ELSEVIER

Available at www.ComputerScienceWeb.com POWERED BY SCIENCE @ DIRECT®

Computer Networks 42 (2003) 375–403

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

An introduction to the testing and test control notation (TTCN-3)

Jens Grabowski [a,*], Dieter Hogrefe [a], György Réthy [b], Ina Schieferdecker [c], Anthony Wiles [d], Colin Willcock [e]

[a] *Institute for Informatics, University of Göttingen, Lotzestrasse 16–18, 37083 Göttingen, Germany*
[b] *Ericsson Hungary, Laborc Street 1, 1037 Budapest, Hungary*
[c] *Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany*
[d] *European Telecommunications Standards Institute (ETSI), 650, route des Lucioles, 06921 Sophia-Antipolis Cedex, France*
[e] *Nokia Research Center Bochum, Meesmannstrasse 103, 44807 Bochum, Germany*

Abstract

The *testing and test control notation* (TTCN-3) is a new test specification and test implementation language that supports all kinds of black-box testing of distributed systems. TTCN-3 was developed in the years 1999 to 2002 at the *European Telecommunications Standards Institute* (ETSI), as a redesign of the *tree and tabular combined notation* (TTCN) standard (ITU-T Rec. X.292). TTCN-3 is built from a textual core language that provides interfaces to different data description languages and the possibility of different presentation formats. This makes TTCN-3 quite universal and application independent. TTCN-3 is being published as the ITU-T Rec. Z.140 series. This paper provides an introduction to TTCN-3. This includes an overall view of the TTCN-3 core language, a description of the existing presentation formats, an explanation of the implementation of TTCN-3-based test systems and a discussion about the current usage and the future of the language. The authors all participated in the work within ETSI.

Keywords: TTCN; TTCN-3; Test specification; Test implementation; Test languages; Black-box testing; Distributed systems testing; Standardization; ETSI

1. Introduction

The *testing and test control notation* (TTCN-3) [1–3] was designed in such a way that a broad user community is addressed. TTCN-3 has removed many of the peculiarities of the specific application domain of OSI and conformance testing that were present in previous versions of the *tree and tabular combined notation*, TTCN [4]. [1] The textual syntax looks similar to a typical programming language like C, C++ or Java and should therefore be easy

[*] Corresponding author.

[1] ITU-T Rec. X.292 is identical to part 3 of ISO/IEC IS 9646 [5]. The latest corrections of TTCN have been published as ETSI Technical Report 101 666 [6].

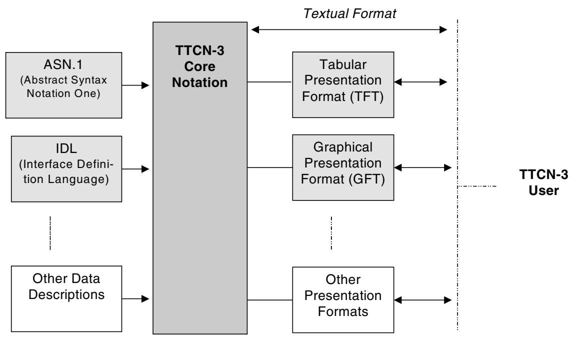

Textual Format
ASN.1
(Abstract Syntax Notation One)
IDL
(Interface Definition Language)
Other Data Descriptions
TTCN-3 Core Notation
Tabular Presentation Format (TFT)
Graphical Presentation Format (GFT)
Other Presentation Formats
TTCN-3 User

Fig. 1. Overall picture of TTCN-3.

to understand and apply for someone familiar with programming.

TTCN-3 is a flexible and powerful language, 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, testing of CORBA based platforms and the testing of APIs. As shown in Fig. 1, TTCN-3 is built from a textual core language that provides interfaces to different data description languages and the possibility of different presentation formats. From a syntactical point of view TTCN-3 is very different from the earlier TTCN. However,the well-proven basic functionality of TTCN has been retained, and in some cases enhanced.

This paper is organized in the following manner. In the first section, the key concepts associated with TTCN-3 are introduced together with their respective semantics. Building on these concepts the next section describes the TTCN-3 core language by using the example of a simplified automated teller machine (ATM). The next section of the paper considers the TTCN-3 presentation formats including the tabular presentation format TFT and the graphical presentation format GFT. In Section 5 the use of TTCN-3 with other languages is described and in Section 6, the execution interfaces of TTCN-3 are introduced, namely the TTCN-3 runtime interface (TRI) and the TTCN-3 control interface (TCI). Section 7 provides some examples of the application of TTCN-3 in industrial environments and standardization. The last section provides a summary and outlook.

2. Concepts of TTCN-3

TTCN-3 is based on concepts, which are independent of any syntax. These concepts are related to *modules*, *test cases*, *test systems*, *test verdicts*, *components* and *defaults*. This section presents these concepts. In some cases, the explanations use terms, which are keywords in the TTCN-3 core language. These terms are printed **bold-face**.

2.1. Modules and test cases

The principle building-blocks of TTCN-3 are modules. A module is a self-contained and complete specification: it can be parsed and compiled as a separate entity. A module consists of a *module definitions part* and a *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 module control part. The module control part is the main program of a TTCN-3 module. It describes the execution sequence (possibly repetitious) of the actual test cases. Test cases are defined in the module definitions part and then called in the control part.

The *test cases* define the behaviours, which have to be executed to check whether the *system under test* (SUT) passes the test or not. Like a module, a test case is considered to be a self-contained and complete specification that checks a test purpose. The result of a test case execution is a *test verdict* (Section 2.3).

2.2. Test system

A test case is executed by a *test system*. TTCN-3 allows the specification of dynamic and concurrent test systems. A test system consists of a set of interconnected *test components* with well-defined communication *ports* and an explicit *test system interface*, which defines the boundaries of the test system (Fig. 2).

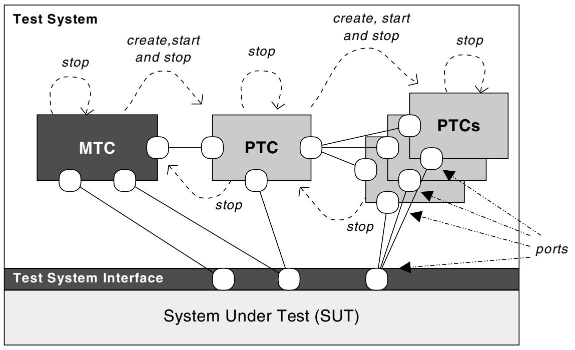


Fig. 2. Dynamic and concurrent test system.

Within every test system, there is one *main test component* (MTC). All other test components are called *parallel test components* (PTCs). The MTC is created and started automatically at the beginning of each test case execution. A test case terminates when the MTC terminates. The behaviour of the MTC is specified in the body of the test case definition. During the execution of a test case, PTCs can be created, started and stopped dynamically. A test component may stop itself or can be stopped by another test component.

For communication purposes, each test component owns a set of local *ports*. Each port has an *in*- and an *out-direction* (Fig. 3). The in-direction is modelled as an infinite FIFO queue, which stores the incoming information until it is processed by the test component owning the port. The out-direction is directly linked to the communication partner, i.e., outgoing information is not buffered.

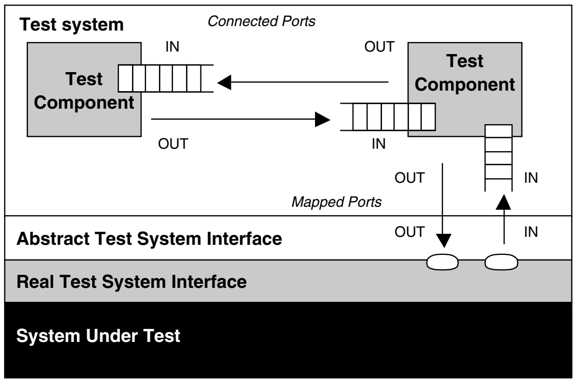


Fig. 3. Conceptual view of a TTCN-3 test system.

During test execution, TTCN-3 distinguishes between *connected* and *mapped ports*.

Connected ports are used for the communication with other test components. If two ports are connected, the in-direction of one port is linked to the out-direction of the other, and vice versa.

A mapped port is used for the communication with the SUT. The mapping of a port owned by a test component to a port in the *abstract test system interface* can be seen as pure name translation defining how communication streams should be referenced. As shown in Fig. 3, TTCN-3 distinguishes between the *abstract* and the *real test system interface*. The abstract test system interface is modelled as a collection of ports that defines the abstract interface to the SUT. The real test system interface is the application specific part of a TTCN-3-based test environment. It implements the real interface of the SUT. The rules for the implementation of TTCN-3 test system interfaces are defined in parts 5 and 6 of [2] (see also [7,8] and Section 6 below).

In TTCN-3, connections and mappings are created and destroyed dynamically at runtime. There are no restrictions on the number of connections and mappings a component may have. A component (and even a port) may be connected to itself. One-to-many connections are allowed, but TTCN-3 only supports one-to-one communication: during test execution the communication partner has to be specified uniquely.

For the communication among test components or between test components and the SUT, TTCN-3 supports *message-based* and *procedure-based* communication. Message-based communication is based on an asynchronous message exchange and the principle of procedure-based communication is to call procedures in remote entities.

2.3. Meaning and calculation of test verdicts

TTCN-3 provides a special test verdict mechanism for the interpretation of test runs. This mechanism is implemented by a set of *predefined verdicts*, *local* and *global test verdicts*, and operations for reading or setting local test verdicts.

The predefined verdicts are **pass**, **inconc**, **fail**, **error** and **none**. They can be used for the judgment of complete and partial test runs. A **pass** verdict denotes that the SUT behaves according to the test purpose, a **fail** indicates that the SUT violates its specification, an **inconc** (inconclusive) describes a situation where neither a **pass** nor a **fail** can be assigned and the verdict **error** indicates an error in the test devices. The verdict **none** is the initial value for *local* and *global test verdicts*, when no other verdict has been assigned yet.

During test execution, each test component maintains its own local test verdict. A local test verdict is an object that is instantiated automatically for each test component at the time of component creation. A test component can retrieve and set its local verdict. The verdict **error** is set automatically by the TTCN-3 run-time environment if an error in the test equipment occurs, and is not allowed to be set by a test component.

When changing the value of a local test verdict, special *overwriting rules* are applied. The overwriting rules only allow that a test verdict becomes worse, e.g., a **pass** may change to **inconc** or **fail**, but a **fail** cannot change to a **pass** or **inconc**.

In addition to the local test verdicts, the TTCN-3 run-time environment maintains a *global test verdict* for each test case. The global test verdict is not accessible by the test components. It is updated according to the overwriting rules when a test component terminates. The final global test verdict is returned to the module control part when the test case terminates.

2.4. Components and behaviour definitions

In TTCN-3, behaviour is related to a *control component* and to *test components*. The control component executes the control part of a module (Section 2.1) and the test components execute the test cases (Section 2.2). Behaviour definitions for control and test components are the *module control part*, *test cases*, *altsteps* and *functions*.

The main program of a TTCN-3 module is defined in the module control part. It is executed on a control component when the module is invoked. The behaviour of the module control part may call *functions* and *altsteps* specified in the module definitions part. TTCN-3 functions are similar to functions in typical programming languages, because they can be used to structure computation or to calculate a single value. Altsteps are functions with special semantics. Their meaning will be explained in Section 2.6. Functions and altsteps may be recursive and may call further functions and altsteps.

The call of a test case by the control component can be seen as the invocation of an independent program. This means, the control component only knows the signature of the test case, but cannot communicate with the test components. It receives the test result (the test verdict and parameter values) from the TTCN-3 runtime environment that manages the test case execution.

The call of a test case causes the creation of an MTC. Immediately after its creation, the MTC starts to execute the behaviour defined in the body of the test case. This behaviour may be structured into functions and altsteps.

During test execution, the MTC and all running PTCs may create and start further PTCs. TTCN-3 distinguishes between the creation and the start of a PTC. This allows to create a PTC and to connect or map the ports of the new component before its execution is started. The behaviour of the new component is defined by a function, which is referenced in the start operation.

2.5. Alternatives and snapshots

A specific feature of the TTCN-3 semantics are *snapshots*. Snapshots are related to the behaviour of components. They are needed for the branching of behaviour due to the occurrence of timeouts, the termination of test components and the reception of messages, procedure calls, procedure replies or exceptions. In TTCN-3, this branching is defined by means of **alt** statements.

An **alt** statement describes an ordered set of alternative branches of behaviour called *alternatives* (Fig. 4a). [2] Each alternative has a *guard*. A guard consists of several *preconditions*, which may refer to the values of variables, the status of timers,

[2] In the TTCN-3 core language, the order of the alternatives is identical to the order of their specification.

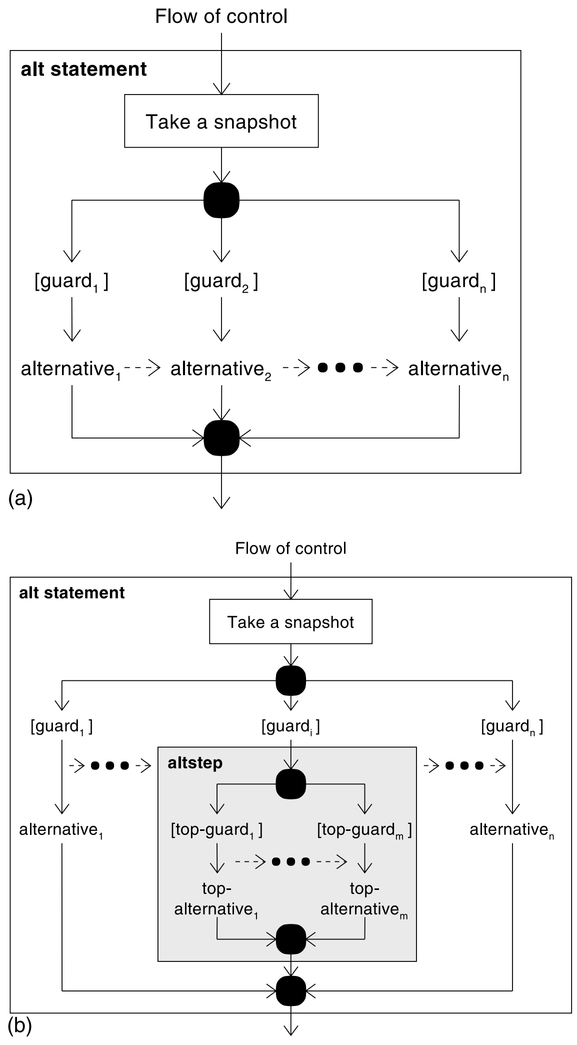


Fig. 4. Meaning of **alt**-statements and altsteps: (a) **alt**-statement, (b) Altstep within an **alt**-statement.

the contents of port queues and the identifiers of components, ports and timers. The same precondition can be used in different guards. An alternative becomes *executable*, if the corresponding guard is fulfilled. If several alternatives are executable, the first executable alternative in the list of alternatives will be executed. If no alternative becomes executable, the **alt** statement will be executed again.

The evaluation of several guards needs some time. During that time, preconditions may change dynamically. This will lead to inconsistent guard evaluations, if a precondition is verified several times in different guards. TTCN-3 avoids this problem by using snapshots. Snapshots are partial module states, which include all information necessary for the evaluation of **alt** statements. A snapshot is taken, i.e., recorded, when entering an alternative. For the verification of preconditions, only the information in the current snapshot is used. Thus, dynamic changes of preconditions do not influence the evaluation of guards.

2.6. The meaning of altsteps

In Section 2.4, altsteps have been introduced as function-like descriptions for structuring component behaviour. The precise semantics of altsteps is closely related to alternatives and snapshots. Like an **alt** statement, an altstep defines an ordered set of alternatives, the so-called *top alternatives.* [3] The difference is that no snapshot is taken when entering an altstep. The evaluation of the top alternatives is based on an existing snapshot.

An altstep is always called within an **alt** statement, which provides the required snapshot (Fig. 4b). Conceptually, the top alternatives of the altstep are inserted into the alternatives of the **alt** statement. Within the core language, the user can specify where the top alternatives shall be placed into the list of alternatives. It is also possible to call an altstep like a function. In this case, the altstep is interpreted like an **alt** statement which only invokes the altstep: the top alternatives are the only alternatives of the **alt** statement.

2.7. Default handling

In TTCN-3, defaults are used to handle communication events which may occur, but which do not contribute to the test objective. Default behaviour can be specified by altsteps (see Sections 2.4 and 2.6) and then *activated* as defaults. For each test component, the defaults (the activated altsteps) are stored as a list. The defaults are listed

[3] The term *top alternative* is used to distinguish alternatives of altsteps from alternatives of **alt** statements that may be embedded in top alternatives.

in the order of their activation. The TTCN-3 operations **activate** and **deactivate** operate on the list of defaults. An **activate** operation appends a new default to the end of the list and a **deactivate** operation removes a default from the list.

At the end of each **alt** statement, if the default list is not empty and with the current snapshot none of the alternatives is executable, the default mechanism is evoked. The default mechanism invokes the first altstep in the list of defaults and waits for the result of its termination. The termination can be *successful* or *unsuccessful*. Unsuccessful means that none of the top alternatives of the altstep defining the default behaviour is executable, successful means that one of the top alternatives has been executed.

In case of an unsuccessful termination, the default mechanism invokes the next default in the list. If the last default in the list has terminated unsuccessfully, the default mechanism will return to the **alt** statement, and indicate an *unsuccessful default execution*. Unsuccessful default execution causes the **alt** statement to be executed again (Section 2.5).

In case of a successful termination, the default may either stop the test component by means of a stop statement, or the main control flow of the test component will continue immediately after the **alt** statement from which the default mechanism was called. If the test component is to execute the **alt** statement again, this has to be specified explicitly by means of a **repeat** statement. If the selected top alternative of the default ends without a **repeat** statement the control flow of the test component will continue immediately after the **alt** statement.

2.8. Semantics of timers

TTCN-3 provides a *timer* mechanism. Timers are local to components. A component can start and stop a timer, check if a timer is running, read the elapsed time of a running timer and process timeout events after timer expiration.

3. The TTCN-3 core language

This section explains the TTCN-3 core language by using the example of a simplified ATM.

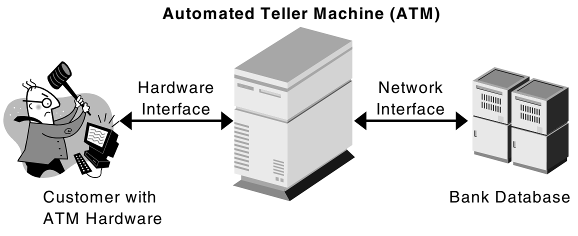


Fig. 5. Structure of an ATM system.

We introduce the example and describe the TTCN-3 core language while presenting a TTCN-3 test suite for the ATM example.

3.1. ATM example

An ATM as shown in Fig. 5 is a computer, which is connected with a *bank database* and *ATM hardware*. The connections among these components are realized by a *hardware interface* and a *network interface*. The ATM system allows a customer to withdraw money from his account. The ATM hardware gives a customer access to the ATM system. The bank database is needed to check and update the credit balance of a customer.

A complete and successful money withdrawal procedure is shown in Fig. 6. Our example abstracts from hardware interactions like putting the credit card into the ATM hardware, typing the PIN number or delivering money, by modelling such interactions in form of messages. The withdrawal

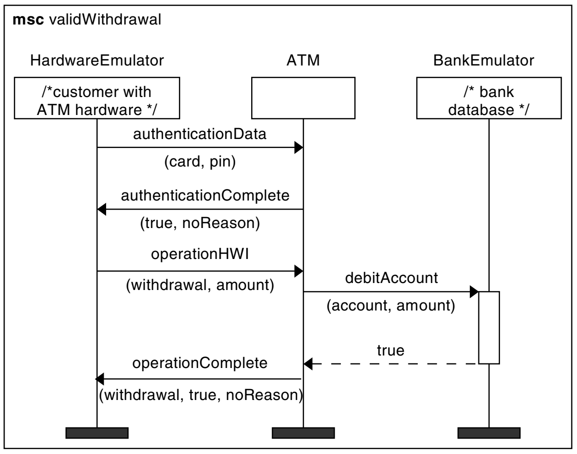


Fig. 6. Withdrawal of money.

of money starts with the authentication of the customer. The customer puts the card into the ATM hardware, the ATM hardware reads the card data, collects the PIN number from the customer and transmits both to the ATM (`authenticationData(card, pin)`). The ATM decodes the card data, verifies the PIN number against the decoded card data and returns the result of this verification to the ATM hardware `authenticationComplete(true, noReason)`.

Afterwards, the customer sends a withdrawal request together with the amount to be withdrawn to the ATM (`operationHWI(withdrawal, amount)`). The ATM debits the account identified by the card and the PIN number in the bank database (`debitAccount(account, amount)`) and the bank database acknowledges the successful debit operation (`true` reply arrow). Finally, the ATM acknowledges the completion of the debit operation (`operationComplete(withdrawal, true, noReason`).

Not shown in Fig. 6 are failures of the authentication, such as entering a wrong PIN or trying to debit more money than is in the account. In both these cases, the error is indicated and the withdrawal procedure terminated: a warning is displayed, the card is returned to customer and the ATM system returns to an idle state.

A test system for this example is shown in Fig. 7. It consists of an MTC *hardware emulator* and a PTC *bank emulator*. During the test, the hardware emulator emulates the behaviour of the customer and the ATM hardware. The bank emulator emulates the bank database. The *hardware interface* is realized by the two ports `hwiCom` and `hwiSUT`. The *network interface* is implemented by be two ports `niSUT` and `niCom`. Both test components coordinate themselves by using the coordination ports `coHWE` and `coBE`.

3.2. Test suite and module structure

Writing a TTCN-3 test suite for the ATM example means to specify one or more TTCN-3 modules. The ATM example test suite is structured into three modules. The module `ATM_Test`, which is shown in Fig. 8 is the main module. The other modules are referenced in **import** statements (lines 11 and 13, see also next section). They provide definitions related to the *hardware interface* and the *network interface*. As explained in Section 2.1, the module `ATM_Test`, is structured into a module definitions part (lines 2–47) and a module control part (lines 48–58). In TTCN-3 modules, the control part is optional, because pure library modules may not define any execution order for test cases.

The definitions in the module definitions part specify or reference all information needed for the execution of the test cases in the module control part. Elements of the module definitions part are definitions of data (constants, data types, test data), test behaviour (functions, test cases and altsteps) and references to definitions outside of the module.

As shown in the lines 21–32 of Fig. 8, definitions can be collected in named groups. Groups may be nested: a group may contain other groups. Grouping is done to aid readability and to add logical structure to a module.

3.3. Import of definitions from other modules

The test suite for the ATM example is structured into three modules. TTCN-3 supports such a structuring by providing a very powerful **import**

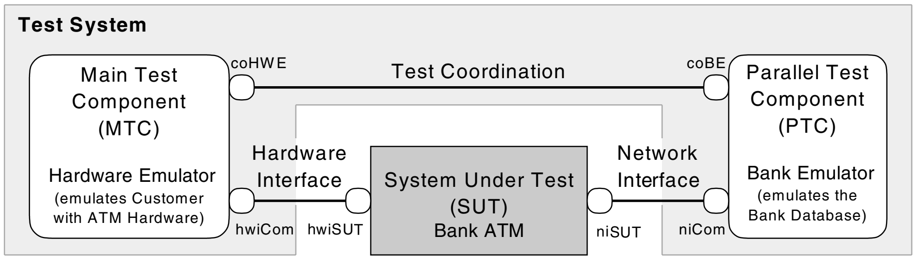


Fig. 7. Test system for the ATM example.

```
 (1) module ATM_Test {
 (2)
 (3)   modulepar { // Module parameter Definition
 (4)     float         maxDurOfTC_Par := 1.8;  // Parameter with default value
 (5)     card_Type     validCard_Par;
 (6)     integer       validPin_Par, validAmount_Par;
 (7)   }
 (8)
 (9)   external const float TestExecutionTime;  // External constant
(10)
(11)   import from Network_Interface all;    // Import of all definitions
(12)
(13)   import from Hardware_Interface {      // Import of selected definitions
(14)     const all;
(15)     group withdrawal_data except {      // Excluding definitions
(16)       type operationHWI_Type;
(17)       template validWithdrawalOp_Template;
(18)     }
(19)   }
(20)
(21)   group Test_Behaviour_Definitions {   // Group definition
(22)
(23)     // Test case for the ATM example
(24)     testcase validWithdrawal ... {
(25)       ... // The complete definition can be found in figure 14.
(26)     }
(27)
(28)     // Behaviour for authentication procedure
(29)     function authentication ... {
(30)       ... // The complete definition can be found in figure 15.
(31)     }
(32)
(33)     // Behaviour for the bank emulator test component
(34)     function BE_validWithdrawal ... {
(35)       ... // The complete definition can be found in figure 15.
(36)     }
(37)
(38)     // Default behaviour for the hardware emulator test component
(39)     altstep HWE_Default ... {
(40)       ... // The complete definition can be found in figure 15.
(41)     }
(42)   } // end group
(43)
(44)   // Further Definitions of the module, e.g., constants, test cases,
(45)   // functions, altsteps, templates, etc. Some of these definitions will
(46)   // be presented in the following sections.
(47)
(48)   control { // begin of the module control part
(49)     var verdicttype testCaseVerdict := none;
(50)     var reason_Type reason := noReason;
(51)
(52)     testCaseVerdict := execute(validWithdrawal(reason),TestExecutionTime);
(53)
(54)     if (testCaseVerdict == pass) {
(55)       execute(invalidWithdrawal(),TestExecutionTime);
(56)     }
(57)
(58)   } // end control
(59) } // end module
```

Fig. 8. TTCN-3 module `ATM_Test`.

statement, which allows the import of single definitions, groups of definitions, definitions of the same kind or of all definitions from other modules. In addition, it supports the import from non-TTCN-3 descriptions, such as ASN.1 and IDL definitions (Section 5). The TTCN-3 module `ATM_Test` (Fig. 8) imports all definitions from module `Network_Interface` (line 11) and some selected definitions from module `Hardware_Interface` (lines 13–19).

3.4. Parameterization of modules and external definitions

The example TTCN-3 module `ATM_Test` (Fig. 8) may have to be executed in several banks and on different computer hardware. For the adaptation of TTCN-3 modules to different environments, modules can be parameterised. A module parameter is a constant at runtim: it cannot change its value during the test execution. Our example module `ATM_Test` has the module parameters `maxDurOfTC_Par`, `validCard_Par`, `validPin_Par` and `validAmount_Par` (lines 3–7 in Fig. 8).

Some constants or functions may not be test specific, but hardware specific. TTCN-3 allows to specify such definitions as **external**. For example, the constant `TestExecutionTime` (line 9) specifies the maximal duration of a test case. This information is considered to be test equipment specific and the constant is therefore declared as **external**.

3.5. Data types, procedure signatures and test data

For testing the ATM in the example, the information exchanged between the test components and other test components or the SUT has to be specified. Our example includes two types of communication (see Figs. 6 and 7):

(i) message-based communication via the *hardware interface* and for the coordination among the test components;
(ii) procedure-based communication via the *network interface*.

In TTCN-3, message-based communication is related to *data types* and procedure-based communication is related to *procedure signatures*. A special *template* mechanism can be used to specify test data for both communication mechanisms.

3.5.1. Data types

TTCN-3 has no special message type. A message can be any value of a valid TTCN-3 data type. For the definition of data types, TTCN-3 supports a number of *simple basic*, *basic string*, *structured*, *special data*, *special configuration* and *special default types*. An overall view of all types is given in Fig. 9. Most types (such as **boolean**, **integer** or **record**) are known from typical programming languages and need no further explanation. The special configuration and special default types will be explained in Sections 3.6.1 and 3.8.1. The **verdicttype** and the **anytype** have a short description here.

The type **verdicttype** is an enumeration type with the values **pass**, **fail**, **inconc**, **fail**, **error** and **none**. These values can be used for the judgement of complete and partial test runs. Their meaning has already been explained in Section 2.3. TTCN-3 provides the operations **setverdict** and **getverdict** for setting and retrieving the actual value of local test verdicts.

The special data type **anytype** is defined as a shorthand for the union of all known types in a TTCN-3 module. The fieldnames of the **anytype** are uniquely identified by the corresponding type names.

3.5.2. Procedure signatures

As already mentioned, procedure signatures are needed for procedure-based communication. The **signature** definition of the `debitAccount` procedure, which is used for the communication at the

Class of type	Keyword
Simple basic types	**integer**, **char**, **universal char**, **float**, **boolean**, **objid**, **verdicttype**
Basic string types	**bitstring**, **hexstring**, **octetstring**, **charstring**, **universal charstring**
Structured types	**record**, **record of**, **set**, **set of**, **enumerated**, **union**
Special data types	**anytype**
Special configuration types	**address**, **port**, **component**
Special default types	**default**

Fig. 9. Overall view of TTCN-3 types.

```
 (1)   // Data Type for a message
 (2)   type record operationHWI_Type {
 (3)     HWI_ops  operation,
 (4)     integer  argument
 (5)   }
 (6)
 (7)   // Message template
 (8)   template operationHWI_Type validWithdrawalOp_Template := {
 (9)     operation := withdrawal,
(10)     argument  := validAmount_Par
(11)   }
(12)
(13)   // Signature definition
(14)   signature debitAccount(account_Type account, integer amount)
(15)     return boolean exception (reason_Type);
(16)
(17)   // Signature template
(18)   template debitAccount valid_Debit := {
(19)     account := validAccount_Par,
(20)     amount  := 200
(21)   }
```

Fig. 10. Data type, signature and template definitions for the ATM example.

network interface of the ATM example (see Fig. 6) can be found in lines 13–15 of Fig. 10. A procedure signature has a name, a list of parameters, a return value and a list of exceptions. Parameters, return value and exceptions are optional.

3.5.3. Specification of test data

The TTCN-3 template mechanism provides the possibility to specify, organize and structure test data in a very comfortable way. A template either describes a concrete value or specifies a subset of values of a given data type or signature. Therefore, templates can be used to define concrete values to be transmitted or to describe conditions to be matched by received values. The latter is done by using *matching mechanisms*. Furthermore, templates can be parameterised and provide a simple form of inheritance. This enables the adaptation of templates to different testing situations and avoids the duplication of similar test data. Templates can be specified for any TTCN-3 data type or procedure signature. Type-based templates used for message-based communication and *signature templates* used for procedure-based communication called *message templates* in the following.

An example for the definitions of a **record** type and a corresponding message template is shown in the lines 1–11 of Fig. 10. The record `operationHWI_Type` specifies the type of the `operationHWI` message in Fig. 6. The template `validWithdrawalOp_Template` provides values for the fields of the record.

The template mechanism may also be used to specify signature templates. Signature templates are instances of procedure parameter lists with actual values. Signature templates may be defined for any procedure by referencing the associated signature definition. A template for the `debitAccount` procedure is shown in the lines 17–21 of Fig. 10.

Matching mechanisms are used to describe conditions on values of a given type and can be used in templates. They are comparable to regular expressions, but they mainly work on value sets and do not provide possibilities to describe complex patterns within strings. For string types, TTCN-3 also supports regular expressions (annex B.1.5 of part 1 of [1]).

3.6. Defining the test system

In addition to data types, procedure signatures and test data, the test system for our ATM example (Fig. 7) has to be defined. For this, TTCN-3 provides special configuration types: **address**, **port** and **component**—see Fig. 9) and the configuration operations **create**, **connect**, **map**, **start** and **stop**.

For the ATM example, the **address** type is not needed. TTCN-3 provides this type to address entities within the SUT. This can be necessary, if the SUT has several distinct entities known by the test system.

3.6.1. Defining ports and components

Instances of the special configuration types **port** and **component** are created, when a test case is invoked or a test component is created.

Ports are specified by means of port type definitions (see lines 1–15 in Fig. 11). Ports are either *message-based*, *procedure-based* or *mixed*. Message-based ports are used for communication by means of message exchange. Procedure-based ports are used for communication by means of remote procedure calls. A mixed port is a shorthand for a message-based and a procedure-based port with the same name. Ports are directional. Each port may have an **in** list (for the *in-direction*), an **out** list (for the *out-direction*) and an **inout** list (for both directions) of allowed messages or procedures. The port is unidirectional, if one of the lists is empty.

The port type definitions for the ATM example are shown in Fig. 11. The port type for the *hardware interface* is a message-based port `hardwareInterface_Ptype`. Messages of type `authenticationData_Type` and `operationHWI_Type` can be sent and messages of type `authenticationComplete_Type`, `operationComplete_Type` and `status_Type` can be received. A procedure-based port of type `networkInterface_Ptype` is used for the communication between ATM and the bank emulator test component. The port type refers to the procedure with the name `debitAccount` in the **in**-direction. This means that the ATM may invoke the `debitAccount` procedure in the bank emulator test component. Test coordination among test components may be performed by using message-based ports of type `testCoordination_PType`. Ports of this type can be used to send and receive messages of type `reason_Type`.

In TTCN-3, each test component is an instance of a corresponding **component** type. The **component** type definition declares the constants, variables, timers and ports owned by a component instance of that type. The component type

```
(1)  type port hardwareInterface_PType message {  // Port type for the Hardware
(2)    out authenticationData_Type,               // Interface
(3)        operationHWI_Type;
(4)    in  authenticationComplete_Type,
(5)        operationComplete_Type,
(6)        status_Type
(7)  }
(8)
(9)  type port testCoordination_PType message { // Port type for test coordination
(10)   inout reason_Type
(11) }
(12)
(13) type port networkInterface_PType procedure { // Port type for the Network
(14)   in debitAccount                            // Interface
(15) }
(16)
(17) type component hardwareEmulator_CType { // Component Type for the Hardware
(18)   port hardwareInterface_PType hwiCom;  // Emulator
(19)   port testCoordination_PType coHWE;
(20)
(21)   timer testCaseGuard := maxDurOfTC_Par;
(22) }
(23)
(24) type component bankEmulator_CType { // Component Type for Bank Emulator
(25)   port networkInterface_PType niCom;
(26)   port testCoordination_PType coBE;
(27) }
(28)
(29) type component ATM_Interface_CType {   // Component Type for the Test System
(30)   port hardwareInterface_PType hwiSUT; // Interface
(31)   port networkInterface_PType niSUT;
(32) }
```

Fig. 11. Port and component types for the ATM example.

definitions for the ATM example are shown in Fig. 11. The component type for the hardware emulator component has the name `hardwareEmulator_CType` (lines 17–22). A test component of that type owns two message-based ports with the name `hwiCom` and `coHWE`. As shown in Fig. 7 the port `hwiCom` is used for the communication with the ATM. Port `coHWE` is used for the exchange of coordination messages with the bank emulator test component. The component type `bankEmulator_Ctype` defines the type of the bank emulator test component.

A **component** type definition is also used to define the *abstract test system interface* (Section 2.2) because, conceptually, the abstract test system interface can be seen as a test component, i.e., a collection of ports. The component type for the abstract test system interface for the ATM example is shown in lines 29–32 of Fig. 11.

Variables of a component type are references to component instances. Component references are the result of **create** component operations (see next section). They may be used for addressing purposes in communication or configuration operations. The abstract test system interface and the MTC are created automatically when a test case starts its execution. Their references can be retrieved by the predefined operations **system** and **mtc**. Additionally, the **self** operation allows a test component to retrieve its own reference.

3.6.2. Configuration operations

Configuration operations are concerned with setting up and controlling test components. During the execution of a test case, the actual test system (Section 2.2) is created dynamically by performing configuration operations. Configuration operations are **create**, **connect**, **map**, **start** and **stop**.

The usage of the configuration operations for the ATM example can be seen in Fig. 14. The MTC of type `hardwareEmulator_CType` (see **runs on** clause in line 2) is the only test component that is created automatically when a test case starts. The PTC of type `bankEmulator_CType` is created in line 9. As indicated earlier, the **create** operation returns a unique reference to the newly created instance, which is stored in variable `BE_PTC`. This reference is used for connecting ports (line 13), for mapping ports (lines 12 and 14), for starting the component (line 16) and for stopping the component (line 53). In addition, component references may also be used for addressing purposes in communication operations. The meaning of the **map** and **connect** operation has already been explained in Section 2.2.

The **start** operation binds the behaviour to a component by referring to a function after it has been created and possibly after the ports of the newly created component have been mapped or connected to other ports. For example, the **start** operation `BE_PTC. start(BE_validWithdrawal())` in line 16 of Fig. 14 starts the function `BE_validWithdrawal()` on the bank emulator test component of the ATM test case example.

By using the **stop** operation, a test component is able to stop itself (lines 26 and 55) or other components. In the latter case, the **stop** operation has to be prefixed by a component reference (line 25 and 53). Furthermore, a test component can stop the whole test case by stopping the MTC and the MTC may stop all PTCs at once by using an **all component** prefix. The control component of a TTCN-3 module can also stop itself by using a **stop** operation without a prefix. Nevertheless, the control component cannot stop test components and cannot be stopped by any test component.

3.7. Communication

Both message-based and procedure-based communication mechanisms are handled in a uniform manner by providing symmetric peers of sending and receiving operations. The operation peers are **send/receive** for the exchange of messages, **call/getcall** for the handling of procedure calls, **reply/getreply** for the treatment of replies to procedure calls and **raise/catch** for dealing with exceptions.

3.7.1. Message-based communication

Message-based communication is communication based on an asynchronous message exchange. Message-based communication is non-blocking on the **send** operation, as illustrated in Fig. 12 where processing in the `SENDER` continues immediately

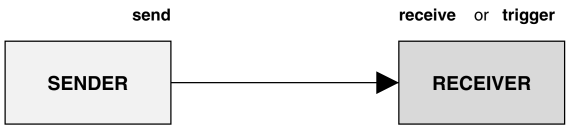


Fig. 12. Illustration of message-based communication.

after the **send** operation. The RECEIVER is blocked on the **receive** operation until it receives a message. In addition to the **receive** operation, TTCN-3 provides a **trigger** operation that filters messages with certain matching criteria from a stream of received messages on a given incoming port. All messages that do not fulfil the matching criteria are removed from the port without any further action.

Message-based communication by means of **send** and **receive** operations is frequently used in the ATM test case example. In line 30 of Fig. 14, a message defined by the message template validWithdrawalOp_Template (lines 8–11 in Fig. 10) is sent to the ATM. In line 39 of Fig. 14, the test component receives any message of type operationComplete_Type. The **trigger** operation is not used in the ATM example.

3.7.2. Procedure-based communication

The principle of procedure-based communication is to call procedures in remote entities. TTCN-3 supports *blocking* and *non-blocking* procedure-based communication. Blocking procedure-based communication is blocking on the calling and the called side, whereas non-blocking procedure-based communication is blocking on the called side only.

The communication scheme of blocking procedure-based communication is shown in Fig. 13a. The CALLER calls a remote procedure in the CALLEE by using the **call** operation. The CALLEE accepts the call by means of a **getcall** operation and reacts by either using a **reply** operation to answer the call or by raising (**raise** operation) an exception. The CALLER handles a reply or exception by using **getreply** or **catch** operations. In Fig. 13a, the blocking of CALLER and CALLEE is indicated by means of dashed lines.

The communication scheme of non-blocking procedure-based communication is shown in Fig.

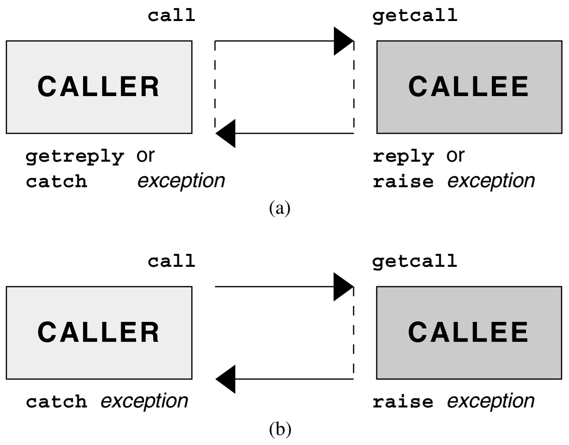


Fig. 13. Illustration of procedure-based communication: (a) blocking, (b) non-blocking.

13b. The CALLER calls a remote procedure in the CALLEE by using the **call** operation and continues its own execution, i.e., does not wait for a reply or exception. The CALLEE accepts the call by means of a **getcall** operation and executes the requested procedure. If the execution is not successful, the CALLEE may raise an exception to inform the CALLER. The CALLER may handle the exception afterwards by using a **catch** operation in an **alt** statement (Section 2.5). In Fig. 13b, the blocking of the CALLEE until the end of the call handling and possible raise of an exception is indicated by a dashed line.

In the ATM example, procedure-based communication is used at the *network interface* (Fig. 7). The *bank emulator* PTC plays the role of a callee in this communication. Therefore, the behaviour definition of the PTC (lines 14–38 in Fig. 15) only includes examples for the usage of **getcall** (lines 18, 22 and 27), **reply** (line 19) and **raise** (lines 23 and 28) operations.

3.8. Defining test behaviour

The features described allow specification of a complete test case for the ATM example in Section 3.1. Basically, the example test case tests the valid withdrawal of money, which is visualized by the message sequence charts (MSC) in Fig. 6. The test behaviour of the example test case is structured into one test case definition, two function definitions and one **altstep** definition. This structure is indicated in the definitions part of the module `ATM_Test` (lines 21–42 in Fig. 8). Functions and altstep are invoked inside the test case definition.

3.8.1. Test case definitions

The TTCN-3 test case `validWithdrawal` for testing the valid withdrawal of money is shown in Fig. 14. The test case is parameterised with the **inout** parameter `reason` of type `reason_Type`.

```
testcase validWithdrawal (inout reason_Type reason)
 runs on hardwareEmulator_CType system ATM_Interface_CType {

  var boolean result;
  var operationComplete_Type erroneousWithdrawal;
  var default theDefault := activate(HWE_Default());

  // Creation of PTC
  var bankEmulator_CType BE_PTC:=bankEmulator_CType.create;

  // Mapping & Connecting ports
  map(system:niSUT, BE_PTC:niCom);
  connect(self:coHWE, BE_PTC:coBE);
  map(system:hwiSUT, self:hwiCom);

  BE_PTC.start(BE_validWithdrawal()); // start PTC

  testCaseGuard.start; // start guarding timer

  // invocation of authentication procedure
  result := authentication(validCard_Par, validPin_Par, reason);

  if (result != true) { // test case fails in authentication procedure
    setverdict(inconc);
    BE_PTC.stop;  // stop the bank emulator test component
    stop;
  }

  // start of withdrawal procedure
  hwiCom.send(validWithdrawalOp_Template);

  reason := unknown; // return value if MTC fails in default
  alt {
    [] hwiCom.receive(operationComplete_Type:{withdrawal, true, ?}) {
         reason := noReason;
         setverdict(pass);
         all component.done; // wait for the termination of PTC
       }
    [] hwiCom.receive(operationComplete_Type:?) -> value erroneousWithdrawal {
         setverdict(fail);
         reason := erroneousWithdrawal.reason;
       }
    [] coHWE.receive(reason_Type:?) -> value reason {
         setverdict(fail);
       }
    [] testCaseGuard.timeout {
         reason := notInTime;
         setverdict(fail);
       }
  }
  deactivate(theDefault);
  if (reason != noReason) { // Stopping PTC in case of a failure
    BE_PTC.stop;
  }
  stop;
}
```

Fig. 14. TTCN-3 test case `validWithdrawal`.

```
// Behaviour for authentication procedure
function authentication(card_Type card, integer pin, out reason_Type reason)
 runs on hardwareEmulator_Ctype return boolean {

  var authenticationComplete_Type authenComplete;

  hwiCom.send(authenticationData_Type:{card, pin});
  hwiCom.receive(authenticationComplete_Type:?) -> value authenComplete;

  reason := authenComplete.reason;
  return authenComplete.success;
}

// Behaviour for the bank emulator test component
function BE_validWithdrawal() runs on bankEmulator_CType {

  alt {
    []  niCom.getcall(debitAccount: {validAccount_Par, validAmount_Par}) {
          niCom.reply(debitAccount: {-, -} value true);
          setverdict(pass);
        }
    []  niCom.getcall(debitAccount: {validAccount_Par, ?}) {
          niCom.raise(debitAccount, reason_Type: amountNotValid);
          coBE.send(reason_Type: amountNotValid);
          setverdict(fail);
        }
    []  niCom.getcall(debitAccount: {?, validAmount_Par}) {
          niCom.raise(debitAccount, reason_Type: accountNotFound);
          coBE.send(reason_Type: accountNotFound);
          setverdict(fail);
        }
    []  coBE.receive {
          setverdict(inconc);
          coBE.send(reason_Type: unknown);
        }
  }
  stop;
}
// Default behaviour for the hardware emulator test component
altstep HWE_Default() runs on hardwareEmulator_CType {
  [] hwiCom.receive (status_Type:?) {
       repeat;
     }
  [] hwiCom.receive {
       setverdict(fail);
       stop;
     }
  [] coHWE.receive {
       setverdict(fail);
       stop;
     }
}
```

Fig. 15. Function and altstep definitions.

The **runs on** clause following the parameter refers to the component type `hardwareEmulator_CType` (lines 17–22 of Fig. 11), which is the type of the MTC. The **system** clause specifies the type of the test system interface: `ATM_Interface_CType`. The **runs on** and **system** clauses uncover declarations of the component types inside the body of the test case definition. The test case body defines the behaviour of the MTC. The MTC will be created and started automatically when the test case is invoked.

The test case body starts with variable declarations (lines 4, 5, 6 and 9 of figure Fig. 14). The variable `theDefault` is of the special default type **default** and initialised with the result of a default activation. The default mechanism has been explained in Section 2.7. The **activate** operation refers to the **altstep** definition HWE_Default (Fig. 15 lines 39–52). The result of the **activate** operation is reference for the activated default, which later on is used to deactivate the default (line 51 of Fig. 14). The variable BE_PTC, which is declared and initialised in line 9 of the test case definition, is component variable. It is used to store the reference to the PTC of the test system.

The operations **map**, **connect** and **start** in the lines 12–16 of Fig. 14 set up the test system shown in Fig. 7 and start the behaviour of the PTC.

The timer **start** operation in line 18 starts the component timer testCaseGuard with its default value. Timer and default value are declared in the component type of the MTC at line 21 of Fig. 11: hardwareEmulator_CType.

Afterwards, the function authentication is called (line 21 of Fig. 14). This function defines the authentication procedure for the ATM example. The result of the function is stored in variable result. The lines 23–27 check if the authentication was successful. If not, the local test verdict is set to **inconc**, the PTC is stopped and then, the test case stops itself.

The withdrawal procedure starts in line 30 by sending a withdrawal request to the ATM. As a result of this request, the ATM may react in different ways. These alternatives are defined in the **alt** statement shown in the lines 33–50. The expected alternative, which leads to a **pass** verdict, is shown in the lines 34–38. The **done** operation in line 37 allows waiting for the correct termination of PTCs. In this case the usage of the **all component** shorthand is shown. It means that all PTCs have to terminate before the execution of the MTC continues. The **done** operation may also be used in combination with a component reference. The other alternatives of the **alt** statement cover the reception of other messages at the ports of the MTC and the expiration of the testCaseGuard timer (lines 46–49). This example shows that timeout events of timers are handled in the same manner as messages: a **timeout** operation is comparable to a **receive** operation.

After completion of the withdrawal procedure, the default is deactivated (line 51) and the PTC is stopped if the withdrawal procedure was not successful (lines 52–54). Finally, the MTC stops itself (line 55).

3.8.2. Function definitions

TTCN-3 functions are used to express test behaviour or to structure the computation within a module. A function may be parameterised and may return a value. As shown in the function definition of authentication in Fig. 15, the return value is defined by the **return** keyword followed by a type identifier.

If a function defines test behaviour, the type of the test component on which the behaviour is executed has to be specified by means of a **runs on** clause (see function authentication in lines 1–12 of Fig. 15). This type reference uncovers declaration of the component type inside the behaviour definition of the function.

The functions in Fig. 15, authentication and BE_validWithdrawal, define test behaviour for the ATM test case validWithdrawal in Fig. 14. Function authentication specifies the authentication procedure at the hardware interface and function BE_validWithdrawal specifies the behaviour of the bank emulator test component.

Basically, the authentication procedure of the ATM example consists of a **send** operation, where authentication data is sent and a **receive** operation as the result of the authentication. The stand-alone **receive** operation in line 8 of Fig. 15 is considered to be shorthand for an **alt** statement with only one alternative. Therefore, the default, which has been activated in the test case definition before the authentication function is called, may cover the reception of unexpected messages.

Function BE_validWithdrawal (lines 14–38 of Fig. 15) defines the behaviour of the bank emulator test component and is therefore responsible for the communication at the procedure-based port niCom. It plays the role of the callee and therefore reacts on calls received from the ATM at port niCom and possible messages from the MTC at coordination port coBE. Therefore the body of BE_validWithdrawal mainly consists of a large **alt** statement. It should be noted that syntax and usage of the **getcall**, **reply** and **raise** operations is comparable to syntax and usage of **receive** and **send** operations.

3.8.3. Altstep definitions

Altsteps are used to structure **alt** statements and to define default behaviour. The signature of an altstep definition is very similar to a function definition (see altstep `HWE_Default` in lines 40–52 of Fig. 15). An altstep has a name, and may have parameters and a **runs on** clause. The only difference is that altsteps have no return values. The altstep `HWE_Default` defines the default for the MTC of the ATM test case example. It removes status messages from the `hwiCom` port without stopping the test case and covers all unexpected events by setting the local verdict to **fail** and stopping the test case.

3.9. Execution of test cases

Test cases are defined in the module definitions part and executed in the control part of a module. A test case is invoked using an **execute** statement. As the result of the execution of a test case, a test verdict is returned and may be assigned to a variable for further processing. Optionally, the **execute** statement allows the supervision of a test execution by means of a timer duration. If the test case does not end within this duration, the result of the test case execution is an **error** test verdict and the test system terminates the running test case.

The module control part of the module `ATM_Test` is shown in the lines 48–58 of Fig. 8. It includes two **execute** statements in the lines 52 and 55. Both statements guard the execution of the test cases with the duration `TestExecutionTime`, which is declared to be an **external** constant of type **float**. In this example, the test case `invalidWithdrawal` is executed only if the test case `validWithdrawal` terminates with a **pass** verdict.

3.10. Further TTCN-3 constructs

This paper can only give an impression of the TTCN-3 core language. In addition to the presented language constructs, TTCN-3 provides a variety of conversion functions for the data handling, additional operations for controlling the port communication and examining the contents of ports, further constructs for defining behaviour (such as **while**-loop, **goto**, **label** or **for**-loop) and special test constructs for logging and defining interleaved behaviour. An explanation and examples for the usage all these constructs can be found in the TTCN-3 language definition [1].

4. The TTCN-3 presentation formats

TTCN-3 offers various presentation formats to serve the needs of different TTCN-3 application domains and users (Fig. 1). The programming-like textual core notation can be developed within a text editor of the users' choice and enables an easy integration into an overall test environment. The graphical format of TTCN-3 is based on MSC [9] based and aids the visualization of test behaviour. The tabular presentation format highlights the structural aspects of a TTCN-3 module and in particular of structures of types and templates. The graphical format eases the reading, documentation and discussion of test procedures and is also well suited to the represention of test execution and analyzing of the test results. Where needed, additional presentation formats to highlight further specific aspects of TTCN-3 modules can be defined and integrated into a TTCN-3 development environment.

4.1. The tabular presentation format

The *tabular presentation format for TTCN-3* (TFT) is defined in part 2 of [1]. It is designed for users that prefer the TTCN style of writing test suites [4]. TFT presents a TTCN-3 module as a collection of tables. In the following, some selected TFT tables for the ATM example (Section 3.1) are presented. The structured type definition `operationHWI_Type` and a template of this type are shown in Figs. 16 and 17.

The structured type definition and the corresponding template definition are examples for TTCN-3 language constructs that are presented in one table per definition. Some TFT tables allow to present several definitions of the same kind in one table. An example for such a compact description is shown in Fig. 18. The table presents the module parameter definitions of the ATM example.

Structured Type		
Name	: operationHWI_Type	
Group	:	
Structure	: record	
Encoding	:	
Comments	:	
Field Name	**Field Type**	**Field Encoding**
operation	HWI_ops	
argument	integer	
Detailed Comments :		

Fig. 16. Structured type definition in tabular presentation format.

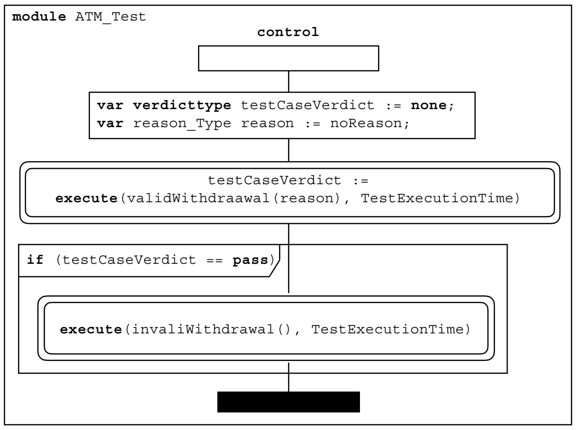
4.2. *The graphical presentation format*

Part 3 of [1] defines the *graphical presentation format for TTCN-3* (GFT) [10,11]. GFT provides a visualization of TTCN-3 behaviour definitions in an MSC-like manner. For each kind of TTCN-3 behaviour definition, GFT provides a special diagram type: *control diagrams* for the control part of a module; *function diagrams* for functions; *test case diagrams* for test cases; *altstep diagrams* for altsteps.

The control part of the ATM example (Section 3.1) is visualized by the GFT control diagram shown in Fig. 19. The control diagram uses the module identifier in its heading and has a specific control instance for the behaviour of the control part. At first, the variables `testCaseVerdict`

Structured Template		
Name	: validWithdrawalOp_Template	
Group	:	
Type/Signature	: operationHWI_Type	
Derived From	:	
Encoding	:	
Comments	:	
Element Name	**Element Value**	**Element Encoding**
operation	withdrawal	
argument	validAmount_Par	
Detailed Comments :		

Fig. 17. Template definition in tabular presentation format.



```
var verdicttype testCaseVerdict := none;
var reason_Type reason := noReason;

testCaseVerdict :=
execute(validWithdraawal(reason), TestExecutionTime)

if (testCaseVerdict == pass)

execute(invaliWithdrawal(), TestExecutionTime)
```

Fig. 19. GFT control diagram for the ATM example.

Module Parameters				
Name	**Type**	**Initial Value**	**PICS/PIXIT Ref**	**Comments**
maxDurOfTC_Par	float			
validCard_Par	card_Type			
validPin_Par	integer			
validAmount_Par	integer			
Detailed Comments :				

Fig. 18. Module parameter definitions in tabular presentation format.

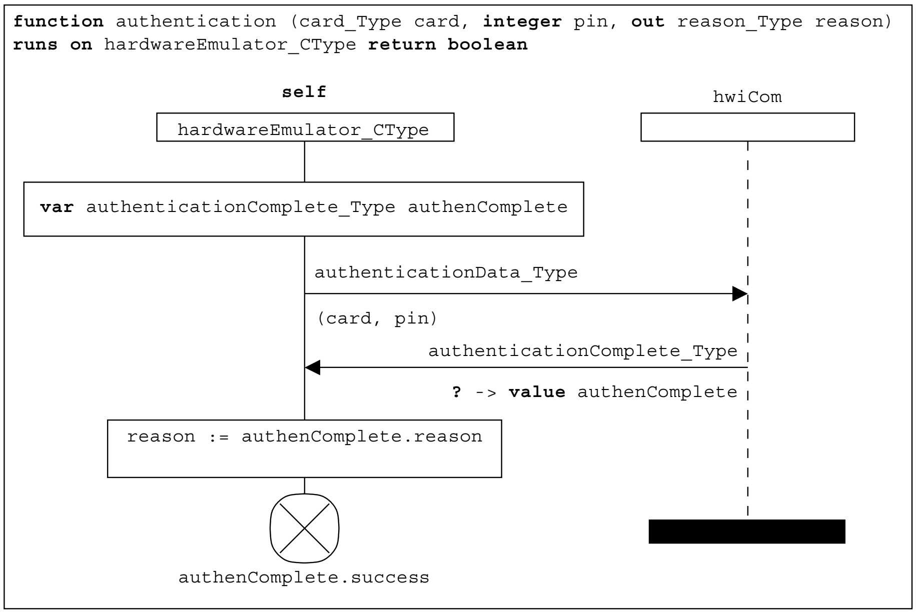

function authentication (card_Type card, integer pin, out reason_Type reason)
runs on hardwareEmulator_CType return boolean
self
hardwareEmulator_CType
hwiCom
var authenticationComplete_Type authenComplete
authenticationData_Type
(card, pin)
authenticationComplete_Type
? -> value authenComplete
reason := authenComplete.reason
authenComplete.success

Fig. 20. GFT representation of function authentication.

and `reason` are declared and assigned with an initial value. Afterwards, the test case `validWithdrawal` is executed. Only if this test case completed successfully, the second test case `invalidWithdrawal` will be invoked.

Another example of GFT is given by the function diagram for the `authentication` function in Fig. 20. The diagram header declares the **function** `authentication` including the function parameters, the **runs on** component type and the return type for this function. A **self** instance represents the test component that performs this function. At first, the variable `authenComplete` is defined. Then, a message of type `authenticationData_Type` is sent to the port `hwiCom`, which is represented by a port instance. Such GFT port instances highlight the interactions being performed at the component ports. A response message of an arbitrary value (indicated by the **?** wildcard) has to be received. Its value is bound to the local variable `authenComplete`. Finally, `reason` (being an **out** parameter) is calculated and the result of the authentication is returned (with returning `authenComplete. success`).

5. Using other languages with TTCN-3

TTCN-3 supports the referencing of objects defined in other languages (*foreign* objects) from within TTCN-3 test suites. This is illustrated in Fig. 1. Foreign objects can be used in TTCN-3 only if they have a *TTCN-3 view*.

The term TTCN-3 view can be best explained by considering the case when the definition of a TTCN-3 object is based on another TTCN-3 object, the information content of the referenced object shall be available and is used for the new definition. For example, when a template is defined based on a structured type, the identifiers and types of fields of the base type shall be accessible and are used for the template definition. In a similar way, when the base type is a foreign object it shall provide the same information content as would be required from a TTCN-3 type declaration.

The foreign object, naturally, may contain more information than required by TTCN-3. The TTCN-3 view of a foreign object means that part of the information carried by the object, which is necessary to use it in TTCN-3. Obviously the TTCN-3 view of a foreign object may be the full set or a subset of the information content of the object, but never a superset.

There may be foreign objects without a TTCN-3 view (zero TTCN-3 view), where for some reason no TTCN-3 definition could be based on them. We show an example of objects with different levels of TTCN-3 view in Section 5.1.

The use of foreign objects in TTCN-3 modules is supported in two ways. Firstly, the language allows importing and using them (by referencing), secondly special attribute strings are defined which assure that a TTCN-3 module referring foreign objects will be portable to any tool supporting the other language.

To make declarations of foreign object visible in TTCN-3 modules, their names shall be imported just like declarations in other TTCN-3 modules. When imported, only the TTCN-3 view of the object will be seen from the TTCN-3 module. There are two main differences between importing TTCN-3 items and objects defined in other languages:

- to import from a non-TTCN-3 module the import statement shall contain an appropriate language identifier string.
- only foreign objects with a TTCN-3 view are importable into a TTCN-3 module.

Importing can be done automatically using the **all** directive, in which case all importable objects shall automatically be selected by the testing tool, or done manually by listing names of items to be imported. Naturally, in the second case only importable objects are allowed in the list. Currently, standard language identifier strings are specified for ASN.1 only. The usage of TTCN-3 in combination with OMG IDL and XML has been studied [12,13] and the corresponding extensions of the TTCN-3 standard are under discussion.

5.1. Support of ASN.1

In several important application domains that are considered to be amongst the first potential users of TTCN-3 (like UMTS) ASN.1 [14] is used to describe the data structure of messages. For this reason, TTCN-3 provides sophisticated support to use ASN.1 together with TTCN-3. It allows the reference to ASN.1 definitions from within TTCN-3 modules and the specifying of encoding rules for imported ASN.1 definitions including the dynamic control of encoding options.

When all ASN.1 definitions of a given module are imported by a single "**all**" statement, imported objects can (only) be referenced by the same name as the original ASN.1 name with the exception that all dash characters (allowed in ASN.1 names) are changed to an underscore. This change is because dash is not allowed in TTCN-3 names to prevent mismatch with the minus sign. The change also applies to field identifiers of imported `SEQUENCE`, `SET` and `CHOICE` types, identifiers of associated types for `EMBEDDED PDV` and `EXTERNAL` and for enumeration identifiers of imported `ENUMERATED` types.

The TTCN-3 default import mechanism (nonrecursive import with importing internal type information) is almost the same as the ASN.1 import mechanism, used when ASN.1 definitions are imported from one ASN.1 module to another. The only difference is that ASN.1 allows re-exporting imported definitions, whereas TTCN-3 does not.

The use of ASN.1 definitions in TTCN-3 is more complex than just importing names. When importing from ASN.1, imaginary associated definitions are created behind the scene. Because ASN.1 is richer in types and type construction than TTCN-3, it is necessary to perform a sequence of transformations over the ASN.1 module. When all transformations are executed only types and constructs supported by TTCN-3 shall remain in the virtual ASN.1 module. Replacing remaining ASN.1 types with their TTCN-3 equivalents results in the associated type and value definitions for each ASN.1 name (their TTCN-3 views). In TTCN-3 all operations on imported ASN.1 definitions such as using them in value assignments, passing them as actual parameters, template matching, decision on type compatibility etc. shall be done according to the relevant associated definition.

The ASN.1 abstract syntax, in general, permits defining types (value sets are also considered being in this category), values and information object (IO) kind definitions like information objects, information object classes, and information object sets. ASN.1 definitions may be parameterised that also requires careful consideration.

The situation with the use of information objects, information object classes and information object sets is straightforward. There are no declarations of a similar kind in TTCN-3, and for this reason their TTCN-3 views are zero. This means no TTCN-3 definition could (directly) be based on them, hence they are not importable on their own. However, as will be shown later, their information content may be embedded into other (ASN.1) declarations and imported indirectly. This is similar to the use of information objects and IO sets used within ASN.1, where the notations for the object class field type and information from the object are used to map between information objects and X.680-based type and value definitions.

The use of non-parameterised ASN.1 values is simple and also straightforward: they can be used just like TTCN-3 constants. In other words their TTCN-3 view is full.

It is more difficult with the use of non-parameterised ASN.1 type declarations. Any imported ASN.1 type can be used as the base type when defining values or templates, types listed in port type declarations (to be allowed to be sent/received over that port), etc., but not as formal parameters in any other definition. TTCN-3 objects are only allowed to have value-type formal parameters and not type formal parameters. Therefore, the only relevant information content is that necessary for value notations (because in port type definitions only the names of types are listed, so their information content is extraneous). In general, the TTCN-3 views of most ASN.1 types is less than 100%. For example no tagging information, extension marker, exception rule or default value specification is transferred to the TTCN-3 view of an ASN.1 type. The example in Fig. 21 illustrates these points.

Named numbers and named bits need special attention. ASN.1 allows assigning specific names to integer values within `INTEGER` type definitions or to bit positions in `BIT STRING` type definitions. These names can be used in ASN.1 to reference values of the given type (in the context of the type and only in this context), but cannot be used when such an ASN.1 type is used in TTCN-3. TTCN-3 does not support named numbers or named bits in its type definition system, hence they are outside the TTCN-3 views of ASN.1 types. If these named numbers or bits are used within ASN.1, they will be changed to the literal value they represent at import.

Several ASN.1 types are nonexistent in TTCN-3 such as `EXTERNAL`, `EMBEDDED PDV`, `CHARACTER STRING`, `RELATIVE-OID`, `NULL` and most of the restricted character string types. Also ASN.1 provides a wider range of type construction mechanisms than TTCN-3 does (like `COMPONENTS OF`) and forms of type constraints. The TTCN-3 view either replaces an unsupported type with a construct allowing value notation for the type, or replaces it with a built-in TTCN-3 type (sometimes constrained) that provides an equivalent set of values to the ASN.1 type. Unsupported subtyping mechanisms (like user defined constraints, content constraints, pattern constraint, inner subtyping, etc.) are ignored at import. This means that the test suite writer has to use value range or value list matching mechanisms according to the neglected subtype specification instead of some more convenient matching mechanisms of TTCN-3 like `AnyValue` or `AnyOrNone`.

When they are used in TTCN-3, parameterised ASN.1 type or value declarations shall always be provided with actual parameters. This is obvious because “use” in this case means referencing the given definition. The situation is twofold. As TTCN-3 will only reference these definitions, it shall always provide the actual parameter when referencing. In TTCN-3, only values (and templates) are allowed as actual parameters, not types.[4] Hence we have no problem when the ASN.1 definition has formal value parameter(s) and none of the formal parameters are used in a subtype specification of the parameterised type. The actual value shall simply be passed when the

[4] Port and timer parameters are not considered here, because they are irrelevant from the point of view of using ASN.1.

```
-- A sample ASN.1 module:
ASN1-module-with-example-definitions
DEFINITIONS IMPLICIT TAGS ::=

BEGIN

My-Seq-Type1 ::= SEQUENCE {
    field-1  [0]    My-Int1      OPTIONAL,
    field-2  [1]    My-Int2      DEFAULT 7,
    ...!1
}

My-Int1 ::= INTEGER (2..3)
My-Int2 ::= INTEGER (2..7)

END

// A TTCN-3 module importing ASN.1 definitions:
module A_ttcn3_module {//start of the module

  import from   ASN1_module_with_example_definitions   language "ASN.1:1997"
               { type My_Seq_Type1 }
      // The TTCN-3 view (equivalent TTCN-3 definition) of My-Seq-Type1 is:
      // record My_Seq_Type1 {
      //   integer field_1 (2..3) optional, integer field_2 (2..7) optional }

      // Note, that by importing the ASN.1 type "My-Seq-Type1" by the default
      // import mechanism, the type information of field-1 and field-2 also
      // known in the TTCN-3 module but the names "My-Int1" and "My-Int2" are
      // not; TTCN-3 objects with the names My_Int1 and My_Int2 can be
      // defined without any name clash:

  const    integer     My_Int1    := 0;
  const    integer     My_Int2    := 1;

  template My_Seq_Type1 MyTemplate1 := { field_1 := My_Int1, field_2 := My_Int2 }
     // the values of field_1 and field_2 will be 0 and 1 accordingly

}//end of the module
```

Fig. 21. Importing ASN.1 types.

ASN.1 definition is used. TTCN-3 does not allow using formal parameters in subtype specifications. When a formal parameter is used in the subtype definition of an imported ASN.1 type the subtype specification remains invisible for TTCN-3 (when such type is used in another imported ASN.1 type declaration providing actual values the subtype information will implicitly be imported).

It is more complicated to use parameterised ASN.1 type definitions expecting value sets, types, information objects, IO classes or IO sets as actual parameters or to use ASN.1 value definitions with information object formal parameter(s). Such kind of actual parameters cannot be passed from TTCN-3 definitions. Hence such ASN.1 definitions do not have a TTCN-3 view and cannot therefore be imported. However, this does not mean losing the information they carry. This implies that only importable ASN.1 definitions referencing these parameterised objects (inside ASN.1) can be imported into and used in TTCN-3 not the parameterised definition itself.

The above discussion might lead the reader to the conclusion that the TTCN-3 views of ASN.1 objects do not contain enough information for correct encoding of values based on ASN.1 definitions. This is correct but we shall not forget that the only reason we introduced the *TTCN-3 view* was to specify an unambiguous way of creating new TTCN-3 declarations when they are based on ASN.1 definitions. Naturally, when dealing with ASN.1, test tools shall internally handle all the information needed for correct encoding. However, this remains behind the scenes and the (TTCN-3) user need not care about the way it is implemented by a specific tool.

The support of ASN.1 goes beyond *just* importing and allowing the use of ASN.1 objects in TTCN-3 modules. The standard defines encoding attribute and (encoding) variant attribute strings.

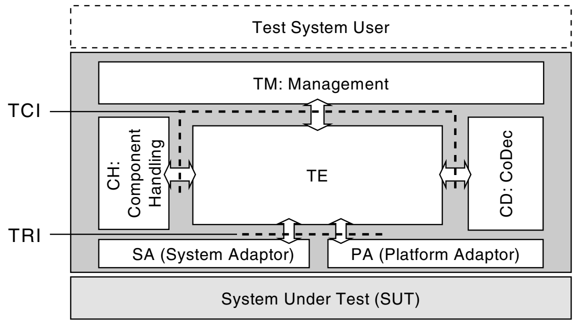
```
-- A sample ASN.1 module:
ASN1-module-with-example-definitions-2
DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

My-Seq-Type2 ::= SEQUENCE {
   a  [0] INTEGER,
   b      OCTET STRING
  }

My-Seq-Type3 ::= SEQUENCE {
   COMPONENTS OF My-Seq-Type2,
   c      BOOLEAN,
   d      BIT STRING
  }

END


// A TTCN-3 module importing ASN.1 definitions:
module A_second_ttcn3_module { //start of the module

 import from  ASN1_module_with_example_definitions_2  language "ASN.1:1997"  all
     with { encode "BER:1997";
            variant (My_Seq_Type2) "length form 1";
            variant (My_Seq_Type3) "length form 2"; }

      // The associated type are:
      //   type record My_Seq_Type2 {
      //      integer      a,
      //      octetstring  b
      //   }
      // and
      //   type record My_Seq_Type3 {
      //      integer      a,
      //      octetstring  b,
      //      boolean      c,
      //      bitstring    d
      //   }
      // The encode attribute causes all items imported shall be encoded in BER
      // with the restrictions implied by the variant attribute; namely allow the
      // short length form at encoding and decoding for values My_Seq_Type2 and
      // the long length form for values of My_Seq_Type3; as the attribute is
      // applied to the converted type, all fields of My_Seq_Type3 (a, b, c and d)
      // shall be encoded/decoded using the long length form

  template My_Seq_Type2 MyTemplate3 := { a := 5, b := '7F'O }

  template My_Seq_Type3 MyTemplate4 := { a:= 5, b:= '7F'O, c:= true, d:= '11011'B }

}//end of the module
```

Fig. 22. Example use of encoding attributes.

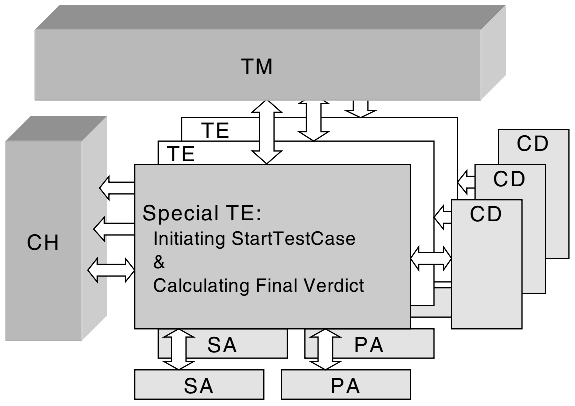
Adding these to imported ASN.1 definitions or templates based on ASN.1 definitions allows dynamically control the encoding rule used on a per message or even a per message part basis. Fig. 22 shows an example for the use of such attribute strings at import.

6. The execution interfaces of TTCN-3

The TTCN-3 execution interfaces *TTCN-3 runtime interface* (TRI) and *TTCN-3 control interfaces* (TCI) unify the way of realizing TTCN-3 based test systems [2,7,8]. They define a standardized adaptation of the test system for communication, management, component handling, external data and logging. The well-defined interfaces provide sets of operations, which are independent of the target (the SUT, processing platform, implementation language, etc.). Code from any TTCN-3 compiler or interpreter that supports these interfaces, can be executed on any test platform or test device, which supports the same interfaces.



(a)



(b)

Fig. 23. General structure of TTCN-3 test systems: (a) overall view; (b) distributed test system.

A TTCN-3 test system (Fig. 23a) can be conceptually thought of as a set of interacting entities where each entity corresponds to a particular aspect of functionality in a test system implementation. These entities manage test execution, interpret or execute compiled TTCN-3 code, realize proper communication with the SUT, handle types, values and test components, implement external functions, and handle timer operations.

The part of the test system that deals with interpretation and execution of TTCN-3 modules is the *TTCN-3 executable* (TE). Within the TE, individual structural elements can be identified, like control, behaviour, components, types and values and queues. The structural elements within the TE represent functionality that is defined within a TTCN-3 module or by a TTCN-3 specification itself. For example, the structural element "Control" represents the control part within a TTCN-3 module, while the structural element "Queues" represent the requirement on a TE that each port of a test component maintains its own port queue. While the first is specified within a TTCN-3 module the later is defined by the TTCN-3 specification. The TE corresponds typically to the executable code produced by a TTCN-3 compiler or a TTCN-3 interpreter. The TE can be executed centralized (on a single test device) or distributed (on different physical test devices). Although the structural entities of the TE implement a complete TTCN-3 module, single structural entities might be distributed over several test devices.

The TE implements a TTCN-3 module on an abstract level. The other entities of a TTCN-3 test system make these abstract concepts concrete. For example, the abstract concept of sending an event or receiving a timeout cannot be implemented within the TE. The platform adaptors of the test system realize the encoding of the message then its sending over concrete physical means, or measuring the time and determining when the timer has expired.

The *system adaptor* (SA) for the communication with the SUT and the *platform adaptor* (PA) for the realization of timers and their interaction with the TE are defined by the TRI (see part 5 of [2] and [8]). The interaction between the TE and the *test management and control* (TMC) entities covering functionality related to test execution management (TM), component handling (CH), and coding and decoding handling (CD) are defined by the TCI (see part 6 of [2] and [7]). The TRI and TCI operations are mainly atomic operations in the calling entity. The called entity, which implements a TRI/TCI operation, returns control to the calling entity as soon as its intended effect has been accomplished or if the operation cannot be completed successfully. The called entity is not blocked, so that performant test systems are enabled.

Both, TRI and TCI are defined by a set of abstract types, a set of operations at required and provided sub-interfaces, and a set of scenarios to show the use of abstract types and operations. The types and operations are provided in OMG IDL [15]. Language mappings to Java and C show the realization of TRI and TCI on specific test platforms.

6.1. The TTCN-3 runtime interfaces

The TRI SA adapts message and procedure based communication of the TTCN-3 test system with the SUT to the particular execution platform of the test system. It realizes the mapping of communication ports of TTCN-3 test components to test system interface ports. It is responsible to propagate send requests and SUT actions from the TTCN-3 executable (TE) to the SUT, and to notify the TE of any received test events by putting them into the respective port queue.

The PA realizes TTCN-3 external functions and provides a TTCN-3 test system with one notion of time. It enables the invocation of external functions, the starting, reading, and stopping of timers as well as the inquiring of their status. The PA notifies the TE of expired timers, i.e., timeouts.

6.2. The TTCN-3 control interfaces

The TCI *test management* (TM) entity is responsible for the overall management of a test system. After the test system has been initialised, test execution starts within the TM entity. The entity is responsible for the proper invocation of TTCN-3 modules, i.e., propagating module parameters to the TE if necessary. Typically, this entity would also implement a test system user interface. In addition, the TM entity performs test event logging and presentation to the test system user.

As the TE can be distributed among several test devices, the CH is responsible to implement the distribution and communication between the distributed entities. The CH provides the means to synchronize the different entities of the test system being potentially located on several nodes. The general structure of a test system distributed via several nodes is depicted in Fig. 23b.

On each node, a TE together with SA, PA and CD is performed. The entities CH and TM mediate the test management and test component handling between the TEs on each node. There is a special TE that is identified to be the TE that started a test case [5] and that is responsible for calculating the final verdict of that test case. Besides this, all TEs are handled the same.

The *coding/decoding* (CD) entity is responsible for the encoding and decoding of values into bitstrings suitable to be sent to the SUT. The TE determines which codecs shall be used and passes the TTCN-3 data to the appropriate codec in order to obtain the encoded data. Received data is decoded in the CD entity by using the appropriate decoder, which translates the received data into TTCN-3 values.

7. Putting TTCN-3 into practice

Even though TTCN-3 is a new language, it is already in use. In this section, we present first experiences with TTCN-3 and explain the motivation for moving from TTCN to TTCN-3.

7.1. Usage in standardization

For over 10 years, ETSI technical bodies have been producing conformance test specifications for leading-edge technologies such as GSM, UMTS, DECT, INAP, TETRA, B-ISDN, Hiperlan/2, VB-5, FSK and VoIP. The large majority of these test specifications are written in TTCN [4]. These test specifications play a key role in overall testing strategies to ensure interoperability of products based on ETSI technologies. TTCN-3 was developed by ETSI TC MTS (methods for testing and specification) in response to a demand from the ETSI membership for a modern, general purpose testing language based on the well-proven principles of TTCN.

Since the publication of TTCN-3, ETSI has produced two major test suites in this language. One for SIP (session initiation protocol, IETF RFC3261) and one for OSP (open settlement protocol, ETSI TS 101 321). Our experience is that the TTCN-3 fulfils its promise of application to new areas of testing and that its modern syntax

[5] Please note that in course of executing a TTCN-3 module there can be at most one test case being executed.

makes it more acceptable to the IP community. Features such as the trigger mechanism, interleaving, regular expressions, interface mapping, address types and powerful type and template matching mechanisms make TTCN-3 more adapted to expressing tests for protocols such as SIP and OSP than TTCN.

There is a significant investment in TTCN test suites both in industry and in standardization bodies such as the ITU-T and ETSI. The purpose of TTCN-3 is to expand the use of test languages into areas where it is currently not used. It is the policy that TTCN and TTCN-3 will exist in parallel for several years to come. [6] It is probable that technologies that have made extensive use of TTCN in the past (such as ISDN, GSM, Hiperlan/2, IN and DECT^{TM}) will continue to use TTCN in order to preserve that investment. However, as more TTCN-3 test systems are developed it is expected that new ETSI test specifications will be written in TTCN-3.

In the near future, ETSI is preparing to apply TTCN-3 for testing IPv6, Sigtran (signal translation) and SIP in UMTS (3GPP). There are no plans for a wholesale upgrade of existing TTCN specifications to TTCN-3, though this may be done on an as needed basis for individual test suites. The use of TTCN-3 is still at an early stage and success of the language will be closely linked to the availability of TTCN-3 tools and compilers. During the past year interest in the language has grown considerably, accompanied by a number of sophisticated TTCN-3 tools and test systems appearing on the market. ETSI sees this as a very promising development for the use of TTCN-3 in all kinds of testing standards.

7.2. Usage within Nokia

Nokia has been active in the development of the TTCN-3 from the very start, both in terms of participation in the language standardization and in developing the associated commercial tools. The past year has shown an increasing awareness of the importance of TTCN-3 within Nokia. This period has not only included the major milestone of the first product development tester using TTCN-3 but also witnessed major efforts to develop a definitive overall migration plan for the transition from TTCN to TTCN-3. At present most TTCN-3 work within Nokia is focused on improving existing protocol testing activities, however significant research has also been started into using TTCN-3 in a much broader scope.

7.3. Exploiting TTCN-3 in Ericsson

TTCN-3 is a very powerful and promising new language that has been tried out on different application areas and test methods. Most obviously the first application areas are those in which significant new developments are elaborated, like IP based networks and the UMTS domain or new test methods, like automated performance testing, are required.

Ericsson is using TTCN-3 for testing different protocols of IP routers and servers for more than two years. The first IPv6 test suites were written using the first edition of TTCN-3. Newer suites are created using TTCN-3 edition two and it has proven to be even more powerful and flexible than the previous edition [16]. In IP test suites, single test equipment is able to simulate a complex multiple-node network by means of multiple test components, which significantly decreases costs. Another huge benefit is that TTCN-3 tests are automated and need minimal human intervention only (to select test cases to be executed and analyze the reasons of failed or inconclusive test results). This approach was successfully applied in several interoperability test events. The logging functionality of TTCN-3 test tools eliminates the need for monitoring equipment and allows a high degree of documentation and unique style of test reports. It is also beneficial that the same test environment and language can be used for functional, conformance and performance testing. Ericsson has promising results in using TTCN-3 for performance tests and load generation.

Ericsson has been applying TTCN-3 for testing UMTS network elements for more than half year.

[6] A seamless migration from TTCN to TTCN-3 is possible by an automated translation to TTCN-3.

TTCN-3 has proven to be reliable and efficient even in executing the most complex test scenarios like inter-Mobile Switching Center handover between UMTS access networks or between UMTS and GSM. Technically UMTS is one of the most complex telecommunication systems. Both core network and access network nodes handle several protocols simultaneously, from which the access stratum (AS) ones, like RANAP, NBAP, RNSAP and RRC are especially complicated. These protocols are using ASN.1 information object definitions and are heavily parameterized. This puts very high requirements on TTCN-3 tools for their support of ASN.1. On the other hand, a sophisticated TTCN-3 test tool may significantly decrease costs and lead-time of the preparations as there is no need to *teach* message structures to the test tool and no need to update the tool internal representation of message structures when protocols are changing. The ASN.1 specifications of the protocols can directly be used for this purpose. TTCN-3 also has shown to provide higher degree of automation and reusability of test cases than several other test tools and test methods. This has a considerable effect in regression testing and in creating new test cases from existing ones.

8. Outlook

TTCN-3 is continually being maintained. ETSI provides a change request procedure and a mailing list. The change request procedure allows the collection of errors in the language, ambiguities in the standard and proposals for language extensions. An ETSI specialist task force (STF) takes care of all change requests and implements accepted change requests into the standard. The mailing list is a discussion forum, which allows asking general questions as well as discussing sophisticated language details. The latest news about TTCN-3, the status of change requests and information about the TTCN-3 mailing list can be found on the webpage http://www.etsi.org/ptcc/ptccttcn3.htm.

Further developments of TTCN-3 as a language include the real-time aspects. Some real-time aspects are already included in the language, such as timers. However, others, such as timestamp and logging mechanisms for an off-line evaluation after test execution, could be improved and may be subject to further versions of the language [17,18]. Another important activity is the harmonization of TTCN-3 with other important specification techniques, in particular in the UML context. Currently the OMG is in the process of developing a testing profile for UML 2.0, where TTCN-3 is an important input [19]. TTCN-3 has started off great interest and motivation to further development and refinement of tools for the language and its integration with other specification tools.

References

[1] ITU-T Recommendations Z.140-142 (2002): The Testing and Test Control Notation Version 3 (TTCN-3), Rec. Z.140: TTCN-3 Core Language, Rec. Z.141: Tabular Presentation Format for TTCN-3 (TFT), Rec. Z.142: Graphical Presentation Format for TTCN-3 (GFT). ITU-T, Geneva (Switzerland).

[2] ETSI European Standard (ES) 201 873 (2002/2003): The Testing and Test Control Notation Version 3 (TTCN-3), Part 1: TTCN-3 Core Language, Part 2: Tabular Presentation Format for TTCN-3 (TFT), Part 3: Graphical Presentation Format for TTCN-3 (GFT), Part 4: Operational Semantics, Part 5: The TTCN-3 Runtime Interface (TRI), Part 6: The TTCN-3 Control Interfaces (TCI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France).

[3] J. Grabowski, A. Wiles, C. Willcock, D. Hogrefe, On the design of the new testing language TTCN-3, in: H. Ural, R.L. Probert, G. von Bochmann (Eds.), Testing of Communicating Systems–Tools and Techniques, vol. 13, Kluwer Academic Publishers, Dordrecht, 2000.

[4] ITU-T Recommendation X.292 (1998): OSI Conformance Testing Methodology and Framework for Protocol Recommendations for ITU-T Applications—The Tree and Tabular Combined Notation (TTCN). ITU-T, Geneva (Switzerland).

[5] ISO/IEC International Multipart Standard 9646 (1994–1998): Information Technology—Open Systems Interconnection—Conformance Testing Methodology and Framework. International Organization for Standardization (ISO), Geneva (Switzerland).

[6] ETSI Technical Report (TR) 101 666 (1999-2005): Information Technology—Open Systems Interconnection Conformance Testing Methodology and Framework; The Tree and Tabular Combined Notation (TTCN) (Ed. 2++). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France).

[7] I. Schieferdecker, T. Vassiliou-Gioles, Realizing distributed TTCN-3 test systems with TCI, in: D. Hogrefe, A. Wiles, (Eds.), Testing of Communicating Systems—Next

Generation Testing for Next Generation Networks, vol. 15, Kluwer Academic Publishers, Dordrecht, 2003.

[8] S. Schulz, T. Vassiliou-Gioles, Implementation of TTCN-3 test systems using the TRI, in: I. Schieferdecker, H. König, A. Wolisz (Eds.), Testing of Communicating Systems—Application to Internet Technologies and Services, vol. 14, Kluwer Academic Publishers, Dordrecht, 2002.

[9] ITU-T Recommendation Z.120 (11/99): Message Sequence Chart (MSC). ITU-T, Geneva (Switzerland).

[10] P. Baker, E. Rudolph, I. Schieferdecker, Graphical test specification—the graphical format of TTCN-3, in: R. Reed, J. Reed (Eds.), SDL 2001—Meeting UML, Lecture Notes in Computer Science, Springer, London, 2001.

[11] I. Schieferdecker, J. Grabowski, The graphical format of TTCN-3 and its relation to UML and MSC, in: Proceedings of the 3rd International Workshop on SDL and MSC (SAM'2002). Aberystwyth, UK, June 2002.

[12] M. Ebner, A. Yin, M. Li, Definition and utilisation of OMG IDL to TTCN-3 mappings, in: I. Schieferdecker, H. König, A. Wolisz (Eds.), Testing of Communicating Systems—Application to Internet Technologies and Services, vol. 14, Kluwer Academic Publishers, Dordrecht, 2002.

[13] I. Schieferdecker, B. Stepien, Automated testing of XML/SOAP based web services, in: Proceedings of the 13th Fachkonferenz der Gesellschaft für Informatik (GI) Fachgruppe "Kommunikation in verteilten Systemen" (KiVS), Leipzig (Germany), 26–28 February 2003.

[14] ITU-T Recommendations X.680-683 (2002): Information Technology—Abstract Syntax Notation One (ASN.1) Rec. X.680: Specification of Basic Notation Rec. X.681: Information Object Specification Rec. X.682: Constraint Specification Rec. X.683: Parameterization of ASN.1 Specifications. ITU-T, Geneva (Switzerland).

[15] OMG CORBA v2.2: The Common Object Request Broker: Architecture and Specification, Section 3: Interface Definition Language (IDL). Object Management Group (OMG), 1998.

[16] J.Z. Szabó, Experiences of TTCN-3 test executor development, in: I. Schieferdecker, H. König, A. Wolisz (Eds.), Testing of Communicating Systems—Application to Internet Technologies and Services, vol. 14, Kluwer Academic Publishers, Dordrecht, 2002.

[17] Z.R. Dai, J. Grabowski, H. Neukirchen, Timed TTCN-3—a real-time extension for TTCN-3, in: I. Schieferdecker, H. König, A. Wolisz (Eds.), Testing of Communicating Systems—Application to Internet Technologies and Services, vol. 14, Kluwer Academic Publishers, Dordrecht, 2002.

[18] Z.R. Dai, J. Grabowski, H. Neukirchen, Timed TTCN-3 based graphical real-time test specification, in: D. Hogrefe, A. Wiles (Eds.), Testing of Communicating Systems—Next Generation Testing for Next Generation Networks, vol. 15, Kluwer Academic Publishers, Dordrecht, 2003.

[19] I. Schieferdecker, Z.R. Dai, J. Grabowski, A. Rennoch, The UML 2.0 testing profile and its relation to TTCN-3, in: D. Hogrefe, A. Wiles (Eds.), Testing of Communicating Systems—Next Generation Testing for Next Generation Networks, vol. 15, Kluwer Academic Publishers, Dordrecht, 2003.

Jens Grabowski graduated from the University of Hamburg with a diploma degree in Computer Science. He spent two years at SIEMENS AG in Munich focusing on protocol specification and protocol validation based on Petri Nets, SDL and MSC. During 1990–1995 he was research scientist at the University of Berne, where he received his Ph.D. in 1994. From 1995 to 2003, Jens Grabowski worked as researcher and lecturer at the Institute for Telematics at the University of Lübeck, where he received his habilitation in 2003. Since April 2003, he is associate professor (C3) at the Institute for Informatics at the University of Göttingen. Since 1995 he worked as expert in several ETSI standardization projects. He was a member of the ETSI experts team, which developed TTCN-3.

Dieter Hogrefe was born on 23 September 1958 in Göttingen, Germany. He graduated from Philips Exeter Academy, USA in 1976. He studied Computer Science and Mathematics at the University of Hannover, Germany, where he graduated with a diploma degree and later received his Ph.D. His research activities are directed towards Computer Networks and Software Engineering. He has published numerous papers and two books on analysis, simulation and testing of formally specified communication systems. From 1983 to 1986 he was with the SIEMENS research centre in Munich and worked in the area of analysis of telecommunication systems. He was responsible for the protocol simulation and analysis of the CCS No. 7. He had professorships at the Univerity of Hamburg, Bern, Lübeck and since 2002 he is full professor (C4) for Telematics at the University of Göttingen. Prof. Hogrefe represents the IITB (Fraunhofer Institute for information and data processing) in the European Telecommunication Standards Institute, ETSI, where he is chairman of the Technical Committee Methods for Testing and Specification.

György Réthy graduated from the Telecommunication University of Moscow with the degree M.Sc. in 1986. He continued his studies at the same University as a postgraduate student by researching the area of transmission quality of mixed analogue/digital telecommunication networks. He defended his Ph.D. and Candidate of Technical Sciences thesis in 1990. He joined the PKI Telecommunication Research Institute of the Hungarian Telecommunication Company (MATAV) and technically lead the introduction of ISDN in Hungary. From mid 1990s studied ATM and broadband ISDN networks and services and from 1998, he led the department responsible for the technical introduction of broadband services in MATAV's network. He joined Ericsson Hungary in 1999. His main focus is testing of telecommunication networks and network components. From 2001 he has been actively working in the field of TTCN-3 concentrating both on the language aspects, methods and usability aspects. From 1991 he has also been participating in different technical bodies of the European Telecommunications Standards Institute (ETSI) and from 1992 in the work of the Telecommunication Standardization Sector of the International Telecommunication Union (ITU-T). In particular, in 2002 he was member of the ETSI project team responsible for the development and maintenance of the TTCN-3 language.

Ina Schieferdecker is heading the Competence Center for Testing, Interoperability and Performance (TIP) of the Fraunhofer Institute for Open Communication Systems (FOKUS), lecturer at Technical University Berlin and co-founder of Testing Technologies IST GmbH, Germany. Her working areas include system specification and validation, test development, test execution and test tools for IT and Telco systems. She actively contributes to test technologies developed at ETSI, ITU and OMG such as TTCN-3 and the UML Testing Profile.

Anthony Wiles is originally from the UK. He received an M.Sc. in Physics and Computer Science from Uppsala University, Sweden in 1984. He is currently manager of the Protocol and Testing Competence Centre (PTCC) at the European Telecommunications Standards Institute in Sophia Antipolis, France. The centre supports ETSI technical bodies on the use of modern specification techniques such as SDL, UML, ASN.1 and TTCN in ETSI standards. Application areas include GSM, UMTS, Hiperlan/2, VoIP, DECT, IN, OSA and B-ISDN. The centre also manages the ETSI specialist task forces (STF) that produce the test specifications for these technologies.He has been involved in the development of TTCN since the language first originated in the late 1980s and was the ISO editor of the original TTCN language specification (ISO/IEC 9646-3). More recently, he was leader of the TTCN-3 development team since its start in 1999 and continues in that role as the work progresses in an improvement and maintenance phase. Other activities include the application of TTCN-3 to VoIP-related test specifications such as SIP, OSP and SIGTRAN. He is also leading an ETSI STF whose mandate includes a study of the use of TTCN-3 for testing IPv6.

Colin Willcock is a Research Manager at Nokia Research center. He received a B.Sc. from Sheffield University in 1986 an M.Sc. from Edinburgh University in 1987 and his Doctorate in parallel computation from the University of Kent in 1992. He is currently working on testing methodology, tool development and standardization. He is part of the core ETSI team which developed the TTCN-3 language and is currently participated in ETSI STF213 which is maintaining the language. In addition he is the Rapporteur for the TTCN-3 runtime interface (TRI) within ETSI MTS.