest, and simplest way of describing TTCN Access would be to state that TTCN Access is a TTCN compiler. Unfortunately this statement is not fully true and also somewhat misleading as it might, conceptually, bring the reader (and TTCN Access users) towards the domain of executable test suites and thereby not reveal all the other possibilities that TTCN Access brings. A more correct statement would thereby be that TTCN Access is a half compiler, more precisely the first phase of a compiler, often called the *front-end*. For a general explanation of compiler theory and compiler front-ends, see chapter 21, *Basic Compiling Theory*. The fact is that the TTCN to C compiler is built on top of TTCN Access.

TTCN Access and the TTCN Analyzer

All the components and definitions mentioned in chapter 21, *Basic Compiling Theory*, together build up the basics for compiling theory. As mentioned, TTCN Access is a C++ application programmers interface towards a test suite written in TTCN. It reveals the content of the test suite in a high level abstraction and allows various users to access the content in a read-only manner.

This section will once again mention these components, but this time put them in the context of the TTCN suite and the functionality that it provides, thereby explaining the relationship between the TTCN suite and TTCN Access.

Lexical Analysis

The lexical analysis in the TTCN suite is done in two phases, depending on the format of the abstract test suite.

- If the source code is in MP format, a basic lexical analysis is done at the import stage, verifying that the imported test suite is written in correct MP format. A full lexical analysis is done when using the Analyzer. The complete lexical analysis is the first phase in the analysis.

  For more information see "Importing a TTCN-MP Document" on page 1149 in chapter 25, *The TTCN Browser (on UNIX)*.

- If the source code is written directly into the TTCN suite via one of the editors, a lexical analysis is done by using the Analyzer as mentioned above.

  For more information see chapter 27, *Analyzing TTCN Documents (on UNIX)*.

Syntax Analysis

The syntax analysis is done when executing the Analyzer and it is done in the second phase. This second phase also contains a semantic analysis of the test suite, all in order to verify that the test suite complies with the standardized notations for TTCN and ASN.1.

Parse Tree

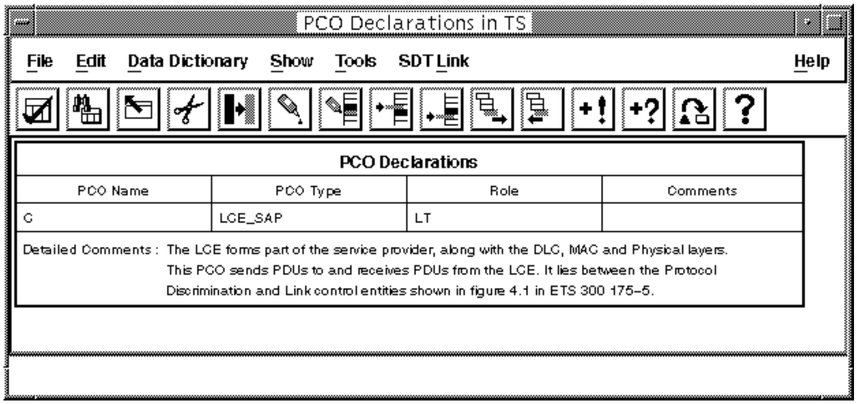
The last phase of the analysis is to generate a parse tree. This can only be done if the lexical and syntax analysis have been successfully completed.

Symbol Table Management

During execution of an TTCN Access application the symbol table is accessible at any time.

Example of TTCN Access Functionality

As TTCN Access looks upon a test suite as a parse tree, it provides the required primitives and functionality for parse tree handling. This includes parse tree traversing and hooks into the parse tree as well as templates for main() and several examples in source code format. The best way to visualize the functionality of TTCN Access is by using an example:



PCO Declarations

PCO Name	PCO Type	Role	Comments
C	LCE_SAP	LT	

Detailed Comments : The LCE forms part of the service provider, along with the DLC, MAC and Physical layers. This PCO sends PDUs to and receives PDUs from the LCE. It lies between the Protocol Discrimination and Link control entities shown in figure 4.1 in ETS 300 175-5.

Figure 181: The PCO Declarations table

The PCO Declarations table, containing just one PCO declaration will generate the parse tree visualized below:

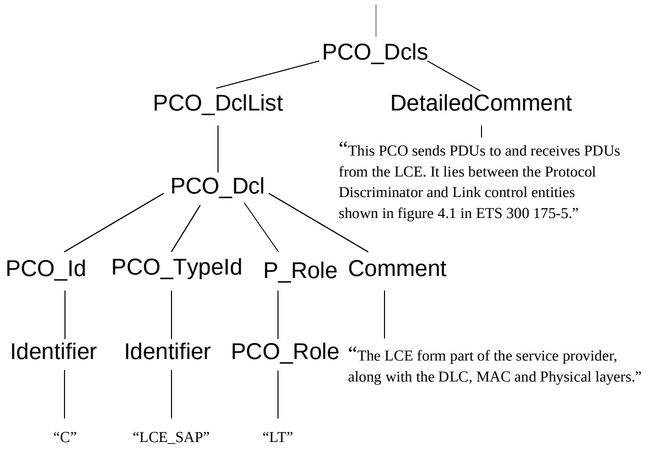

PCO_Dcls
PCO_DclList
DetailedComment
"This PCO sends PDUs to and receives PDUs from the LCE. It lies between the Protocol Discriminator and Link control entities shown in figure 4.1 in ETS 300 175-5."
PCO_Dcl
PCO_Id
PCO_TypeId
P_Role
Comment
Identifier
Identifier
PCO_Role
"The LCE form part of the service provider, along with the DLC, MAC and Physical layers."
"C"
"LCE_SAP"
"LT"

Figure 182: Parse tree

Traversing

The basics of parse tree functionality has to include traversing primitives. TTCN Access therefore provides a special *visitor* class with a *default traverser* that, given any node in the parse tree, traverses that parse tree in a *Left-Right-Depth-First* manner. In the parse tree previously described, the default traversing order for the sub tree *PCO_Dcl* will be:

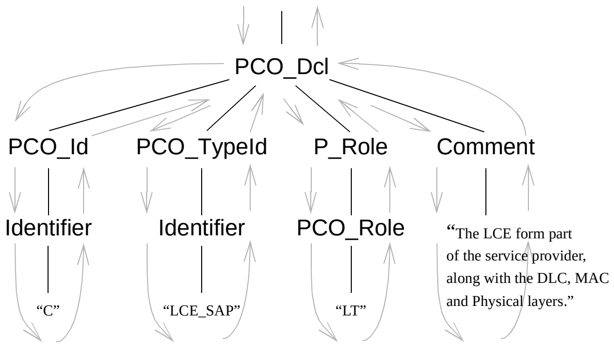

PCO_Dcl
PCO_Id
PCO_TypeId
P_Role
Comment
Identifier
Identifier
PCO_Role
"The LCE form part of the service provider, along with the DLC, MAC and Physical layers."
"C"
"LCE_SAP"
"LT"

Figure 183: Default traversing tree

This default translating algorithm may be *customized* in order to fit any user specific traversing order. For the above example, a customized traverser could be implemented as traversing sub tree *PCO_TypeId* before sub tree *PCO_Id* and ignoring sub tree *Comment*.

Translating

The second most important functionality when discussing parse trees, is the possibility to access specific nodes in the parse tree in order to generate side effects, such as executing external code, print parse tree information, etc. By being able to modify the default traverser, TTCN Access provides the user with the possibility of defining such side effects.

Example of TTCN Access Usability

To visualize the strength of TTCN Access, this section will discuss a small TTCN Access application. The example is a generic encoder.

The Encoder

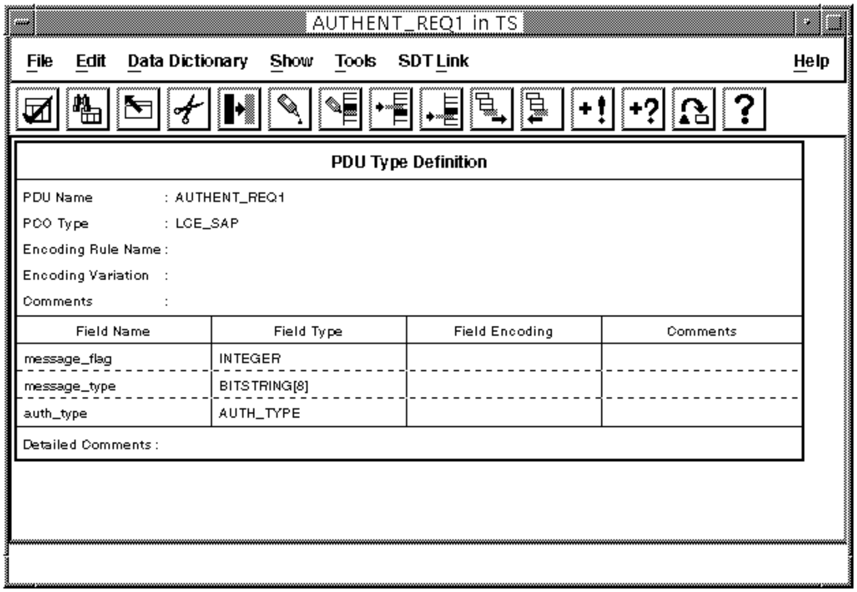
Transforming an abstract test suite into an executable test suite will eventually involve the problem of representing the actual ASP and PDU signals as bit patterns. The step from abstract syntax (TTCN) to transfer syntax (bit patterns) has to be provided and implemented by someone with vast knowledge of the actual protocol as well as the test system and the test environment.

The encoder described in this small example will assume the following:

- Elements of the same type are encoded identically.
- Encoding an ASP or PDU is performed by encoding its components recursively.

These assumptions tell us that encoding functions are data driven and that the encoding function has to be derived from the type definitions. It also tells us that the encoding of base types, that is INTEGER, BOOLEAN, etc., are identical.

Before looking at the algorithm for the generic encoder, let us have a look at a specific type and how the corresponding encoding function may be implemented. The type will be a small PDU as below:



PDU Type Definition

PDU Name	: AUTHENT_REQ1
PCO Type	: LCE_SAP
Encoding Rule Name	:
Encoding Variation	:
Comments	:

Field Name	Field Type	Field Encoding	Comments
message_flag	INTEGER		
message_type	BITSTRING[8]		
auth_type	AUTH_TYPE		

Detailed Comments :

Figure 184: TTCN PDU type definition

The corresponding encoding function may look like:

```
void AUTHENT_REQ1_encode( AUTHENT_REQ1* me, char* enc_buf )
{
  INTEGER_encode( &me->message_flag, enc_buf );
  BITSTRING_encode( &me->message_type, enc_buf );
  AUTH_TYPE_encode( &me->auth_type, enc_buf );
}
```

It is a data driven encoding function that given any element of type `AUTHENT_REQ1` will encode them all identical. The first argument is a pointer to the element that shall be encoded, the second argument is the buffer where the result of the encoding shall be stored.

The function may of course contain more code and more information, but for this example, the above is the smallest encoder function needed.

The following example will give an algorithm for how the encode functions may be generated via TTCN Access. The algorithm is followed by a small TTCN Access application generating the actual encoder.

```
FUNCTION GenerateEncoder( type : typedefinition )
BEGIN
/* Generate encoder function header such as: */
/* void type_encode( type* me, char* enc_buf ) */
/* { */
FOR( “every element in the structured type” )
/* Generate a call to the element type encoder
/* function
/* such as: */
/* element_type_encode( &type->element, enc_buf );
END
/* Generate encoder function footer such as: */
/* } */
END
```

The above algorithm will generate generic encoders for any type. Of course, header files and base type encoder functions must be provided/generated as well. Below is a nearly complete TTCN Access application that generates encoders for any TTCN PDU type definition. Observe how the visitor class is customized to traverse and generate side effects for the specific parts we are interested in.

```
// Start by customizing the default visitor class

class Trav : public AccessVisitor
{
public:
  void VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me );
  void VisitPDU_FieldDcl( const PDU_FieldDcl& Me );
};

void Trav::VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me )
{
  // Generate the header part

  cout << Me.pdu_Identifier( ) << “_encode( const “;
  cout << Me.pdu_Identifier( ) << “& me, char* enc_buf )”;
  cout << “\n{“ << endl;

  // Traverse fields using default traverser

  AccessVisitor::VisitPDU_FieldDcls( Me.pdu_FieldDcls( ) );

  // Generate the footer part

  cout << “}” << endl;
}

void Trav::VisitPDU_FieldDcl( const PDU_FieldDcl& Me )
{
  Astring type = get_type( Me.pdu_FieldType( ) );
  Astring name = get_name( Me.pdu_FieldId( ) );

  cout << “void “ << type << “_encode( me->”
       << name << “( ), enc_buf );” << endl;
}
```

All that remains now is a main function for the generic encode generator.

```
#include     <stdio.h>
#include     <iostream.h>
#include     <time.h>
#include     <ITEXAccessClasses.hh>
#include     <ITEXAccessVisitor.hh>

class Trav : public AccessVisitor
{
public:
  void VisitTTCN_PDU_TypeDef( const TTCN_PDU_TypeDef& Me );
  void VisitPDU_FieldDcl( const PDU_FieldDcl& Me );
};

int main( int argc, char **argv )
{
  //The actual suite

  AccessSuite suite;

  if ( argc != 2 )
    {
      fprintf( stderr, "usage: %s suitename\n", argv[ 0 ] );
      return 0;
    }

  /* Open the suite and go for it! */

  if ( suite.open( argv[ 1 ] ) )
    {
      Trav trav;

      trav.Visit( suite );
      suite.close( );
    }
}
```

The `get_type` and `get_name` functions can be simple or complex. See the example below:

```
Astring get_type( const PDU_FieldType& Me )
{
  TypeAndAttributes ta = Me.typeAndAttributes( );
  switch ( ta.choice( ) )
    {
    case Choices::c_TypeAndLengthAttribute:
      return get_type( ta.typeAndLengthAttribute( ) );
    default:
      cerr << "ERROR: Type not supported: "
           << Me.content( ) << endl;
      return "***#***";
    }
}


Astring get_type( const TypeAndLengthAttribute& Me )
{
  const TTCN_Type tt = Me.ttcn_Type( );

  switch ( tt.choice( ) )
    {
```

```
    case Choices::c_PredefinedType:
      if ( tt.predefinedType( ).choice( ) ==
           Choices::c_INTEGER )
        return "INTEGER";
      break;
    case Choices::c_ReferenceType:
      return tt.referenceType( ).identifier( );
    default:
      break;
    }
  cerr << "ERROR: Type not supported: "
       << Me.content( ) << endl;
  return "***#***";
}


Astring get_name( const PDU_FieldId& Me )
{
  const PDU_FieldIdOrMacro pfid = Me.pdu_FieldIdOrMacro( );
  if ( pfid.choice( ) != Choices::c_PDU_FieldIdAndFullId )
    {
      cerr << "ERROR: Slot name not supported: "
           << Me.content( ) << endl;
      return "***#***";
    }

  return pfid->pdu_FieldIdAndFullId( ).pdu_FieldIdentifier( );
}
```

TTCN Access in Relation to TTCN and ASN.1

TTCN Access is created for the purpose of accessing and traversing the contents of a TTCN test suite. As a basis for the development of TTCN Access, the TTCN-MP syntax productions in BNF [ISO/IEC 9646-3, appendix A] together with the ASN.1 standard [ISO/IEC 8824] have been used.

Differences

Every node in TTCN Access reflects a rule in the BNF. However there are some small differences between the BNF and TTCN Access:

The Present Nodes

The intention is to map the ASN.1 and TTCN standards as good as possible. In a perfect world this would mean that no extra nodes could be found in TTCN Access and that each rule in the standards would have exactly one corresponding node in TTCN Access. Some differences between the perfect world and our world are listed in detail below. Differences exist due to redundancies in the standardized grammar or design decisions made during the development of TTCN Access. The goal of any implementation of TTCN Access is to make as few changes compared to the perfect world as possible. However, some extra nodes have been added to TTCN Access, and in a few places the exact calling order of the pre/post functions has been altered to what we feel is a more straight forward approach for executable languages.

The SEQUENCE OF

The TTCN and ASN.1 notations at times allow for arbitrary many nodes to be grouped in lists of nodes or sequences of nodes. For example there is the SubTypeValueSetList rule of the ASN.1 standard and there is the AssignmentList rule of the TTCN standard. In both BNF notations a difference is made between lists that are allowed to be empty and those which must contain at least one element, corresponding to the {} and {}+ notation in the TTCN standard.

In TTCN Access no difference is made between the two forms of lists, since a well defined and simple access method is of prime interest. All lists are allowed to be empty, and it is the work of the Analyzer to ensure non-empty lists where appropriate. In the ASN.1 standard the non-empty lists are sometimes written by re-grouping already existing rules. This re-grouped structure will not be represented in TTCN Access. The decision does not alter the number of nodes or the names of nodes, but merely the visiting order when traversing the nodes, i.e. SubTypeValueSetList before SubTypeValueSet.

Example 161 ––

ASN.1 rule SubTypeSpec:

```
SubTypeSpec ::= ( SubTypeValueSet SubTypeValueSetList )
```

becomes:

```
SubTypeSpec ::= ( SubTypeValueSetList )
```

––

In the TTCN standard there are nodes which contain an implicit grouping.

Example 162 ––

TTCN BNF rule:

```
TestStepLibrary ::= ({TestStepGroup | TestStep})
```

becomes:

```
TS_ConstDcls ::= {TS_ConstDcl}+ [Comment].
```

––

Furthermore there are nodes that contain groupings without the list postfix name convention.

Example 163 ––

TTCN BNF rule:

```
TimerOps ::= TimerOp {Comma TimerOp}.
```

––

Whenever necessary, a grouping or/and choice node will be inserted between rules in the TTCN standard. When nodes are created without corresponding node in any of the standards, the nodes will be named according to the convention used in the ASN.1 standard, i.e. these nodes will have postfix “List”. Nodes which are a grouping but do not follow the naming convention will be left as they are.

OPTIONAL

In the BNF rules for TTCN-MP, the use of `[ abc ]` implies zero or one instance of `abc`. In the definition of TTCN Access the same effect is achieved by the use of the symbol “OPTIONAL” after the TTCN Access definition. If a TTCN Access element defined as “OPTIONAL” is not present in a specific instance, no TTCN Access representation for that specific field will be available. To detect whether or not a TTCN Access node defined as “OPTIONAL” is present or not, a boolean TTCN Access function will be necessary to apply to that TTCN Access node in order to verify the presence/absence of the parse tree related to the specific slotname.

FIELD

A difference between TTCN Access and the BNF rules is that TTCN Access views every BNF production reflecting a field in the TTCN-GR format to be optional even though the field is not defined as such in the BNF. The reason for this is that no TTCN Access tree can be built for a field unless a successful analysis has been performed on that field. If the analysis failed, the field will not carry any TTCN Access information and the field may not be accessed.

In TTCN Access all fields are marked with the symbol “FIELD”. This “FIELD” symbol implies that the field may be empty. The field may be empty even though the field is not defined as optional in the BNF. To detect whether or not a field is present or not present, a boolean TTCN Access function will be necessary to apply to that TTCN Access node in order to verify the presence/absence of the parse tree related to the specific field. This extension is applicable to all fields except if a field is implemented in TTCN Access as a node of type TERMINAL. All TERMINAL types are strings and TTCN Access sees the absence of a TERMINAL as an empty string.

The Value Notation

The Common Value Notation

The value notations of the two standards must be unified. Some values are clearly within one standard only (SetValue for instance), while others are in both standards (7 for instance). Those values that have the same syntax in both standards are considered to belong to the TTCN standard. Note that the ASN.1 standard might specify some other chain of productions to reach such a value and that this chain of productions is not represented in TTCN Access.

The Expression Tree

The expression rule has been re-written since it is undesirable to have the flat representation from the standard. Instead a normal tree oriented representation of an expression is used. This means that the rules making up an expression are heavily changed.

The Base Nodes

Base nodes are the representation of the terminals in the TTCN Access tree. They can in most aspects be treated as strings, they can be printed directly for example. The base nodes are:

- Identifier
  Has an associated type
- Number
- All nodes having a single child of type BoundedFreeText (FullIdentifier or SO_SelExprId for example).
- Ostring, Cstring, Bstring and Hstring

Note:

The Keyword class in ITEX Access 1.0 does not exist in ITEX Access 2.0. Neither of the standards has the notion of a keyword class/rule, and therefore there is no node/class named Keyword in ITEX Access 2.0. It has been replaced with ITEX Access node TERMINAL.

Tree Traversing in the Dynamic Part

An extension to the BNF adds the possibility to access the content of test cases, test steps and defaults, by using the logical tree structure of the test. This is achieved by adding a slot in the node type definition BehaviourLine named “children”. By accessing that slot an TTCN Access object is returned that holds a vector of children to the current statement line. A child is a BehaviourLine with a level of indentation one greater that the preceding BehaviourLine.

Naming Conventions

As a basis for naming TTCN Access nodes, the names in the BNF for TTCN and ASN.1 have been used. There are some differences though, and they will be discussed and explained in this subsection.

- No slot name, type name or type definition contains the symbol “&” as it is a reserved symbol in C++. It is replaced with “and” or “And” where appropriate.
- All type definitions start with an upper case letter (and are written in bold).
- All slot names start with a lower case letter.
- All type names start with an upper case letter.

The TTCN Access Notation

The notation used to describe a node in TTCN Access is described as below (a simple BNF for an example node):

```
Node           ::= TypeReference Assign TypeAssignment
TypeReference  ::= Identifier
            --has to start with an upper case letter
Assign         ::= “::=”
TypeAssignment ::= ClassType ClassBody | “TERMINAL”
ClassType      ::= “SEQUENCE” | “SEQUENCE OF” |
“CHOICE”
ClassBody      ::= “{“ { Slot }+ “}”
Slot           ::= SlotName TypeReference
                   [ “OPTIONAL” | “FIELD”]
SlotName       ::= Identifier
             --has to start with a lower case letter
```

Example 164 ––

```
StructTypeDef ::= SEQUENCE {
    structId          FullIdentifier  FIELD
    comment           Comment         FIELD
    elemDcls          ElemDcls
    detailedComment   DetailedComment FIELD
}
```

––

- A TTCN Access node type is defined in **Bold** starting with an upper case letter.
- A node can be of type SEQUENCE, SEQUENCE OF or CHOICE. A SEQUENCE contains an ordered collection of elements, a SE-

QUENCE OF a vector of elements (possibly empty) and a CHOICE is a collection of possible elements.

- Each slot in a *TypeAssignment* starts with a *SlotName* followed by a *SlotType*. Each *SlotType* has a corresponding node in TTCN Access.
- The *SlotType* can be followed by a `FIELD` symbol meaning that the slot is associated with a field in the TTCN-GR format.
- The *SlotType* can be followed by a `OPTIONAL` symbol meaning that the slot can be absent.
- Each slot is followed by a page reference to the node corresponding to the *SlotType*. This page reference is not a part of the Abstract Data Structure.

For more information see chapter 24, *The TTCN Access Class Reference Manual*.

TTCN Access Primitives

Each node in TTCN Access is translated to a C++ class definition. Each instance of a specific C++ class definition contains one or more elements as defined in the specific C++ class definition. For each type definition there are a collection of TTCN Access functions.

In this chapter names written in *italic* are referred to as meta names. Words written in `courier` are taken from the definition of TTCN Access.

Note:

For further details see the TTCN Access include file `access.hh`.

General TTCN Access Functions Description

- For every node definition with assignment of type SEQUENCE or SEQUENCE OF there exists a general method

  SlotType TypeReference.SlotName()

  that returns an object of type *SlotType*. This type is the type of the corresponding slot in TTCN Access.

  Example 165 –––

  ```
  Attach MyRepeat.attach()
  ```

  where MyRepeat is of type Repeat and the return value is of type Attach.

  ––

- For every *TypeAssignment* of type `CHOICE` there is a general method

  Choices::choice TypeReference.choice()

  that returns the allowed SlotName type for current object. It is then possible to use the general method

  SlotType TypeReference.SlotName()

  that returns an object of type SlotType.

  Example 166 –––

  ```
  switch ( MyEvent.choice() ) {
          case Choice::c_send:
                  do_something( Me.send() );
                  break;
          case Choice::c_receive:
                  do_something( Me.receive() );
                  break;
          case Choice::c_otherwise:
                  do_something( Me.otherwise() );
                  break;
          case Choice::c_timeout:
                  do_something( Me.timeout() );
                  break;
          case Choice::c_done:
                  do_something( Me.done() );
                  break;
          default:
                  do_something_default();
                  break;
  }
  ```

  ––

- For every node of type `SEQUENCE OF` there is a general method

  `SlotTypeList TypeReference.SlotName()`

  that returns an object of type *SlotTypeList* which is a vector of elements of type *SlotType*.

  Example 167 ––

  Assume that `MyEvent.choice()` in the previous example returned the value `Choice::c_done`. Then, TTCN Access method `MyEvent.done().tcompIdList` will then be a valid method and return an object of type `TCompIdList`.

  ––

- Every object of type *SlotTypeList* has a method

  `int Object.nr_of_items()`

  that returns the number of elements in the vector. The elements can be accessed with the method

  `SlotType Object[ index ]`

  where *Object* is an element of type *SlotTypeList* and index starts with 0 for the first element. It returns an element of type *SlotType*.

  Example 168 ––

  According to the above example the method `MyTCompIdList.nr_of_items()` returns an integer value of the number if items in the vector object `MyTCompIdList` and `MyTCompIdList[ 2 ]` will return the third element in the vector.

  ––

- For slots with ending `OPTIONAL` or `FIELD` there is a general method

  `Boolean TypeReference.is_present_SlotName()`

  used for verifying if an object is present or not. It is the responsibility of the TTCN Access programmer to ensure that all calls to optional slots are preceded with a call to the `is_present` method of that slot. It is a fatal error to attempt to access a non-present optional slot.

- For slots with ending `FIELD` and slots residing in a sub tree to a field there is a general method

  `Astring` *`Node`*`.content()`

  that returns an object of type `Astring` holding the content of the current field in a vector of characters.

  Example 169 –––

  The following C++ code will print out the content of a behavior line:

  ```
  BehaviourLine BLine = MyLine;

    cout << BLine.line( ).content() << endl;
  ```
  ––

Terminal Nodes in TTCN Access

Every TTCN Access node that represents a leaf in the parse tree is considered to be a *terminal* TTCN Access node. A terminal TTCN Access node is a node that does not have any TTCN Access child nodes and can therefore not be accessed any further.

Terminal TTCN Access children are nodes containing identifiers, strings, keywords as well as numbers (i.e. TTCN Access nodes of type `Identifier`, `IA5String`, `INTEGER`, `NUMBER`, etc.). However, the content of such terminal nodes can be accessed through the methods supplied by the inherited class `Astring`.

TTCN Access Class Astring

Every terminal TTCN Access node carries information in string format. In order to access that information, every terminal TTCN Access node inherits a class named `Astring`. This class contains various methods for treating string information.

For terminal TTCN Access nodes, the following operations are available:

- Initialization operations

Example 170 ––

The following C++ code will create and initiate a variable of type `Astring`:

```
Astring tmp1;               // tmp1 is empty
Astring tmp2( "test1" );    // tmp2 contains "test1"
Astring tmp3 = "test2";     // tmp3 contains "test2"
Astring tmp4 = tmp2;        // tmp4 contains "test1"
```

––

- Relational operations

Example 171 ––

The following relational operations are available between Astring elements:

```
==, !=
```

––

- `Astring` objects are type cast equivalent with `const char*`

Example 172 ––

The following code is valid C++ code:

```
Astring myString( "test" );

cout << myString;
printf( "%s", ( const char* ) myString );
```

––

- Address the nth character in the string, starting with 0.

Example 173 ––

The following is valid C++ code:

```
char c = myString[ 2 ]; // save the third character
of the string
```

––

Note:

For further details see class `Astring` in TTCN Access include file `ITEXAccessClasses.hh`.

Direct Access

AccessSuite

The `AccessSuite` object is the TTCN Access representation of a TTCN test suite. The test suite is in turn contained in a TTCN suite data base. The *AccessSuite* services include opening and closing the TTCN suite data bases as well as services to start an TTCN Access application and accessing the symbol table manager.

Example 174 ––

Opening and closing the test suite *Test.itex* for use in TTCN Access:

```
AccessSuite suite;

Boolean ok_open = suite.open( "Test.itex" );
if( ok_open )
{
   // do something
   Boolean ok_close = suite.close();
}
```

––

After opening a test suite for TTCN Access use, it is possible to use the following methods to get a handle to the contents of the data base file:

- Get a handle to the root node of the document.

  Example 175 ––

  Getting a handle to the root object of a document:

```
AccessSuite suite;
Boolean ok_open = suite.open( "Test.itex" );
if( ok_open ) {
    const AccessNode node = suite.root();
    // do something with NSAPaddr
    Boolean ok_close = suite.close();
}
```

  ––

Note:

For further details see class `AccessSuite` in TTCN Access include file `access.hh`.

- Ask the `AccessSuite` object for a specific TTCN Access object in the test suite. The object asked for must be a global object in the test suite. This is performed by using TTCN Access class `AccessNode`.

  Example 176 ––

  Find the TTCN Access object `NSAP` in test suite *Test.itex*:

  ```
  AccessSuite suite;
  Boolean ok_open = suite.open( "Test.itex" );
  if( ok_open ) {
      const AccessNode node = suite.find("NSAPaddr");
      // do something with NSAPaddr
      Boolean ok_close = suite.close();
  }
  ```

  ––

The `AccessNode` object now holds the TTCN Access item corresponding to the name NSAPaddr, if any.

To gain access to the data in an AccessNode, you must now find out the runtime type of the object. Based on that type, you will be able to use the conversion routine for an object of the corresponding type:

Example 177 ––

Find the type of the TTCN Access node corresponding to a name:

```
extern void HandleSimpleType( const SimpleType * );

AccessSuite suite;
Boolean ok_open = suite.open( "Test.itex" );
if( ok_open ) {
   AccessNode node = suite.find( "SomeName" );
   switch( node.choice() ) {
     case Choice::_SimpleType:
       HandleSimpleType(node.SimpleType());
       break;
     default:
       break;
   }}
```

––

Note:

For further details see class `AccessNode` in TTCN Access include file `access.hh`

The AccessVisitor Class

TTCN Access is a large class library, with over 600 classes, and therefore we also need suitable tools for simplifying the creation of TTCN Access applications. The preferred solution is the usage of the class AccessVisitor, which definition is available in the C++ header file `ITEXAccessVisitor.hh`.

The AccessVisitor class is a very close relative to the design pattern ‘Visitor’ (described by, for instance, Gamma, Helm, Johnson and Vlissides in ‘Design Patterns - Elements of reusable software’, Addison-Wesley1994). The difference is that the TTCN Access classes contain runtime type information which eliminates the need for them to have dependencies to the Visitor class (and therefore there is no need for ‘Accept’ methods in the visited classes).

An object of a class which is derived from the class AccessVisitor is later on referred to as a ‘visitor’.

AccessVisitor Class Members

Common Classes

The AccessVisitor class has two methods for visiting objects of common classes AccessSuite and AccessNode. These are declared

```
public:
void Visit( const AccessNode );
void Visit( const AccessSuite & );
```

and calling them with, will start a chain of calls in the visitor which effectively is a pre-order traversal of the subtree of the AccessNode, or the complete syntactical tree of the AccessSuite.

TTCN/ASN.1 Derived Classes

The AccessVisitor class has one virtual member function for each TTCN and ASN.1 derived class in TTCN Access. Each of the member functions are declared

```
public:
virtual void Visit<class>( const <class> & );
```

Example 178 ––

Visitor member method for an Identifier (excerpt from ITEXAccess-Visitor.hh):

```
class AccessVisitor
{
public:
    ...
  virtual void VisitIdentifier(const Identifier&);
  virtual void VisitVerdict(const Verdict&);
    ...
};
```

–––

The base class implementation of this method calls the related Vis-it<child-class> function for all of the child objects to the current object.

If you need to change the behavior, for instance in order to generate some code or report from the TTCN Access Suite, just derive a new class from the AccessVisitor class and override the relevant method(s). Call the base class implementation of the method to traverse the children if needed.

Data Members

The AccessVisitor class contains no explicitly declared data members and is therefore stateless. It therefore makes program re-entrance possible, and several visitors may be active in the same tree/document at the same time if needed.

Using the AccessVisitor

The intended usage of the AccessVisitor class is by derivation. Derive one or several specialized classes for each of the purposes which you use TTCN Access. Override the relevant methods.

Use Case – Information Collection

It is simple to create a visitor class for collection of some kind of information from the test suite. The basic method for this is to have a handle or actual information in the visitor class.

Example 179 ---

An information collecting visitor class:

```
#include <time.h>
#include <iostream.h>
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

//
// A visitor which counts the number of testcases
// in a test suite.

class TestCounter : public AccessVisitor
{
public:
  TestCounter() : _count( 0 ) { }
  ~TestCounter() { cout << _count << endl; }
  void VisitTestCase(const TestCase&) { _count++; }
private:
  unsigned int _count;
};

// Small example usage, no real fault control...

int main( int argc, char ** argv )
{
  AccessSuite suite;
  if ( suite.open( argv[1] ) ) {
    TestCounter testCounter;
    testCounter.Visit( suite );
  }
  return 0;
}
```

Use Case – Code Generation

It is likewise simple to create a class for simple code generation. The following example is a class for generation of a c-style declaration which contains a list of all SimpleType names in a test suite.

Example 180 ––

An information collecting visitor class:

```
#include <stdio.h>
#include <time.h>
#include <ITEXAccessClasses.hh>
#include <ITEXAccessVisitor.hh>

// A visitor which generate a c-style declaration
// of a null-terminated array of all Simple Type
// Declarations (identifiers) in a test suite.

class ListGenerator : public AccessVisitor
{
public:
  ListGenerator(FILE * f) : _file( f ) {
          printf( "const char *ids[]={\n");
        }
  void VisitSimpleTypeId(const SimpleTypeId& id) {
          printf( "  \"%s\",\n", (const char*)
                   id.simpleTypeIdentifier());
        }
  ~ListGenerator() {
          printf( "  NULL\n};\n");
        }
};

// Small example usage, no real fault control...

int main( int argc, char ** argv )
{
  AccessSuite suite;
  if ( suite.open( argv[1] ) ) {
    ListGenerator generator;
    generator.Visit( suite );
  }
  return 0;
}
```

––

The last two examples are not the most efficient implementations due to the fact that they traverse the whole tree even though we are only interested in small parts, and therefore we may want to improve the efficiency. This may be done using any of the techniques described in “Optimizing Visitors” on page 972.

Advanced Use Case – Combined Visitors

More advanced designs may be possible by combining several visitors for performing more advanced tasks. You may for instance have visitors which are parameterized with other visitors for specialized tasks, where you still would want to maintain decent performance and yet not have trade-offs in clarity of the design.

Example 181 ––

A TTCN Interpreter would need a mechanism for building an internal representation of values. Values are always built by using the same structure, but you may wish to have several possible representations of atomic values. A visitor could be used to generate the value objects, where one visitor is used for building the overall structure, and another is used for building individual fields. There are at least two choices available in the design:

- To inherit the structure building class into the class which builds the atomic values
- To parameterize the structure building class with the atomic value handing class (which gives you a possibility to tune the behavior at runtime).

In this example, the value building class inherits from the AccessVisitor class and the Visit<xxx> functions are used to generate a new value. The class GciValueBuilder in the example, may visit any structured type and will on the Visit<class> function either create itself a data value, or if it is a structured type, create a dynamic array, and then invoke a new visitor for each of the fields, which results will later be assigned to each of the fields in the same array. The implementation is not present in the example. It is just outlined below.

The solution of using visitors for building the values, removes the switch statements, which otherwise would undoubtedly clobber an implementation which uses Direct Access as described in a previous section.

```
class GciValueBuilder : public AccessVisitor
{
  // basic structure for building/matching values
  virtual void VisitXxxx( const Xxxx & );
  ...
  // built value
  GciValue * _value;
};

class GciValueBinaryBuilder : public GciValueBuilder
{
  // suitable overrides for efficient binary value
  // handling
  ...
};

class GciValueBigNumBuilder : public GciValueBuilder
{
  // suitable ovverides for a ‘bignum’
  // implementation
  ...
};
```

Optimizing Visitors

A visitor is a potentially inefficient way to find and process objects, since it in its unmodified version traverses the whole document, without regarding what parts of the document the inherited visitor is interested in. All optimizations are in the domain of limiting the subtree for which we are traversing. The following examples shows methods for avoiding unnecessary traversal and optimally, constant time access.

Example 182

Optimizing a visitor class to avoid traversal of all parts but the declarations part, may improve performance for suite traversal by over 100 times for some fairly representable suites (since most suites contain more and larger syntactical trees in the dynamic part than in all other parts, possibly with the exception of ASN.1 constraint declarations).

```
class DeclarationChecker : public AccessVisitor
{
 public:
  void VisitSuiteOverviewPart(
                const SuiteOverViewPart& ) { }
  void VisitConstraintsPart(
                const ConstraintsPart & ) { }
  void VisitDeclarationsPart (
                const DeclarationsPart & ) { }

  // Add the functions which actually do processing
  // below ...
};
```

Optimizing a visitor class to traverse only parts we are interested in, is also possible, by using one or a few levels of Direct Access instead of the default traversal.

Example 183 ---

This example skips all traversal down to the Simple Type Definitions table, and only traverses those.

```
class SimpleTypeIdPrinter : public AccessVisitor
{
 public:
  void VisitASuite( const ASuite& );
  void VisitSimpleTypeId ( const SimpleTypeId & );
};

void
SimpleTypeIdPrinter::VisitASuite(const ASuite & s)
{
  VisitSimpleTypeDefs( s.declarationsPart().
                        definitions().
                        ts_TypeDefs().
                        simpleTypeDefs());
}

void
SimpleTypeIdPrinter::VisitSimpleTypeId( const
                                SimpleTypeId & id )
{
  cout  << "Id: "
        << id.simpleTypeIdentifier()
        << endl;
}
```

Finally, you may combine several visitors into one visitor, thus avoiding multiple passes when processing a suite. This implies that you define several classes which are not truly visitors, and define a inherited class from AccessVisitor which calls methods in these classes.

Example 184 –––

Two objects driven by a visitor, thus performing two passes on one traversal.

```
class IdOperation
{
 public:
  virtual void AtIdentifier(const Identifier&) = 0;
};

// A IdOperation

class IdCounter : public IdOperation
{
 public:
  IdCounter( ) : _count( 0 ) { }
  void AtIdentifier( const Identifier & id )
     { _count++; }
  ~IdCounter( ) { cout << _count << endl; }
 private:
  unsigned int _count;
};

// Another IdOperation

class IdPrinter : public IdOperation
{
  public:
   void AtIdentifier( const Identifier & id )
        { cout << id << endl; }
};

// A class which drives up to IdOpDriverMax
// IdOperations

const int IdOpDriverMax = 10;

class IdOpDriver : public AccessVisitor
{
 public:
  IdOpDriver() : _ops_used(0) { }
  void VisitIdentifier( const Identifier & id ) {
    for (unsigned int op = 0 ; op < _ops_used; ++op)
      _ops[op].AtIdentifier( id );
  }
  void AddIdOp( IdOperation * op )
        { _ops[_ops_used++] = op; }
 private:
  IdOperation  _ops[IdOpDriverMax];
  unsigned int _ops_used;
};

// Main routine which opens a suite and applies two
// IdOperations.

int main( int argc, char ** argv )
{
  AccessSuite suite;
  if ( suite.open( argv[1] ) ) {
    IdCounter  counter;
    IdPrinter  printer;
    IdOpDriver driver;
    driver.AddIdOp( &counter );
    driver.AddIdOp( &printer );
    driver.Visit( suite );
    suite.close( );
  }
  return 0;
}
```

Common Class Definitions

This part contains the declaration of the three common TTCN Access classes `AccessSuite`, `AccessNode` and `Astring`. For further information see TTCN Access include file `access.hh`.

AccessSuite

```
class AccessSuite
{
public:
  AccessSuite();
  ~AccessSuite();
  AccessSuite( const AccessSuite& orig );
  void operator=( const AccessSuite& orig );

  Boolean open( const char* suite_name );
  Boolean open( Suite* suite );
  Boolean close();

  const AccessNode root();
  const AccessNode find( const Identifier & id );
  const AccessNode find( const char* id );
 };
```

AccessNode

```
class AccessNode
{
public:
  AccessNode();
  AccessNode(NodeInfo nodeinfo);
  ~AccessNode();
  AccessNode(const AccessNode& Me);

  int operator==(const AccessNode& o) const;
  void operator=(const AccessNode& orig);

  Boolean is_equal(const AccessNode& o) const;
  Choices::Choice choice() const;
  Boolean ok() const;

};
```

Astring

```
class Astring
{
public:
  Astring();
  Astring(const char* s);
  // end should point to the char after the last
char
  Astring(const char* begin, const char* end);
  Astring(Field* field, PT* pt);
  Astring(const Astring& orig);

  ~Astring();

  //operators
  Astring* operator->();
  const Astring* operator->() const;
  operator const char*() const;
  char& operator[](unsigned i) ;
  char operator[](unsigned i) const ;

  void operator=(const Astring&);
  void operator=(const String&);
  void operator=(const char*);
  void operator=(const char);

  int operator==(const Astring& s) const;
  int operator!=(const Astring& s) const;
  int operator==(const char* cs) const;
  int operator!=(const char* cs) const;
};
```

Getting Started with TTCN Access

Setting Up the TTCN Access Environment

When building applications using TTCN Access, the compiler will need to find a few files, the `ITEXAccessClasses.hh` include file and the `libaccess.a` library file. Normally these files are found in the `.../itex/include/CC` and `.../itex/lib/CC` directory respectively. The include file must be included in every TTCN Access application and the library `libaccess.a` must be used when linking.

Note:

For further information, contact your system administrator.

TTCN Access operates on the TTCN suite data bases. These data bases must have passed analysis and be saved before using TTCN Access. If the data base contains a TTCN test suite that is not analyzed, the TTCN Access application can not reach into the fields of the tables. The default behavior of TTCN Access is to simply skip those fields that are not analyzed. De-referencing a particular field in an non-analyzed data base, will result in undefined behavior.

TTCN test suite data bases are managed by the AccessSuite object, which has member functions for opening and closing the TTCN suite data bases and also for starting traversing. From an AccessSuite object it is also possible to access tables in a random manner via the symbol table manager.

Using Example Applications

TTCN Access is delivered with some simple example applications. To compile the examples, the installation directory, where the `ITEXAccessClasses.hh` and `libaccess.a` files reside, must be known to the makefiles. Do this by setting the environment variable ACCESS or by explicitly filling in the local variable ACCESS in every makefile or using make with the syntax `make ACCESS=$telelogic/itex/access`.

Starting an TTCN Access Application

You have to retrieve the TTCN Access license when you start the TTCN suite. To do this, start the TTCN suite from the command line with the switch `-access`.

Then you can execute TTCN Access applications from the command line or from the window manager.

You can also select *Start Application* in the *Access* menu in the Browser. This will open a dialog in which you may change settings and start the TTCN Access application.

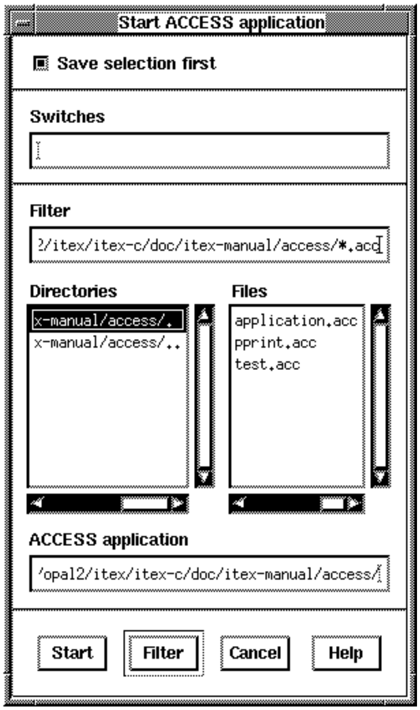


Figure 185: The Start Application dialog

Save selection first

Select this to save the current selection in the Browser before executing the TTCN Access application.

Switches

Switches may be passed to the chosen TTCN Access application. The switches are written as free text. As the last argument is the name of the the TTCN suite database passed. This name is always passed to the TTCN Access application.

Note:

Observe that it is the name of the (working) *database* file and not the name of the test suite that is passed to the TTCN Access application.

Filter

Sets the filter for the files that will be displayed in the *Suites* list. There are no predefined naming conventions for TTCN Access applications.

For example the name filter `*.acc` will cause only those files whose names end with `.acc` to be displayed.

Access application

Displays the selected application.