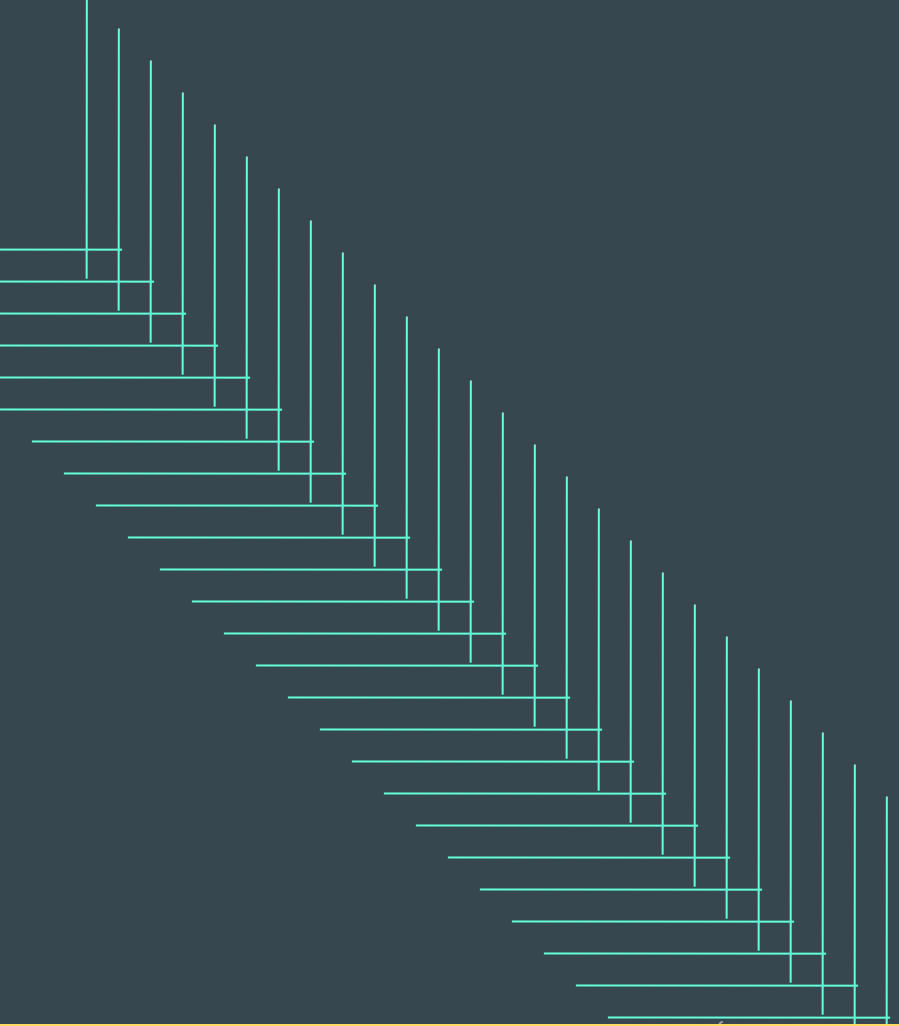


TELCOMA
AN ISO CERTIFIED COMPANY

TTCN 3

Full course at
https://telcomaglobal.com

TELCOMA

Overview

Copyright © TELCOMA. All Rights Reserved

What is TTCN 3 ?

Testing and Test Notation Version 3 (TTCN 3) is a strongly typed test scripting language.

It is developed and maintained by the TC-MTS at ETSI.

It is platform independent testing technology.

TTCN has grown into a global testing language used well beyond telecommunication and standardisation.

Copyright © TELCOMA. All Rights Reserved

TTCN 3:

Area of testing:

- Regression Testing
- Conformance Testing
- Functionality Testing
- Interoperability and integration testing
- Load / Stress Testing

Copyright © TELCOMA. All Rights Reserved

Development of TTCN 3:

TTCN 1:

Developed in 1992

Tree and Tabular Combined Notation

Published as an ISO standard

Used for protocol layer testing of OSI layer

Widely used for testing of telecommunication protocols.

Copyright © TELCOMA. All Rights Reserved

Development of TTCN 3:

TTCN 2:

Developed in 1998

Tree and Tabular Combined Notation version 2

Support Concurrent and multiple modularization.

Developed by ISO and ITU.

Copyright © TELCOMA. All Rights Reserved

Development of TTCN 3:

TTCN 3:

Developed in 2000

Testing and Test Control Notation version 3

Written by ETSI and standardized by ITU

More Generic Testing Language.

Widely used for conformance testing of communication systems.

TTCN 3 has its own data types and can be combined with ASN.1, IDL and XML type definition.

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Standards :

Part 1: “TTCN 3 Core Language”

Part 2: “TTCN 3 Tabular presentation Format (TFT)- not supported as of version 4.2.1”

Part 3: “TTCN 3 Graphical Presentation Format (GFT)”

Part 4: “TTCN 3 Operational Semantics”

Part 5: “TTCN 3 Runtime Interface (TRI)”

Part 6: “TTCN 3 Control Interface (TCI)”

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Standards :

Part 7: “Using ASN.1 with TTCN-3”

Part 8: “The IDL to TTCN-3 Mapping”

Part 9: “Using XML Schema with TTCN 3”

Part 10: “TTCN 3 Documentation Comment Specification”

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Core Language Versions :

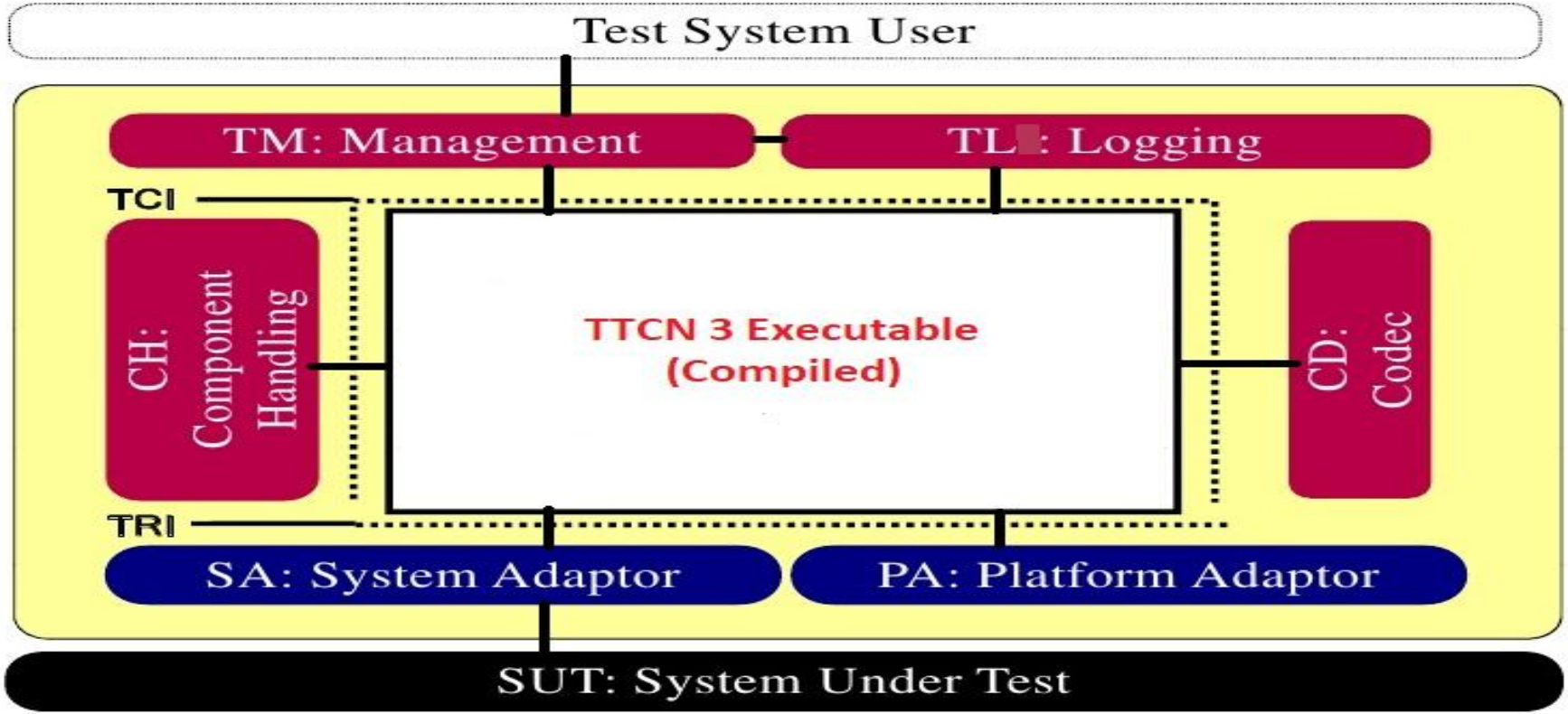
- TTCN-3: 2001 (v.1.1.2)
- TTCN-3: 2003 (v.2.2.1)
- TTCN-3: 2005 (v.3.1.1)
- TTCN-3: 2007 (v.3.2.1)
- TTCN-3: 2008 (v.3.3.2)
- TTCN-3: 2008 Amendment 1 (v.3.4.1)
- TTCN-3: 2009 (v.4.1.1)
- TTCN-3: 2010 (v.4.2.1)

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Architecture

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Test System Architecture:



TTCN 3 Test System Architecture:

TRI:

TTCN 3 Runtime Interface (TRI) is a standardized interface that helps to connect system adapter with the system under test (SUT).

TCI:

TTCN 3 Control Interface (TCI) is a standardized interface helps to connect external logger, test management and codecs.

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Test System Architecture:

TRI:

- Test Management (TM)
- Test Logging (TL)
- Coding and Decoding (CD)
- Component Handling (CH)

TCI:

- System Adapter (SA)
- Platform Adapter (PA)

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Test System Requirements:

TTCN 3 test system require:

- TTCN-3 test suite
- TTCN-3 tool plus execution environment.
- Codecs.
- SUT Adapter
- Platform Adapter

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Language Representation:

TTCN 3 language is represented in two forms:

1. Core Notation - Textual Format
2. Presentation format

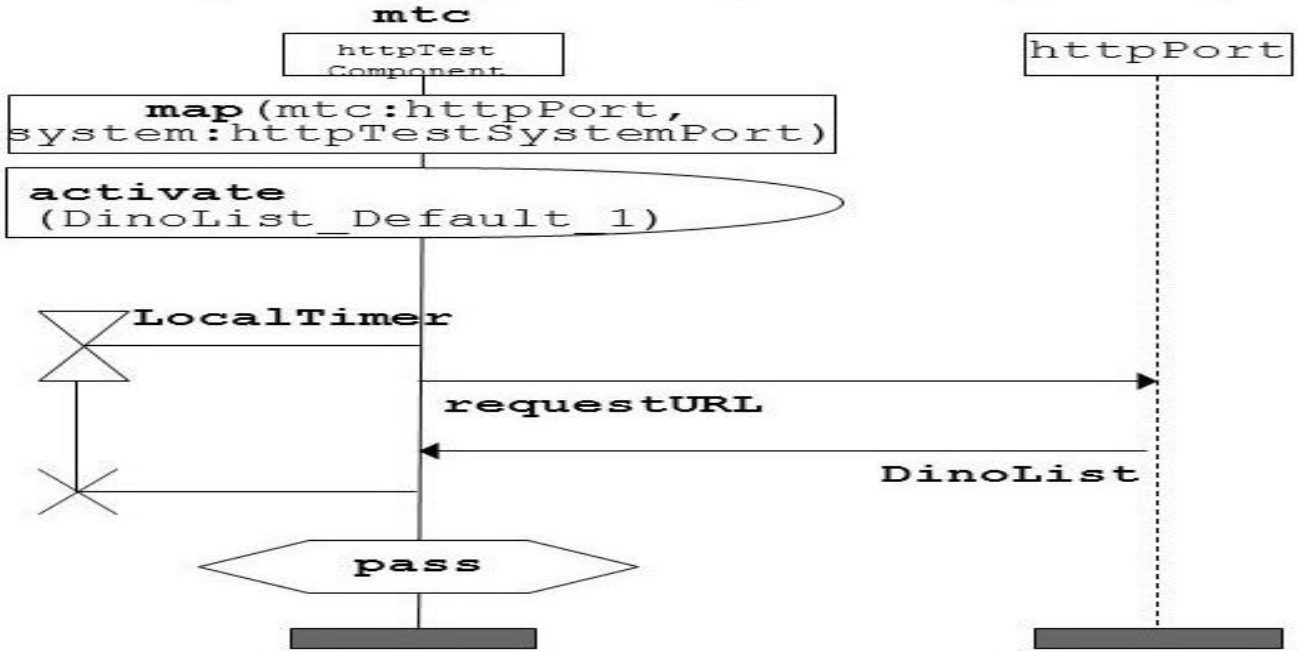
Further Presentation Format represent two forms:

1. Tabular Presentation Format
2. Graphical Presentation Format

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Language Representation:

```
testcase DinoList_Test_1
runs on httpTestComponent system httpTestSystemComponent
```



Application Area of TTCN 3:

TTCN 3 testing language is used in various large scale project

- Telecom System (ISDN, UMTS, GSM, ATM)
- Internet (IP and IP based applications & protocols)
- Software System (Java & XML)

Other application areas of TTCN 3 are automotive, railway and financial etc.

Copyright © TELCOMA. All Rights Reserved

TTCN 3
Modules

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Major Elements:

Data types: Built-in and user-defined Generic Data types (e.g. to define messages, service primitives, information elements, PDUs)

Test Data: Actual test data transmitted/ received during testing.

Testing Configuration: Definition of components and communication ports that are used to build various testing configuration.

Test Behaviour: Specification of the dynamic test system behaviour

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Structure:

TTCN 3 consist of **Module**:

- Module Definitions
- Module Control
- Attributes

Module Definition consist of:

- Data types
- Constants and Variables
- Templates
- Signature
- Component
- Ports
- Functions
- Alt Steps
- Test Cases

Copyright © TELCOMA. All Rights Reserved

Modules:

Modules are the building blocks of all TTCN 3 specifications.

A test suite is a module

Module consist of definition part and a control part.

Features of modules:

- Modules can be parameterised.
- Modules can import definitions from other modules.
- Modules also contain attributes.

Copyright © TELCOMA. All Rights Reserved

Modules:

```
/* File saved as "filename.ttcn" */

module TESTcase {
        // Module definition Part

    control {
        // Control Part
    }
}
```

Identifiers:

Identifiers are used to uniquely identify named entities in the codes.

TTCN-3 identifiers must consist of alphanumeric characters and may contain underscores.

Identifiers always start with a letter and are case-sensitive.

Example: test_case, TEST_case etc.

Copyright © TELCOMA. All Rights Reserved

Scope:

Scope in TTCN 3 is defined as code blocks enclosed in curly brackets.

The outermost and top-level scope is the actual module.

The blocks of code can contain new individual code statements or new nested blocks.

Scoping is used to control the visibility of particular language statements.

There are nine basic scope units.

Copyright © TELCOMA. All Rights Reserved

TTCN 3
Constants, Variable , Comments & Data Types

Copyright © TELCOMA. All Rights Reserved

Constants:

Constants are defined with "*const*" keyword and a value.

It is defined in the definitions part of a module and inside the test functions and test components.

The value of the constant must be assigned at the point of declaration and is not allowed to change after the assignment.

In the example, the integer constant cmax_number define the number of names that shall be resolved.

Copyright © TELCOMA. All Rights Reserved

Constants:

```
/* File: testcase2.ttcn */

module TEST_case {
    const integer cmax_number    := 10;

    control {
        var charstring v_IP;
        var integer v_no          : 0;
        var integer v_No          : cmax_number;

        const integer c_result    : cmax_number / 2;
    }
}
```

Variables:

Variables are defined with “*var*” keyword

It can be defined at any scope level except at the top module level.

Variables are used to save temporary values at run time during program execution.

When a variable is declared, it can be initialised with a value of the appropriate type.

Variables can be used only within control, function, altstep and testcase component type definition.

Copyright © TELCOMA. All Rights Reserved

Variables:

```
/* File: testcase2.ttcn */

module TEST_case {
    const integer cmax_number    := 10;

    control {
        var charstring v_IP;
        var integer v_no          : 0;
        var integer v_No          : cmax_number;

        const integer c_result    : cmax_number / 2;
    }
}
```

Comments:

In TTCN-3 comments contain graphical character defined in ISO/IEC 10646.

The language offers both line and block comments.

A block comment starts with /*, can extend over several lines and ends with */.

A line comment starts with // and extends to the end of the line and ends with //.

Copyright © TELCOMA. All Rights Reserved

Comments:

```
/* File saved as "filename.ttcn" */          Block Comment

module TESTcase {
        // Module definition Part            Line Comment

    control {
        // Control Part
    }
}
```

Data Types:

TTCN-3 is a typed language with a large number of built-in types.

The Data types are:

1. Integer
2. Boolean
3. charstring.

Copyright © TELCOMA. All Rights Reserved

Data types:

```
module TEST_case {
    const integer cmax_number    := 10;

    Control  {
        var integer v_no;
                    v_no := 0;
        var boolean v_resolved := false;
        var charstring v_IP := "192.168.14.110";
    }
}
```

Data type:

1. **Integer**:

   Integer data types in the program represents the numerical value. Values of type integer can be positive whole number or negative whole numbers, including zero.

2. **Boolean**:

   The type boolean consists of the two distinguished values TRUE and FALSE.
   The variable of type boolean is used to handle conditional operations.

Copyright © TELCOMA. All Rights Reserved

Data type:

3. **Charstring**:

   The data type charstring is used to represents a sequence of ASCII characters.
   Values of charstring are denoted by an arbitrary number of (printable) characters preceded and followed by double quotes.

Copyright © TELCOMA. All Rights Reserved

Functions

Copyright © TELCOMA. All Rights Reserved

Functions:

Functions are defined in the module definitions part

It is defined using "*function*" keyword,

It is consist of a unique name, a parameter list, an optional return value and the function body.

A function body may contain local constant and variable definitions and statements to express behaviour.

Copyright © TELCOMA. All Rights Reserved

Functions:

```
function f1() runs on sys1 {
    alt {
        [] port1.receive(s1)
            { setverdict (pass, "Hello World:", s1);
            }
        [] port1.receive
            {setverdict (fail, "cannot connect:",s1);
            }
        }
}
```

Functions:

The function body also contain the return statement.

For example

```
function f1 () return integer {
return 0; }
```

Functions may also be defined externally by using the "*external*" keyword in front of the function prototype.

Copyright © TELCOMA. All Rights Reserved

Functions:

Parameters passing modes in function:

1. In mode: signifies value parameter (value taken from the caller to the called entity)
2. Inout mode: signifies reference parameter (value is taken from the caller to th called entity and value is returned from the called entity to the caller)
3. Out mode: signifies reference parameter (value returned from the called entity to the caller)

Copyright © TELCOMA. All Rights Reserved

Functions:

For example:

```
function f_myFunc (
        In integer a,
        Inout integer seq,
        Out charstring result ).
```

Copyright © TELCOMA. All Rights Reserved

Pre-Defined Functions:

Pre-defined functions have an extensive set of “value conversion functions”.

for example it is used to convert an integer to a character, string handling functions, length and size functions, codec functions, as well as some other special functions,

For example:

```
var integer v_i4  := float2int(4.5);     // i4== 4
var float v_f4 := int2float(3);          //f4== 3.0
```

Copyright © TELCOMA. All Rights Reserved

Statement
Part 1

Copyright © TELCOMA. All Rights Reserved

Statements:

1. Operator, Expression and assignment
2. Conditional statement
3. Loops statement
4. Label statement
5. GoTo statement
6. Log statement
7. Control Part
8. Pre-Processing Macros

Copyright © TELCOMA. All Rights Reserved

Operators:

In TTCN 3, operator are used to define expression by combining data.

The operators are divided into the categories:

- Arithmetic (+, -, *, /, mod, rem)
- Relational (==, <, >, != , >=, <=)
- Logical (not and, or, xor)
- binary string (not4b, and4b, xor4b, or4b)
- string (&, <<, >>, <@, @>)

For example:

+op or op1 + op2

op1 > op2

Copyright © TELCOMA. All Rights Reserved

Operators:

Priority	Operator type	Operator
Highest	Unary	-, +
	Binary	*, /, mod, rem
	Binary	+ , -, &
	Unary	not4b
	Binary	and4b
	Binary	xor4b
	Binary	or4b
	Binary	<<, >>, <@, @>

Copyright © TELCOMA. All Rights Reserved

Operators:

Priority	Operator type	Operator
	Binary	<, >, <=, >=
	Binary	==, !=
	Unary	not
	Binary	and
	Binary	xor
	Binary	or
Lowest		

Copyright © TELCOMA. All Rights Reserved

Expressions:

Expressions are created by grouping of various operators and operands acc to priority rules.

Grouping can be achieved using parentheses.

These operations are evaluated from left to right, acc to priority of operators.

For example:

*2*f1(v1,v2)+1*

x+y<z

Copyright © TELCOMA. All Rights Reserved

Assignment:

Variables are updated by assignments using this “ := “ operation.

While execution an assignment, the RHS of the assignment must evaluate to a value, which is of a similar type to the LHS.

After the evaluation of an assignment statement, the variable stores the result of expression on the RHS.

For example:

LHS := RHS

V := 5 (!= is used for a value “not equal to”)

Copyright © TELCOMA. All Rights Reserved

Conditional Statement:

The Conditional statement in the TTCN-3 language is represented by:

1. The if-else statement
2. The select-case statement

Syntax of if-else statement:

```
If (condition)
        {statement 1}
[ else {statement 2}]
```

Copyright © TELCOMA. All Rights Reserved

Conditional Statement:

Syntax for Select-case statement:

```
Select (expression) {
        case (template) {statement 1}
    [ case (template-list) {statement 2} ]
    [ case else {statement 3} ]
    }
```

Copyright © TELCOMA. All Rights Reserved

Loop:

In TTCN 3 the Iterative or Repetitive behaviour can be constructed by using these three different loop constructs:

1. The for statement
2. The do-while statement
3. The while statement.

In addition to these, TTCN-3 also offers break and continue statements

Copyright © TELCOMA. All Rights Reserved

Loop:

Syntax: for loop statement:

```
for (init; condition; expression)
    {statement 1}
```

Syntax: do-while loop statement:

```
do
    {statement}
    while (condition);
```

Copyright © TELCOMA. All Rights Reserved

Loop:

Syntax: while loop statement:

```
while (condition)
        {statement};
```

Copyright © TELCOMA. All Rights Reserved

Loop:

In TTCN 3, there are two statements break; and continue;

These two statements are used to alter the normal flow of a program.

The break; statement terminates a loop (for, while and do..while loop) and a select statement immediately when it appears.

The continue statement is used to skip a certain test condition within a loop.

Copyright © TELCOMA. All Rights Reserved

Statement
Part 2

Copyright © TELCOMA. All Rights Reserved

Label & Goto:

TTCN-3 define the label and goto mechanism which help in jumping from one part of the program to another part.

The label statement defines a unique label in a logical statement block.

The goto statement allows the execution to jump directly to the position of that label in the same statement block

Syntax:

Label <labelname>;

Goto <labelname>;

Copyright © TELCOMA. All Rights Reserved

Label & Goto:

To prevent the abuse of this mechanism, there exist a few restrictions on the use of the goto statement:

- Jumping out of or into functions, test cases, or the control part is not allowed.
- It is similarly forbidden to jump into a loop or into an if-else statement.

Copyright © TELCOMA. All Rights Reserved

Log:

The log statement provides the means to write logging information to the logging interface of a test system.

The format of the logged values is dependent on the logging interface implementation used with the test system.

Syntax:

```
log (a);
log (a,...);
log (“a=”,a);
```

Copyright © TELCOMA. All Rights Reserved

Control Part:

The control part of a module is the entry point for execution in a TTCN-3 program.

The control part contains an many control statements and function calls that reflect the dynamic behaviour of the test system.

The main role of the control part is to control and sequence the execution of test cases.

Copyright © TELCOMA. All Rights Reserved

Control Part:

Syntax:

```
module test_case {
        function f1 {
        }
        control {
        }
}
```

Copyright © TELCOMA. All Rights Reserved

Pre-Processing Macros

Copyright © TELCOMA. All Rights Reserved

Pre-Processing Macros:

The TTCN-3 core language there are number of predefined precompiler-like macros which are named as Preprocessing Macros.

These macros are used in definitions or the control part .

Each macro starts and ends with a '_'-character and between these characters the name of the macro is stated.

There are four different types of definition IDs in TTCN 3:

Copyright © TELCOMA. All Rights Reserved

Pre-Processing Macros:

1. The Module-Name Macro _MODULE_

In TTCN 3, if the compiler passes the module-name macro then it is replaced by the name of the TTCN-3 module it was found in.

The name is inserted into the source code as a charstring value.

In a module named ‘U3Tester’ the statement:

log(_MODULE_); will evaluate to: *log("U3Tester");*

Copyright © TELCOMA. All Rights Reserved

Pre-Processing Macros:

2. The File-Name Macro _FILE_

The _FILE_ macro is replaced with a charstring containing the absolute file name of the source file it has been found in including the path.

For example,

log(_FILE_); will evaluate to:

log("/root/LTE_tester/Conn_Tests/10_06_19/Test1.ttcn");

Copyright © TELCOMA. All Rights Reserved

Pre-Processing Macros:

3. The File-Name Macro _BFILE_

Here the compiler replaces the macro with the file name of the source file without its path, i.e. the basic filename.

The exact format of the filename is dependent on the compiler implementation.

For Example:

log(_BFILE_); will evaluate to: *log("Test1.ttcn");*

Copyright © TELCOMA. All Rights Reserved

Pre-Processing Macros:

4. The Line Macro _LINE_

The compiler will exchange this macro with the integer value of the current line number in the file.

A file starts with line number 1. Each new line, including commented lines increase the line number by 1.

Copyright © TELCOMA. All Rights Reserved

Pre-Processing Macros:

5. The Scope Macro _SCOPE_

The scope macro _SCOPE_ is of two type:

An unnamed scope is simply a statement block between a pair of curly brackets: { . . . }.

A named scope is any kind of function, a control part or a module definitions part.

Copyright © TELCOMA. All Rights Reserved

Single Component

Copyright © TELCOMA. All Rights Reserved

Ports
&
Components

Copyright © TELCOMA. All Rights Reserved

Ports:

In TTCN-3, Ports are used to send and receive messages

The messages sent via a port are delivered without delay to the destination.

Messages received at a port are stored in a message queue.

A port type defines which messages can be sent through this port and which messages can be received.

Ports can be bidirectional, which is used to sent and received messages.

Copyright © TELCOMA. All Rights Reserved

Ports:

Syntax:

```
type port m1 message {
        out send                // test system to SUT
        };
type port m2 message {
        in recieve              // SUT to test system
        };
type port m3 message {
        !out sendrecieve        //bidirectional port
        };
```

Copyright © TELCOMA. All Rights Reserved

Component:

Components are used to execute the test behaviour of a test systems.

Each component can have its own local state, which consists of constants, variables and timers.

Component type is defined with the ports of the corresponding component instances by indicating their name and type.

Copyright © TELCOMA. All Rights Reserved

Components:

Syntax:

```
type component  TESTsys1 {
    const   integer         c_maxAmount := 800;
    var     Receipt         v_rec;
    timer                   t_inactive := 6.0;

    port  Request     pt_request;
    port  Receipt     pt_rec;
    port  Amount      pt_amt
};
```

Copyright © TELCOMA. All Rights Reserved

Templates
&
Timer

Copyright © TELCOMA. All Rights Reserved

Templates:

A template defines one or more values of a specific type.

Template are defined by three ways in TTCN-3:

1. define a template as a single value.
2. define a template that contain values of all type. (it is defined by using the wildcard character '?' standing for any value.)
3. define a template consisting of several specific values, by enumerating them in the template definition as a *value list* .

Copyright © TELCOMA. All Rights Reserved

Templates:

Syntax:

```
// Single value template
    template  Receipt     a_recA := "A";
    template  Money       a_mon10 := 10;
    template  Money       a_mon20 := 20;
    template  Money       a_mon50 := 50;
    template  Money       a_mon100 := 100;
    template  Money       a_mon200 := 200;
// any value template
    template  Money       a_monAny := ?;
```

Copyright © TELCOMA. All Rights Reserved

Templates:

Syntax:

```
// value list template
    template  Money     a_smallmon := ( 10, 20, 50 );
    template  Money     a_allmon := ( 10, 20, 50, 100, 200 );
```

Copyright © TELCOMA. All Rights Reserved

Templates:

Templates can be passed as *in* parameters to functions, test cases and so on using additional keyword *“in template”*.

Syntax:

```
testcase t_purchase ( in template Receipt p_rec ) runs on TESTsys1 {
        // the test case body
}
```

Copyright © TELCOMA. All Rights Reserved

Timer:

In TTCN-3 ,Timer is used to describe the timing properties.

Timers are started with arbitrary durations and the execution of code is blocked after a timer expires.

One of the most common uses of timers in TTCN-3 is to guard against the inactivity of the SUT.

To achieve such a guard, the handling of the responses from the SUT are combined with the possible timing out of an inactivity timer.

Copyright © TELCOMA. All Rights Reserved

Timer:

In TTCN-3, durations for timers are given as non-negative float values with a unit of time as seconds (example: duration of 1 ms is given as 0.001)

Timers can be declared in the type definition of a component, in a test case, functions, altstep or in the control part of a module.

A timer is started by the start operation with the timer duration as an optional parameter.

Copyright © TELCOMA. All Rights Reserved

Timer:

Syntax:

```
testcase t_purchaseA () runs on TESTsys1 {
        timer   t_receipt;
        timer   t_mon := 5.0;

        // request the receipt
                pt_request.send ( a_receiptA );
                t_receipt.start ( 2.0 );
                t_receipt.timeout;
```

.....Continued

Copyright © TELCOMA. All Rights Reserved

Timer:

Syntax:

```
// pay for receipt
        pt_mon.send( a_mon50 );
        t_mon.start;
        t_mon.timeout;
        pt_mon.send( a_mon100 );

        pt_receipt.receive( a_receiptA );

        setverdict( pass );
};
```

Copyright © TELCOMA. All Rights Reserved

Test Cases
Part 1

Copyright © TELCOMA. All Rights Reserved

Test Cases:

A test case is a behaviour description.

It is used to describe that how to stimulate the SUT and the expected reactions of the SUT to the stimulator.

“Verdict” are assigned according to the reaction.

For example, a test case use passed or failed verdict used to describe the reaction of SUT.

Copyright © TELCOMA. All Rights Reserved

Main Test Case:

In TTCN-3, test case defines the behaviour of the main test component.

The interface between the test system and the SUT in single test component configuration is TSI (test system interface) (TSI)

TSI is completely defined by the ports of the main test component.

“runs on” clause in the test case defines the component type on which the test behaviour is executed

Copyright © TELCOMA. All Rights Reserved

Main Test Case:

Syntax:

Test case with a empty behaviour

```
testcase t_empty () runs on TESTsys1 { };
```

Copyright © TELCOMA. All Rights Reserved

Test Verdict:

Verdicttype is a type of variable which is implicitly defined in each test component

This implicit variable called the local verdict of a test component.

The setverdict is used to set the operation of local verdict.

The getverdict is used to retrieve the operation of local verdict and result of this verdict can also be logged with the log statement.

The verdicts are ordered from none to pass, corresponding to the following relation: none > pass > inconc > fail > error.

Copyright © TELCOMA. All Rights Reserved

Test Verdict:

Syntax:

Test case with a pass local verdict:

```
testcase t_pass () runs on TESTsys1 {
    setverdict( pass );
    };
```

Copyright © TELCOMA. All Rights Reserved

Test Verdict:

Syntax:

Test case with a fail local verdict:

```
testcase t_fail () runs on TESTsys1 {
        var verdicttype v := getverdict;            // v == none
                setverdict( fail );
        v := getverdict;                            // v == fail
                setverdict( pass );
        v := getverdict;                            // v == fail
        log("The initial MTC verdict is: ", getverdict ); // logging via operation
};
```

Copyright © TELCOMA. All Rights Reserved

Test Cases
Part 2

Copyright © TELCOMA. All Rights Reserved

Test Case Innovation:

In TTCN-3, test cases are invoked explicitly from the control part.

A test case is invoked with the "*execute*" operation.

The return value of the execute operation is the overall test case verdict.

It is not necessary to store the result of the execute operation in a variable.
It is also possible to just execute a test case and discard the return value.

Copyright © TELCOMA. All Rights Reserved

Test Case Innovation:

Syntax: invoking test case without return value

```
control {
        var verdicttype v;
        v := execute ( t_empty () );        // v == none
        v := execute ( t_fail () );              // v == fail
        v := execute ( t_pass () );              // v == pass

        execute( t_empty () );
};
```

Copyright © TELCOMA. All Rights Reserved

Test Case Parameters:

As function have parameters similarly test cases also have parameters.

These parameters are in, out and inout.

The values for in parameters are passed by value and the values for out and inout parameters are passed by reference.

Parameters of test cases are used to exchange data between subsequently invoked test cases.

Copyright © TELCOMA. All Rights Reserved

Test Case Parameters:

Syntax:

```
testcase   t_parameter     ( in integer p_1,
                             out integer p_2,
                              inout integer p_3 )
```

Copyright © TELCOMA. All Rights Reserved

Test Case Termination:

The execution of a test case terminates when its last statement has been executed.

The execution of a setverdict operation in test case does not terminate the execution of a test case.

The stop operation allows a test case to be terminated at any point in its execution.

As a result of stopping the test case, the overall verdict is automatically returned to the control part.

Copyright © TELCOMA. All Rights Reserved

Test Case Termination:

The execution of a test case may come to a state, where it is considered as erroneous and from which it does not make sense to continue the computation.

The operation *testcase.stop* sets the verdict to error and terminates test case execution in a single operation.

Copyright © TELCOMA. All Rights Reserved

Message Based Communication

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

Messages are exchange between the test system and SUT to test the SUT in TTCN-3,

The two most important operations are used for this testing:

1. *send* operation
2. *receive* operation.

The *send* operation is used to send a message to the SUT. The *receive* operation is used to compares a received message against a template.

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

Send Message:

The *send* operation transmits a message to the SUT via the specified port.

The message is given by a template, which has to define a unique value.

As the message is delivered to SUT, the send statement is executed successfully and then further execution proceeds.

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

Syntax:

```
testcase t_purchaseA () runs on TESTsys1 {
    // request the receipt
        pt_request.send( a_receiptA );
    // pay the receipt
        pt_money.send( a_mon50 );
        pt_money.send( a_mon100 );
    // continued
        setverdict( pass );
    };
```

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

***Receive* Message:**

Messages are received by using the receive operation.

This operation different from the send operation both syntactically and semantically:

1. It can have a template as its parameter that describes more than a simple unique value.
2. The receive operation is a blocking operation. The receive operation compares the message at the head of the message queue of the indicated port with its parameter.

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

Syntax:

```
testcase t_purchaseA () runs on TESTsys1 {
    // request the receipt
        pt_request.send( a_receiptA );
    // pay the receipt
        pt_money.send( a_mon50 );
        pt_money.send( a_mon100 );
        pt_receipt.receive( a_receiptA );

        setverdict( pass );
    };
```

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

***Check* Operation:**

Check operation is used to match the templates with the messages stored in queue.

If the first message in a port queue matches the template of a receive operation, then that message is removed from the queue.

This operation allows the inspection of the head of the message queue associated with a port without removing it.

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

***Check* Operation:**

The check operation will block if there is no message in the queue or when the message at the head of the message queue does not match.

A check operation is written as an operation on a port with the receive statement and its parameters.

If the head of the message queue matches the template in the receive statement, the check statement is said to be successfully executed.

If the message does not match, then the check statement will block.

Copyright © TELCOMA. All Rights Reserved

Message Based communication:

Syntax:

```
pt_money.check( receive ) ;
pt_money.check( receive ( a_mon50 ) );
pt_money.check( receive ( a_monAny ) -> value  v_returnedMoney );
```

Copyright © TELCOMA. All Rights Reserved

Alt Statement and Alt-step

Copyright © TELCOMA. All Rights Reserved

Alt Statement:

The Alt statement expresses sets of possible alternatives that form a tree of possible execution paths.

Alt statement specifies alternative behaviour.

It is related to the use of the TTCN-3 operations receive, trigger, getcall, getreply, catch, check, timeout, done and killed.

The alt statement is used in places where several blocking events (alternatives) can occur.

Copyright © TELCOMA. All Rights Reserved

Alt Statement:

Syntax:

```
testcase t_purchaseA () runs
on TESTsys1 {
    timer t_guard;
//continued
    t_guard.start( 30.0 );
    alt {
        [] pt_receipt.receive(
a_receiptA ) {
            T_guard.stop;
            setverdict( pass )
            };
        [] t_guard.timeout {
        setverdict( fail ) }
            }
    };
```

Copyright © TELCOMA. All Rights Reserved

Alt Statement:

Boolean Guard:

Boolean Guard statement is an optional boolean expression enclosed in []. If the expression is false the alternative cannot be activated.

The alt statement can contain special guard :[else] . An else branch always chosen when none of the preceding alternatives is selected.

Copyright © TELCOMA. All Rights Reserved

Alt Statement:

Syntax:

```
alt {
    [x > 2] pt_p.receive( a_m1 )
        { setverdict( pass ) };
    [x < 0] pt_p.receive( a_m2 )
        { setverdict( pass ) };
    [else]
        {setverdict( fail ) }
    };
```

Copyright © TELCOMA. All Rights Reserved

Alt Statement:

Repeat statement:

The repeat statement is used for a re-evaluation of an alt statement.

A repeat statement is used within the alternatives of either an alt statement or within the alternatives of an altstep

Copyright © TELCOMA. All Rights Reserved

Alt Statement:

Syntax:

```
alt {
    [] pt_cash.receive( a_coinAny ) -> value v_retnC
        { v_retnCA := v_retnCA + v_retnC;
    if ( v_returnedCA == 50 )
        {setverdict( pass)}       //correct amount of money returned
    else if ( v_retnCA > 50 )
        {setverdict( fail) } //too much money returned
    else {repeat}                 // wait for more cash
    } }
```

Copyright © TELCOMA. All Rights Reserved

Alt Steps:

TTCN-3 uses altsteps to specify default behaviour or to structure the alternatives of an alt statement.

To structure alternative behaviour to re-usable package

Syntax:

```
altstep alt_timeGuard ( inout timer p_t ) {
    [] p_t.timeout
        { setverdict( fail ) }
    };
```

Copyright © TELCOMA. All Rights Reserved

Multi-Component

Copyright © TELCOMA. All Rights Reserved

Difference

Copyright © TELCOMA. All Rights Reserved

Difference:

1. **Sequential /concurrent behaviour:**

   In the single component case, the main test component (MTC) is the only component. Therefore, the behaviour of the whole test case is sequential.

   In multi components case,the behaviour can be executed sequentially as well as concurrently.

Copyright © TELCOMA. All Rights Reserved

Difference:

2. **Combination of verdicts:**

   In the single component case, the verdict of the MTC becomes the overall verdict of the test case.

   In the multi component case, the local verdicts of all the test components contribute to the overall verdict of the test case.

Copyright © TELCOMA. All Rights Reserved

Difference:

3. **Explicit TSI:**

   In the single component case, it is optional to explicitly define the test system interface

   In the multi component case, it is compulsory to define a test system interface explicitly and to map the ports of test components to ports of the test system interface.

Copyright © TELCOMA. All Rights Reserved

Difference:

4. **Dynamic configurations:**

   In the single component case, there is one test component throughout a test case.

   In the multi component case, parallel test components can be created and terminated throughout the test case execution.

Copyright © TELCOMA. All Rights Reserved

Difference:

5. **Test component execution:**

   Test components are restricted to execute only one single behaviour by default.

   The creation of alive components enables the execution of multiple behaviours sequentially.

Copyright © TELCOMA. All Rights Reserved

Difference:

6. **Data sharing:**

   On a single test component, data can be passed as parameters or the component variables can be used to pass data from one part of the code to another.

   Data shared by several test components has to be exchanged by explicitly passing it around in messages

Copyright © TELCOMA. All Rights Reserved

Test Components
Part 1

Copyright © TELCOMA. All Rights Reserved

MTC & TSI:

A multi component TTCN-3 test case starts with the execution of MTC behaviour as in the single component case.

The type of this MTC is specified by the “*runs on*” clause of the test case definition.

The type of the TSI is indicated by the system clause.

Copyright © TELCOMA. All Rights Reserved

MTC & TSI:

Syntax:

```
testcase TEST_sys2 () runs on ComponentA system
DNSTestSystemInterface {
        // ...
}
```

Copyright © TELCOMA. All Rights Reserved

Parallel Test Component:

Parallel testing is used by automated systems for simultaneously testing multiple applications or components.

The combination of automation and multiple test systems makes it possible to run many more tests than the serial testing.

Executing Parallel test component in test case also reduces the time required for testing to a fraction of that required for the equivalent serial tests.

Copyright © TELCOMA. All Rights Reserved

Parallel Test Component:

Syntax:

```
type component TESTEntity {
    port TESTPort pt;
    }
```

Copyright © TELCOMA. All Rights Reserved

Test Component Creation:

The MTC is created implicitly when a test case starts its execution in TTCN-3.

Parallel test components need to be created explicitly first by the MTC but they may in turn also create other parallel test components.

The parallel test components are created by using the "*create"* operation with a component type .

Copyright © TELCOMA. All Rights Reserved

Test Component Creation:

Syntax:

```
testcase TEST_sys2 () runs on ComponentA system
DNSTestSystemInterface {
        var DNSEntity     v_client;
        var DNSEntity     v_root;
        var DNSEntity     v_remote;
            v_client :=   DNSEntity.create("Client");
            v_root :=     DNSEntity.create("Root");
            v_remote :=   DNSEntity.create("Remote");
        // ...
}
```

Copyright © TELCOMA. All Rights Reserved

Alive Test Component:

Alive Component reuse the component ports while starting new the behaviour.

It also preserve the state of most of the components.

Alive components are used in the same manner as regular parallel test components, it is safe to always use the “*alive”* keyword in component creation.

Copyright © TELCOMA. All Rights Reserved

Alive Test Components:

Syntax:

```
testcase TEST_sys2 () runs on ComponentA system
DNSTestSystemInterface {
        var DNSEntity     v_client       := DNSEntity.create("Client") alive;
        var DNSEntity     v_root         := DNSEntity.create("Root") alive;
        var DNSEntity     v_remote       := DNSEntity.create("Remote") alive;
        // ...
}
```

Copyright © TELCOMA. All Rights Reserved

Test Components Part 2

Copyright © TELCOMA. All Rights Reserved

Component Reference:

Component references refer to an instance of a component type, that is to the MTC or to a parallel test component.

pre-defined operations for component reference:

- *mtc*: returns a reference to the MTC of a test case.
- *system*: returns a reference to the test system interface. This reference is needed when mapping ports of parallel test components to the test system interface.

Copyright © TELCOMA. All Rights Reserved

Component Reference:

- *self*: returns a reference to the test component on which this operation is executed.
- *null* : it is used to initialise variables for component references.

Copyright © TELCOMA. All Rights Reserved

Starting Parallel Test Components:

Syntax:

```
// start the behaviour on the parallel test components
v_client.start( f_client ( c_clientQuestion, c_clientAnswer,
c_identification ) );
v_root.start ( f_server ( c_rootQuestion, c_rootAnswer ) );
v_remote.start( f_server ( c_clientQuestion, c_clientAnswer ) );
// ...
}
```

Copyright © TELCOMA. All Rights Reserved

Stopping Parallel Test Component:

A component can also be stopped explicitly using the *“stop”* operation.

The stop operation can be called without a qualifying component reference.

The test component instance on which this stop statement is executed terminates its behaviour.

- *client.stop* : used to stop the behaviour of the test component client.
- *component.stop* : used to stop all parallel test components at once.

Copyright © TELCOMA. All Rights Reserved

Stopping Parallel Test Component:

- *mtc.stop*: used to stop the MTC which terminates all test cases.
- *testcase.stop*: used to terminate test case and overall verdict is set to error
- *kill*: operation is used to remove the test cases with alive test components.

The difference in the effect of stop and kill operations is that a stopped component can be restarted, whereas, a killed component can no longer be restarted.

Copyright © TELCOMA. All Rights Reserved

Await Termination of Test Component:

The done operation resembles the timeout operation for timers.

It can be used to wait until a component has terminated.

- *vclient.done*: block other operation until the component client has terminated its behaviour.
- *All component.done*: used to wait until all parallel test components have terminated execution.
- *Any component.done*: used to wait for any parallel test component to terminate.
- The done operation can be used with *receive* statements in alt statements and altsteps.

Copyright © TELCOMA. All Rights Reserved

Checking Execution Status:

The *running* operation can be used to check whether a component is currently executing the behaviour.

This operation is non blocking and returns a Boolean value indicating the status of a test component.

True value is returned when the component is already started to execute behaviour and not yet terminated.

Copyright © TELCOMA. All Rights Reserved

Checking Execution Status:

false value is returned if the component has been created already but not yet started to execute behaviour or if the component has already terminated.

For example: any component.running, all component.running

Copyright © TELCOMA. All Rights Reserved

Verdict Computation:

In multi component test case, if one of the parallel test components has a verdict fail whereas all the other PTC and the MTC have the verdict pass, then the overall verdict of the test case will be fail.

Here the MTC has its own local verdict.

A test case can result in a fail verdict even when the verdict of the MTC is pass.

Copyright © TELCOMA. All Rights Reserved

Mapping
&
Connection

Copyright © TELCOMA. All Rights Reserved

Mapping:

Map operation is used for mapping port of test component to a port of TSI.

Syntax:

```
map ( v_client : pt,        system : pt_client );
map ( v_root : pt,          system : pt_root );
map ( v_remote : pt,        system : pt_remote );
```

Copyright © TELCOMA. All Rights Reserved

Mapping:

Unmap operation is used to undo the mapping during run time.

Unmap statement are used to end the test case.

To unmap all ports of the test system: "*unmap(all component : all port)*"

Syntax:

```
unmap ( v_client : pt,      system : pt_client );
unmap ( v_root : pt,        system : pt_root );
unmap ( v_remote : pt,      system : pt_remote );
```

Copyright © TELCOMA. All Rights Reserved

Connection:

Connect operation is used to connect the ports of two test components directly to exchange messages.

Disconnect operation is used to remove the connection between the ports during execution of test cases.

Difference between *connect* operation and *map* operation:

In *map* operation, port of test component is mapped to port of TSI whereas In *connect* operation, ports of two test component are connected together.

Copyright © TELCOMA. All Rights Reserved

Connection:

Syntax:

```
connect ( v_clienta : pt_time,   mtc : pt_time );
connect ( v_clientb : pt_time,   mtc : pt_time );
connect ( v_clientc : pt_time,   mtc : pt_time );
```

Copyright © TELCOMA. All Rights Reserved

Many to one:

In TTCN-3, it is possible to mapped or connect several ports to one port. This is called a many-to-one mapping.

Send statement with a "*to*" clause is used to send a message to one of several components i.e connected to the same port.

This *"to"* clause indicates the destination test component to whom the message should be sent.

If the test component specified in the to clause does not exist or does not have a connection, this will cause a run-time system error.

Copyright © TELCOMA. All Rights Reserved

Many to one:

Syntax:

```
pt_time.send( t_answer ) to v_client;
```

Or

```
pt_time.send( t_answer ) to (v_clienta, v_clientb); // multicast
```

Copyright © TELCOMA. All Rights Reserved

Procedure Based Communication

Copyright © TELCOMA. All Rights Reserved

Difference:

Message Based Communication

- Communication between client/server messages are sent and received using the same primitives (*send, receive*) regardless of the role of the communication partner.

Procedure-based communication

- Makes a clear distinction between client/server roles: for each communication act, there exists a client that calls a remote procedure, and a server that processes this invocation and eventually returns a reply.

Copyright © TELCOMA. All Rights Reserved

Difference:

Message Based Communication

- It asynchronous in nature.
- The sender of a message will proceed with its behaviour before its message has been answered

Procedure-based communication

- It is synchronous in nature.
- The caller of a remote procedure will block until a reply has been returned or an exception has been raised.

Copyright © TELCOMA. All Rights Reserved

Signature:

First, define the interfaces that will be used to communicate with the SUT.

Signatures are used to define the remote procedures which is used in testing.

A signature consist of a name, a possibly empty sequence of parameters with their types and passing modes, an optional return type, and a possibly empty list of exception type

Copyright © TELCOMA. All Rights Reserved

Signature:

Syntax:

```
signature   lookup ( in charstring key ) return charstring
exception   ( NotFound, SessionExpired );
signature   update ( in charstring key, inout charstring val )
exception   ( NotAllowed, SessionExpired );
signature   logout();
```

Copyright © TELCOMA. All Rights Reserved

Non blocking Signature:

A signature may be declared as non-blocking if it does not specify a return type.

It has no *out* or *inout* parameters whereas *in* parameters are permitted.

The invocation of such a remote procedure does not allow passing information back from the callee to the caller other than that the invocation has been received and possibly processed.

A non-blocking signature is declared using the noblock keyword:

```
signature unackedLogout() noblock;
```

Copyright © TELCOMA. All Rights Reserved

Ports:

A procedure-based port may be mapped to a procedure-based port on the test system interface (TSI)

if:

- each *in* or *inout* signature at the TSI port type is matched by an *in* or *inout* signature at the test component's port type
- each *out* or *inout* signature at the test component's port type is matched by a corresponding *out* or *inout* signature at the TSI port type.

Copyright © TELCOMA. All Rights Reserved

Ports:

Syntax:

```
type port DirectoryC procedure {
        out lookup, update, logout
        }
type port DirectoryS procedure {
        in lookup, update, logout
        }
```

Copyright © TELCOMA. All Rights Reserved

Modes of Communication:

Modes of Procedure-based communication:

- calling a signature (client to server);
- replying to a call (server to client); and
- indicating an exceptional condition (server to client).

Six communication operations in TTCN-3:

call and *getcall* , *reply* and *getreply*, *raise* and *catch*.

Copyright © TELCOMA. All Rights Reserved

Procedure Based Communication On Client Side

Copyright © TELCOMA. All Rights Reserved

Call Statement:

In Procedure-based communication, the *call* statement is used to invoke a signature on a declared port.

Call Statement specifies the signature of the procedure to call and the actual values for the signature parameters are given in the form of a template.

Inline templates specify all procedure parameters directly after the signature identifier within the call statement.

The port specified in the statement must be connected or mapped.

Copyright © TELCOMA. All Rights Reserved

Call Statement:

Specific values are specified for each *in* or *inout* parameter of the signature.

A hyphen '-' is used to avoid specifying a *out* parameter value.

Syntax:

```
// pt is a mapped or connected port of type DirectoryC
pt.call( lookup : {"ATM Pin"} )
        { ... }
```

Copyright © TELCOMA. All Rights Reserved

Getreply Operation:

The *getreply* operation is used to specifies the port on which user get the reply.

In this operation, the same port must be there as the initial call and also the template used to specify the expect reply must have the same signature as the initial call.

Syntax:

```
[] pt.getreply ( update : {-, ""} )       { ... }          or
[] pt.getreply (lookup : {-} )       { ... }
```

Copyright © TELCOMA. All Rights Reserved

Getreply Operation:

Syntax:

Value Redirection

```
var charstring v_oldv;
    pt.call( ... ) {
        [] pt.getreply( update : {-,?} ) -> param (v_oldV := val ) { }
        }
```

Copyright © TELCOMA. All Rights Reserved

Catch Operation:

Catch operation is used to specifies a port and the signature type, plus a template that constrains the exception value that shall be caught.

When used in the body of a call statement, the catch operation must specify the same port and the same signature type as the call operation.

Copyright © TELCOMA. All Rights Reserved

Catch Operation:

Syntax:

```
[] pt.catch ( update, NotAllowed : ? ) {
        setverdict( fail );
        }
[] pt.catch ( update, SessionExpired : ? ) {
        setverdict( fail );
        }
```

Copyright © TELCOMA. All Rights Reserved

Catch Operation:

Syntax:

Return value redirection

```
var charstring v_returnVal;
    pt.call( ... ) {
        [] pt.getreply ( lookup : [-] value ? ) -> value v_returnVal [ ]
    }
```

Copyright © TELCOMA. All Rights Reserved

Deadlocks:

During the evaluation of the body of a call statement, all active defaults are ignored.

When none of the alternatives of the call statement body matched, the execution of the current component blocks until re-evaluation of the alternatives.

This result in deadlocking of the test system or at least one test component.

Copyright © TELCOMA. All Rights Reserved

Timeout:

In Timeout setting, if no reply or exception is received within particular period, the test system will generate a timeout exception.

The timeout period is specified by a float value measured in seconds.

The timeout exception is handled with the catch operation, using the keyword "*timeout*".

Syntax:

```
[] pt.catch( timeout ) {
        setverdict( inconc );
        }
```

Copyright © TELCOMA. All Rights Reserved

Procedure Based Communication on Server side

Copyright © TELCOMA. All Rights Reserved

Procedure Based Communication:

The TTCN-3 operations for the server side are:

- *Getcall:* to receive incoming procedure invocations
- *Reply:* to dispatch the corresponding invocation result.
- *Raise*: used to send exceptions back to the invoking client.

The concepts and syntax for the server-side communication are very similar to those for client-side communication

Copyright © TELCOMA. All Rights Reserved

Getcall Operation:

The *getcall* operation is used to accept incoming calls from other components or the SUT.

With this operation a port which is specified must be connected or mapped and must have an underlying port type, that is of procedure kind

It must also lists the expected signature among its in signatures.

A template for the signature of the incoming call must also specified, either an inline or explicit template.

Copyright © TELCOMA. All Rights Reserved

Getcall Operation:

Syntax:

```
alt {
    [] pt.getcall ( lookup : [?] ) {
    // deal with the lookup procedure
        }
    [] pt.getcall ( update : [?,?] ) {
    // deal with the update procedure
        }
}
```

Copyright © TELCOMA. All Rights Reserved

Reply Operation:

Reply operation is used to send a reply back to the client.

With this operation, the specified port is added which is mapped or connected, with a procedure kind and also with a list the signature among its *in* signatures.

The template used in the reply operation must specify the signature for which the reply is sent and give fully defined values for each *out* or *inout* parameter of the signature.

Copyright © TELCOMA. All Rights Reserved

Reply Operation:

Syntax:

```
const charstring c_noPreviousValue := " " ;
pt.reply ( update : { - , c_noPreviousValue } ) ;
```

// syntax to specify a return value if the signature defines a return type.

```
pt.reply ( lookup : {-} value "secret" );
```

Copyright © TELCOMA. All Rights Reserved

Raise Operation:

Errors that occur during the execution of a remote procedure are often signalled to the client using exceptions.

In TTCN-3, such an exception is generated with the *raise* operation.

With this operation, specified port must be mapped or connected and must have underlying port type, that is of procedure kind.

It must lists the given signature among its *in* signatures.

The signature, for which the exception is generated, has to be specified together with a fully defined implicit or explicit template.

Copyright © TELCOMA. All Rights Reserved

Raise Operation:

Syntax:

```
pt.raise( lookup, NotFound:{} );
```

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Modularity

Copyright © TELCOMA. All Rights Reserved

Modularity:

Modularity is one of the key feature of TTCN-3

Modularity and modularisation of TTCN-3 code are important because they can provide the key to a successful testing project.

Modularisation allows easier distribution of code development and maintenance when various developer are working on TTCN-3 code.

Copyright © TELCOMA. All Rights Reserved

Modules:

A module definition starts with the *module* keyword followed by an identifier, that tells the module's name.

While structuring test suite into several modules, each module must have a unique name.

The module body is delimited by curly brackets

It may contain an arbitrary number of definitions and most important is control part definition.

Copyright © TELCOMA. All Rights Reserved

Module:

Syntax:

```
module TEST_syst1 {
        // definition part
    Control {
        // control part
    }
}
```

Copyright © TELCOMA. All Rights Reserved

Group:

The definition part within a module of TTCN-3 code can be structured using *groups*.

Groups have little logical significance

It play vital role while importing of definitions from module to module and they also provide easy navigation in TTCn-3 code.

A *group* definition starts with the "*group*" keyword followed by a group identifier and the contained definitions between curly braces.

Copyright © TELCOMA. All Rights Reserved

Groups:

Syntax:

```
module TEST_syst1 {
    group consts {
        group basic {
            const integer c_value1 := 50;
            const integer c_value2 := 65;
            }
        }
    ..... //continued
    }
```

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Importing Modules

Copyright © TELCOMA. All Rights Reserved

Import Operation:

Importing is used when there is need to declare definitions from one module to another module.

Import definition starts with the keywords "*import*" followed by the name of the module that is to import, followed by a specification of the definitions that user want to import from particular module.

Definitions from a module are made 'visible' in another module by importing explicitly.

Mostly import definition is declared in beginning of the module definition part.

Copyright © TELCOMA. All Rights Reserved

Import Operation:

```
module TEST_syst1 {
        import from TEST_syst2 all;
        Control{
                // control part
        }
}
```

```
module TEST_syst2 {
        Function f1() runs on
    TEST_sys2{
        // function part
    }
}
```

Copyright © TELCOMA. All Rights Reserved

Visibility:

TTCN-3 consist of three types of visibility i.e. defined at the top-level in the module definitions part:

1. Public
2. Friend
3. Private

A group combines definitions that can have different types of visibility.

Copyright © TELCOMA. All Rights Reserved

Visibility:

1. Public: If no visibility type is specified with the definition then it is *public* by default and it can be imported by any other module.
2. Friend: Modules having visibility *friend* it mean they have friend-relationship and can import each other's definitions
3. Private:A definition having visibility *private* can only be used locally in that specific module and not be imported by any other module.

Copyright © TELCOMA. All Rights Reserved

Visibility:

Syntax:

```
module ATM_mac {
    friend module ATMpin;
    friend type boolean ATMStarted;
        public group Money{
            public type integer Withdraw;
            friend type boolean Deposit;
            private type charstring LeftAmount; }
    }
```

Copyright © TELCOMA. All Rights Reserved

Import from other languages:

TTCN-3 is designed to be extensible towards other programming languages and type systems.

The import statement plays the key role in the extension of TTCN-3 code to import other languages.

Syntax:

```
import from DNS language "ASN.1:2002" all;
```

Copyright © TELCOMA. All Rights Reserved

TTCN 3
Restriction on Import Operation

Copyright © TELCOMA. All Rights Reserved

Restriction on Import Operation:

Benefits of Restriction imposed over *import* operation:

1. Restrictive imports result in smaller ‘interfaces’ between modules, which help in better maintainability.
2. Restricted imports decrease the amount of workload over TTCN-3 tools it perform the processing of a module prior to the execution of the test system.

Copyright © TELCOMA. All Rights Reserved

Restriction on Import Operation:

Restriction by kind:

TTCN-3 restrict the imports on two levels:

1. to restrict the import to certain kinds of definitions;
2. to import specific definitions identified by their name.

Syntax:

```
import from SysTypes {
        type all;
        const all;}
```

Copyright © TELCOMA. All Rights Reserved

Restriction on Import Operation:

Restriction by Name:

It is possible to import the definition by name if user required a more control over the imported definition.

Syntax:

```
import from SysTypes {
    const c_defaultPort;
    type Meg1, RawMsg2;
    type Identification; }
```

Copyright © TELCOMA. All Rights Reserved

Restriction on Import Operation:

Restriction on importing groups:

To import sub-group directly from module SysTypes:

```
module SysTypes {
    group consts {
    group basic { /* ... */ } }
        group types {
            group basic { /* ... */ }
            group structured { /* ... */ }
    } }
```

Copyright © TELCOMA. All Rights Reserved

Restriction on Import Operation:

Syntax:

Importing sub-group structured directly:

```
import from SysTypes { group structured }
```

Importing sub-group basic directly: (as there are two sub-group of this name it is importing by defining its parent group name)

```
import from SysTypes { group basic }          // ERROR
import from SysTypes { group consts.basic }   // OK
```

Copyright © TELCOMA. All Rights Reserved

Restriction on Import Operation:

Restriction on importing *“ALL EXCEPT”*:

Restriction of importing *“all”* definitions of a certain module, kind or group except for an explicitly excluded list of definitions.

It is indicated using the “*except*” keyword.

Syntax:

```
import from SysTypes {
type all except Identification, RawMsg ;
const all except c_unsignedshortMax ; }
```

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Module Parameter

Copyright © TELCOMA. All Rights Reserved

Module Parameters:

It is necessary to provide certain parameters to a test suite, to execute it successfully in different environments.

Such parameters are called module parameter in TTCN-3 code.

Module parameters are used to provide external parameters to a TTCN-3 test suite at execution time.

These parameters are passed without re-processing the TTCN-3 code.

Module parameters work like constants, which can be overwritten externally by the test system user upon test system execution.

Copyright © TELCOMA. All Rights Reserved

Module Parameters:

The change of a module parameter during execution time result as an error by a TTCN-3 system.

Module parameters are declared on the module level, using the "*modulepar*" keyword followed by one or more module parameters between curly brackets.

Each module parameter is declared with a type and may have optionally default value.

Copyright © TELCOMA. All Rights Reserved

Module Parameters:

Values to module parameters are specified externally via command line parameters , configuration files or through TCI

If no actual value for a module parameter can be found at runtime, the default value will be used.

If no default value has been specified either, then the first access of the module parameter value will cause a test case error.

Copyright © TELCOMA. All Rights Reserved

Module Parameters:

Syntax:

```
module SysParameters {
    import from SysTypes { const c_defaultPort }
    modulepar charstring mp_bAddress, mp_aAddress;
    modulepar integer mp_aPort := 1000;
    modulepar integer mp_bPort := c_defaultPort;
}
```

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Attributes

Copyright © TELCOMA. All Rights Reserved

Attributes:

Attributes are used to specify the meta-information inside the TTCN-3 core notation.

This information cannot be read or written by any statement and it is used by test system entities externally.

Attributes are assigned to an (import) definition, group or module using the keyword “*with*”.

Copyright © TELCOMA. All Rights Reserved

Attributes:

Attributes of five different kinds:

- ***Display***: specify information i.e. related to different TTCN-3 presentation formats.
- ***Encode***: specify information which is used by the codec implementation
- ***Variant***: specify information which is used to selects a different variation within the selected encoding.
- ***Extension***: used for user- or tool-specific purposes.
- ***Optional***: indicate whether optional fields are set to be absent by leaving them out from definition or by setting them to omit

Copyright © TELCOMA. All Rights Reserved

Attributes:

Syntax:

```
type integer Identification ( 0 .. 65535 ) with { variant "unsigned 16 bit" }
```

Copyright © TELCOMA. All Rights Reserved

Attributes:

Attribute defining encoding:

Attributes are used to give complete encoding information for the text-based protocol, in the form of encoding attributes.

Syntax:

```
type record PortA {
        charstring name,
        unsignedshort portNo optional }
                with { encode ( portNo ) ": _" ;  };
```

Copyright © TELCOMA. All Rights Reserved

TTCN 3
Subtypes

Copyright © TELCOMA. All Rights Reserved

Subtypes:

Subtypes are created to allow restriction to a certain subset in the range of value.

Subtype specify the range of values, by specifying an upper and lower bound for the allowed values.

A new subtype is defined by using the keyword “*type*”, followed by the parent type, the name for the newly defined type, and the subtype’s restriction.

Copyright © TELCOMA. All Rights Reserved

Subtypes:

Syntax:

```
type charstring HostName ( "www.google.com", "www.yahoo.com",
                           "www.facebook.com" );
```

Copyright © TELCOMA. All Rights Reserved

Subtypes:

TTCN-3 has a wide range of subtype definitions:

- Aliasing – giving a new name to an already defined type.
- Value lists – restricting a type to a list of admissible values.
- Value ranges – restricting an ordered type to a certain range.
- Field value constraints – restricting values of selected fields for structured types.
- Character set restrictions – restricting the admissible characters in a character string type.
- Length restrictions – restricting the number of elements in strings or list types.

Copyright © TELCOMA. All Rights Reserved

Subtypes:

1. **Type Aliasing:**

   It is define by giving new name to the type without restricting the admissible set of values.

   Syntax:

   ```
   type integer HexadecimalInt;
   type integer OctalInt;
   ```

Copyright © TELCOMA. All Rights Reserved

Subtypes:

2. **Value List:**

   A value list subtype restricts the values to a fixed list of allowed values.

   These values are explicitly enumerated in the type definition.

   A value list subtype can be defined for all types.

   Syntax:

   ```
   type charstring SIPmessages ( "REGISTER", "INVITE", "ACK", "BYE",
   "CANCEL", "OPTIONS" );
   ```

Copyright © TELCOMA. All Rights Reserved

Subtypes:

3. **Value Range:**

   Value ranges are used to define subtypes of float and integer types.

   Half-open ranges can be defined using infinity and -infinity.

   The boundary values must be included in the range.

   Syntax:

   ```
   type integer IntCode ( 100 .. 609 );
   ```

Copyright © TELCOMA. All Rights Reserved

Subtypes:

4. **Type List:**

   The type list notation is used for basic types, structured types and anytype.

   The type list notation specify a new subtype which is based on two or more existing subtypes.

   Type lists may also contain additional length restrictions using *length* keyword followed by subtype identifier.

Copyright © TELCOMA. All Rights Reserved

Subtypes:

5. **Character Set Restriction:**

   Character set restriction are used to restrict the set of allowed characters for a string value.

   Syntax:

   ```
   type charstring AlphaString ( "0" .. "9", "A" .. "Z", "a" .. "z" );
   ```

Copyright © TELCOMA. All Rights Reserved

Subtypes:

6. **Length Restriction:**

   Length restriction is used to specify the exact length or a length range by using the "*length*" keyword.

   Infinity can be used to specify upward open ranges.

   Syntax: *type charstring NonEmptyString length (1 .. infinity);*

   Length restriction subtypes can also be specified for the other TTCN-3 string types such as universal charstring, bitstring, hexstring and octetstring.

Copyright © TELCOMA. All Rights Reserved

Subtypes:

7. **Type Conversion:**

   In TTCN-3, Implicit type conversion or ‘casting’ is not allowed.

   The mixing of integer and float in arithmetic expressions is also not allowed.

   Example: 9 + 2.0 is rejected as ill-typed.

   TTCN-3 offers a rich set of explicit conversion functions that allow for controlled conversion between various types.

Copyright © TELCOMA. All Rights Reserved

TTCN-3
Built-in Data types

Copyright © TELCOMA. All Rights Reserved

Boolean Type:

Boolean built-in type in TTCN-3 is used to assume the two truth values *true* and *false*.

The Boolean operators and, or, xor and not are used to form Boolean expressions.

Copyright © TELCOMA. All Rights Reserved

Boolean Type:

Syntax:

```
const boolean c_B1 := true;
const boolean c_B2 := false;
Var B3 v_B3 := (10 > 2) ?;
var boolean := true : false ;
```

Copyright © TELCOMA. All Rights Reserved

Integer type:

TTCN-3 provides only a single built-in type for integral numbers: *integer*.

TTCN-3 tools support only signed 32-bit or signed 64-bit integer values.

Syntax:

```
type integer byte ( -128 .. 127 );
type integer unsignedbyte ( 0 .. 255 );
type integer unsignedshort ( 0 .. 65535 );
type integer long ( -2147483648 .. 2147483647 );
```

Copyright © TELCOMA. All Rights Reserved

Float Type:

Float type represent the real numbers in TTCN-3

Float is an ordered numerical type that has comparison and arithmetic operators.

It does not provide any mathematical functions like (co)sine, exponential or logarithmic.

one useful function that the language does provide is random number generation (rnd).

Copyright © TELCOMA. All Rights Reserved

Float Type:

Syntax:

```
function f_value ( in float p_v ) return integer {
    if ( p_v >= 0.0 ) {
        return ( float2int( p_v + 0.5 ));
    }
    else {
        return ( float2int( p_v - 0.5 ));
    }
}
```

Copyright © TELCOMA. All Rights Reserved

Character String Type:

TTCN-3 has two different character string types: *charstring* and *universal charstring.*

The *charstring* type is restricted to represent 7-bit ASCII strings.

It is used to convert human readable text in Latin alphabet.

The *universal charstring* type is more extensive and contain characters from the Unicode character set.

String literals are enclosed in double quotes (''),

Copyright © TELCOMA. All Rights Reserved

Character String Type:

Syntax:

```
var charstring v_clientName := "alice";
var charstring v_clientRealname := "Alice " "Host" " Bell";
var universal charstring v_finishClientName := "Yrjo Aberg"
```

Copyright © TELCOMA. All Rights Reserved

Verdicttype Type:

TTCN-3 use a type to represent the possible outcomes of test case i.e. verdicts

This type is known as *verdicttype*.

It has five possible values: *none*, *pass*, *inconc*, *fail* and *error*.

Each test component implicitly carries a value of type verdicttype, which stores the current local verdict.

This state can be set with *setverdict* and can be read with *getverdict*.

Copyright © TELCOMA. All Rights Reserved

Verdicttype Type:

Syntax:

```
        setverdict( pass );
;
if ( getverdict() == fail )
        { /* ... */ }
```

Copyright © TELCOMA. All Rights Reserved

Binary String Type:

In TTCN-3, raw binary data is represented using its different binary string types such as bitstring, hexstring or octetstring.

These string types allow the representation of binary data either without grouping, with grouping of 4 bits or with grouping of 8 bits, respectively.

Literals for the binary string types are written as a sequence of binary (for bitstring) or hexadecimal digits (hexstring, octetstring) in single quotes (') followed by the letters ‘B’, ‘H’ or ‘O’.

Copyright © TELCOMA. All Rights Reserved

Binary String Type:

Syntax:

```
const bitstring c_2011_bit := '11111010101'B;
const hexstring c_2011_hex := '7d6'H;
const octetstring c_2011_oct := '07D5'O;
```

Copyright © TELCOMA. All Rights Reserved

TTCN-3
User Defined Type

Copyright © TELCOMA. All Rights Reserved

Enumerated Type:

Enumerated types is used to represent types that have a small, finite set of values.

Enumerations are ordered types and can be compared using the comparison operators "<", ">", "<=" and ">=".

Syntax:

```
type enumerated DnsOption { e_question, e_answer
        // will be interpreted as: e_question( 0 ), e_answer( 1 )
}
```

Copyright © TELCOMA. All Rights Reserved

Record Type:

Records are used to group related fields into a single type.

Field names within a record must be unique and may be re-used in different record type definitions.

The different notations used to specify record values: *value list notation* and *assignment list notation*.

The value list notation specifies values for all the fields of the record in their order of occurrence in the type definition, whereas the assignment list notation explicitly specifies the field names.

Copyright © TELCOMA. All Rights Reserved

Record Type:

Syntax:

```
type record SipStatus {
        float version,
        charstring Phrase
};
```

Copyright © TELCOMA. All Rights Reserved

Set Type:

Set type in TTCN-3 are cryptographically secure hash function.

It is used to make guessing of the shared secret from the hash value difficult.

The only major difference is that set values may not be written using the value list notation.

Copyright © TELCOMA. All Rights Reserved

Set Types:

Syntax:

```
type set Values {
    charstring realm,
    charstring nonce, DigestAlgorithm algorithm,
    charstring opaque optional }
```

Copyright © TELCOMA. All Rights Reserved

Union Type:

Union type is used to combines a group of different types in such a way that exactly one of these types is present at any one time.

Values for union type are written in assignment list notation with only a single field, which specifies the variant.

Access to a variant of a union value is done with the dot operator '.'

Copyright © TELCOMA. All Rights Reserved

Union Type:

Syntax:

```
type union SipReq {
        SipUri      sip,
        SipsUri     sips,
        TelUri      tel,
        FaxUri      fax,
        ModemUri    modem
};
```

Copyright © TELCOMA. All Rights Reserved

Encoding and Decoding:

The invocation of these codec functions is usually performed implicitly when a send or receive statement is executed.

The predefined functions *encvalue* and *decvalue* are used to encode and decode the messages.

Copyright © TELCOMA. All Rights Reserved

TTCN-3
List Type

Copyright © TELCOMA. All Rights Reserved

List type:

List Type is used to collect a bounded or unbounded number of values of the same type into one value.

This is done by using arrays and record-of types for ordered groups, and set-of types for unordered groups.

The operators such rotation ‘<@’ and ‘@>’, and concatenation ‘&’ are used define them together.

Pre-defined functions are used with list values and templates including lengthof, substr and replace functions.

Copyright © TELCOMA. All Rights Reserved

List Type:

1. **Record-of type:**

   Record of type provide the most natural way to define an ordered collection of elements – lists or vectors – of the same type in TTCN-3.

   Record-of types may contain an arbitrary number of elements, but subtyped to fixed length or length ranges.

   The length ranges always contain the boundary values.

   It is also used to represent an IP address.

Copyright © TELCOMA. All Rights Reserved

List Type:

1. **Record-of type:**

   Access to the individual elements of a record-of value is achieved with this operator "[]" with indices starting from 0.

   Syntax:

   ```
   type record length ( 1..infinity ) of ViaHeader ;
   ```

Copyright © TELCOMA. All Rights Reserved

List Type:

2. **Array:**

   Arrays are also used to group values.

   Arrays can be defined either inline in constant or by variable declarations.

   The number of elements in an array is specified between square brackets.

   Each array specify the number of elements using upper and lower bounds

Copyright © TELCOMA. All Rights Reserved

List Type:

2. **Array:**

   The upper and lower bound always belong to the index range.

   While specifying a value for an entire array, the value list notation is used.

Copyright © TELCOMA. All Rights Reserved

List Type:

Syntax:

```
type unsignedshort IPV6 [8]; // explicit array definition, 8x16 bit
var unsignedbyte v_ipv4 [4]; // implicit array definition, 4x8 bit
var IPV6 v_ipv6 := {  65550, 215, 41154, 10, 0, 0, 31, 1 };
    v_ipv4[0] := 100;
    v_ipv4[1] := 0;
    v_ipv4[2] := 0;
    v_ipv4[3] := 1;
```

Copyright © TELCOMA. All Rights Reserved

List Type:

3. **Multi-dimensional Array:**

   Multi-dimensional arrays are used to store multi-dimensional tables or matrices.

   Syntax:

```
type integer TwoByThree [2][3];
type integer ThreeByTwo [0..2][0..1];
const TwoByThree c_2x3 := { [1,2,3], [4,5,6] };
const ThreeByTwo c_3x2 := { [1,2], [3,4], [5,6] };
```

Copyright © TELCOMA. All Rights Reserved

List Type:

4. **Set-of Type:**

   The only difference between set-of and record-of is the notion of equality.

   I.e While two record-of values are equal if they contain the same values in the same order, two set-of values are already considered equal if they contain the same elements in the same multiplicity, but not necessarily in the same order.

Copyright © TELCOMA. All Rights Reserved

Templates

Copyright © TELCOMA. All Rights Reserved

Templates:

The simplest form of TTCN-3 templates defines a unique value, which is mainly used as the argument to sending operations.

The real power of TTCN-3 templates lies in the ability to specify multiple values or variations of a message within one single definition.

Copyright © TELCOMA. All Rights Reserved

Match Operation:

Templates are coupled with built-in *matching* mechanism in TTCN-3

This *matching* mechanism, is automatically invoked when a TTCN-3 *receive* operation is executed.

It can also be invoked directly using the TTCN-3 *match* operation.

The *match* operation takes two parameters:

1. value
2. Template.

Copyright © TELCOMA. All Rights Reserved

Match Operation:

This operation checks if the given value is within the restrictions given by the template.

It returns *true* in the case of a match, otherwise it returns *false*.

Copyright © TELCOMA. All Rights Reserved

Match Operation:

Syntax:

```
const PortA     Subject1 := {
    Subject := "English",           marksObt := 58 }
const PortB     Subject2 := {
    Subject := "Mathematics",       marksObt := omit }
const PortC     Subject3 := {
    Subject := "Science",           marksObt := 58 }
        b := match ( Subject1, Subject3 );      // true
        b := match ( Subject2, Subject1 );      // False
```

Copyright © TELCOMA. All Rights Reserved

Pre-Defined Function:

Valueof:

A template can be turned into a value using the pre-defined function *valueof*, given that the template only specifies a single value for all fields.

Isvalue:

To prevent test case errors, *isvalue* is used for checking the template that are converted with this pre-defined function, which returns either true or false.

Copyright © TELCOMA. All Rights Reserved

TTCN-3
Templates with Match Expression

Copyright © TELCOMA. All Rights Reserved

“Any” Matching Expression:

’*any*’ is denoted by ‘?’ wildcard character.

It is most frequently used matching expression.

It can be applied to any built-in type, string type or user-defined type.

This expression accepts any single value, which is compatible with the underlying type definition.

Copyright © TELCOMA. All Rights Reserved

“Any” Matching Expression:

Syntax:

```
type integer First      ( 0 .. infinity );      // non-negative numbers
template integer        a_anyInt    :=  ?;
template First          a_anyFst    :=  ?;
    b := match ( 1, a_anyFst );                 // b evaluates to true
    b := match ( -1, a_anyFst );                // b evaluates to false,
    b := match ( -1, a_anyInt );                // b evaluates to true
```

Copyright © TELCOMA. All Rights Reserved

“Any” Matching Expression:

Value List:

A template definition with a value list simply specifies all the values that are acceptable.

A received value will match the value list if it matches one of the elements in the list. Then it is set to true otherwise false.

Copyright © TELCOMA. All Rights Reserved

“Any” Matching Expression:

Value Range:

Template definitions with the basic types *integer* and *float* are defined as a range of acceptable values.

A value will match in such template in between the lower and upper limit of the range.

The range can be defined with one or both boundary values included or excluded from the range.

Copyright © TELCOMA. All Rights Reserved

“Any” Matching Expression:

Value Range:

To exclude a boundary value from the range it is preceded by the character ‘!’.

The predefined constants infinity and -infinity aare used if no upper or lower limit are specified for the expected value.

Copyright © TELCOMA. All Rights Reserved

“Any” Matching Expression:

Syntax:

```
template integer a_Int          :=      ( 0 .. infinity );
template integer a_anyCode      :=      ( 100, 180 .. 183 );
b := match ( -1, a_Int );               // b evaluates to false
b := match ( 183, a_anyCode );          // b evaluates to true
b := match ( 110, a_anyCode );          // b evaluates to false,
```

Copyright © TELCOMA. All Rights Reserved

TTCN-3 Template Parameterisation

Copyright © TELCOMA. All Rights Reserved

Template Parameterisation:

Template are parameterise in a similar way to functions.

Parameters are used to pass regular values, other templates and matching expressions into a template definition.

Parameters specify the information that only becomes definite during test system execution.

Parameters of templates are always *in* parameters and cannot be of *out* or *inout* kind.

Copyright © TELCOMA. All Rights Reserved

Template Parameterisation:

1. **Value Parameter:**

   Value parameters are instantiated with proper values.

   Syntax:

```
template Value a_Value1 ( in charstring p_hostName,
                          in integer p_portNumber ) := {
        host := p_hostName & ".com",
        portNumber := p_portNumber
}
```

Copyright © TELCOMA. All Rights Reserved

Template Parameterisation:

2. **Template Parameters:**

   Template parameters are defined by preceding the parameter declaration with the *template* keyword.

   It is possible to pass *in* other templates, matching expressions or *omit* values.

Copyright © TELCOMA. All Rights Reserved

Template Parameterisation:

2. Template Parameters:

Syntax:

```
template Host a_hostPort( template charstring p_hostName,
template integer p_port ) :=
    {
        host := { hostName:= p_hostName },
                portNumber := p_port
    }
```

Copyright © TELCOMA. All Rights Reserved

TTCN-3
Test System

Copyright © TELCOMA. All Rights Reserved

Test System:

A TTCN-3 test system defined as a collection of different test system entities.

These entities are interact with each other during a test suite execution.

Test system architecture consists of three dominant layers.

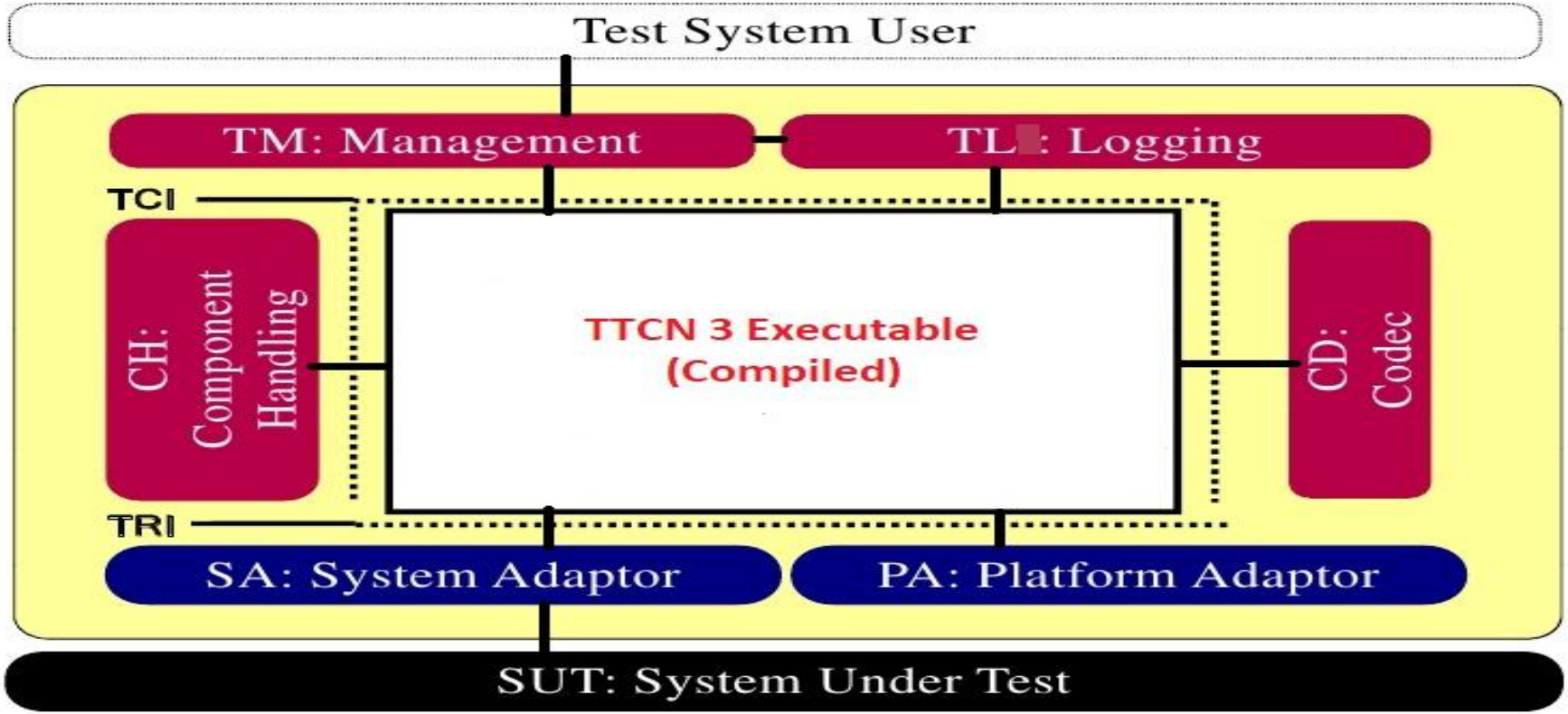
A central layer of the TTCN-3 Executable (TE) and two main layers.

The TTCN-3 Executable (TE), handles the execution of TTCN-3 statements.

The TE depends on a number of services for this operation that are provided by the other two main layers.

Copyright © TELCOMA. All Rights Reserved

TTCN 3 Test System:



Test System:

Test Management and Control (TMC):

This entity is responsible for

- interfacing to the test system user,
- the encoding and decoding of data,
- logging
- deal with distributed execution.

These services are provided by the Test Management (TM), External Codecs (CD), Test Logging (TL) and Component Handling (CH) entities.

Copyright © TELCOMA. All Rights Reserved

Test System:

SUT Adapter (SA) and Platform Adapter (PA):

These two adapter are used by TE for interfacing towards the system under test (SUT) and towards the actual test system operating system,

Communication with the central entity i.e. TE is done via the standardised TCI and TRI.

Copyright © TELCOMA. All Rights Reserved

TTCN-3 Executable:

The TE is located at the heart of a TTCN-3 test system.

The name ‘TTCN-3 Executable’ indicate that this entity is responsible for the execution of the TTCN-3 code.

TE consist of a suitable representation of TTCN-3 test suite plus some mechanisms that execute code as specified by the TTCN-3 core language standard.

Copyright © TELCOMA. All Rights Reserved

TTCN-3 Executable:

Run Time System (RTS) is a mechanism that provide the services provided by the other test system entities in its execution of the TTCN-3 test suite.

The RTS implements all the advanced aspects of TTCN-3 semantics.

for example concurrent test components, snapshots, verdict handling, memory management, dynamic type checking and so on.

Copyright © TELCOMA. All Rights Reserved

Procedure for the test system execution:

1. Initialisation of test system and test cases.
2. Prepare communication channel toward SUT.
3. Adding operations to handle communication toward SUT.
4. Defining and starting of Timer.
5. Operations to handle incoming Communication from SUT.
6. Handling of timeouts and stopping of timer while execution.
7. Unmapping of Communication channels

Copyright © TELCOMA. All Rights Reserved

SUT Adapter:

The role of SA is to provide the means for communication between the TE and the SUT

It also bridge the gap between the (abstract) TRI communication primitives and real communication mechanisms employed by the SUT.

The main task of the SA is to add transport information to encoded messages or calls sent by the TE and send them to the SUT.

Copyright © TELCOMA. All Rights Reserved

SUT Adapter:

Functions of SUT Adapter:

- Executing Threads
- Management of TRI information
- Provide Procedure based communication with the SUT
- Configuration of Dynamic SUT Adapter

Copyright © TELCOMA. All Rights Reserved

Platform Adapter:

The PA is used to implement test system adaptation aspects that are not directly related to the interaction with the SUT.

It also implements the model of time that is used during the execution of TTCN-3 as well as external functions.

Functions:

- TRI timing operation
- Implementation of non-real-time
- Defining external functions

Copyright © TELCOMA. All Rights Reserved

External Codec:

The CD entity perform both the encoding and decoding between the value representation used in the TE and the format expected by the SUT.

Functions:

- Access to TTCN-3 value
- Implementing encoder
- Implementing Decoder

Copyright © TELCOMA. All Rights Reserved