TTCN-3 COURSE

PRESENTATION MATERIAL

TEST COMPETENCE CENTER
ERICSSON HUNGARY

http://ttcn.ericsson.se/

Copyright © Ericsson AB 2018. All rights reserved.

This program and the accompanying materials
are made available under the terms of the Eclipse Public License v1.0
which accompanies this distribution, and is available at

Copyright © Ericsson AB 2018. All rights reserved.

This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at

# CONTENTS

- Protocols and Testing 3
- Introduction to TTCN-3 16
- TTCN-3 module structure 27
- Type system 36
- Constants, variables, module parameters 69
- Program statements and operators 79
- Timers 87
- Test configuration 93
- Functions and testcases 105
- Verdicts 120
- Configuration operations 125
- Data templates 153
- Abstract communication operations 192
- Behavioral statements 206
- Sample test case implementation 231
I. PROTOCOLS AND TESTING

WHAT IS “PROTOCOL”?
DEFINITIONS
PROTOCOL VERIFICATION, TESTING AND VALIDATION

CONTENTS
PROTOCOL
A protocol is a set of rules that controls the communication between entities in different systems.

Protocols define format (syntax), order of messages sent and received among network entities, as well as actions taken on message transmission or reception (behaviour).

Behaviour of the protocols can be defined using natural language (e.g. English) or some formal description technique. Examples for the latter: SDL, Estelle and LOTOS. They are compilable specification languages. None of them has outweighed the others.

- ASN.1 Abstract Syntax Notation One (ITU-T X.680-X.699)
- TTCN-3 Testing and Test Control Notation version 3 (ETSI ES 201 873)
- MSC Message Sequence Charts (ITU-T Z.120-Z.129)
- LOTOS: Language of Temporal Ordering Specifications (ISO8807) is widely used in the academic world. LOTOS is based on communicating processes.
- Estelle (ISO9074) is based on extended finite automata.
PROTOCOL TECHNOLOGY

- Informal specification
- Formal specification
- Implementation
- Test cases

- Ambiguous
- Not complete

- ASN.1, TTCN-3, ...
- UML, SDL, MSC, ...
- Verification, validation
- Conformance tests
TESTING

- Black box testing
  - Implementation/System Under Test
  - Point of Control and Observation

Verdict:
  - pass, fail, inconclusive

- Not possible to test all the situations
  - Test Purposes
ATS: Abstract Test Suite, a collection of Abstract Test Cases.
ETS: Executable Test Suite, a set of Executable Test Cases.
IUT: Implementation Under Test
TEST TYPES

• Conformance testing
  – Function tests
    ▪ Regression tests
  – System tests

• Interoperability testing

• Performance (Load) testing
Black-box testing means that the internal structure of the tested software product is not known: the only way to test it is to send a message ("stimulus") to the system and to analyse the received response. The latter is compared to the due response determined beforehand using the reference specification. If the comparison ("pattern matching") between the real and the expected response fails, the test case is considered as "failed" otherwise "passed".

The test script language must have means to match the expected and the received messages even if the message elements arrive in different order, or some of them (the optional ones) are missing. Usually, there are more than one possible responses; all of them must be accepted. Once the match is determined, the next stimulus is constructed taking into consideration the data having received in the response, and so on.

The test script language must be prepared to determine that the expected response is not received within the specified time frame: it must handle timing ("temporal") requirements.

**3.3.118 test purpose:** A prose description of a well defined objective of testing, focusing on a single conformance requirement or a set of related conformance requirements as specified in the appropriate OSI specification (e.g. verifying the support of a specific value of a specific parameter).

**3.3.3 abstract test case:** A complete and independent specification of the actions required to achieve a specific test purpose, defined at the level of abstraction of a particular Abstract Test Method, starting in a stable testing state and ending in a stable testing state. This specification may involve one or more consecutive or concurrent connections.

Note 1: The specification should be complete in the sense that it is sufficient to enable a test verdict to be assigned unambiguously to each potentially observable test outcome (i.e. sequence of test events).

Note 2: The specification should be independent in the sense that it should be possible to execute the derived executable test case in isolation from other such test cases (i.e. the specification should always include the possibility of starting and finishing in the "idle" state).

**3.3.31 executable test case:** A realization of an abstract test case.

**3.3.107 test case:** An abstract or executable test case.

**Abbreviations**

IUT: Implementation Under Test

SUT: System Under Test
3.3.121 testing state: A state encountered during testing, comprising the combination of the states of the SUT, the test system, the protocols for which control and observation is specified in the ATS, and, if relevant, the state of the underlying service.

3.3.93 stable testing state: A testing state which can be maintained, without prescribed Lower Tester behaviour, sufficiently long to span the gap between one test case and the next in a test campaign.

3.3.47 initial testing state: The testing state in which a test body starts.

3.3.110 test event: An indivisible unit of test specification at the level of abstraction of the specification (e.g. sending or receiving a single PDU).

3.3.117 (test) preamble: The sequences of test events from the starting stable testing state of the test case up to the initial testing state from which the test body will start.

3.3.105 test body: The sequences of test events that achieve the test purpose.

3.3.116 (test) postamble: The sequences of test events from the end of the test body up to the finishing stable testing state(s) for the test case.
ATS is exhaustive if all test cases are exhaustive (all passing implementations are compliant)

ATS is sound if all test cases are sound (all implementations that do not pass are not compliant)

ATS is complete if all test cases are both sound and exhaustive
PHASES OF BLACK-BOX (FUNCTIONAL) TESTING

- Test purpose definition
  - Formally or informally
- TTCN-3 Abstract Test Suite (ATS)
  - design or generation
- Executable Test Suite (ETS) implementation
  - using the Means of Testing (MoT)
- Test execution against the Implementation Under Test (IUT)
  - with MoT
- Analysis of test results
  - verdicts, logs (validation)
Once the protocol specification is formalised, it is theoretically possible to generate executable test cases automatically. However, this procedure, called Computer Aided Test Generation (CATG) is only being developed.

Otherwise, one needs to design abstract test cases manually. Manual test suite design starts with the formulation of test purposes from protocol specification. Test purposes are implemented in test cases.
The test system is the link between “abstract” and “executable”. It derives executable test cases from abstract test cases and executable test suites (ETSs) from abstract test suites (ATSs). The test system and any additional equipment and procedures that may be required for the execution of test cases together are called the Means of Testing.
II. INTRODUCTION TO TTCN-3

HISTORY OF TTCN
TTCN-2 TO TTCN-3 MIGRATION
TTCN-3 CAPABILITIES, APPLICATION AREAS
PRESENTATION FORMATS
STANDARD DOCUMENTS

CONTENTS
Test notation is used to describe abstract test cases. The test notation can be an informal notation (without formally defined semantics) or a Formal Description Technique (FDT). TTCN-2 is an informal notation with clearly defined, but not formally defined semantics.

The International Organization for Standardization (ISO*) has standardised first two versions of TTCN. The very same standard has been adopted as ITU-T and ETSI standard. Data structure definitions written in ASN.1 can be imported to TTCN-2.

TTCN-2 test cases can be edited using special software, e.g. ITEX. Executable test cases are produced and run with help of e.g. SCS.

Abbreviations:

ETSI
IEC International Engineering Consortium
ITU-T International

SCS System Certification System
(Ericsson’s TTCN test case execution platform)

ITEX Interactive TTCN Editor and eXecutor
(from the Swedish firm Telelogic)

* Because “International Organization for Standardization” would have different abbreviations in different languages (“IOS” in English, “OIN” in French for Organisation internationale de normalisation), it was decided at the outset to use a word derived from the Greek isos, meaning “equal”. Therefore, whatever the country, whatever the language, the short form of the organization’s name is always ISO.
Language development was being done in the following framework:
ETSI MTS/STFs 133, 156, 213, 253

TTCN-3 can be used for protocol testing (for mobile and Internet protocols), supplementary service testing, module testing, the testing of CORBA-based platforms, the testing of Application Programming Interfaces (APIs) and many more applications. The language is not restricted to conformance testing, but can be used for interoperability, robustness, regression, system, and integration testing.

The syntax of TTCN-3 is new, but the language has retained (and improved upon) much of the well proven capabilities of its predecessors. Its main features include:

- Dynamic, concurrent testing configurations
- Synchronous and asynchronous communication mechanisms
- Encoding information and other attributes (including user extensibility)
- Data and signature templates with powerful matching mechanisms
- Type and value parameterization
- Assignment and handling of test verdicts
- Test suite parameterization and test case selection mechanisms
- Combined use of TTCN-3 with ASN.1
- Well defined syntax, interchange format and static semantics
- Optional presentation formats (e.g. tabular conformance presentation format, MSC (Message Sequence Chart) format)
- Precise execution algorithm (operational semantics)
- Execution and control of test cases
The **Core Language** has a textual format, that, as opposed to the mp format of the TTCN-2 language, can be read by humans.

Tabular format was originally meant to facilitate the migration from TTCN-2 to TTCN-3. It is sparingly used nowadays.

In the graphical format (similarly to MSC) it is not possible to define types, templates etc. **User Defined Formats** are open to anyone.
Core Language is the basic language. White space or new line characters are not taken into consideration; it makes it similar to a programming language. Different TTCN-3 applications use it for data interchange.

You should not strive to understand the example, rather get a look and feel of it. It looks like any ordinary programming language.
Tabular Presentation Format resembles the most the TTCN-2 format, it is specified mainly for compatibility reasons. Editing is done in strictly specified tables, but data is saved in Core Language.

The example shows the same extract in Tabular Format: we can fill in the name of the test case, any comments, the type of the variables. The behaviour is specified as text in the next raw.
Graphical Presentation Format reminds the Test Sequence Chart or MSC. The messages sent and received are represented by arrows; there are additional special symbols for dynamic behaviour, cycles, decisions. For the time being, no editing program handling this format is known to us, however, there are programs capable of displaying Core Language programs in Graphical Format.

The perpendicular lines symbolize the components or, more precisely, the ports of the components. The horizontal arrows represent the messages sent and received. Boxes of various shape are representing the diverse operations coded in the Core Language.
The most important language TTCN-3 can interwork with is ASN.1. TTCN-3 has been designed from the beginning to ensure that definitions written in ASN.1 can be imported into test suites without the need for any modifications. With other words, when a protocol is specified in ASN.1 there is no need to rephrase it. Likewise, information in other format can be reused, e.g. functions written in C++ can be called from within the TTCN-3 module. It is planned to harmonize TTCN-3 with XML (eXtended Markup Language) and IDL (Interface Definition Language), but it can be harmonized with other ‘type & value’ system.
TTCN-3 is a procedural language, i.e., using the concept of the unit and scope. Unit corresponds to TTCN-3 modules, which are built of procedures (functions). Scope is the viewing range of a definition. There are seven scoping units in TTCN-3; they are dealt with later.

**Abstract Data Types**
Data can be specified independently from its coding and physical representation.

**Templates**
When sending a message, templates make possible to parameterise the message. When receiving a message, parameters or wildcards in templates render possible to accept or reject (‘to match’) a group of possible messages.

**Event handling**
While executing the program, we can wait for different events. The incidental arrival of these independent events influences the further program execution. Events are among others: reception of a message, completion of a test component, timer expiration.

**Timer management**
Timers can be started, stopped. The actual value of a timer can be read as well whether a given timer is running. The expiration of a timer can be checked.

**Verdict management**
Test verdict can be pass, fail, inconclusive, none or error. The final verdict is determined with regard to the outcome of each test step.

**Abstract communication**
Between the test executor system and the implementation under test there are two different communication possibilities. Message based communication is asynchronous while procedure based communication is synchronous. There is communication also between components.

**Concurrency**
Parallel test components (PTCs) are working concurrently, they can be created and destroyed.

**Test specific constructions: alt, interleave, default, altstep**
…are used to specify message reception behavior
TEST ARRANGEMENT AND ITS TTCN-3 MODEL

Test System | SUT | IUT
---|---|---
PCO | Network | Test Port
SAP | ASPs | ASPs

MTC

IUT

Network

System

Port
The principal building blocks of TTCN-3 are modules.
The module definitions part specifies the top-level definitions of the module and may import identifiers from other modules. TTCN-3 does not support the definition of variables in the module definitions part. This means that global variables cannot be defined in TTCN-3.
The module control part may contain local definitions and describes the execution order of the actual test cases. A test case shall be defined in the module definitions part and called in the control part.
General syntax rules describe the file format, capitalisation, delimiters, identifiers etc.
The module parameter list defines a set of values that are supplied by the test environment at run-time. During test execution these values shall be treated as constants. Module parameters shall be defined within the module definition part only.
Keywords are listed in table A.3 of the ETSI standard 201 873-1. These words must not be used as identifiers.

Identifiers are case sensitive and may only contain lowercase letters (a-z) uppercase letters (A-Z) and numeric digits (0-9). Use of the underscore ( _ ) symbol is also allowed. An identifier shall begin with a letter.

Comments written in free text may appear anywhere in a TTCN-3 specification.
A test suite consists of one or more modules. There is no hierarchy between modules. Modules are written as free text files: line breaks or paragraph marks may be used without restrictions. A module consists of a (optional) definitions part, and a (optional) module control part. Usually, the definitions part is longer, the control part only states the execution order of the test cases. Module parameters are supplied to the module at run-time and are considered constant during test execution. Module attributes give additional information, like coding rules or the size of a table.

The beginning of a module is indicated in the header by the keyword "module" followed by the module name (here: modulename ). Thereafter between curly brackets appears the definitions part followed by the control part. Module attributes (here: the encoding rule valid for the whole module) may be given after the closing curly bracket of the module. Attributes are introduced by the keyword "with" whereas the attributes themselves are listed between curly brackets.
Module Parameters are supplied by the test environment at run-time and are treated as constants during test execution.

Data Types: a common name for simple basic types, basic string types, structured types, the special data type and all user defined types based on them.

Procedure Signatures (or signatures for short) are needed for procedure-based communication.

Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification. A template can be thought of as being a set of instructions to build a message for sending or to match a received message. Message Templates are used over message based ports, whereas Signature Templates are used over procedure based ports.

Test components are connected via their Communication Ports. Each port is modelled as an infinite FIFO queue which stores the incoming messages or procedure calls until they are processed.

Test Components are the owner of the ports. Each test component has a unique reference created during the execution of a test case.

Altsteps are special functions used to specify and structure test behaviour.

Test Cases are functions running on MTC and returning the result of the test ("verdict").
The module control part manages the execution of the test cases. In the module control part the `execute` statement is used to start test cases. Program statements may be used in the control part of a module to specify such things as the order in which the test cases are to be executed or the number of times a test case may be run. Variables, timers etc. (if any) defined in the control part of a module are only locally visible, i.e., they shall be passed into the test case by parameterization when required.

As the result of the execution of a test case a test case verdict of either none, pass, inconclusive, fail or error shall be returned.
Modules can import definitions from any module. Identifiers imported from other modules are globally visible throughout the importing module. It is possible to import to various extent:

- single definitions;
- groups of definitions;
- all templates, functions and types;
- all definitions.

The default import mechanism imports referenced definitions without their identifier. A recursively imported definition, in turn, is imported together with all referenced definitions, i.e. the identifier of all referenced definitions becomes visible and usable in the importing module.
It is possible to re-use definitions specified in different modules using the `import` statement. An import statement can be used anywhere in the module definitions part. It shall not be used in the control part.

TTCN-3 supports the import of the following definitions: module parameters, user defined types, signatures, constants, external constants, data templates, signature templates, functions, external functions, altsteps and test cases.

The rules of importing are depicted in the chapter 7.5 of ETSI standard ES 201 873-1.

Legend: the import options preceded by comments in red are not implemented in the TITAN environment.
It is possible to re-use definitions specified in different modules using the **import** statement. An import statement can be used anywhere in the module definitions part. It shall not be used in the control part.

TTCN-3 supports the import of the following definitions: module parameters, user defined types, signatures, constants, external constants, data templates, signature templates, functions, external functions, altsteps and test cases.

The rules of importing are depicted in the chapter 7.5 of ETSI standard ES 201 873-1.

Legend: the import options preceded by comments in red are not implemented in the TITAN environment.
This classical example illustrates how many definitions should be made to complete a module.

The main point is the testcase called `HelloW`. The message is sent over the port `My_PCO` defined previously.

The port, component, testcase definition form the module definitions part followed by the module control part.
IV. TYPE SYSTEM

OVERVIEW
BASIC AND STRUCTURED TYPES
VALUE NOTATIONS
SUB-TYPING

CONTENTS
TTCN-3 supports a number of predefined basic types. These basic types include ones normally associated with a programming language, such as integer, boolean and string types, as well as some TTCN-3 specific ones such as objid and verdicttype. Structured types such as record types, set types and enumerated types can be constructed from these basic types.

User-defined type is defined by subtyping of a basic type, defining a structured type or constraining the anytype to a single type by the dot notation.

Definitions in the module definitions part may be made in any order but forward references should be avoided for readability reasons.

Sub-types are user-defined types formed from simple basic and basic string types using lists, ranges and length restrictions.

Parameterisation: all user-defined type definitions support static value parameterization (i.e. this parameterization shall be resolved at compile-time); template, signature, testcase, altstep and function support dynamic value parameterization (i.e. this parameterization shall be resolvable at run-time).

Type compatibility: TTCN-3 is not strongly typed. For non-structured variables, constants, templates etc. the value "b" is compatible to type "A" if type "B" resolves to the same root type as type "A" and it does not violate subtyping (e.g. ranges, length restrictions) of type "A". In the case of structured types (except the enumerated type, that is never compatible with other basic or structured types) a value "b" of type "B" is compatible with type "A" if the effective value structures of type "B" and type "A" are compatible. The communication operations are exceptions to the weaker rule of type compatibility and require strong typing.
**Integer**: a type with distinguished values which are the positive and negative whole numbers, including zero.

**Float**: a type to describe floating-point numbers. Floating point numbers are represented in TTCN-3 as: `<mantissa> × 10<exponent>`.

**Boolean**: a type consisting of two distinguished values: true, false.

**Objid**: a type whose distinguished values are the set of all object identifiers conforming to clause 6.2 of ITU-T Recommendation X.660.

**Verdicttype**: a type for use with test verdicts consisting of 5 distinguished values.
**Bitstring:** a type whose distinguished values are the ordered sequences of zero, one, or more bits.

**Hexstring:** a type whose distinguished values are the ordered sequences of zero, one, or more hexadecimal digits, each corresponding to an ordered sequence of four bits.

**Octetstring:** a type whose distinguished values are the ordered sequences of zero or a positive even number of hexadecimal digits (every pair of digits corresponding to an ordered sequence of eight bits).
Universal charstring: The "quadruple" is capable to denote a single character and denotes the character by the decimal values of its group, plane, row and cell according to ISO/IEC 10646.
**SPECIAL TYPES (1)**

- **anytype**
  - Introduced to allow mapping of CORBA IDL to TTCN-3;
  - Defined as a shorthand for the union of all known types in a TTCN-3 module, where known type embraces all built-in types, user-defined types, imported ASN.1 and other imported external types.
  - The fieldnames of the anytype shall be uniquely identified by the corresponding type names using the “dot” notation.
  - Performance problems – not to use unless explicitly necessary!
    - all used types must be listed at the end of the module
    - with {extension “anytype ...”}

```idl
module my_Module {
  type integer money;
  type record MyRec {
    integer i,
    float f
  }
  control {
    var anytype v_any;
    v_any.integer := 3;
    // ischosen(v_any.integer) == true
    v_any.MyRec := {{ 1,true} }
  }
  with {
    extension “anytype charstring, MyRec”
  }
}
```

CORBA Common Object Request Broker Architecture

IDL Interface Description Language

The specification of CORBA IDL can be read by following the Uniform Resource Locator:
http://www.omg.org/technology/documents/idl2x_spec_catalog.htm

```idl
module my_Module {
  type integer money;
  type record MyRec {
    integer i,
    float f
  }
  control {
    var anytype v_any;
    v_any.integer := 3;
    // ischosen(v_any.integer) == true
    v_any.charstring := “three”;  
  }
  with {
    extension “anytype integer, charstring” // adds two fields
    extension “anytype MyRec” // adds a third field
    extension “anytype money” // adds the money type
  }
}
```
Address shall only be used in receive and send operations of ports mapped to test system interface. Only one definition of type address may exist in a test suite.

SUT: System Under Test

Each port type definition shall have list(s) indicating the allowed collection of message types and/or procedures together with the allowed communication direction.

Component definitions shall be made in the module definitions part. It is possible to define constants, variables and timers local to a particular component.
Received messages are usually examined in an alt statement. When no branch of the alt matches the received message, the previously activated default(s) are examined. It is possible to have several defaults activated at same time and deactivate them one by one.
OVERVIEW OF STRUCTURED TYPE SYNTAX

• General syntax of structured type definitions:

  type <kind> [element-type] <identifier> [ { body } ];

• kind is mandatory, it can be:
  record, set, union, enumerated, record of, set of

• element-type is only used with record of, set of

• body is used only with record, set, union, enumerated;
  it is a collection of comma-separated list of elements

• Elements consist of <field-type> <field-id> [optional]
  except at enumerated

• element-type and field-type can be a reference to any basic or user-defined data type or an embedded type definition

• field-ids have local visibility (may not be globally unique)
In the above example, "type" of the elements is integer or boolean, their "identifier" is field1 or field2. The same identifiers may be used in both record and set, because it is not mandatory to use globally unique names.

Optional elements may or may not be present when assigning value to the constructs. A record or a set may be an element of another record or set.
The main difference between record and set is the following: elements of a record must be referenced in the same order as defined, whereas elements of a set may be referenced in arbitrary order. In other words, the ordering of the set fields is not significant.
Value notation: notation by which an identifier is associated with a given value or range of a particular type

Assignment notation: in the curly brackets following the name of the record or set, the element identifier must be present to designate which element is the value is assigned to. It is important to know that every identifier of the record or set must be listed. Omitted optional elements must be given the value "omit" otherwise its value remains undetermined (unbound), resulting in run-time error.
Value-list notation: in the curly brackets following the name of the record, values of the elements are listed one by one. Every identifier of the record must be listed. Omitted optional elements must be given the value "omit" otherwise its value remains undetermined (unbound), resulting in run-time error. In contrast to value assignment notation, all elements must appear inside the initializer. Application of the hyphen (-) leaves the corresponding field unchanged. Attention! Such a field is unbound unless it has been given a value earlier. It is not allowed to mix value-list notation and assignment notation in the same context! The not-used symbol is only valid in value-list notation.
STRUCTURED TYPES – NESTED VALUES

```haskell
type record InternalType {
    boolean field1,
    integer field2 optional
};
type record RecType {
    integer field1,
    InternalType field2
};
const RecType c_rec := {
    field1 := 1,
    field2 := {
        field1 := true,
        field2 := omit
    }
};
// same as previous, but with value list
const RecType c_rec2 := { 1, { true, omit } }
```
FIELD REFERENCES

- Reference or "dot" notation
  - Can not be used at specification, only for previously defined variables
  - Referencing structured type fields
  - Applicable in dynamic parts (e.g. function, control) only

```plaintext
v_myRecord2.field1 := omit;
v_mySet1.field1 := v_myRecord2.field1;
```

```plaintext
type record R1 {
  integer i,
  boolean b
}
type record R2 {
  R1 r1,
  integer i2
}"
```
Union type is useful to model a structure which can take one of a finite number of known types.
For the **union** type, assignment notation and dot notation may be used. (First, respective second row in the example on the middle of the slide.) Value-list notation (listing of element values without their identifiers) must not be used.
The only difference between record of and set of appears when comparing them. Two records of are only equal when they contain the equal elements in the same order. Two sets of are equal if there is exactly one pair for each element.

These records and sets can be considered similar to an ordered array and an unordered array respectively.
STRUCTURED TYPES – NESTED TYPES

• Similarly to other notations (e.g. ASN.1) TTCN-3 type definitions may be nested (multiple times)
• The embedded definition have no identifier associated

```c
// nested type definition:
// the inner type "set of integer" has no identifier
type record of set of integer OuterType;

// ...could be replaced by two separate type definitions:
type set of integer InnerType;
type record of InnerType OuterType;
```
When indexing a string type element, index corresponds to different units of length in function of the string type. A bitstring is indexed by bits, a hexstring by hexadecimal digits, an octetstring by octets and finally a character string by characters.
NOTE1: The comments at the assignment examples of r2 and r3 might be misleading: an unbound value never can be a right-hand-side value, not even for relational operators! It causes a run time error!

NOTE2: Just for convenience: the typedefs from one of the earlier slides:

```c
// example record type def.
type record MyRecordType {
    integer field1 optional,
    boolean field2
}

// example set type def.
type set MySetType {
    integer field1 optional,
    boolean field2
}
```
For each enumeration without an assigned integer value, the system successively associates an integer number in the textual order of the enumerations, starting at the left-hand side, beginning with zero, by step 1 and skipping any number occupied in any of the enumerations with a manually assigned value. These values are only used by the system to allow the use of relational operators.
Although the TTCN-3 standard does not require it, it is a good practice to begin user-defined type names with uppercase letters and to use lowercase letters as the first letter of element, variable and constant names. That's why weekdays are written in small letters violating English orthography.

Comparison is only possible between two elements of the same enumeration type.
One way to create user-defined types is sub-typing a basic type. (The two other ways already discussed are defining a structured type or constraining the anytype to a single type by the dot notation.) By sub-typing the value set of the original type is restricted to certain values. In case of string types also the length of the string can be restricted. Mathematically spoken, the set $D(\text{New})$ is the proper subset of set $D(\text{basic})$ and has the same type as the original basic type.

Universal charstring / charstring types can be sub-typed with patterns (not supported in TITAN yet, as of v1.6.pl3 (R6D))
TTCN-3 permits the specification of a range of values of type integer, charstring, universal charstring and float. The lower boundary and the upper boundary are included in the range of permitted values. In the case of charstring and universal charstring types, the boundaries mean character positions according to the coding rules of the respective character set.

The keyword infinity may be used in order to specify an infinite integer or float range.
The subtype defined by this list enumerated in parentheses restricts the allowed values of the subtype to those values in the list. The values in the list shall be of the root type and shall be a true subset of the values defined by the root type.

For values of type integer, charstring, universal charstring and float it is possible to mix lists and ranges. Within charstring and universal charstring subtype definitions, lists and ranges shall not be mixed in the same subtype definition. For values of type bitstring, hexstring, octetstring it is possible to mix lists and length restrictions.

Note: in sub-typing we use parentheses around the value list, while in value-notation we use curly braces around the value lists.
For the upper bound the keyword *infinity* may also be used to indicate that there is no upper limit for the length. The upper boundary shall be greater than or equal to the lower boundary. The lower boundary and the upper boundary are included in the range of permitted values.

Length restriction can only be either a concrete number or a range. Other (e.g. value list) not allowed

type octetstring MyOct length(4 .. 8, 11);
type octetstring MyOct length(4 , 8);
Both wrong
According to table 3 in chapter 6.0 of ETSI ES 201 873-1 V2.2.1 length restriction of the structured types `record of` and `set of` is considered as sub-typing. Chapter 6.2.0, on the other hand, only allows sub-typing of on simple basic and basic string types.
**SUB-TYPING: PATTERNS**

- *charstring* and *universal charstring* types can be restricted with patterns (→ *charstring value patterns*)
- All values denoted by the pattern shall be a true subset of the type being sub-typed

```cpp
// all permitted values have prefix abc and postfix xyz
type charstring MyString (pattern "abc*xyz");
// a character preceded by abc and followed by xyz
type charstring MyString2 (pattern "abc?xyz");
// all permitted values are terminated by CR/LF
type charstring MyString3 (pattern "*\r\n")
```

```cpp
// causes an error because MyString does not contain a value starting with character 'd'

type MyString MyString3 (pattern "d*xyz");
```

---

*type charstring* MyString2 (pattern "abc?q(0,0,1,113)");
/* causes an error because a universal char {0,0,1,113} is not allowed in the charstring type */

// all permitted universal string values are terminated by CR/LF
*type universal charstring* MyUString (pattern ""\r\n")
Type aliasing is defined in TTCN-3 BNF only, but it is implemented in TITAN.
NOTE:
List subtyping of the types "record", "record of", "set", "set of", "union", "enumerated", "anytype" are possible when defining a new constrained type from an already existing parent type but not directly at the declaration of the first parent type.
Type compatibility is a language feature, which allows to use values or templates of a given type as actual values of another type (e.g. at assignments, as actual parameters at calling a function, referencing a template etc. or as a return value of a function).

An example for type compatibility of structured types is given in chapter 6.7.2 of ETSI ES 201 873-1.

```plaintext
int2char(65) == "A" // ASCII(65): letter A
int2str(65) == "65"
hex2str('FABABA' H) == "FABABA"
```
Conversion functions span the gap between different simple variable types.
A function at the intersection of a given column and a row has an input parameter indicated in the column header and returns the value type indicated in the row header.

The detailed description of predefined functions is given in annex C of the ETSI standard ES 201 873-1.

Green letters indicate TITAN extensions, not included in the standard.

Difference between functions with 'str' and 'char' in their names is explained with the following examples:

int2char (66) = "B", int2str (66) = "66".
V. CONSTANTS, VARIABLES, MODULE PARAMETERS

CONSTANT DEFINITIONS
VARIABLE DEFINITIONS
ARRAYS
MODULE PARAMETER DEFINITIONS

CONTENTS
Constants defined in module definitions part are globally (anywhere in the module) visible. Those defined in the module control part, test cases, functions and altsteps are only locally (within the same scope unit) visible. The ones defined in component type definitions are visible in functions, test cases and altsteps referencing that component type by a `runs on`-clause.

No forward referencing allowed in constant definitions except in module definition part.
Both assignment notation and the short-hand value list notation may be used when assigning value to a constant.
Variables defined in the module control part, test cases, functions and altsteps are only locally (=within the same scope unit) visible. The ones defined in component type definitions are visible in functions, test cases and altsteps referencing that component type by a runs on-clause. An initial value may be assigned to the variable.

The naming convention (ETH/R-04:000010 Uen rev. A) generally requires that the variable names should be prefixed by 'v'. However, the prefix may be omitted for non-protocol related variables like loop counters, for loop control variables, variables used in calculations etc.
Forward references shall never be made inside the module control part, test case definitions, functions and altsteps. This means forward references to local variables, local timers and local constants shall never occur.

Although initial value assignment is optional, a variable defined must receive a value assigned somewhere in the program, otherwise a reference to it results in run-time error (reference to an unbound value).

In the last example, v_myInt1 remains unbound, while v_myInt2 has the value \(2 \cdot c\_myConst = 6\).
It is important to realize that a single figure in brackets specifies the number of elements (=array dimension). When a range is given, however, the two figures give the lower respective the upper index value.

In the first case, the maximum index value is one less then the figure indicated in the brackets; in the latter case, the maximum index value equals to the last figure indicated in brackets.
A multidimensional array may be replaced by nested `record` of types. The number of `record` of types equals to the number of indices of the array. The length of the individual records correspond to the value of the array indices.
The module parameter list defines a set of values that are supplied by the test environment at run-time. During test execution these values shall be treated as constants. Module parameters are defined by listing their identifiers and types following the keyword `modulepar`. Module parameters shall be defined within the module definition part only. Redefinition of module parameters is not allowed.

It is allowed to specify default values for module parameters.
The **scope unit** is the region of the TTCN-3 source within which (constant, timer, variable, etc.) definitions may have effect, within which multiple definitions of the same name are prohibited, and outside of which definitions inside the unit do not have effect.

Definitions made in the **module definition part** but outside of other scope units are globally visible in the module. So are imported identifiers.

Definitions made in the **module control part** have local visibility, i.e. can be used within the control part only.

Definitions made in a test **component type** may be used only in functions, test cases and altsteps referencing that component type by a runs on-clause.

**Functions**, **altsteps** and **test cases** are individual scope units without any hierarchical relation between them, i.e. definitions made at the beginning of their body have local visibility.

Definitions within **block of statements** (e.g. for, if-else, while, do-while, alt, interleave) have local visibility within the statement concerned.
VISIBILITY MODIFIERS

• On module level
  - public  definition is visible in every module importing the module. (default)
  - private the definition is only visible within the same module.
  - friend  the definition is only visible within the friend declared module.

```plaintext
module module1
{
friend module module2;
type integer module2Type;
public type integer module2TypePublic;
friend type integer module2TypeFriend;
private type integer module2TypePrivate;
} // end of module
```

```plaintext
module module2
{
import from module1 all;
const module2Type c_m2t := 1;
//OK, type is implicitly public
const module2TypePublic c_m2tp := 2;
//OK, type is explicitly public
const module2TypeFriend c_m2tf := 3;
//OK, module1 is friend of module2
const module2TypePrivate c_m2tpv := 4;
//NOK, module2TypePrivate is private to module2
```
VI. PROGRAM STATEMENTS AND OPERATORS

EXPRESSIONS
ASSIGNMENTS
PROGRAM CONTROL STATEMENTS
OPERATORS
EXAMPLE

CONTENTS
**Basic program statements** can be used in the module control part, functions, altsteps and test cases.

**Expressions** are specified using the operators shown on the following two slides.

An **assignment** binds the variable on the left side to the value of the expression on the right side.

**Logging** enables to write a string or a variable value to a log file in an implementation dependent manner.
An **if-else** statement is used to denote branching in the program execution based on a Boolean expression (condition).

The **select-case** statement permits branching based on the calculated value of an expression. The statement block of the first branch containing a matching template inside its **case** is executed. The statement block of the **case else** is run when none of the cases match.

The select case statement is an alternative to using if .. else if .. else statements when comparing a value to one or several other values. The statement contains a header part and zero or more branches. **Never more than one of the branches is executed.**

The **for** statement defines a counter **loop**. The first statement (init) is used to initialize the counter variable. If the Boolean expression (cond) is true, the loop terminates. The second assignment (expr) is used to manipulate (increase or decrease) the index variable.

A **while loop** is executed as long as the loop condition holds.

The **do while loop** is identical to a **while** loop with the exception that the loop condition shall be checked at the end of each loop iteration. This means that the instruction is executed at least once.

**Label definition** allows the specification of labels (a specific place in the program code).

**Jump to a label** performs a jump to a previously defined label.

Used in the control part of a module, the **stop** statement terminates the **execution** of the module control part. When used in a test case, altstep or function with runs on clause, it terminates the relevant test component.
break AND continue

• **break**
  - Leaves innermost loop
  - or alternative within `alt` or `interleave` statement
• **continue**
  - Forces next iteration of innermost loop

• **continue**
  Forces next iteration of innermost loop
  Not for taking new snapshot in `alt` or `interleave` statement -> repeat
Operands of arithmetic operators shall be of type integer or float, except for `mod` and `rem` which shall be used with integer types only. The result is of the same type as the operands, operands must not have different types. Both `mod` and `rem` have the same result for positive arguments but they differ for negative ones. See Table 7 in 7.1.1 in ETSI ES 201 873-1 V4.4.1 (2012-04).

The operators `rem` and `mod` compute on operands of type integer and have a result of type integer. The operations `x rem y` and `x mod y` compute the rest that remains from an integer division of `x` by `y`. Therefore, they are only defined for non-zero operands `y`. For positive `x` and `y`, both `x rem y` and `x mod y` have the same result but for negative arguments they differ.

Formally, `mod` and `rem` are defined as follows:

\[
x \text{ rem } y = x - y \times \left(\frac{x}{y}\right) \]
\[
x \text{ mod } y = x \text{ rem } |y| \quad \text{when } x \geq 0
\]
\[
= 0 \quad \text{when } x < 0 \text{ and } x \text{ rem } |y| = 0
\]
\[
= |y| + x \text{ rem } |y| \quad \text{when } x < 0 \text{ and } x \text{ rem } |y| < 0 \text{ ETSI}
\]

Effect of `mod` and `rem` operator

\[
\begin{array}{c|c|c|c|c|c|c}
 x & -3 & -2 & -1 & 0 & 1 & 2 & 3 \\
 x \text{ mod } 3 & 0 & 1 & 2 & 0 & 1 & 2 & 0 \\
 x \text{ rem } 3 & 0 & -2 & -1 & 0 & 1 & 2 & 0 \\
\end{array}
\]

**Concatenation** is performed from left to right on compatible string types. The result type is the root type of the operands. The relational operators `equal` and `not equal` may be applied on all compatible types. All other relational operators shall have only operands of type integer, float, or instances of the same enumerated types. The result type of these operations is `boolean`. 

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
<th>Format</th>
<th>Type of operands and result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetical</td>
<td>Addition</td>
<td><code>+op_1 op_2</code></td>
<td><code>op_1, op_2, result: integer, float</code></td>
</tr>
<tr>
<td></td>
<td>Subtraction</td>
<td><code>-op_1 op_2</code></td>
<td><code>op_1, op_2, result: integer, float</code></td>
</tr>
<tr>
<td></td>
<td>Multiplication</td>
<td><code>op_1 * op_2</code></td>
<td><code>op_1, op_2, result: integer</code></td>
</tr>
<tr>
<td></td>
<td>Division</td>
<td><code>op_1 / op_2</code></td>
<td><code>op_1, op_2, result: integer</code></td>
</tr>
<tr>
<td></td>
<td>Modulo</td>
<td><code>op_1 mod op_2</code></td>
<td><code>op_1, op_2, result: integer</code></td>
</tr>
<tr>
<td></td>
<td>Remainder</td>
<td><code>op_1 rem op_2</code></td>
<td><code>op_1, op_2, result: integer</code></td>
</tr>
<tr>
<td>String</td>
<td>Concatenation</td>
<td><code>op_1 &amp; op_2</code></td>
<td><code>op_1, op_2, result: *string</code></td>
</tr>
<tr>
<td>Relational</td>
<td>Equal</td>
<td><code>op_1 == op_2</code></td>
<td><code>op_1, op_2, all; result: boolean</code></td>
</tr>
<tr>
<td></td>
<td>Not equal</td>
<td><code>op_1 != op_2</code></td>
<td><code>op_1, op_2, all; result: boolean</code></td>
</tr>
<tr>
<td></td>
<td>Less than</td>
<td><code>op_1 &lt; op_2</code></td>
<td><code>op_1, op_2, integer, float, enumerated</code></td>
</tr>
<tr>
<td></td>
<td>Greater than</td>
<td><code>op_1 &gt; op_2</code></td>
<td><code>op_1, op_2, integer, float, enumerated</code></td>
</tr>
<tr>
<td></td>
<td>Less than or equal</td>
<td><code>op_1 &lt;= op_2</code></td>
<td><code>op_1, op_2, integer, float, enumerated</code></td>
</tr>
<tr>
<td></td>
<td>Greater than or equal</td>
<td><code>op_1 &gt;= op_2</code></td>
<td><code>op_1, op_2, integer, float, enumerated</code></td>
</tr>
</tbody>
</table>
The operands and the result of logical operations shall be of type boolean.
The bitwise operators perform the operations of bitwise not, bitwise and, bitwise or and bitwise xor. The unary operator not4b inverts the individual bit values of its operand. The operands shall be of type bitstring, hexstring or octetstring. The result type shall be the root type of the operands.

**Shift** operators perform the shift left and shift right operations. Their left-hand operand shall be of type bitstring, hexstring or octetstring. Their right-hand operand shall be of type integer and its value of e.g. 1 means a shift of one bit, one hexadecimal digit and one octet, respectively, according to the three possible left-hand operand types. The result type shall be the same as that of the left operand.

**Rotate** operators perform the rotate left and rotate right operations. Their left-hand operand shall be of type bitstring, hexstring, octetstring, charstring or universal charstring. Their right-hand operand shall be of type integer and its value of e.g. 1 means a rotate of one bit, one hexadecimal digit, one octet and one character, respectively, according to the possible left-hand operand types. The result type shall be the same as that of the left operand.
### OPERATOR PRECEDENCE

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator type</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td><em>parentheses</em></td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>Unary</td>
<td>+, -</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>*, /, mod, rem</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>+, -, &amp;</td>
</tr>
<tr>
<td></td>
<td>Unary</td>
<td>not &amp;b</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>and &amp;b</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>xor &amp;b</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>or &amp;b</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>&lt;&lt;, &gt;&gt;, &lt;&amp;, @&gt;</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>==, !=</td>
</tr>
<tr>
<td>Lowest</td>
<td>Unary</td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>xor</td>
</tr>
<tr>
<td></td>
<td>Binary</td>
<td>or</td>
</tr>
</tbody>
</table>

Note: The assignment symbol `:=`, structure field symbol `.`, function calling `()`, indexing `[]` are not operators!
Is the value of j is less than \( pl_y \), then x will get the value of j multiplied by the parameter \( pl_y \), otherwise it will have the value of three times j. The value x will only be converted to a character string and logged when the flag equals true.

The procedure described above will be executed in a for loop. The number of executions is controlled by the value of the parameter \( pl_i \).

The whole process is called in a function (f_MyFunction). The function has two parameters: \( pl_y \) sets the multiplication factor of j, while \( pl_i \) controls how many times the calculation is repeated.
VII. TIMERS

TIMER DECLARATIONS
TIMER OPERATIONS

CONTENTS
Timers can be defined and used in the module control part, test cases, functions and altsteps. Additionally, timers can be defined in component type definitions. These timers can be used in test cases, functions and altsteps which are running on the given component type.
When starting a timer, the optional timer value parameter shall be used if no default duration is given, or if it is desired to override the default value specified in the timer definition. When a timer duration is overridden, the new value applies only to the current instance of the timer, any subsequent start operation for this timer, which do not specify a duration, shall use the default duration.

The start operation may be applied to a running timer, in which case the timer is stopped and re-started.
The **timeout** operation allows to check expiration of a timer, or of all timers, in a scope unit in which the timeout operation has been called. The **timeout** shall not be used in a **boolean** expression, but it can be used to determine an alternative in an **alt** statement.
The **stop** operation is used to stop a running timer. The elapsed time of a stopped timer is set to the float value zero (0.0). An already stopped timer may be stopped again, although it does not have any effect.

RTE: Run Time Environment
The **running** timer operation is used to check whether a timer has been started and has neither timed out nor been stopped.

The **read** operation is used to retrieve the time that has elapsed since the specified timer was started. The operation returns a value of type **float**. Applying the **read** operation on an inactive timer will return the value zero.
VIII. TEST CONFIGURATION

TEST COMPONENTS AND COMMUNICATION PORTS
TEST COMPONENT DEFINITIONS
COMMUNICATION PORT DEFINITIONS
EXAMPLES

CONTENTS
The abstract test configuration consists of **components**. The components are interconnected by means of **ports**. In the course of the test, the components themselves may emerge and disappear, their interconnection vary, in other words, the test configuration is dynamic.

The tested implementation (IUT, Implementation Under Test) is considered a black box, i.e., its internal structure is hidden from the tester. A special test component, called the test system interface (or System for short) interfaces the ports of the real world to the abstract world of components.
In most of the cases Tester behaves as a peer entity of the IUT/SUT

Main Test Component (mtc)
System Component (system)
mtc and system are of the same type
The Implementation Under Test (IUT) is usually located inside the System Under Test (SUT). The test system is connected to the SUT through a Network. The connection points between the IUT and the Network respective between the test system and the network are called Service Access Points (SAPs).

Communication between the Abstract Test System Interface (mapping the Real Test System Interface to the abstract world) and the Test Components is carried in Abstract Service Primitives (ASPs). ASP is an implementation-independent description of an interaction between the test system and the SUT. ASPs are usually described in the specification of the tested protocol.

Communication within the test system (between the components) runs through associated ports. The association between components (on the slide: Parallel Test Components [PTCs] and the Main Test Component [MTC]) is called connection and is set up using the `connect` keyword. The association between components and the Abstract Test System Interface is called mapping and is set up using the `map` keyword.
GRAPHICAL REPRESENTATION OF COMPONENTS AND PORTS

- vc_myComp is the component reference identifying this particular component.
- comtype_CT is the component type definition, which among others specifies the types of P1_PCO and P2_PCO.
- This is a port instance in vc_myComp with the name P1_PCO; its type is defined in a separate port type definition.
- P2_PCO is a port instance in vc_myComp; its type is defined in a separate port type definition.
- This is a component instance of type comtype_CT.
The components are interconnected via test ports. TTCN-3 defines the port communication model through which messages are exchanged (message based ports) or procedures are called (procedure based ports). The interconnection is called mapping between System and components and connecting between components.
Information (messages, procedure calls or both) are exchanged between associated communication ports of the components. Internal (component-to-component) communication happens between connected ports whereas external (component-to-system) communication happens between mapped ports.

Ports are bidirectional, but have a list enumerating the allowed messages together with their direction (in, out, inout).

The infinite FIFO queue stores the incoming messages or procedure calls until they are processed by the component owning that port. A queue overflow (in a real implementation a queue is never infinite) is treated as a test case error.
When defining a **message** based **port type**, the messages allowed to pass that port must be listed together with their direction. When defining a **procedure** based **port type**, the procedure signatures allowed must be listed. A **mixed** port a shorthand notation for two ports, i.e. a message-based port and a procedure-based port with the same name. The **attributes** defined with the keyword **with** may define e.g. the coding rules used for the messages passing the port. Such a rule may be for example whether the most or the less significant bit should be sent first through the port.
A message based port is defined by enumerating the allowed message types together with their direction.
The abstract test configuration consists of **components**. The components are interconnected by means of **ports**. In the course of the test, the components themselves may emerge and disappear, their interconnection vary, in other words, the test configuration is dynamic.

Within every test configuration there shall be one (and only one) main test component (MTC) created automatically at the start of each test case execution.

Parallel test components (PTCs) can dynamically be created during execution of a test case by the explicit use of the `create` operation.

The tested implementation (IUT, Implementation Under Test) is considered a black box, i.e., its internal structure is hidden from the tester. A special test component, called the test system interface (or System for short) interfaces the ports of the real world to the abstract world of components.
A test configuration consists of a set of inter-connected test components with well-defined communication ports.

Test **component type** definitions shall be made in the module definitions part. The actual configuration of components is achieved by performing `create` operations within the test case behavior.

The component type defines which ports are associated with a component. The port names in a component definition are local to that component i.e. another component may have ports with the same names.

It is possible to define constants, variables and timers local to a particular component.

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

It does not make sense to define timers, variables or constants in the system component as the latter serves as an image of the physical world.
The component type MyComponentType_CT owns a port called PCO and a port array PCO_Array containing 10 ports of type MyPortType_PT.

In each component instance of this type local copies of the ports, the variable (v_MyVar) and the timer (T_MyTimer) are generated, and the constant (c_MyConst) will be visible.
IX. FUNCTIONS AND TESTCASES

OVERVIEW OF FUNCTIONS
FUNCTION DEFINITIONS
PARAMETERIZATION
PREDEFINED FUNCTIONS
TESTCASE DEFINITIONS
VERDICT HANDLING
CONTROLLING TEST CASE EXECUTION

CONTENTS
In TTCN-3, functions are used to specify and probe behavior and to structure computation in a module.

Usually, a function is defined in TTCN-3 (using the keyword `function`) but may be defined as an external function (using the keyword `external`) implemented in one or more C++ source files.

A function must be defined with reference to a component ("runs on") if the function uses variables, constants, timers and ports that are defined in a component type definition.

Parameter passing mechanism (by value or by reference) can be chosen for each parameter separately. Parameters passed by value are read-only parameters. Those passed by reference may even be altered by the function.
The function header:
• contains the list of formal parameters of the function. When no parameters are used, empty brackets must be written;
• the usually optional runs on clause must be present if the function uses variables, constants, timers and ports that are defined in a component type definition;
• the keyword return is only used if the function returns a parameter. A function can only return a single value of a given type.

The local definitions are optional. When present, the constants, variables and timers defined here are only visible within the function.

The keyword return must conclude the program part. It must be followed by an expression resulting in the same type as defined in the header when the return keyword was used in the header. Notice that the bold and underscored “return” keyword has two different meanings!
The formal parameters of the function \texttt{f\_MyF\_1} are \texttt{pl\_1} and \texttt{pl\_2}. Their types are \texttt{integer} and \texttt{boolean}, respectively. When invoking the function, the actual parameter list contains the parameters of the corresponding type in the same order as defined.

By the way: the program part of the function defined is empty, in other words, the function does not do anything.

The formal parameter list of the function \texttt{f\_MyF\_2} is empty thus it is invoked with two brackets after the function name standing for an empty parameter list. The program always return the integer value 28 (see the code between the curly brackets). The returned values is of integer type (cf. the function definition) and that’s why it can be assigned to the variable \texttt{v\_two}, the latter being of the same type.
Functions with a return value may be invoked in expressions. On the slide above, the function \( f_3 \) returns the value 2 if the parameter is true, otherwise the value returned will be 0.

The first summand has the value of two times two, the second summand equals zero, thus, the variable \( i \) results in four.

The function \( f_4 \) is defined with reference to a component (MyCompType_CT) because it makes use of the ports having been defined in that component.
By default, parameters are passed by value (optionally denoted by the keyword `in`). To pass parameters by reference, the keywords `out` or `inout` shall be used.

- **In** parameters may only be read inside the parameterized function, i.e., the parameter is only allowed on the right-hand side of an assignment.
- **Out** parameters may only be written inside the parameterized function, i.e., the parameter is only allowed on the left-hand side of an assignment.
- **Inout** parameters may only be both read and written inside the parameterized function, i.e., the parameter is only allowed on the both sides of an assignment.
DEFAULT VALUES

• in parameters may have default values
• at invocation
  – “-” (hyphen) skips the parameter with default value
  – simply leaving out (if it is the last, or all the following have default values)
  – default value may be overwritten

```c
function f_MyFDef (integer i, integer j:=2, integer k){}
function f_MyFDef2 (integer i, integer j:=2, integer k:=3){}

// invocation
f_MyFDef(1,-,3); // f_MyFDef(1,2,3);
f_MyFDef(1,5,3); // f_MyFDef(1,5,3);
f_MyFDef2(1,5,7); // f_MyFDef2(1,5,7);
f_MyFDef2(1,5); // f_MyFDef2(1,5,3);
f_MyFDef2(1); // f_MyFDef2(1,2,3);
```
The functions `lengthof` resp. `sizeof` give the length of a string respective the number of elements in the referenced constructed type.

The functions `regexp` and `substr` return a specific part of the referenced string.

The function `ischosen` returns the Boolean value `true` if the element given in the parameter is selected in the union. The parameter contains the the reference to the union element in dot notation format.

The function `ispresent` returns the Boolean value `true` if the optional field given in the parameter is present in the record or set. The parameter contains the the reference to the record or set field in dot notation format.

The `rnd` function returns a pseudorandom float number $r$ where $1 > r \geq 0$. The function may optionally be initialized by a seed value. The same seed value results in the same sequence of pseudorandom numbers.

The `testcasename` function returns the unqualified name of the actually executing test case.

The detailed description of predefined functions is given in annex C of the ETSI standard ES 201 873-1.
Conversion functions span the gap between different simple variable types. A function at the intersection of a given column and a row has an in parameter indicated in the column header and returns the value type indicated in the row header.

The detailed description of predefined functions is given in annex C of the ETSI standard ES 201 873-1.

Green letters indicate TITAN extensions, not included in the standard.

Difference between functions with ‘str’ and ‘char’ in their names is explained with the following examples:

- `int2char (66) = "B", int2str (66) = "66".`
NEW PREDEFINED FUNCTIONS

`log2str(log-arguments) return charstring`
Returns formatted output of arguments instead of placing them to log file (TITAN)

```cpp
// Save output of log statement instead of
var charstring str
str := log2str("Value of v is:", v);
```

`enum2int(enumeration-reference) return integer`
Gives integer value associated with enumeration item

```cpp
type enumerated E { zero, one, two, three };  
var E e := one;
integer i := enum2int(one);  // i == 1
```

`isvalue(inline-template) return boolean`
Returns true if argument template contains specific value or omit
The Main Test Component (MTC) and Test System Interface (TSI or System for short) are implicitly instantiated (created) when the test case is started. TSI may be omitted if only the MTC is instantiated during test execution. In this case, MTC type defines the TSI ports implicitly.

A testcase has no return clause, must not use the `return` statement. Instead, the result of the test case execution is done in a verdict type variable. This internal verdict variable is associated with each component instance and the MTC determines the final verdict based on the verdicts returned by the Parallel Test Components and the Main Test Component.

TC can be started directly from control part, or from a function running on the control part (i.e., MTC is not yet created) using the `execute()` statement.
The **testcase header**:

- contains the list of formal parameters of the test case. When no parameters are used, empty brackets must be written;
- the mandatory **runs on** clause specifies the Main Test Component which the test case is running on. This makes the test ports visible to the MTC;
- the keyword **system** is only used if a distinct Test System Interface (TSI) is used. Otherwise, MTC type defines the TSI ports implicitly.
- the **local definitions** are optional. When present, the constants, variables and timers defined here are only visible within the test case.
- the program part (test case body) defines the behavior of the Main Test Component (MTC)
The first example shows a configuration where both the Main Test Component (here: MyMTCTYPE_CT) and the Test System Interface (here: MyTestSystemType_SCT) are present.

The second example shows a configuration where only the Main Test Component is present.
Timer may be used to supervise the execution of a test case. This may be done using an explicit timeout in the execute statement. If the test case does not end within this duration, the result of the test case execution shall be an error verdict and the test system shall terminate the test case. The timer used for test case supervision is a system timer and need not be declared or started.
The module control part describes the execution order of the actual test cases. The instruction after the first comment executes the test case (tc_MyTestCase) and stores the resulting verdict in a variable (vl_MyVerdict).

The next instruction shows how to put an optional time limit (here: 0.5 second) on the test case execution time. When the time limit expires without a returned verdict, the final verdict is set to "error" and the test components are stopped.

The third program statement executes the test case (tc_MyTestCase) ten times.

In the last example the test case (tc_MyTestCase) is only executed when the variable vl_SelExpr has the value true.
X. VERDICTS

verdicttype VS. BUILT-IN VERDICT OPERATIONS FOR BUILT-IN VERDICT MANAGEMENT VERDICT OVERWRITING LOGIC

CONTENTS
Local variables of type `verdicttype` can be used to store verdicts. The value of such a variable can be manipulated using common assignments. Assigning a different value to a `verdicttype` variable always overwrites the existing value.
MTC and PTCs each have a built-in or local verdict. The test case author can alter local verdict during test case execution in each component using the following operations.

The `setverdict` operation is used to set local verdict in test cases, allsteps and functions. The operation may be applied several times in a component resulting in a final local verdict determined according the rules shown on the next slide. "Local" means local to a component.

The `getverdict` operation returns current value of the built-in verdict of the component.
The **verdict overwriting logic** determines the resulting verdict in function of the former verdict every time the operation `setverdict` is applied in a module. The verdict only can change for the worse, i.e., the following sequence alone is possible: `none > pass > inconc > fail > error`. 
Test case (global) verdict is computed based on the local verdicts of involved test components. The execute statement returns the global verdict following the test case termination.
XI. CONFIGURATION OPERATIONS

CREATING AND STARTING OF COMPONENTS
ADDRESSING AND SUPERVISING COMPONENTS
CONNECTING AND MAPPING OF COMPONENTS
PORT CONTROL OPERATIONS
EXAMPLE

CONTENTS
Dynamic nature of test configurations means that parallel test components may be created and destroyed as needed. The same is valid for the connections between components.
Ports and components are used to set up test configurations. Components are the owner of the ports. Test components are working concurrently, they can be created and destroyed.

The MTC is the only test component which is automatically created when a test case starts. All other test components (the PTCs) shall be created explicitly at any point in a behavior description by any other (running) component using the create operation. A component is created with its full set of ports and empty input queues. All component variables and timers are reset to their initial value (if any) and all component constants are reset to their assigned values.

The create operation shall return the unique component reference of the newly created instance. The unique reference to the component will typically be stored in a variable and can be used for connecting instances and for communication purposes such as sending and receiving. Variables holding component references shall be of a previously defined component type (and not one of the built-in component type).
**COMPONENT NAME AND LOCATION**

- ~ can be specified at component creation

```java
// Specifying component name
ptc1 := new1_CT.create(“NewPTC1”);
// Specifying component name and location
ptc2 := new1_CT.create(“NewPTC2”, ”1.1.1.1”);
// Name parameter can be omitted with dash
ptc3 := new1_CT.create(-, ”hostgroup3”);
```

- **Name:**
  - appears in printout and log file names (meta character `\%n`)
  - can be used in test port parameters, component location constraints and logging options of the configuration file

- **Location:**
  - contains IP address, hostname, FQDN or refers to a group defined in groups section of configuration file

---

**Fully Qualified Domain Name (FQDN)**
When defining a variable to store a component reference, care must be taken to use the same component type as has the component to be created.

```
var ComponentType_CT vc_CommpReference;
vc_CommpReference := ComponentType_CT.create;
```
A connection can forward messages, procedure calls or both depending on the operation type of the involved ports. The direction of the message flow (in: incoming, out: outgoing, inout: both ways) can be limited at port definition.

The connect operation can only connect consistent ports of test components. It means that on outgoing port may only be connected to an incoming port and vice versa. Another condition is that the messages defined for both ports must match, i.e., the incoming port must be able to receive all outgoing messages from the connected port. A connection can be set up between a pair of running ports at any time.

Limitations: A port owned by component A shall not be connected with two or more ports owned by A or component B. If a port has more than 1 connections then all outgoing messages must be explicitly addressed.

Connections between two test components can be manipulated by a 3rd component as well.
Mappings carry data between Test System and the Implementation (or System) Under Test (IUT/SUT).

Mappings and connections are equivalent from the abstract communication’s point of view. It is not allowed, however, to connect to a mapped port or to map to a connected port.

Connections ("loop back") within the test system interface are not allowed.
BASIC EXAMPLES FOR VALID CONNECTIONS

- A
- B
- A
- B
- A
- B
- C
- A
- A

© Ericsson 2002-2013.  |  LZT 123 7751 Uen, Rev R1N  |  2013-02-07  |  Page 133
VALID MAPPINGS

A

system

A

system

A

B

system
INVALID CONNECTIONS AND MAPPINGS

A

A

B

A

A

system

system

system

© Ericsson 2002-2013. | L2T 123 7751 Uen, Rev R1N | 2013-02-07 | Page 135
DYNAMIC TEST CONFIGURATION

• Creating or destroying connection between two ports of different parallel test components

```plaintext
connect(vc_A : A1_PCO, vc_B : B1_PCO);
disconnect(vc_A : A1_PCO, vc_B : B1_PCO);
```

• Creating or destroying connection between a port of SUT and a port of a TTCN-3 test component

```plaintext
map(system:SYS_PCO, vc_B:B1_PCO);
unmap(system:SYS_PCO, vc_B:B1_PCO);
```

• Where vc_A, vc_B are component references, A1_PCO and B1_PCO are port references
Once a component has been created and connected, the execution of its behavior has to be started. This is done by using the start operation. Every component can only be started once. The function start() is non-blocking, execution continues immediately.
TERMINATING COMPONENTS

- MTC terminates when the executed **testcase** finishes
- PTC terminates when the function that it is executing has finished (implicit stop) or the component is explicitly stopped/killed using the **stop/kill** operation
- PTCs cannot survive MTC termination: the RTE kills all pending PTCs at the end of each test case execution.
- The **stop** operation releases all resources of a ephemeral PTC; alive PTC resources are suspended but remain preserved
- The **kill** operation releases all resources of the PTC

```plaintext
self.kill; // suicide of a test component
vc_A.stop; //terminating a component with reference vc_A
all component.stop; //terminating all parallel components
```

Using the **all component** keyword, all (parallel) components may only be stopped from the Main Test Component (MTC).

stop ≠ self.stop
WAITING FOR A PTC TO TERMINATE

- The **done** operation
  - blocks execution while a PTC is running;
  - does not block otherwise (finished, failed, stopped or killed)
- The **killed** operation
  - blocks while the referred PTC is alive
  - does not block otherwise
  - is the same as **done** on normal PTC

```plaintext
vc_A.done; // blocks execution until vc_A terminates

all component.done; // blocks the execution until all
                     // parallel test components terminate

vc_B.killed; // wait until vc_B alive component is killed
```
The **running** operation returns a Boolean value depending on the active or passive state of the referenced component. The **done** operation blocks the execution until the referenced component has terminated when used as a stand-alone statement. (It can also be used as an alternative in an **alt** statement.)

Components can be in following states:

- non-existing or not created (**running** == error, **done** == error)
- created but not yet started (**running** == false, **done** blocks execution)
- started and running (**running** == true, **done** blocks execution)
- finished execution or stopped or a test case error occurred (**running** == false, **done** does not block)

When the **all component** keyword is used instead of a component reference in the **running** operation (allowed only in the Main Test Component [MTC]), it will return **true** if all PTCs started but not stopped explicitly by another component are executing their behavior.

When the **any component** keyword is used instead of a component reference in the **running** operation (allowed only in the MTC), it will return **true** if at least one PTC is executing its behavior.

When the **all component** keyword is used instead of a component reference in the **done** operation (allowed only in the MTC), execution continues if no one PTC is executing its behavior or if no PTC has been created or started.

When the **any component** keyword is used instead of a component reference in the **done** operation (allowed only in the MTC), execution continues if at least one PTC has terminated or stopped.
PTC STATE MACHINE

create/creation of a non-alive PTC

Inactive

stop"component terminates" (see note 2a)

klll"component terminates" (see note 2b)

Running

start"component executes function"

done/no match killed/no match

running/false alive/true

"run-time error"/error

Killed

start/error

start/error

ERROR

(note 2c)

NOTE 1: (a) Stop can be either a stop, self-stop or a stop from another test component;
(b) Kill can be either a kill, self-kill, a kill from another test component or a kill from the test system (in error cases)

NOTE 2: (a) Stop can be from another test component only;
(b) Kill can be from another test component or from the test system (in error cases) only.

NOTE 3: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.
ALIVE PTC STATE MACHINE

inactive

running

killed

stopped

create alive/creation of an alive PTC

domino match killed/no match running/false alive/false

start/component executes function

domino match killed/no match running/error alive/error

"run-time error/error"

stop/component stops (see note 2a)

kill/component terminates (see note 2b)

"return from function"/"component terminates"

"completion of function"/"component terminates"

start/error (see note 2c)

stop milling (see note 2c)

kill milling (see note 2c)

domino match killed/no match running/false alive/false

NOTE 1: (a) Stop can be either a stop, self stop or a stop from another test component;
(b) Kill can be either a kill, self kill, a kill from another test component or a kill from the test system (in error cases).

NOTE 2: (a) Stop can be from another test component only;
(b) Kill can be from another test component or from the test system (in error cases) only.

NOTE 3: Whenever a test component enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.
MTC STATE MACHINE

NOTE 1: (a) Stop can be either a stop, self-stop, a stop from another test component; (b) Kill can be either a kill, self-kill, a kill from another test component or a kill from the test system (in error cases).
NOTE 2: All remaining PTCs shall be killed as well and the test case terminates.
NOTE 3: Whenever the MTS enters its error state, the error verdict is assigned to its local verdict, the test case terminates and the overall test case result will be error.
**SPECIAL FEATURES OF COMPONENT HANDLING**

- The **running**, **alive**, **done**, **killed** and **stop** operations can be combined with the special **any component** or **all component** as well as with the **self** and **mtc** keywords.

<table>
<thead>
<tr>
<th>Operation</th>
<th>any component</th>
<th>all component</th>
<th>self</th>
<th>mtc</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>running alive</strong></td>
<td>YES*</td>
<td>YES*</td>
<td>YES*</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td><strong>done killed</strong></td>
<td>YES*</td>
<td>YES*</td>
<td>YES*</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td><strong>stop kill</strong></td>
<td>NO</td>
<td>YES*</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

**YES* = from MTC only!**  **YES* = from PTCs only!**
The mtc and system components are automatically created in the beginning of test case execution and destroyed when the test execution finishes. The test case itself is executed on the mtc. The system component does not run any behavior as it acts as a logical model of the IUT.

The runs on clause of the executed test case determines the component type of the mtc, while the system clause specifies the component type used for system.

The component type definition enlists the resources of a particular type component, e.g. how many and what kind of interfaces the component has.

The port type definition declares operation mode of the interface (message=asynchronous, procedure=synchronous) and enlists the type of messages (or signatures at a procedural port), which can traverse the port.
ELEMENTARY STEPS OF SETTING UP THE TEST CONFIGURATION

1) Create PTCs (ports of components are created and started automatically) – create
2) Establish connections and mappings – connect or map
3) Start behavior on PTCs – start
4) Wait for PTCs to complete – done or all component done
Elementary steps of setting up the test configuration:

1) Create PTCs (ports of components are created and started automatically)
2) Establish connections and mappings
3) Start behavior on PTCs remotely
4) Wait for PTCs to complete
EXTENDING COMPONENT TYPES

- Reuse of existing component type definitions:
  - "Derived" component type inherits all resources (ports, timers, variables, constants) of extended "parent" component type(s)
- Restrictions:
  - no cyclic extensions
  - avoid name clashes between different definitions

```plaintext
type component old1_CT {
  var integer i;
  port MyPortType P;
}

type component old2_CT {
  timer T;
  port MyPortType Q;
}

type component new_CT extends old1_CT, old2_CT {
  port NewPortType R; // includes P, Q, R, i and T!
}
```
"RUNS ON-COMpatibility"

- Function/altstep/testcase with “runs on” clause referring to an extended component type can also be executed on all derived component types

```javascript
function f() runs on old1_CT {
  P.receive(integer:?) -> value i;
}

ptc := new1_CT.create;
ptc.start(f()); // OK: new1_CT is derived from old1_CT
```
The **scope unit** is the region of the TTCN-3 source within which (constant, timer, variable, etc.) definitions may have effect, within which multiple definitions of the same name are prohibited, and outside of which definitions inside the unit do not have effect.

Definitions made in the **module definition part** but outside of other scope units are globally visible in the module. So are imported identifiers.

Definitions made in the **module control part** have local visibility, i.e. can be used within the control part only.

Definitions made in a test **component type** may be used only in functions, test cases and altsteps referencing that component type by a **runs on**-clause.

**Functions**, **altsteps** and **test cases** are individual scope units without any hierarchical relation between them, i.e. definitions made at the beginning of their body have local visibility.

Definitions within **block of statements** (e.g. **for**, **if-else**, **while**, **do-while**, **alt**, **interleave**) have local visibility within the statement concerned.
Ports are already running when the component is started. All ports are automatically stopped by the run-time environment when their owner component has finished execution.

None of the above operations affect connections and mapping of ports.

Receiving operations block on stopped ports until the port is restarted (provided no defaults are active).

The contents of port queue can still be matched and read on halted ports.

```
A_PCO.halt;  //no new messages can get into port queue
A_PCO.stop;  //no more activity on A_PCO
A_PCO.clear;  //removes all messages from port queue
A_PCO.start;  //clears port queue and restarts port
```
**Configuration operations** are used to set up and control test components. These operations shall only be used in test cases, functions and altsteps (i.e. not in the module control part).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create new parallel test component</td>
<td><code>CT.create</code></td>
</tr>
<tr>
<td>Create an alive component</td>
<td><code>CT.create alive</code></td>
</tr>
<tr>
<td>Connect two components</td>
<td><code>connect(c1:p1, c2:p2)</code></td>
</tr>
<tr>
<td>Disconnect two components</td>
<td><code>disconnect(c1:p1, c2:p2)</code></td>
</tr>
<tr>
<td>Connect (map) component to system</td>
<td><code>map(c1:p1, c2:p2)</code></td>
</tr>
<tr>
<td>Unmap port from system</td>
<td><code>unmap(c1:p1, c2:p2)</code></td>
</tr>
<tr>
<td>Get MTC address</td>
<td><code>mtc</code></td>
</tr>
<tr>
<td>Get test system interface address</td>
<td><code>system</code></td>
</tr>
<tr>
<td>Get own address</td>
<td><code>self</code></td>
</tr>
<tr>
<td>Start execution of test component</td>
<td><code>ptc.start(f())</code></td>
</tr>
</tbody>
</table>

Where `CT` is a component type definition; `ptc` is a PTC; `f()` is a function; `c, c1, c2` are component references and `p, p1, p2` are port identifiers.
Configuration operations are used to set up and control test components. These operations shall only be used in test cases, functions and altsteps (i.e. not in the module control part).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check termination of a PTC</td>
<td>ptc.running</td>
</tr>
<tr>
<td>Check if a PTC is alive</td>
<td>ptc.alive</td>
</tr>
<tr>
<td>Stop execution of test component</td>
<td>c.stop</td>
</tr>
<tr>
<td>Kill an alive component</td>
<td>c.kill</td>
</tr>
<tr>
<td>Wait for termination of a test component</td>
<td>ptc.done</td>
</tr>
<tr>
<td>Wait for a PTC to be killed</td>
<td>ptc.killed</td>
</tr>
<tr>
<td>Start or restart port (queue is cleared!)</td>
<td>p.start</td>
</tr>
<tr>
<td>Stop port and block incoming messages</td>
<td>p.stop</td>
</tr>
<tr>
<td>Pause port operation</td>
<td>p.halt</td>
</tr>
<tr>
<td>Remove messages from the input queue</td>
<td>p.clear</td>
</tr>
</tbody>
</table>

Where \( c \) is a component reference; \( ptc \) is a PTC and \( p \) is a port identifier.
XII. DATA TEMPLATES

INTRODUCTION TO TEMPLATES
TEMPLATE MATCHING MECHANISMS
INLINE TEMPLATES
MODIFIED TEMPLATES
PARAMETERIZED TEMPLATES
PARAMETERIZED MODIFIED TEMPLATES
TEMPLATE HIERARCHY

CONTENTS
### TEMPLATE CONCEPT

<table>
<thead>
<tr>
<th>Message to send</th>
<th>Acceptable answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE: REQUEST</td>
<td>TYPE: RESPONSE</td>
</tr>
<tr>
<td>ID: 23</td>
<td>ID: SAME as in REQ.</td>
</tr>
<tr>
<td>FROM: 231.23.45.4</td>
<td>FROM: 230.x – 235.x</td>
</tr>
<tr>
<td>TO: 232.22.22.22</td>
<td>TO: 231.23.45.4</td>
</tr>
<tr>
<td>FIELD1: 1234</td>
<td>FIELD1: 800-900</td>
</tr>
<tr>
<td>FIELD2: &quot;Hello&quot;</td>
<td>FIELD2: Do not care</td>
</tr>
</tbody>
</table>
Template: something that establishes or serves as a pattern.

Templates are used either to test whether a set of received values matches the template specification or to transmit a set of distinct values.

Templates used to receive messages have the advantage that all valid message variants may be described in a single template. When a message arrives, the program can decide whether it is a valid one or not. This procedure is called matching.

Templates used to send messages are advantageous because they can be parameterized, thus, reused. All fields of these templates must have a determined value at the point when a message is sent using them. These templates may be used to receive messages as well, but only when all fields of the expected message are fixed and known beforehand.
The runtime environment (RTE) compares the received message with the predefined template describing all valid message variants. When the message is one of the valid messages (it fits into the template), the match is successful.
**Template Syntax**

```
template <type> <identifier> [ formal parameter list ]
  [ modifies <base template identifier> ] := <body>
```

- **Type** can be any simple or structured type;
- **<body>** uses the assignment notation for structured types, thus, it may contain nested value assignments;
- the optional **formal parameter list** contains a fixed number of parameters; the formal parameters themselves can be templates or values;
- the optional **modifies** keyword denotes that this template is derived from an existing **<base template identifier>** template;
- constants, matching expressions, templates and parameter references shall be assigned to each field of a template.

**Type** determines the structure of the template, i.e., its fields.
**Identifier** is the name of the template. It is used when we want to refer to the template.

The **formal parameter list** provides the list of the parameters of the template. These optional parameters are used to alter the template at every invocation.

The keyword **modifies** denotes derived template where only some of the fields of the original template are changed. Both templates have the same fields.

The template **body** lists the permitted values for all fields.
First, we define a record (MyMessageType) containing three fields, the first one being optional.

The type of the template will be the one just defined. The template we'll define is called tr_MyTemplate. In the template name prefix, 't' stands for 'template' and 'r' for receiving.

The template accepts the following messages: the first field must be present, but its content is don't care. The second field may have the value B, O or Q. The value of the last field must be in function of the parameter pl_param either true or false.

The template can be used for receiving only, because it contains an undefined field (the first one).
Matching checks whether the received message fits in the set of accepted messages. The check is done for each field of the template independently. A message is accepted ("matches") when all fields contain accepted values.

The matching mechanisms are depicted in the annex B.1 of ETSI ES 201 873-1.
SPECIFIC VALUE TEMPLATE

- Contains constant values or `omit` for optional fields
- Template consisting of purely specific values is equivalent to a constant → use the constant instead!
- Applicable to all basic and structured types
- Can be sent and received

```c
// Template with specific value and the equivalent constant
template integer Five := 5;
const integer Five := 5; // constant is more effective here

// Specific values in both fields of a record template
template MyRecordType SpecificValueExample := {
    field1 := omit,
    field2 := false
};
```
The simplest template lists all discrete message values that will be accepted. Complemented values list lists the values which will not be accepted. Both lists refer to fields of the template, i.e., both notations may be mixed in different fields of the same template.
Range indicates the upper and the lower boundaries of acceptable values. An expression evaluating to a specific integer or float value can be used when setting the boundaries. The lower boundary (written after the left parenthesis) must be less than the upper boundary (written before the right parenthesis).
Note: The syntax differs from the intermixed value list and value range subtype construction's notation:

```plaintext
type integer Intermixed (0..127,255);
```
The matching symbol “?” (AnyValue) is used to indicate that any valid incoming value is acceptable. It can be used on values of all types. A template field that uses the any value mechanism matches the corresponding incoming field if, and only if, the incoming field evaluates to a single element of the specified type.
The matching symbol "*" (*AnyValueOrNone) is used to indicate that any valid incoming value, including omission of that value, is acceptable. It can be used on values of all types, provided that the template field is defined as optional.

A template field that uses this symbol matches the corresponding incoming field if, and only if, either the incoming field evaluates to any element of the specified type, or if the incoming field is absent.

Note: The template tr_AnyBitstring can only be used as an optional field of another template.
The matching symbol "?" is used to indicate that it replaces single elements of a string (except character strings), a record of, a set of or an array. It shall be used only within values of string types, record of types, set of types and arrays.

The matching symbol "***" is used to indicate that it replaces none or any number of consecutive elements of a string (except character strings), a record of, a set of or an array. The "***" symbol matches the longest sequence of elements possible, according to the pattern as specified by the symbols surrounding the "***".
Character patterns can be used in templates to define the format of a required character string to be received.

TTCN-3 pattern expressions have little common with standard regular expressions!

Note: pattern matching for universal charstring is not implemented in TITAN yet!
In addition to literal characters, character patterns allow the use of meta-characters. If it is required to interpret any metacharacter literally it should be preceded with the metacharacter \\

```
- means a range, if before and after there is no space!
inside [ ] char set may be defined e.g. [a ft] --- a or f or t
[a d -] a or d or – (- can be only at the LAST position!)
```
The pattern used in template tr_3 explained: it begins with a capital letter, followed by (zero or more hyphen and at least one letter or number) and the section inside the parentheses may be repeated several times.
The function is used to extract a substring from the input string (on the slide: v_string). It is used mainly with textual protocols.

The substring to be extracted is the one matching the regular expression (on the slide: v_regexp). The last argument of the function (on the slide: 0) denotes the cardinal number of the group in the regexp, 0 being the first match. A group is enclosed in parentheses, where the first parenthesis must not be preceded by a '#' or a '\'.

```plaintext
function regexp(<input-string>, <regexp>, <group-number>)
return <type of input-string>;
• returns a substring of <input-string>, which is the content of <group-number> + 1th group matching the <regexp>
• <input-string> type can be any (universal) charstring
• the type of returned value equals to the type of the input string

control {
    var charstring v_string := "0036 (1) 737-7698";
    var charstring v_regexp :=
        "0036 #,(,)\((\d#(1,))\) #,(,)\[d-]\#(1,)"
    var charstring v_result := regexp(v_string, v_regexp, 0);
} // v_result contains the number in parentheses, i.e. 1
```
MATCHING MECHANISMS (2)

• Value attributes on field level:
  – length restriction;
  – ifpresent modifier.

• Special matching for set of types:
  – subset and superset matching.

• Special matching for record of types:
  – permutation matching.

• Predefined functions operating on templates:
  – match()
  – valueof()
The length restriction attribute is used to restrict the length of string values and the number of elements in a set of, record of or array structure.
A template field that uses `ifpresent` matches the corresponding incoming field if, and only if:

- the incoming field matches according to the associated matching mechanism, or
- if the incoming field is absent.

Not to be confused with the predefined function `ispresent()` which checks whether an optional field is present in the actual instance of the referenced data object.
A field that uses SubSet matches the corresponding incoming field if, and only if, the incoming field contains only elements defined within the SubSet, and may contain less.

A field that uses SuperSet matches the corresponding incoming field if, and only if, the incoming field contains at least all of the elements defined within the SuperSet, and may contain more.

∀value ∈ set of : value ⊆ subset: For all value in set of such that value is a subset of subset.

In the superset example, the group \{4,3,2\} does not match because '1' is missing. The excess '4' would not hinder the match.
PERMUTATION

- Applicable to record of types only
- permutation matches all permutations of enlisted elements (i.e. the very same elements enlisted in any order)

```plaintext
type record of integer ROI;
template ROI tr_ROIa := { permutation (1, 2, 3) };
// Matches {1,3,2} and {2,1,3}
// Does not match {4,3,2}, {0,1,2,3} and {1,2} (3 is missing)
```

© Ericsson 2002-2013.
Specific value template, mentioned in the first column, matches the corresponding incoming field value if, and only if, the incoming field value has exactly the same value as the value to which the expression in the template evaluates. Thus, it cannot be regarded as a veritable matching mechanism, as it only accepts a fixed value.
The function can be interpreted as an extended 'equality' operation. It compares the value of a variable with a template and returns 'true' if the template matches the value of the variable as it is the case in the example on the slide.
Specific values template means that each field of the template shall resolve to a single value.
TEMPLATES ARE NOT VALUES

- Value types in TTCN-3

```c
1 // literal value
const integer c := 1; // constant value
modulepar integer mp := 1; // module parameter value
var integer v := 1; // variable value
```

- Specific value templates vs. general (receive) templates

```c
template integer t1 := 1; // specific value template
template integer t2 := ?; // receive template
```

- Comparing values with values or templates

```c
c == 1 and c == mp and mp == v // true: all values
t1 == c // error: comparing template with a value
valueof(t1) == v // true: t1 may be converted to a value
valueof(t2) == v // error:t2 cannot be converted to a value
match(mp,t2) == true // true: mp matches t2
```
TEMPLATE VARIANTS

- Inline templates
- Inline modified templates
- Template modification
- Template parameterization
- Template hierarchy
Inline templates do not have identifiers and are valid for that single operation. Inline templates must not have parameters.

The type identifier may be omitted when the value unambiguously identifies the type, see Ex2 on the slide.

The typical use is depicted in Ex1. It is used mainly for value redirect and sender redirect.
Instead of specifying a new template, it is possible to modify an existing template when only a few fields change.

The `modifies` keyword denotes the parent template from which the new, or modified template shall be derived.

This parent template may be either an original template or a modified template.
INLINE MODIFIED TEMPLATES

- Defined directly in the communication operation
- Valid only for that one operation (No identifier, no reusability)
- Can not be parameterized
- Usually ineffective, not recommended to use!

```c
template MyRecordType t_1 := {
    field1 := omit,
    field2 := false
}
control {
    ... port_PCO.receive(modifies t_1 := { field1 := * });
    ...
}
```
Templates for both sending and receiving operations can be parameterized. On the slide, the first one is appearing. This slide shows the use of value parameters.

The message sent on `P1_PCO` will have the following structure:

the 1\textsuperscript{st} field is integer, its value equals to 1;
the 2\textsuperscript{nd} field is structured (MyMsgType) and has two subfields:

its 1\textsuperscript{st} subfield is integer, its value is determined by the variable `vl_integer_2`;
its 2\textsuperscript{nd} subfield is not present.
It is not allowed to modify a field, which is parameterized in the parent template. Thus, in the example on the slide field1 and field2 cannot be modified while field3 can.
• **Template** formal parameters can accept as actual parameter:
  - literal values
  - constants, module parameters & variables, omit
  + matching symbols (?, *, etc.) and templates

```cpp
// Template-type parameterization
template integer tr_Int := ( (3..6), 88, 555 ) ;
template MyIEType tr_TemplPm(template integer pl_int) :=
  { f1 := 1, f2 := pl_int }

// Can be used:
P1_PCO.send(tr_TemplPm( 5 ) );
P1_PCO.receive( tr_TemplPm( ? ) );
P1_PCO.receive( tr_TemplPm( tr_Int ) );
P1_PCO.receive( tr_TemplPm( (3..55) ) );
P1_PCO.receive( tr_TemplPm( complement (3,5,9) ) );
```

Note the **template** keyword!
RESTRICTED TEMPLATES

Templates can be restricted to
- (omit) evaluate to a specific value or omit
- (present) evaluate to any template except omit
- (value) specific value but the entire template must not be omit

Applicable to any kind of templates (i.e. template definitions, variable templates and template formal parameters)

<table>
<thead>
<tr>
<th>Template</th>
<th>(omit)</th>
<th>(present)</th>
<th>(value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>omit</td>
<td>Ok</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>Specific value</td>
<td>Ok</td>
<td>Ok</td>
<td>Ok</td>
</tr>
<tr>
<td>template</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>error</td>
<td>Ok</td>
<td>error</td>
</tr>
</tbody>
</table>

```plaintext
function f_omit(template (omit) integer p) {}
function f_present(template (present) integer p) {}
function f_value(template (value) integer p) {}
```
// omit restriction
function f_omit(template (omit) integer p) {}
    f_omit(omit); // Ok
    f_omit(integer:1); // Error
    f_omit(1); // Ok

// present restriction
function f_present(template (present) integer p) {}
    f_present(omit); // Error: omit is excluded
    f_present(integer:1); // Ok
    f_present(1); // Ok

// value restriction
function f_value(template (value) integer p) {}
    f_value(omit); // Error: entire argument must not be omit
    f_value(integer:1); // Error: not value
    f_value(1); // Ok
TEMPLATE VARIABLES

- Templates can be stored in so called template variables
- Template variable
  - may change its value several times
  - assignment and access to its elements are permitted
    (e.g. reference and index notation permitted)
  - must not be an operand of any TTCN-3 operators

```plaintext
control {
  var template integer vt := ?;
  var template MySetType vs :=
    { field1 := ?, field2 := true };
  vt := (1, 2, 3); // Ok
  vs.field1 := 2; // Ok
}
```
**TEMPLATE HIERARCHY**

- Practical template structure/hierarchy depends on:
  - Protocol: complexity and structure of ASPs, PDUs
  - Purpose of testing: conformance vs. load testing
- Hierarchical arrangement:
  - Flat template structure – separate template for everything
  - Plain templates referring to each other directly
  - Modified templates: new templates can be derived by modifying an existing template (provides a simple form of inheritance)
  - Parameterized templates with value or template formal parameters
  - Parameterized modified templates
- Flat structure → hierarchical structure
  - Complexity increases, number of templates decreases
  - Not easy to find the optimal arrangement
TEMPLATE HIERARCHY - TYPICAL SITUATIONS

modified template

parametrized template

template parameter
XIII. ABSTRACT COMMUNICATION OPERATIONS

ASYNCHRONOUS COMMUNICATION
SEND, RECEIVE, CHECK AND TRIGGER OPERATIONS
PORT CONTROL OPERATIONS (START, STOP, CLEAR)
VALUE AND SENDER REDIRECTS
SEND TO AND RECEIVE FROM OPERATIONS
SYNCHRONOUS COMMUNICATION

CONTENTS
ASYNCHRONOUS COMMUNICATION

send

non-blocking

MTC

receive

blocking

PTC
**send** AND **receive** SYNTAX

- `<PortId>.send(<ValueRef>)`
  where `<PortId>` is the name of a `message` port containing an `out` or `inout` definition for the type of `<ValueRef>` and `<ValueRef>` can be:
  - Literal value; constant, variable, specific value template (i.e. send template) reference or expression

- `<PortId>.receive(<TemplateRef>)` or `<PortId>.receive`
  where `<PortId>` is the name of a `message` port containing an `in` or `inout` definition for the type of `<TemplateRef>` and `<TemplateRef>` can be:
  - Literal value; constant, variable, template (even with matching mechanisms) reference or expression; inline template
Send and receive operations can be used only on connected ports
- Sending or receiving on a port, which has neither connections nor mappings results in test case error
- The send operation is non-blocking
- The receive operation has blocking semantics (except if it is used within an alt or an interleave statement)
- Arriving messages stay in the incoming queue of the destination port
- Messages are sent and received in order
- The receive operation examines the 1st message of the port’s queue, but extracts this only if the message matches the receive operation’s template
SEND AND RECEIVE EXAMPLES

```
MSG.send("Hello!");
MSG.receive("Hello!");

MSG.send("Hi!");
MSG.send("Hello!");
MSG.receive("Hello!");
```

Blocked!
CHECK-RECEIVE AND TRIGGER VS. RECEIVE

- Check-receive operation blocks until a message is present in the port’s queue, then it decides, if the 1st message of the port’s queue matches our template or not; 
  The message itself remains untouched on the top of the queue!
  - Usage:
    `<PortId>.check(receive(<TemplateRef>));`
    `<PortId>.check;`
    any port.check;

- Trigger operation blocks until a message is arrived into the port’s queue and extracts the 1st message from the queue:
  - If the top message meets the matching criteria → works like receive
  - Otherwise the message is dropped without any further action
  - Usage:
    `- <PortId>.trigger(<TemplateRef>);`
    `- <PortId>.trigger;` (equivalent to `<PortId>.receive;`)

`<PortId>.check;` checks if there is anything waiting in the queue.
TRIGGER EXAMPLES

MSG. send("Hello!");

MSG. trigger("Hello!");

MSG. send("Hi!");

MSG. send("Hello!");

MSG. trigger("Hello!");
VALUE AND SENDER REDIRECT

- Value redirect stores the matched message into a variable
- Sender redirect saves the component reference or address of the matched message's originator
- Works with both `receive` and `trigger`

```
template MsgType MsgTemplate := { /* valid content */ } 

var MsgType MsgVar;
var CompRef Peer;
// save message matched by MsgTemplate into MsgVar
PortRef.receive(MsgTemplate) -> value MsgVar;
// obtain sender of message
PortRef.receive(MsgTemplate) -> sender Peer;
// extract MsgType message and save it with its sender
PortRef.trigger(MsgType:?) -> value MsgVar sender Peer;
```

// obtain sender of message in queue w/o removing it
PortRef.check(receive(MsgTemplate) -> sender Peer);
send to AND receive from

- Components A, B, C are of the same type
- P has 2 connections and 1 mapping in component A
- How does component A tell to the RTE that it waits for an incoming message from component B?
  
  ```java
  p.receive(TemplateRef) from B;
  ```

- How does component A send a message to system?
  
  ```java
  p.send(Msg) to system;
  ```

// send a reply for the previous message

```java
  p.receive(Request_Msg) -> sender CompVar;
  p.send(Msg) to CompVar;
```
EXAMPLES OF ASYNCHRONOUS COMMUNICATION OPERATIONS

MyPort_PCO.send(f_Myf_3(true));

MyPort_PCO.receive(tr_MyTemplate(5, v_MyVar));

MyPort_PCO.receive(MyType:? --> value v_MyVar; // !!

MyPort_PCO.receive(MyType:? --> value v_MyVar sender Peer;

any port.receive;

MyPort_PCO.check(receive(A < B)) from MyPeer;

MyPort_PCO.trigger(5) --> sender MyPeer;
## SUMMARY OF ASYNCHRONOUS COMMUNICATION OPERATIONS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send a message</td>
<td>send</td>
</tr>
<tr>
<td>Receive a message</td>
<td>receive</td>
</tr>
<tr>
<td>Trigger on a given message</td>
<td>trigger</td>
</tr>
<tr>
<td>Check for a message in port queue</td>
<td>check</td>
</tr>
</tbody>
</table>
SYNCHRONOUS COMMUNICATION

MTC

- call
- getreply
- catch
- exception
- blocking

PTC

- getcall
- reply
- raise
- exception
- blocking
signature MyProc3 (out integer MyPar1, inout boolean MyPar2)  //
signature definition
  return integer
  exception (charstring);

// Call of MyProc3
MyPort.call(MyProc3:{ -, true }, 5.0) to MyPartner {
  [] MyPort.getreply(MyProc3:{?, ?}) -> value MyResult param
          (MyPar1Var,MyPar2Var) { }
  [] MyPort.catch(MyProc3, “Problem occured”) {
    setverdict(fail); stop; }
  [] MyPort.catch(timeout) {
    setverdict(inconc); stop; }
}
// Reply and exception to an accepted call of MyProc3
MyPort.reply(MyProc3:{5,MyVar} value 20);  // reply
MyPort.raise(MyProc3, “Problem occured”);  // exception
SUMMARY OF SYNCHRONOUS COMMUNICATION OPERATIONS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoke (remote) procedure call</td>
<td>call</td>
</tr>
<tr>
<td>Reply to a (remote) procedure call</td>
<td>reply</td>
</tr>
<tr>
<td>Raise an exception</td>
<td>raise</td>
</tr>
<tr>
<td>Accept (remote) procedure call</td>
<td>getcall</td>
</tr>
<tr>
<td>Handle response from a previous call</td>
<td>getreply</td>
</tr>
<tr>
<td>Catch exception (from called entity)</td>
<td>catch</td>
</tr>
<tr>
<td>Check reply or exception</td>
<td>check</td>
</tr>
</tbody>
</table>
XIV. BEHAVIORAL STATEMENTS

SEQUENTIAL BEHAVIOR
ALTERNATIVE BEHAVIOR
ALT STATEMENT, SNAPSHOT SEMANTICS
GUARD EXPRESSIONS, ELSE GUARD
ALTSTEPS
DEFAULTS
INTERLEAVE STATEMENT

CONTENTS
SEQUENTIAL EXECUTION BEHAVIOR FEATURES

- Program statements are executed in order
- Blocking statements block the execution of the component
  - all receiving communication operations, timeout, done, killed
- Occurrence of unexpected event may cause infinite blocking

```cpp
// x must be the first on queue P, y the second
P.receive(x); // Blocks until x appears on top of queue P
P.receive(y); // Blocks until y appears on top of queue P
// When y arrives first then P.receive(x) blocks -> error
```
PROBLEMS OF SEQUENTIAL EXECUTION

- Unable to prevent blocking operations from dead-lock
  i.e. waiting for some event to occur, which does not happen

  ```
  // Assume all queues are empty
  P.send(x); // transmit x on P -> does not block
  T.start; // launch T timer to guard reception
  P.receive(x); // wait for incoming x on P -> blocks
  T.timeout; // wait for T to elapse
  // ^^^ does not prevent eventual blocking of P.receive(x)
  ```

- Unable to handle mutually exclusive events

  ```
  // x, y are independent events
  A.receive(x); // Blocks until x appears on top of queue A
  B.receive(y); // Blocks until y appears on top of queue B
  // y cannot be processed until A.receive(x) is blocking
  ```
SOLUTION: ALTERNATIVE EXECUTION
- alt STATEMENT

- Go for the alternative that happens earliest!
- Alternative events can be processed using the alt statement
- alt declares a set of alternatives covering all events, which …
  - can happen: expected messages, timeouts, component termination;
  - must not happen: unexpected faulty messages, no message received
  ... in order to satisfy soundness criterion
- All alternatives inside alt are blocking operations

The format of alt statement:

```java
alt { // declares alternatives
  // 1st alternative (highest precedence)
  // 2nd alternative
  // ...
  // last alternative (lowest precedence)
} // end of alt
```
ALTERNATIVE EXECUTION BEHAVIOR EXAMPLES

- Take care of unexpected event and timeout:

```c
P.send(req)
T.start;
// ...
alt {
[] P.receive(resp) { /* actions to do and exit alt */ }
[] any port.receive { /* handle unexpected event */ }
[] T.timeout { /* handle timer expiry and exit */ }
}
```
The execution of alt starts with taking a “snapshot”. The snapshot represent the current state of the test system including timers, port queues, components, etc. The alternatives enlisted within the alt statement are evaluated on the contents of the snapshot.

When none of the alternatives are successful, the run-time environment takes a new snapshot and the execution resumes with the first alternative.

The execution proceeds until a single successful alternative is found or when the run-time environment can determine that no alternative can ever be successful. In the former case the statement block of the successful alternative is executed. Then, the next statement following the alt is executed. In the latter case the execution terminates with dynamic test case error.

The snapshot is only valid until the execution gets to the statement block! That is why the alt statement can be nested.
The alt statement consists of alternatives. Alternatives normally consist of guard, event and statement block. The event used in alt can only be a receiving (or blocking) event. The semantics of these blocking statements change when used within the alt statement!
alt STATEMENT EXECUTION SEMANTICS

- Alternatives are processed according to snapshot semantics
  - Alternatives are evaluated in the same context (snapshot) such that each alternative event has “the same chance”
- alt waits for one of the declared events to happen then executes corresponding statement block using sequential behavior!
  - i.e. only a single declared alternative is supposed to happen
- alt quits after completing the actions related to the event that happened first
- First alternative has highest priority, last has the least
- When no alternatives apply → programming error (not sound) → dynamic testcase error!
The repeat keyword can appear only as the last statement within statements blocks of `alt` statements. Then, instead of jumping to the next statement following the `alt`, the execution is continued from the beginning of the `alt` with a new snapshot.
The repeat keyword can appear only as the last statement within statements blocks of alt statements. Then, instead of jumping to the next statement following the alt, the execution is continued from the beginning of the alt with a new snapshot.
The else guard does not have an accompanying event because it is always successful.
Local definitions within altsteps are deprecated. When initializing a local variable with a function having side-effect (i.e. doing something else in addition to initializing the variable) then this side-effect may be executed multiple times. Consequently, variables should be initialized with constant only!

Side-effect is for instance the sending of a message. In the above situation we could not know how many times this message is sent!
THREE WAYS TO USE \texttt{altstep}

- Direct invocation:
  - Expands dynamically to an \texttt{alt} statement

- Dynamic invocation from alt statement:
  - Attaches further alternatives to the place of invocation

- Default activation:
  - Automatic attachment of activated \texttt{altstep} branches to the end of each \texttt{alt}/blocking operation
USING `altstep` – DIRECT INVOCATION

```c
// Definition in module definitions part
altstep as_MyAltstep(integer pl_i) runs on My_CT {
    [] PCO.receive(pl_i) {...}
    [] PCO.receive(tr_Msg) {...}
}

// Use of the altstep
testcase to_101() runs on My_CT {
    as_MyAltstep(4); // Direct altstep invocation...
}

// ... has the same effect as
testcase to_101() runs on My_CT {
    alt {
        [] PCO.receive(4) {...}
        [] PCO.receive(tr_Msg) {...}
    }
}
```
USING `altstep` - INVOCATION IN `alt`

```
alt {
    [guard₁] port1.receive (cR_T) block of statements₁
    [guard₂] optional block of statements₂
    [guard₃] port2.receive block of statementsₓ block of statements₂
    [guard₄] port3.receive block of statementsᵧ block of statements₂
    [guard₅] timerₓ.timeout block of statementsₙ
}

as_myAltstep () {
    optional local definitions
    [guard₂] port2.receive block of statementsₓ
    [guard₄] port3.receive block of statementsᵧ
}
```
MOTIVATION - DEFAULTS

- Error handling at the end of each **alt** instruction
  - Collect these alternatives into an **altstep**
  - Activate as **default**
  - Automatically copied to the end of each **alt**

```plaintext
alt {
[] P.receive(1)
{
  P.send(2)
  alt { // embedded alt
      [] P.receive(3) { P.send(4) }
      [] any port.receive { setverdict(fail); }
      [] any timer.timeout { setverdict(inconc) }
      } // end of embedded alt
  }
  [] any port.receive { setverdict(fail); }
  [] any timer.timeout { setverdict(inconc) }
}
```
USING `altstep` - ACTIVATED AS DEFAULT

```plaintext
var default def_myDef := activate(as_myAltstep());
alt {
    [guard_1] port1.receive (cR_T) block of statements_1
    [guard_2] port2.receive(cR2_T) block of statements_2
    local definitions!
    [guard_3] any port.receive block of statements_x
    [guard_4] T.timeout block of statements_y
}
as_myAltstep () {
    optional local definitions
    [guard_5] any port.receive block of statements_x
    [guard_6] T.timeout block of statements_y
}
```

alternatives of activated defaults are also evaluated after regular alternatives
component instance defaults as_myAltstep:
ACTIVATION OF **altstep** TO DEFAULTS

- Altsteps can be used as default operations:
  - **activate:** appends an **altstep** with given actual parameters to the current default context, returns a unique default reference
  - **deactivate:** removes the given default reference from the context

```plaintext
altstep as1() runs on CT {
  [] any port.receive { setverdict(fail)}
  [] any timer.timeout { setverdict(inconc)}
}
var default d1 := activate(as1());
...
deactivate(d1);
```

- Defaults can be used for handling:
  - Incorrect SUT behavior
  - Periodic messages that are out of scope of testing
- There are only **dynamic** defaults in TTCN-3
- The default context of a PTC can be entirely controlled run-time
- Defaults have no effect within an alt, which contains an else guard!

Defaults have no effect within an alt, which contains an else guard!
STANDALONE RECEIVING STATEMENTS VS. alt

- Default context contains a list of altsteps that is implicitly appended:
  - At the end of all alt statements except those with else branch
  - After all stand-alone blocking receive/timeout/done ... operations (!!)

- Any standalone receiving statement (receive, check, getcall, getreply, done, timeout) behaves identically as if it was embedded into an alt statement!

```c
MyPort_PCO.receive(tr_MyMessage);
```

- ... is equivalent to:

```c
alt {
    [] MyPort_PCO.receive(tr_MyMessage) {} 
}
```
STANDALONE RECEIVING STATEMENTS VS. `default`

- Activated default branches are appended to standalone receiving statements, too!

```java
var default d := activate(myAltstep(2));
MyTimer.timeout;
```

- ... is equivalent to:

```java
alt {
    [] MyTimer.timeout {}
    [] MyPort.receive(MyTemplate(2))
        { MyPort.send(MyAnswer); repeat }
    [] MyPort.receive
        { setverdict(fail) }
}
```
MULTIPLE DEFAULTS

• Default branches are appended in the opposite order of their activation to the end of alt, therefore the most recently activated default branch comes before of the previously activated one(s)

```
altstep as1() runs on CT {
    [] T.timeout { setverdict(inconc) }
}
altstep as2() runs on CT {
    [] any port.receive { setverdict(fail) }
}
altstep as3() runs on CT {
    [] FCO.receive(MgmtPDU:?) {}
}

var default d1, d2, d3; // evaluation order
    d1 := activate(as1()); // +d1
    d2 := activate(as2()); // +d2+d1
    d3 := activate(as3()); // +d3+d2+d1
    deactivate(d2); // +d3+d1
    d2 := activate(as2()); // +d2+d3+d1
```
The number of alt statements used for modeling a single interleave statement grows exponentially with the number of blocking operations used within the interleave statement.
Execution segments are shown with arrows. Alternative segments are evaluated using snapshot semantics and executed interleaved.
INTERLEAVE RESTRICTIONS

- Guard must be empty
- No control statements (for, while, do-while, goto, stop, repeat, return) permitted in interleave branches
- No activate/deactivate, no altstep invocation
- No call of functions including communication operations
<table>
<thead>
<tr>
<th>Statement</th>
<th>Keyword or symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential behaviour</td>
<td>...; ...; ...</td>
</tr>
<tr>
<td>Alternative behaviour</td>
<td>alt { ... }</td>
</tr>
<tr>
<td>Interleaved behaviour</td>
<td>interleave { ... }</td>
</tr>
<tr>
<td>Activate default</td>
<td>activate</td>
</tr>
<tr>
<td>Deactivate default</td>
<td>deactivate</td>
</tr>
<tr>
<td>Returning control</td>
<td>return</td>
</tr>
<tr>
<td>Repeating an alt, altstep or default</td>
<td>repeat</td>
</tr>
</tbody>
</table>
XV. SAMPLE TEST CASE IMPLEMENTATION

TEST PURPOSE IN MSC
TEST CONFIGURATION
MULTIPLE IMPLEMENTATIONS

CONTENTS
**SAMPLE TEST CASE IMPLEMENTATION**

- Single component test configuration

- Test purpose defined by MSC:
  - Simple request-response protocol
  - Answer time less than 5 s
  - Result is pass for displayed operation, otherwise the verdict shall be fail
FIRST IMPLEMENTATION
WITHOUT TIMING CONSTRAINTS

```haskell
type port PT message {
    out A, B, C;
    in  X, Y, Z;
}

type component CT {
    port PT P;
}

testcase test1() runs on CT {
    map(mto:P, system:P);
    P.send(a);
    P.receive(x);
    P.send(b);
    P.receive(y);
    P.send(c);
    P.receive(z);
    setverdict(pass);
}
```

- Test case test1 results error verdict on incorrect IUT behavior → test case is not sound!

- Lower case identifiers refer to valid data of appropriate upper case type!
SOUND IMPLEMENTATION

testcase test2() runs on CT {
    timer T:=5.0; map(mtc:P, system:P);
    P.send(a); T.start;
    alt {
        [] P.receive(x) (setverdict(pass))
        [] P.receive (setverdict(fail))
        [] T.timeout (setverdict(inconc))
    }
    P.send(b); T.start;
    alt {
        [] P.receive(y) (setverdict(pass))
        [] P.receive (setverdict(fail))
        [] T.timeout (setverdict(inconc))
    }
    P.send(c); T.start;
    alt {
        [] P.receive(z) (setverdict(pass))
        [] P.receive (setverdict(fail))
        [] T.timeout (setverdict(inconc))
    }
}

\[
\text{type port PT message} \{
\begin{align*}
\text{out A, B, C; } & \\
\text{in X, Y, Z; } & \\
\end{align*}
\}
\text{type component CT} \{
\begin{align*}
\text{port PT F;} & \\
\end{align*}
\}
\]

- This test case works fine, but its operation is hard to follow between copy/paste lines!
ADVANCED IMPLEMENTATION

testcase test3() runs on CT {
    var default d := activate(as());
    map(mtc:P, system:P);
    P.send(a); T.start;
    P.receive(x);
    P.send(b); T.start;
    P.receive(y);
    P.send(c); T.start;
    P.receive(z);
    deactivate(d);
    setverdict(pass);
}

altstep as() runs on CT {
    [] P.receive {setverdict(fail)}
    [] T.timeout {setverdict(inconc)}
}

type port PT message {
    out A, B, C;
    in X, Y, Z;
}

type component CT {
    timer T := 5.0;
    port PT P;
}

• This example demonstrates one specific use of defaults
• Compact solution employing defaults for handling incorrect IUT behavior
COURSE EVALUATION

HELP US TO IMPROVE THE COURSE!

PLEASE FILL IN THE EVALUATION FORM AT

HTTP://TTCN.ERICSSON.SE/ => TRAINING SERVICES => EVALUATION FORM
HTTP://INTERNAL.ERICSSON.COM/PAGE/HUB_INSIDE/SUPPORT/RD/TOOLS/TITAN/TRAINING/EVALUATION_FORM/INDEX.JSP

THE END