TTCN-3 - A new Test Specification Language for Black-Box Testing of Distributed Systems

Jens Grabowski

University of Lübeck, Institute for Telematics,
Ratzeburger Allee 160, D-23538 Lübeck, Germany,
Email: jens@itm.mu-luebeck.de

Abstract

The Tree and Tabular Combined Notation (TTCN) is a well-established notation for the specification of test cases for OSI protocol conformance testing. The third edition of TTCN (TTCN-3) is currently under development by the European Telecommunications Standards Institute. TTCN-3 will be a complete redesign of the entire test specification language. The close relation between tabular and textual representation will be removed, OSI specific language constructs will be cleared away and new concepts will be introduced. The intention of this redesign is to modernize TTCN and to widen its application area beyond pure OSI conformance testing. This paper will motivate the need for TTCN-3 and introduce the principles of TTCN-3.

1 Introduction

The *Tree and Tabular Combined Notation* (TTCN) is defined and standardized in Part 3 of the international standard 9646 *OSI Conformance Testing Methodology and Framework* (CTMF) [7]. OSI conformance testing is understood as functional black-box testing, i.e., an *implementation under test* (IUT) is given as a black-box and its functional behavior is defined in terms of inputs to and corresponding outputs from the IUT. Subsequently, TTCN test cases describe sequences of stimuli to and expected responses from the IUT.

CTMF and TTCN have been used for testing OSI protocols and protocols in systems following the OSI layering scheme, e.g., ISDN or ATM. CTMF principles and TTCN have also been applied successfully to other types of functional black-box testing, e.g., ISDN service testing and interoperability testing.

The requirements on testing are changing. New software architectures, e.g., ODP, CORBA, TINA or DCE, with advanced and time-critical applications like multimedia, home-banking, or video-conferencing require new testing concepts, new testing architectures and a new and powerful test specification language.

Researchers have already started to extend CTMF and TTCN in order to meet the upcoming testing requirements. The proposed extensions were related to testing architectures [12,16], real-time testing [14,15] and performance testing [13]. Some of these ideas will find their way into practice.

The international standardization organizations *International Telecommunication Union* (ITU-T) and *European Telecommunications Standards Institute* (ETSI) have studied the new testing requirements [1] and recognized the urgent need for a modern and powerful test specification language. As a consequence, an experts team was set up by ETSI in October 1998. This team is developing the third edition of TTCN (TTCN-3) [2] which is a complete redesign of the previous TTCN editions [5]. The development of TTCN-3 is an ongoing task and is expected to be completed in October 2000.

TTCN-3 will be looking like a common programming language with test specific extensions. Highlights of these extensions comprise the handling of test verdicts, matching mechanisms to compare the reactions of the IUT with the expected range of reactions, timer handling, distribution of tester processes and the ability to specify encoding information. For the test of communicating systems, TTCN-3 supports synchronous and asynchronous communication as well as monitoring. For its use in the telecommunications area, TTCN-3 can be used in combination with ASN.1 [8].

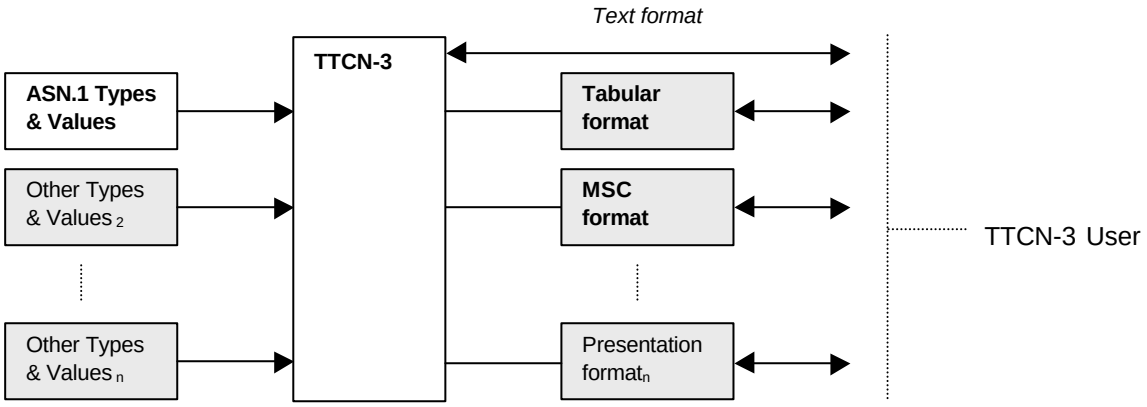

Figure 1: User's view of TTCN-3 and the various presentation formats

2 Adapting TTCN-3 to different application areas

TTCN-3 is meant to be a core testing language which can be adapted to different application areas. The adaptation may be done by providing an application specific presentation format or by defining application specific attributes for TTCN-3 language elements.

2.1 TTCN-3 presentation formats

TTCN-3 may be used as a generalized text-based language in its own right, as a standardized interchange format for tools, or as the semantical basis for various presentation formats.

As shown in Figure 1, the different presentation formats of TTCN-3 shall provide an application oriented view of the TTCN-3 description to the human user. Depending on the application area, either TTCN-3 or a specific presentation format may be used to specify and visualize test cases.

Together with the TTCN-3 standard two special presentation formats will be standardized. A tabular format supporting the conformance testing view of the previous TTCN versions [3] and an MSC/UML [10,11] based presentation format [4,6] providing an MSC/SDL [9] and MSC/UML oriented testing view.

2.2 Application specific attributes in TTCN-3

In most cases, testing an application requires application specific information for testing. In TTCN-3 it is possible to add application specific information by defining attributes which can be associated with TTCN-3 language elements.

Figure 2 presents the definition of a **record** type MyPDU in a TTCN-3 test suite for protocol conformance testing. Attributes are associated to MyPDU by means of the **with** statement. The **display** attribute identifies MyPDU as a Protocol Data Unit (PDU) and the **encode** attribute specifies the encoding of instances of MyPDU according to the ASN.1 Basic Encoding Rules (BER). The attribute values PDU and BER are application specific and are rarely used outside the area of protocol conformance testing. In addition, TTCN-3 allows to specify an **extension** attribute which can be used freely to associate further information to TTCN-3 language elements.

```
type record MyPDU {
   integer       field1;
   ia5string     field2;
   boolean       field3
}
with {
   display       "PDU";
   encode        "BER"
}
```

Figure 2: Defining attributes

```
module MyTestSuite {                // This module contains a definitions part...
    :
   const integer MyConstant := 1;
   type record MyMessageType { … }
   template MyMessage { … }
    :
   function MyFunction1( … ) { … }
   function MyFunction2 { … }
    :
   testcase MyTestcase1 { … }
   testcase MyTestcase2 { … }
   testcase MyTestcase3 { … }
    :
   control {                        // ... and a control part
       :
      var boolean MyVariable := true; // local variable
       :
      MyTestCase1; // sequential execution of MyTestCase1 and MyTestCase2
      MyTestCase2;
      if (MyVariable) MyTestCase3; // conditional execution of MyTestCase3
       :
   }  // End control
}  // End module
```

Figure 3: Structure of a TTCN-3 module

3 TTCN-3 modules

The top-level unit of TTCN-3 is the module. A module cannot be structured into sub-modules, but a module may import definitions from other modules. A module may be parameterized to ease its adaptation to different test environments. As indicated in Figure 3, a TTCN-3 module consists of a definitions part and an (optional) control part.

3.1 Module definitions part

The module definitions part specifies the top-level definitions of a TTCN-3 module like test components, communication ports, data types, constants, test data templates, functions, signatures for remote procedure calls, signature templates, named alternatives or test cases.

The top-level definitions specified in the module definitions part may be used elsewhere in the module, including the module control part. A module definitions part can be structured by means of named groups of definitions. Groups may be nested. TTCN-3 groups are no scope units, but they can be used to refer to all definitions in a group either for associating application specific attributes or for being imported by another TTCN-3 module.

It is possible to re-use top-level definitions specified in other TTCN modules by using **import** statements. TTCN-3 has no explicit export construct and thus, by default, all definitions in the module definitions part may be imported by other modules. As shown in Figure 4, it is possible to import single definitions, all definitions of a module, groups of definitions and definitions of the same kind.

```
import type MyType from MyModuleA;   // imports a single definition
import all from MyModuleB;           // imports all definitions of a module
import group MyGroup from MyModuleC; // imports a group
import all type from MyModuleD;      // imports all type definitions
```

Figure 4: Examples for the use of the **import** construct

3.2 Module control part

The (optional) control part of a TTCN-3 module can be compared to the *main* function of a C or C++ program. A TTCN-3 module control part executes the test cases specified (or imported) in the module definitions part. A TTCN-3 module without control part can be considered to be a test library.

In the control part, (local) variables, constants or timers may be declared and program statements such as **if-else** or **do-while** may be used to specify the selection and (possibly repetitious) execution order of individual test cases. For example, the TTCN-3 module control part in Figure 3 executes the test cases *MyTestCase1* and *MyTestCase2* in sequential order. The test case *MyTestCase3* is only executed if *MyVariable* is **true**. All program statements which can be used in a TTCN-3 module control part can be found in Figure 5.

Variables defined in the module control part are local, i.e., they cannot be accessed by functions or test cases called inside the control part. TTCN-3 does not support global variables. If required, variable values can be passed into test cases and functions as parameters.

Statement	Associated keyword or symbol	Usable in module control part	Usable in functions and test cases
Basic program statements			
Expressions	`(…)`	Yes	Yes
Assignments	`:=`	Yes	Yes
If-else statement	`if (…) {…} else {…}`	Yes	Yes
For loop	`for (…) {…}`	Yes	Yes
While loop	`while (…) {…}`	Yes	Yes
Do while loop	`do {…} while (…)`	Yes	Yes
Label definition	`label`	Yes	Yes
Jump to a label	`goto`	Yes	Yes
Behaviour statements and operations			
Alternative behaviour	`alt {…}`		Yes
Interleaved behaviour	`interleave {…}`		Yes
Activate a default	`activate`		Yes
Deactivate a default	`deactivate`		Yes
Returning control	`return`		Yes
Configuration operations			
Create parallel test component	`create`		Yes
Connect component to component	`connect`		Yes
Map component to test interface	`map`		Yes
Get MTC address	`mtc`		Yes
Get test system interface address	`system`		Yes
Get own address	`self`		Yes
Start execution of test component	`start`		Yes
Stop execution of test component	`stop`		Yes
Check termination of a PTC	`done`		Yes
Communication operations			
Send message	`send`		Yes
Invoke procedure call	`call`		Yes
Reply to procedure call from remote entity	`reply`		Yes
Raise exception (to an accepted call)	`raise`		Yes
Receive message	`receive`		Yes
Trigger on message	`trigger`		Yes
Accept procedure call from remote entity	`getcall`		Yes
Receive reply for a previous procedure call	`getreply`		Yes
Catch exception (from called entity)	`catch`		Yes
Check (current) message/call received	`check`		Yes
Clear port	`clear`		Yes
Clear and give access to port	`start`		Yes
Stop access (receiving & sending) at port	`stop`		Yes
Timer operations			
Start timer	`start`		Yes
Stop timer	`stop`		Yes
Read elapsed time	`read`		Yes
Timeout event	`timeout`		Yes
Verdict operations			
Set local verdict	**verdict.set**		Yes
Get local verdict	**verdict.get**		Yes

Figure 5: Overview of TTCN-3 statements and operations

Class of type	Keywords	Sub-type
Basic types	**integer, float**	range, list
	boolean, objectidentifier, verdicttype, duration	
Basic string types	**bitstring, hexstring, octetstring, numericstring**	list, length
Basic characterstring types	**printablestring, teletexstring, t61string, videotexstring, visiblestring, iso10646string, ia5string, graphicstring, generalstring, bmpstring, universalstring, utf8string**	list, length
User-defined structured types	**record, record of, set, set of, enumerated, union**	

Figure 6: Overview of TTCN-3 data types

4 Data types, messages and message templates

TTCN-3 includes a number of predefined data types. They are shown in Figure 6 and can be used to define messages and message templates.

4.1 TTCN-3 data types

Most of the data types shown in Figure 6 are well known from other programming languages and need no further explanation. The telecommunications history of TTCN-3 is reflected by the types **objectidentifier**, the various basic characterstring types and the structured types **record of** and **set of**. These types are required for compatibility to ASN.1 and to previous TTCN versions.

The **verdicttype** is special to TTCN. It is an enumerated type with the values **none**, **pass**, **inconclusive** , **fail** and **error**. During the execution of a test case (Section 7), each test component (Section 6) keeps track of an implicitly defined **verdict** object. The **verdict** object can be accessed by using the predefined operations **set** and **get**. The rules for setting the value of the **verdict** object by a test component are very simple: A **none** can be overwritten by **pass**, **fail** and **inconclusive**. A **pass** can be overwritten by **inconclusive** and **fail**. An **inconclusive** can be overwritten by **fail**. A **fail** cannot be overwritten.

After termination of a test case, based on the local verdicts of the test components a test case verdict is calculated. This calculation follows the rules above. The value **error** is reserved for the test system and is assigned to a test case if during its execution a dynamic test case error occurs.

4.2 Messages

TTCN-3 supports asynchronous communication by means of message exchange, but there is no explicit *message* data type. In TTCN-3 any value of any type can be used as a message and, if allowed by the test configuration (Section 6), be sent either to the implementation under test or other test components. In most cases, messages will be defined as record types.

4.3 Templates and matching mechanisms

In TTCN-3 the definition of test values can be done by using templates. A **template** is a placeholder for either a single test value or a whole set of test values. Such templates can be used in communication operations to specify a value to be sent or to check whether a received message has the expected value. An example for a **template** definition and its usage is presented in Figure 7.

```
// Given the message definition in Figure 2... a corresponding message template might be ...
template MyPDU MyTemplate {
   field1     *;             // use of 'instead of value' matching mechanism
   field2     "abc*xyz";     // use of 'inside value' matching mechanism
   field3     true           // use of 'specific value' matching mechanism
}

// ... and a corresponding receive operation could be
MyPCO.receive(MyTemplate);
```

Figure 7: Definition and usage of templates

To ease the specification of templates, TTCN-3 provides several *matching mechanisms*. These matching mechanisms can be arranged in four groups:

1. specific values (i.e., an expression that evaluates to a specific value);
2. special symbols that can be used *instead* of values:
   - **(…)**: a list of values;
   - **complement (…)**: complement of a list of values;
   - **omit**: an (optional) value is omitted;
   - **?**: wildcard for any value;
   - *****: wildcard for any value or no value at all (i.e., an omitted value);
   - **(**<lower> **to** <upper>**)**: a range of integer values including the lower- and upper bounds;
3. special symbols that can be used *inside* values:
   - **?**: wildcard for any single element in a string or array;
   - *****: wildcard for any number of consecutive elements in a string or array, or no element at all;
   - **permutation**: a permutation of elements in an array;
4. special symbols which describe *attributes* of values:
   - **length**: restrictions for strings and arrays;
   - **if present**: for matching of optional field values (if not omitted).

5 Procedure signatures

Procedure signatures (or signatures for short) are needed for synchronous communication. A procedure may either be invoked in the IUT, i.e., the test system performs the call, or invoked in the test system, i.e., the IUT performs the call. For both the complete procedure signature has to be defined in the TTCN-3 module.

Within a **signature** definition the parameter list may include parameter identifiers, parameter types and their direction, i.e., **in**, **out**, or **inout**. It should be noted that in a signature definition the direction of the parameters is as seen by the *called* party rather than the *calling* party.

A remote procedure call will result in the called party performing either a **reply** operation (the normal case) or raising an exception (Section 8). Exceptions are represented as values of a specific type, even templates and matching mechanisms can be used. A list of all possible exceptions is included in the signature definition as shown in Figure 8.

```
// Defines the signature of MyRemoteProc which, if called, either returns an
// integer value or may raise exceptions of type ExceptionType1 or ExceptionType2.
signature MyRemoteProc (in integer Par1, out float Par2, inout integer Par3)
   return integer
   exception (ExceptionType1, ExceptionType2);
```

Figure 8: Definition of a **signature**

6 Test configurations

TTCN-3 allows the (dynamic) specification of concurrent test configurations. A test configuration consists of a set of interconnected *test components* with well-defined *communication ports* and an explicit *test system interface* which defines the borders of the test system.

Within every test configuration, there is one *Main Test Component* (MTC). All other test components are *called Parallel Test Components* (PTCs). The MTC is created automatically at the start of each test case execution and the behavior defined in the body of the test case (Section 7) is executed on this component. During execution of a test case PTCs can be created and stopped dynamically by the explicit use of **create** and **stop** operations. The conceptual view of a typical TTCN-3 testing configuration is shown in Figure 9.

6.1 Defining communication port types

Ports facilitate communication between test components and between test components and the test system interface. There are no restrictions on the number of connections a component may have, but a component shall not be connected

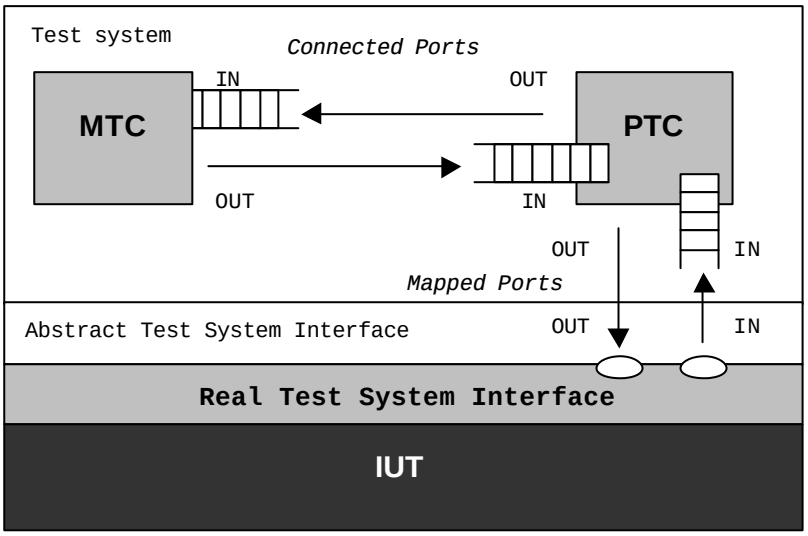

Figure 9: Conceptual view of a typical TTCN-3 testing configuration

to itself. One-to-many connections are allowed, but TTCN-3 only supports one-to-one communication only, i.e., during test execution the communication partner has to be determined uniquely. Each port is modeled as an infinite FIFO queue which stores the incoming messages or procedure calls until they are processed by the component owning that port.

TTCN-3 ports are either message-based or procedure-based. Message-based ports are used for asynchronous communication by means of message exchange. Procedure-based ports are used for synchronous communication by means of remote procedure calls. Ports are directional and each port may have an **in** list (for the *in* direction), an **out** list (for the *out* direction) or an **inout** list (for both directions) of allowed messages or procedures. Figure 10 presents an examples of a port type definition.

```
// Message-based port which allows MsgType1 and MsgType2 to be received,
// MsgType3 to be sent and any integer value to be send and received.
type port MyMessagePortType message {
   in       MsgType1, MsgType2;
   out      MsgType3;
   inout    integer
}
```

Figure 10: TTCN-3 port type definition

6.2 Defining component types and the test system interface

A test case is composed of a set of one or more test components. The test case behavior is executed on these components. The **component** type defines which ports are associated with a component. The port names in a component definition are used in the component behavior definition to address the different ports. Port names are local to a component, i.e., another component may have a port with the same (local) name. Figure 11 shows an example for a component type definition.

A **component** type definition is also used to define the test system interface, because conceptually component type definitions and test system interface definitions have the same form, i.e., both are collections of ports defining possible connection points.

```
// Component type with three ports
type component MyPTCType {
   MyProcedurePortType    PCO1;
   MyMessagePortType      PCO2;
   MyAllMesssagesPortType PCO3
}
```

Figure 11: TTCN-3 component type definition

```
// It is assumed that the ports and components types used are properly defined
 :
var component MyNewComponent := MyComponentType.create; // usage of create
 :
connect(MyNewComponent.Port1, mtc.Port3);// usage of connect and mtc
 :
map(self.Port2, system.PCO1);            // usage of map, self and system
:
MyNewComponent.start(MyCompBehaviour(…));// usage of start operation
 :
if (MyNewComponent.done) {               // usage of done
    :    // Do something
}
 :
if (date = 1.1.2000) stop;               // usage of stop
```

Figure 12 Usage of configuration operations

6.3 Configuration operations

Configuration operations are concerned with setting up and controlling test components. During the execution of a test case, the actual test configuration of components and the connections among them and between the components and the test system interface are created dynamically by performing configuration operations. Configuration operations are **create**, **connect**, **map**, **start**, **stop**, **mtc**, **system**, **self** and **done**.

The create operation

The MTC is the only test component which is created automatically when a test case starts. All other test components are created explicitly during test execution by **create** operations. Since all components and ports are destroyed at the end of a test case, each test case must completely create its required configuration of components and connections.

As shown in Figure 12, the **create** operation returns a unique reference to the newly created instance. The reference can be used for connecting instances and for communication purposes, i.e., for addressing individual components.

Components can be created at any time during a test run providing full flexibility with regard to dynamic configurations, i.e., any component can create any other component. Component references are local to the scope of their creation. In order to reference a component outside its scope of creation, the component reference can be passed as a parameter to a function or remote procedure or can be sent in a message.

The mtc, system and self operations

The operations **mtc** and **system** return the references (or addresses) of the MTC and the system interface. The **self** operation allows a test component to retrieve its own reference, i.e., **self** returns the reference of the component in which **self** is called. The operations **mtc**, **system** and **self** can be used for addressing purposes in communication operations or, as shown in Figure 12, in configuration operations.

The connect and map operations

The ports of a test component can be connected to ports of other components or to the ports of the test system interface. The connection between two test components is done by means of the **connect** operation. When linking a test component to a test system interface, the **map** operation shall be used. As illustrated in Figure 9, the **connect** operation directly connects one port to another with the *in* side of the one port connected to the out side of the other, and vice versa. The **map** operation on the other hand can be seen as a pure name translation defining how communications streams should be referenced. In Figure 12 examples for the usage of **connect** and **map** operations are shown.

The start operation

Once a component has been created and connected the execution of its behavior has to be started. This is done by using the **start** operation. The reason for the distinction between **create** and **start** is to allow connection operations to be done before actually running the test component. The **start** operation binds the behavior to a component by referring to a function (Section 7). An example for the usage of the start operation can be found in Figure 12.

```
testcase MyTestCase(inout integer MyPar)
runs on MyMtcType1           // defines the type of the MTC
system  MyTestSystemType     // defines the test system interface
{
 :       // The behaviour defined here executes on the mtc when the test case invoked
}
```

Figure 13: Example for a test case definition

The stop and done operations

By using the **stop** operation, a test component is able to stop itself. A stopped component disappears from the configuration. The **done** operation allows a test component to ascertain whether another test component has completed, i.e., is stopped. Examples for the usage of **done** and **stop** operations can be found in Figure 12.

7 Test cases and functions

Behavior in TTCN-3 is related to the definition of test cases, functions and named alternatives. Named alternatives are a special form of macros and will be explained in Section 9.2.

7.1 Test cases

The test cases are the probes which have to be executed in order to judge whether an implementation under test passes the test or not. Test cases are defined in the module definitions part and called in the module control part. Each test case returns a test verdict of either **none**, **pass**, **fail**, **inconclusive** or **error** (Section 4). This means a single test case can be considered to be a special kind of function returning a test verdict.

An example of a test case definition is shown in Figure 13. The test case is called *MyTestCase* and has the **inout** parameter *MyPar* of type **integer**. The **runs on** clause following the parameter defines the type of the MTC. The **system** clause specifies the type of the test system interface. The definition body defines the behavior of the MTC and will be started automatically when the test case is called. The MTC type is required to make the port names of the MTC visible inside the behavior definition. The type of the system interface is mandatory, if during the test run several test components are created and stopped dynamically. If the MTC performs the whole test on its own, the type of the test system interface is identical to the MTC type and can be omitted.

7.2 Functions

In TTCN-3, functions are used to express test behavior or to structure computation in a module, for example, to calculate a single value or to initialize a set of variables. A function may be parameterized and may return a value. As shown in the function definition of *MyFunction* in Figure 14, the return value is defined by the **return** keyword followed by a type identifier. If no **return** is specified then the function result is void. An explicit keyword for void does not exist in TTCN-3.

If a function defines test behavior, the type of the test component on which the behavior is executed has to be specified by means of a **runs on** clause. This type reference makes the port names of the component type visible inside the behavior definition of the function. This is shown in the definition of function *MyBehaviour* in Figure 14.

```
// Definition of MyFunction which has no parameters
function MyFunction return integer {
   return 7 // returns the integer value 7 when the function terminates
}

// Definition of MyBehaviour which specifies test behaviour
function MyBehaviour (inout integer MyPar)
runs on MyPTCType
{   :                                  // MyFunction3 doesn't return a value, but does make
   var integer MyVar := 5 * MyPar;     // use of the port operation send and therefore
   PCO1.send(MyVar);                   // requires a runs on clause to resolve the port
    :                                  // identifiers by referencing a component type
}
```

Figure 14: Examples for definitions of functions

8 Communication operations

TTCN-3 supports message-based (asynchronous) and procedure-based (synchronous) communication. As illustrated in Figure 15 asynchronous communication is non-blocking on the **send** operation, where processing in the MTC continues immediately after the **send** operation. The IUT is blocked on the **receive** operation until it receives the send message.

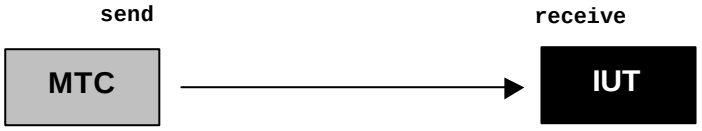

Figure 15: Illustration of the asynchronous **send** and **receive** operations

Synchronous communication in TTCN-3 is related to remote procedure calls. As sketched in Figure 16, the synchronous communication mechanism is blocking on the **call** operation, where the **call** operation blocks processing in the MTC until either a reply or an exception is received from the IUT. Similar to the asynchronous **receive** operation, the **getcall** blocks the IUT until the call is received.

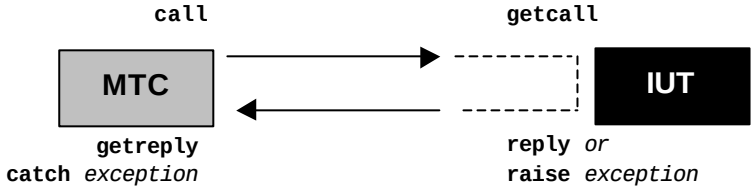

Figure 16: Illustration of a complete synchronous call

8.1 Asynchronous communication

For asynchronous communication, TTCN-3 provides the **send** and **receive** operations. The **send** operation is used to place a value on an outgoing message-based port. The value may be specified by referencing a template, a variable or a constant, or can be defined in-line in form of an expression (which of course can be an explicit value). When defining the value in-line, the optional type field can be used to avoid any ambiguity of the type of the value being sent.

The **receive** operation is used to receive a value from an incoming message port queue. If the top message in the port satisfies all matching criteria associated with the **receive** operation, it is removed from the queue. The matching criteria may be related to the value of the message or the sender of the message. If the match is not successful, the top message is not removed, i.e., an alternative **receive** operation is required to remove the message from the port queue. Examples for the usage of **send** and **receive** operations can be found in Figure 17.

```
MyCL.send(integer 5);  // Sends the integer value 5 via port MyCL.

MyCL.receive(MyTemplate(integer 5, MyVar));
// Specifies the reception of a value which fulfils the conditions defined
// by the template MyTemplate with actual parameters 5 and MyVar.

MyCL.receive(A<B);
// Specifies the reception of a boolean value depending on the value of A<B

MyCL.receive(MyType *) from MyPartner -> MyVar;
// Specifies the reception of an arbitrary value of MyType (from a component
// with an address stored in variable MyPartner) which afterwards is
// assigned to the variable MyVar. MyVar has to be of type MyType.
```

Figure 17: Usage of **send** and **receive** operations

8.2 Synchronous communication

As is shown in Figure 16, for synchronous communication, the calling side and the called side have to be distinguished. In order to test both, TTCN-3 provides communication operations for both sides.

The communication operations for the calling side are the **call** operation to call a remote procedure, the **getreply** operation to handle replies (or answers) to calls and the **catch** operation to handle exceptions which in case of exceptional situations may be received instead of a reply. In addition, TTCN-3 provides special **timeout** exception to cope with situations where the called party neither replies nor raises an exception. The usage of **call**, **getreply** and **catch** is shown in Figure 18.

For the called side, TTCN-3 provides the **getcall** operation to accept calls from remote, the **reply** operation to reply to calls and the **raise** operation to raise exceptions. The usage of these operations is shown in Figure 19.

```
// The following call operation calls the remote procedure MyProc with the in or inout parameters
// 5 and MyVar owned by a component with an address stored in variable MyPartner via the
// communication port MyCL. A special timeout exception with the duration of 30 ms is specified
// and defines the waiting time for either a reply or an exception to the call.
MyCL.call(MyProc(5,MyVar), 30ms) to MyPartner {

   [] MyCl.getreply(MyProc(MyVar1, MyVar2)) -> MyResult param (MyPar1Var,MyPar2Var);
      // handles a reply to the call, where the out and inout parameters have the same value
      // as MyVar1 and MyVar2. The return value of MyProc is assigned to variable MyResult and
      // the out and inout parameters are assigned to the variables MyPar1Var and MyPar2Var.

   [] MyCL.catch(MyProc, MyExceptionOne) // catches an exception and as a result of
         stop;                           // the exception stops the component

   [] MyCL.catch(MyProc, MyExceptionTwo); // catches a second exception

   [] MyCL.catch(timeout) {;  // handling of the timeout exception i.e., the called party does
         verdict.set(fail);   // not react in time or not in an appropriate manner. The local
         stop;                // verdict is set to fail and the component stops execution
      }                       //
}
```

Figure 18: Usage of **call**, **getreply** and **catch** operations

```
MyCL.getcall(MyProc(5, MyVar)) -> sender MySenderVar;
// Will accept a call of MyProc at MyCL with the inout parameters 5 and MyVar. The
// calling party is retrieved by the accept operation and stored in MySenderVar.

MyCL.reply(MyProc(20,MyVar2) value 20) to MySenderVar;
// Replies to the accepted call above. The return value is 20 and the
// values of the two inout parameters are 20 and the value of MyVar2.

MyCL.raise(MyProc, MyVariable + YourVariable - 2) to MySenderVar;
// Raises an exception for an accepted call with a value which is
// the result of the arithmetic expression.
```

Figure 19: Usage of **getcall**, **reply** and **raise** operations

8.3 The check operation

The **check** operation is a generic operation that permits to read the top element of message-based and procedure-based incoming port. The **check** operation has to handle values at message based ports and to distinguish between calls to be accepted, exceptions to be caught and responses from previous calls at procedure-based ports. This is done by using the operations **receive**, **getcall**, **getreply** and **catch** together with their matching and assignment parts to define the condition which has to be checked and to extract the value or values of its parameters if required. Examples for the usage of the **check** operation can be found in Figure 20.

```
MyAsyncPort.check(receive(integer 5));
// Will check for an integer value of 5 as top message in the asynchronous port MyAsyncPort.

MyAsyncPort.check(receive(integer *) -> MyVar);
// Will check for an arbitrary integer value at part MyAsyncPort. If an integer value is
// received, its value will be assigned to MyVar, but not removed from the queue.

MyCL.check(getcall(MyProc(5, MyVar)) from MyPartner);
// Will check for a call of MyProc at MyCL (with the in or inout parameters 5 and MyVar) from
// a peer component which has the address stored in variable MyPartner.
```

Figure 20: Usage of the **check** operation

8.4 Controlling communication ports

TTCN-3 provides the **clear**, **start** and **stop** operations to control communication ports. The **clear** operation removes the contents of an incoming port queue. The **start** operation starts listening at and gives access to a port. The **stop** operation stops listening and disallows **send**, **call**, **reply** and **raise** operations at the port.

9 Special behavior statements in TTCN-3

A complete overview of TTCN-3 statements and operations is presented in Figure 5. The configuration operations, the communication operations, and the verdict operations have already been explained in the previous sections. The basic program statements, the timer operations and some of the behavior statements are well known from programming languages and need no special explanation. Only the handling of alternatives, the interleaved behavior and the handling of defaults are special to TTCN-3 and need some explanation.

9.1 Alternative behaviour

The alternative behavior statement (or **alt** statement for short) describes branching of control flow due to the reception of communication and timer events, i.e., the **alt** statement is related to the use of the TTCN-3 operations **receive**, **trigger**[1], **getcall**, **getreply**, **catch**, **check** and **timeout**.

An example of an **alt** statement is shown in Figure 21. The different branches of the **alt** statement start with square brackets which may include nothing, a boolean expression or the keywords **expand** or **else**. The brackets can be seen as a sort of boolean guard for the following receiving event. Empty brackets denote the value **true**.An **alt** statement is evaluated from top to bottom. A branch is selected when the boolean guard evaluates to **true** and the following **receive**, **trigger, getcall**, **getreply**, **catch**, **check** or **timeout** operation can be executed. A selected branch is executed in the expected manner.

The keyword **expand** denotes a macro expansion and get a description in the next section. The keyword **else** is an unconditional exit of an **alt** statement. The else branch does not have to start with a receiving operation and is always taken if none of the previous branches can be selected.

```
alt {
   []       L1.receive(MyMessage1); {
                : // Do something
            }
   [x>1]    L2.receive(MyMessage2);    // boolean guard/expression
   [x<=1]   L2.receive(MyMessage3);    // boolean guard/expression
   [expand] MyNamedAlt;                // macro expansion
   [else]   stop                       // else branch
}
```

Figure 21: Example of an **alt** statement

9.2 Named alternatives

An **alt** statement which is used in several places can be defined in a named alternative denoted by the keyword pair **named alt**. A **named alt** is a macro definition and causes a textual replacement when it is referenced. It can be referenced at any place in a behavior definition where it is valid to include a normal **alt** construct. Furthermore, it can be used to add alternative branches in an **alt** statement as shown in Figure 21. The definition of a **named alt** statement is shown in Figure 22.

```
named alt MyNamedAlt {
   [] PCO2.receive(DL_EST_IN);
   [] PCO2.receive(DL_EST_CO);
}
```

Figure 22: Definition of a named alternative

[1] The details of the **trigger** operation are not described in this paper, but it is a special form of the **receive** operation.

9.3 Default handling

In TTCN-3 defaults are used to handle communication events which may occur, but which do not contribute to the test objective. For example, when testing the call forwarding feature of an ISDN system, charging information may be received at any time. This information is not relevant for the testing objective and thus, can be ignored in the test evaluation. During the test, execution messages or calls containing such information may be received and have to be handled. This can be done by means of defaults.

The default concept of TTCN-3 is related to the macro expansion concept of named alternatives, i.e., an activated default expands automatically all **alt** statements following the **activate** statement. A default behavior can be defined in-line in an **activate** statement or the **activate** statement can refer to an already defined named alternative. It is also possible to deactivate defaults by using the **deactivate** statement.

9.4 Interleaved behavior

In case of de-coupled ports, messages, calls, replies and exceptions may be received in an arbitrary order. The **interleave** statement allows to express such arbitrary order. Without the **interleave** statement all valid orders have to be specified explicitly. The **interleave** statement can be seen as shorthand a notation for a number of nested **alt** statements. This is shown in the example in Figure 23.

```
// The following interleave statement
interleave {
   [] PCO1.receive(MyMessageOne);
   [] PCO1.receive(MyMessageTwo);
   [] PCO1.receive(MyMessageThree);
}

// ... can be interpreted as a shorthand for
alt {
   [] PCO1.receive(MyMessageOne); {
         alt {
            [] PCO1.receive(MyMessageTwo);
                  PCO1.receive(MyMessageThree);
            [] PCO1.receive(MyMessageThree);
                  PCO1.receive(MyMessageTwo);
         }
      }
   [] PCO1.receive(MyMessageTwo); {
         alt {
            [] PCO1.receive(MyMessageOne);
                  PCO1.receive(MyMessageThree);
            [] PCO1.receive(MyMessageThree);
                  PCO1.receive(MyMessageOne);
         }
      }
   [] PCO1.receive(MyMessageThree); {
         alt {
            [] PCO1.receive(MyMessageTwo);
                  PCO1.receive(MyMessageOne);
            [] PCO1.receive(MyMessageOne);
                  PCO1.receive(MyMessageTwo);
         }
      }
}
```

Figure 23 Meaning of the **interleave** statement

10 Summary and outlook

In this paper a simple and general testing language called TTCN-3 has been presented. The language is currently in the standardisation process at ETSI and ITU-T with the plan to be published in the year 2000 as an European Norm (EN) by ETSI and in 2001 as ITU-T Recommendation Z.140. Next steps will then be the publication of the tabular presentation format [3] and the MSC/UML presentation format [4].

This paper cannot contain a full language description but it is intended to give the reader a flavour of the new language. For details, the language description in [2] is recommended.

Several tool makers have already shown interest in implementing TTCN-3. Many of the tools are embedded in an environment together with SDL [9] and MSC[10]. Therefore it is necessary to consider the interworking between an SDL specification and a TTCN-3 test suite. This will also enable mechanisms for automated test case generation.

There is already research on the way for including real-time [14,15] and performance [13] aspects into TTCN. With this new version, it seems to be feasible to base the performance and real-time extensions on TTCN-3. New research projects are just in the process of being started in this direction.

Acknowledgements

The work presented in this paper has been carried out by the Specialists Task Force (STF) 133/156 as part of the funded work program of ETSI. The author is a member of STF 133/156 and would like to thank Anthony Wiles (STF leader) and Colin Willcock for allowing to present parts of the STF work.

There are many individuals who also contributed to TTCN-3 in several ways. Listing names at this point brings the risk that someone will be forgotten. I therefore constrain myself, with two exceptions, to listing the main contributing organisations: Danet, Ericsson, Expert Telecoms, France Telecom, Fraunhofer Gesellschaft (FhG), GMD Fokus, Motorola, NMG Telecoms, Nokia, Nortel, Tektronix, Telelogic, University of Lübeck (Institute for Telematics). However, out of all the all the individuals who contributed, the engagement of Os Monkewich from Nortel and Dieter Hogrefe from the University of Lübeck should be highlighted.

References

[1] ETSI TC MTS. *Guide for the use of the second edition of TTCN (Revised Version)*.European Guide 202 103, 1998.

[2] ETSI TC MTS. *TTCN-3 - Core Language*. European Norm (EN) 00063-1 (provisional)[2] , 2000.

[3] ETSI TC MTS. *TTCN-3 - Tabular Presentation Format*. EN00063-2 (provisional)[2], 2000.

[4] ETSI TC MTS. *TTCN-3 - MSC Presentation Format*. EN00063-3 (provisional)[2], 2000.

[5] J. Grabowski, D. Hogrefe: *An Introduction to TTCN-3*. In: G. Csopaki, S. Dibuz, K. Tarnay, editors, *Testing of Communicating Systems – Methods and Applications*, Kluwer Academic Publishers, September 1999.

[6] J. Grabowski, T. Walter. *Visualisation of TTCN test cases by MSCs*. In: Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC - SAM'98 (editors: Y. Lahav, A. Wolisz, J. Fischer, E. Holz), Informatik-Berichte Humboldt-Universität zu Berlin, June 1998.

[7] International Standardization Organization. *Information Technology - OSI - Conformance Testing Methodology and Framework - Parts 1-7*. ISO, International Standard 9646, 1994 - 1997.

[8] ITU-T Recommendations X.680-683. *Information Technology - Abstract Syntax Notation One (ASN.1)*. Geneva 1994.

[9] ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*. Geneva 2000.

[10] ITU-T Recommendation Z.120. *Message Sequence Chart (MSC)*. Geneva, 2000.

[11] E. Rudolph, J. Grabowski, P. Graubmann. *Towards a Harmonization of UML-Sequence Diagrams and MSC*. In: R. Dssouli, G. v. Bochmann, Y. Lahav, editors, SDL'99 - The next Millenium, Elsevier, June 1999.

[12] I. Schieferdecker, M. Li, A. Hoffmann. *Conformance Testing of TINA Service Components - the TTCN/CORBA Gateway*. In: Proceedings of the 5th International Conference on Intelligence in Services and Networks, Antwerp, Belgium, May 1998.

[13] I. Schieferdecker, S. Stepien, A. Rennoch. *PerfTTCN, a TTCN Language Extension for Performance Testing*. In: M. Kim, S. Kang, K. Hong, editors, *Testing of Communicating Systems*, volume 10, Chapman & Hall, September 1997.

[14] T. Walter, J. Grabowski. *Real-time TTCN for Testing Real-time and Multimedia Systems*. In: M. Kim, S. Kang, K. Hong, editors, *Testing of Communicating Systems*, volume 10, Chapman & Hall, September 1997.

[15] T. Walter, J. Grabowski. *A Framework for the Specification of Test Cases for Real-Time Systems*. In: *Journal of Information and Software Technology*, Special Issue on Communications Software Engineering, vol. 41, Elsevier, July 1999.

[16] T. Walter, I. Schieferdecker, J. Grabowski. *Test Architectures for Distributed Systems - State of the Art and Beyond*. In: A. Petrenko, N. Yevtushenko, editors, *Testing of Communicating Systems*, volume 11, Chapman & Hall, September 1998.

[2] NOTE: The EN-00063 numbers are only provisional ETSI work item numbers. The actual numbers will not be the same.