1

ON THE DESIGN OF THE NEW TESTING LANGUAGE TTCN-3

Jens Grabowski[a], Anthony Wiles[b], Colin Willcock[c], Dieter Hogrefe[a]

[a]*Institute for Telematics, University of Lübeck, Ratzeburger Allee 160, D-23538 Lübeck, Germany, email: {jens,hogrefe}@itm.mu-luebeck.de*

[b]*European Telecommunication Standards Institute, 650, Route des Lucioles, F-06921 Sophia Antipolis, France, e-mail: Anthony.Wiles@etsi.fr*

[c]*Nokia Research Center, Bochum,Meesmannstr. 103, D-44807 Bochum, Germany, e-mail: Colin.Willcock@nokia.com*

Abstract This paper gives an overview of the main concepts and features of the new testing language TTCN version 3 (TTCN-3). TTCN-3 is a complete new testing language built from a textual core notation on which a number of different presentation formats are possible. This makes TTCN-3 quite universal and application independent. One of the standardised presentation formats is based on the tree and tabular format from previous TTCN versions and another standardised presentation format is based on MSCs. TTCN-3 is a modular language and has a similar look and feel to a typical programming language. However, in addition to the typical programming constructs it contains all the important features necessary to specify test suites.

Keywords: Conformance Testing, ETSI, ITU-T, Programming Languages, TTCN, Test Specification, Telecommunication Systems

1. INTRODUCTION

TTCN-3 is designed in such a way that a broader user community is addressed. The syntax looks similar to typical implementation languages and

should therefore be easy to understand and apply for someone familiar with software development.

The core language is freed from the peculiarities specific to OSI and conformance testing. This makes TTCN-3 flexible, applicable to the specification of all types of reactive system tests over a variety of communication interfaces. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing and testing of CORBA based platforms.

The TTCN-3 standard is separated into three parts. The first part [5] defines the core language. The second part [6] defines the tabular presentation format which is similar in appearance and functionality to TTCN-2 [8]. The third part [7] describes an MSC [9] based presentation format.

The core language serves three purposes. Firstly the core language can be used as a generalised text based test language in its own right. Secondly it is used as a standardised interchange format of TTCN test suites between TTCN tools and thirdly the core language provides the semantic basis for the all the various presentation formats.

The core language is defined by a complete syntax and operational semantics. It contains minimal static semantics which do not restrict the use of the language due to some underlying application domain or methodology.

The next section of this paper provides an overview of presentation formats and then the rest of the paper concentrates on the design of the TTCN-3 core language.

2. PRESENTATION FORMATS AND ATTRIBUTES

Presentation formats provide alternate ways to specify and visualise TTCN-3 test suites, as shown in Figure 1. The tabular format and the MSC format are the first in an anticipated set of different presentation formats. Further presentation formats may be standardised formats or be proprietary formats defined by TTCN-3 users themselves. Use and implementation of all presentation formats is based on the core language.

A presentation format is defined by specifying the graphical representation required for the test suite and the mapping necessary to convert between this graphical form and the TTCN-3 core language.

To enable this mapping, language elements within the core langauge may have attributes associated with them. There are three kinds of attributes:

1. **encode**: allows references to specific encoding rules;

2. **`display`** : allows the specification of display attributes related to specific presentation formats;
3. **`extension`**: allows the specification of user-defined attributes.

Attributes are associated with TTCN-3 language elements by means of **`with`** statements. In the core language the syntax for the argument of the **`with`** statement (i.e. the actual attributes) is simply defined as a free text string. Special attribute strings related to the display attributes for the tabular (conformance) presentation format can be found in [6]. Special attribute strings related to the display attributes for the MSC presentation format can be found in [7].

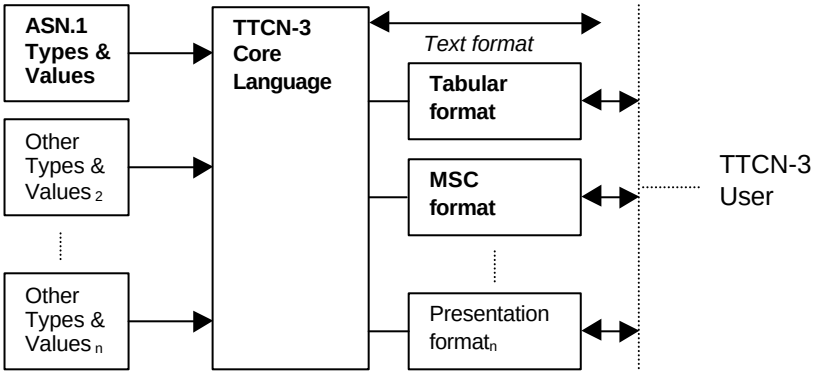

Figure 1. User's view of the core language and the various presentation formats

3. TYPES AND VALUES

TTCN-3 supports a number of predefined basic, structured and special types. Basic types include ones normally associated with a programming language, such as **integer**, **boolean** and string types, as well as TTCN-3 specific ones such as **objectidentifier**, **verdicttype** and **duration**. Structured types such as **record** types, **`set`** types and **enumerated** types can be constructed from these base types. Special types associated with configurations such as **port** and **component** may be used to define the architecture of the test system as described in Section 5.

TTCN-3 is strongly typed and there are a number of predefined functions to handle type conversion.

A special kind of data value called a template provides parameterisation and matching mechanisms for specifying test data to be sent or received over

the test ports. The operations on these ports provide both asynchronous and synchronous communication capabilities (Section 7).

TTCN-3 is fully harmonised with ASN.1 [11,12,13,14] which may optionally be used with TTCN-3 modules as an alternative data type and value syntax. The approach used to combine ASN.1 and TTCN-3 could be applied to support the use of other type and value systems with TTCN-3.

4. MODULES

The principle building-blocks of the TTCN-3 core language are modules. A module is a self-contained and complete specification, i.e. it can be parsed and compiled as a separate entity. A module consists of an (optional) definitions part, and an (optional) module control part.

The module definitions part specifies the top-level definitions of the module. These definitions may be used elsewhere in the module, including the control part.

The module control part describes the execution order (possibly repetitious) of the actual test cases. A test case shall be defined in the module definitions part and then called in the control part. For example:

```
module MyTestSuite {
   // Definitions part
   testcase MyTestcase1()…
   testcase MyTestcase2()…
    :
   // Control part
   control {
      var boolean MyVariable; //Localcontrol variable
       :
      MyTestCase1(); // sequential execution of test cases
      MyTestCase2();
       :
   }
}
```

TTCN-3 does not support the declaration of global variables, therefore declarations of dynamic language elements such as **var** or **timer** is only allowed locally within functions and test cases.

It is possible to re-use definitions specified in other modules using the **import** statement. TTCN-3 has no explicit export construct thus, by default, all module definitions in the module definitions part may be imported. An **import** statement can only be used in the module definitions part.

5. TEST CONFIGURATIONS

TTCN-3 allows the specification of (dynamic) 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 6) 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 2.

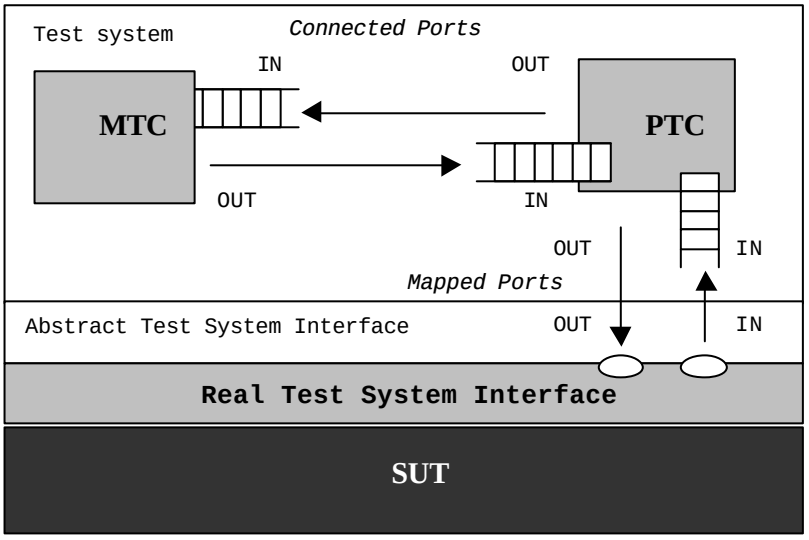

Figure 2. Conceptual view of a typical TTCN-3 testing configuration

5.1 Communication ports

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 to itself. One-to-many connections are allowed, but TTCN-3 only supports one-to-one communication, i.e. during test execution the

communication partner has to be specified 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. For example:

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

5.2 Component types and the test system interface

A test case consists 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. For example:

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

The port names in a component type 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.

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.

5.3 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, the connections among them and the connections 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 3, 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.

```
// usage of create
var component MyNewComponent := MyComponentType.create;
:
// usage of connect and mtc
connect(MyNewComponent.Port1, mtc.Port3);
:
// usage of map, self and system
map(self.Port2, system.PCO1);
:
// usage of start operation
MyNewComponent.start(MyCompBehaviour(…));
:
// usage of done
if (MyNewComponent.done) {
    :   // Do something
}
 :
// usage of stop
if (date = 1.1.2000) stop;
```

Figure 3. Usage of configuration operations

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 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 3 , 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 2, 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 3 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 6). An example for the usage of the start operation can be found in Figure 3.

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 3.

6. 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 8.

6.1 Test cases

The test cases define the behaviour which has 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**. 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 4. 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.

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

Figure 4. Example for a test case definition

6.2 Functions

In TTCN-3, functions are used to express test behaviour 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 5, 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 5.

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

function MyBehaviour (inout integer MyPar)
runs on MyPTCType
{   :                              // MyFunction3 make use of
   var integer MyVar := 5 * MyPar; // the port operation send
   PCO1.send(MyVar);               // and therefore requires a
    :                              // runs on clause to resolve
    :                              // the port identifiers
}
```

Figure 5. Examples for function definitions

7. COMMUNICATION OPERATIONS

TTCN-3 supports message-based (asynchronous) and procedure-based (synchronous) communication. As illustrated in Figure 6 asynchronous communication is non-blocking on the **send** operation, where processing in the MTC continues immediately after the **send** operation. The SUT is blocked on the **receive** operation until it receives the send message.

Synchronous communication in TTCN-3 is related to remote procedure calls. As illustrated in Figure 7, 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 SUT. Similar to the asynchronous **receive** operation, the **getcall** blocks the SUT until the call is received.

7.1 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.

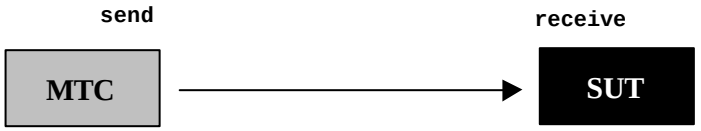

Figure 6. Illustration of the asynchronous **send** and **receive** operations

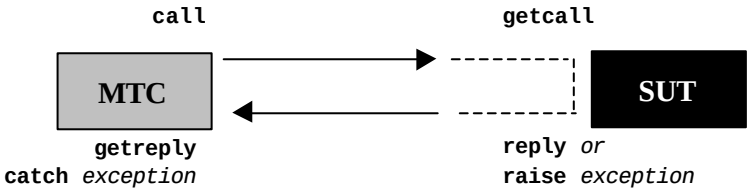

Figure 7. Illustration of a complete synchronous call

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. For example:

```
MyCL.send(integer:5);
// Sends integer value 5 is sent via port MyCL

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

MyCL.receive(MyType:*) from MyPartner -> value MyVar;
// Receives an arbitrary value of MyType from a MyPartner.
// The received value is assigned to MyVar.
```

7.2 Synchronous communication

As is shown in Figure 7, 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. For example:

```
// Calls remote procedure MyProc via MyCl. A timeout
//  exception will be raised after 30 ms
MyCL.call(MyProc(5,MyVar), 30ms) to MyPartner {
   [] MyCl.getreply(MyProc:{MyVar1, MyVar2}) ->
            value MyResult param(MyPar1, MyPar2);
      // Handles a reply to the call. The return value is
      // assigned to MyResult. The out/inout parameters
      // are assigned to MyPar1 and MyPar2.

   [] MyCL.catch(MyProc, MyExceptionOne) { // Exception
         stop                  // Stop of component
      }
   [] MyCL.catch(MyProc, MyExceptionTwo); // Second exception

   [] MyCL.catch(timeout) {   // Timeout exception
         verdict.set(fail);
         stop;
      }
}
```

For the called side, TTCN-3 provides the **getcall** operation to accept calls, the **reply** operation to reply to calls and the **raise** operation to raise exceptions. For example:

```
MyCL.getcall(MyProc:{5, MyVar}) -> sender MySenderVar;
// Accepts a call of MyProc. The calling party is retrieved
// and stored in MySenderVar.

MyCL.reply(MyProcTemp(20,MyVar2) value 20) to MySender;
// Replies to the accepted call above.

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

7.3 The check operation

The **check** operation is a generic operation that permits to read the top element of message-based or 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:

```
MyAsyncPort.check(receive(integer:5) from MyPartner);
// Check for an integer value of 5 from MyPartner in port
//MyAsyncPort.

MyAsyncPort.check(receive(integer:*) -> value MyVar);
// Checks for any integer value at port MyAsyncPort.
```

7.4 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.

8. SPECIAL BEHAVIOR STATEMENTS IN TTCN-3

The configuration operations, the communication operations, and the verdict operations have already been explained in the previous sections. The basic program statements such as **if-else**, **while**-loop, **for**-loop or **goto**, and the timer operations **set cancel** and **readtimer** are well known from other programming and specification languages and need no special explanation. Only the handling of alternatives and the handling of defaults are special to TTCN-3 and need some explanation.

8.1 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**, **getcall**, **getreply**, **catch**, **check** and **timeout**.

An example of an **alt** statement is shown in Figure 8. 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 is described 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

   [x<=1]   L2.receive(MyMessage3);    // boolean guard

   [expand]   MyNamedAlt;              // macro expansion

   [else]   stop                       // else branch
}
```

Figure 8. Example of an **alt** statement

8.2 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 8. For example:

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

8.3 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 named alternatives referenced in an **activate** statement. It is also possible to deactivate defaults by using the **deactivate** statement.

9. CONCLUSIONS AND OUTLOOK

We have presented here a simple and general core testing language called TTCN-3. 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 EN by ETSI (under the work program of Technical Committee MTS) and in the year 2001 as ITU-T standard Z.140. The next steps will then be the publication of the presentation formats. There are concrete plans for the tree and tabular presentation format and the MSC presentation format.

A number of tool makers have already shown interest in implementing the language. Some of the tools are embedded in an environment together with SDL [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 [2,3] and performance [4] aspects into TTCN. With this new version it seems feasible to base the performance and real-time extensions on the core language. New research projects are just in the process of being started in this direction

Acknowledgements

There are many individuals who contributed to TTCN-3 in several ways. Listing names at this point brings the risk, that someone will be forgotten. We therefore constrain ourselves, with one exception, to listing the main contributing organizations: 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, we would like to highlight the engagement of Os Monkewich from Nortel.

TTCN-3 is currently being developed under the work program of ETSI TC MTS (Methods for Testing and Specification). This proposed standard has not yet been published. For the official version of the TTCN-3 standard please contact the ETSI publications office at publications@etsi.fr.

10. REFERENCES

[1] Jens Grabowski, Dieter Hogrefe: An Introduction to TTCN-3. Invited Presentation of the '12th International Workshop on Testing Communicating Systems' (IWTCS'99), Budapest, September 1999.

[2] Thomas Walter, Jens Grabowski.: A Framework for the Specification of Test Cases for Real Time Distributed Systems. In: Information and Software Technology, vol. 41, Elsevier, July 1999.

[3] Thomas Walter, Jens Grabowski: Real-time TTCN for testing real-time and multimedia systems. In: Testing of Communicating Systems (Editors: M. Kim, S. Kang, K. Hong), volume 10, Chapman & Hall, 1997.

[4] Ina Schieferdecker, et al: PerfTTCN, a TTCN Language Extension for Performance Testing. . In: Testing of Communicating Systems (Editors: M. Kim, S. Kang, K. Hong), volume 10, Chapman & Hall, 1997.

[5] EN00063-1 (provisional),TTCN-3 Core Language

[6] EN00063-2 (provisional), TTCN-3 Tabular Presentation Format

[7] EN00063-3 (provisional), TTCN-3 MSC Presentation Format

[8] ISO/IEC 9646-3 (1998): Information technology - Open systems interconnection – Conformance testing methodology and framework - Part 3: The Tree and Tabular combined Notation (TTCN).

[9] ITU-T Recommendation Z.120 (2000): Message sequence Chart (MSC).

[10] ITU-T Recommendation Z.100 (2000): Specification and Description Language (SDL).

[11] ITU-T Recommendation X.680 (1997): | ISO/IEC 8824-1:1998,Abstract Syntax Notation One (ASN.1): Specification of basic notation.

[12] ITU-T Recommendation X.681 (1997) | ISO/IEC 8824-2:1998, Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.

[13] ITU-T Recommendation X.682 (1997) | ISO/IEC 8824-3:1998, Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.

[14] ITU-T Recommendation X.683 (1997) | ISO/IEC 8824-4:1998, Information technology – Abstract Syntax Notation One (ASN.1): Parameterisation of ASN.1 specifications.

NOTE: The EN-00063 numbers are only provisional ETSI Work Item numbers (the actual EN numbers will not be the same)