

Programmers' Technical Reference Guide for the TITAN TTCN-3 Toolset

Jenő Balaskó, Botond Baranyi, Ádám Knapp

Version 11.2.1, 2026-08-05

Table of Contents

1. About the Document . . . 2
   1.1. Purpose . . . 2
   1.2. Target Groups . . . 2
   1.3. Typographical Conventions . . . 2
2. TTCN-3 Limitations in this Version . . . 3
3. Clarifications to the TTCN-3 Standard . . . 7
   3.1. Predefined Function Identifiers . . . 7
   3.2. Meaning of any and all . . . 7
   3.3. Response and Exception Handling Parts . . . 8
   3.4. Variable Lists in param Redirect . . . 8
   3.5. References between Language Elements . . . 8
   3.6. Encoding Rules . . . 9
   3.7. Address Type . . . 10
   3.8. Importing import Statement from TTCN-3 Modules . . . 11
   3.9. Description of Behavior Types Syntax . . . 12
   3.10. Partially initialized structure values . . . 12
   3.11. Concatenation of templates . . . 13
   3.12. The predefined function replace . . . 13
   3.13. ASN.1 extension additions . . . 14
   3.14. Runs-on, mtc, and system compatibility . . . 14
   3.15. Embedded fields . . . 14
   3.16. Handling of `out` parameters . . . 15
   3.17. Referencing elements of non-specific templates . . . 16
4. TTCN-3 Language Extensions . . . 18
   4.1. Syntax Extensions . . . 18
   4.2. Visibility Modifiers . . . 20
   4.3. The `anytype` . . . 22
   4.4. Ports and Test Configurations . . . 24
   4.5. Parameters of create Operation . . . 24
   4.6. Altsteps and Defaults . . . 25
   4.7. Interleave Statements . . . 26
   4.8. Logging Disambiguation . . . 26
   4.9. Value Returning done . . . 28
   4.10. Dynamic Templates . . . 30
   4.11. Template Module Parameters . . . 30
   4.12. Predefined Functions . . . 31
   4.13. Additional Predefined Functions . . . 34
   4.14. Exclusive Boundaries in Range Subtypes . . . 40

4.15. Special Float Values Infinity and not_a_number . . . 40
4.16. TTCN-3 Preprocessing . . . 40
4.17. Parameter List Extensions . . . 41
4.18. `function`, `altstep` and `testcase` References . . . 42
4.19. Function Types with a RunsOn_self Clause . . . 42
4.20. TTCN-3 Macros . . . 45
4.21. Component Type Compatibility . . . 47
4.22. Implicit Message Encoding . . . 49
4.23. RAW Encoder and Decoder . . . 67
4.24. TEXT Encoder and Decoder . . . 126
4.25. XML Encoder and Decoder . . . 134
4.26. JSON Encoder and Decoder . . . 157
4.27. OER Encoder and Decoder . . . 203
4.28. PER Encoder and Decoder . . . 203
4.29. Build Consistency Checks . . . 205
4.30. Negative Testing . . . 209
4.31. Testcase Stop Operation . . . 228
4.32. Catching Dynamic Test Case Errors . . . 228
4.33. Lazy Parameter Evaluation . . . 230
4.34. Differences between the Load Test Runtime and the Function Test Runtime . . . 231
4.35. Profiling and code coverage . . . 236
4.36. Defining enumeration fields with values known at compile time . . . 242
4.37. Ports with translation capability . . . 243
4.38. Real-time testing features . . . 246
4.39. Object-oriented features . . . 248
4.40. Default alternatives of union types . . . 256
4.41. Advanced matching . . . 257
4.42. Logging sensitive data . . . 258
5. Supported ASN.1 Constructs and Limitations . . . 260
6. Compiling TTCN-3 and ASN.1 Modules . . . 262
6.1. Command Line Syntax . . . 262
6.2. The Compilation Process for TTCN-3 and ASN.1 Modules . . . 288
6.3. Particularities of ASN.1 Modules . . . 290
6.4. Using Component Relation Constraints from TTCN-3 . . . 293
7. The Run-time Configuration File . . . 295
7.1. [MODULE_PARAMETERS] . . . 295
7.2. [LOGGING] . . . 300
7.3. `[TESTPORT_PARAMETERS]` . . . 328
7.4. `[DEFINE]` . . . 329
7.5. `[INCLUDE]` . . . 334
7.6. `[ORDERED_INCLUDE]` . . . 335

7.7. [EXTERNAL_COMMANDS] . . . 336
7.8. [EXECUTE] . . . 337
7.9. [GROUPS] (Parallel mode) . . . 338
7.10. [COMPONENTS] (Parallel mode) . . . 339
7.11. [MAIN_CONTROLLER] (Parallel mode) . . . 340
7.12. [PROFILER] . . . 341
7.13. Dynamic Configuration of Logging Options . . . 346
8. The TITAN Project Descriptor File . . . 350
8.1. Project Name . . . 352
8.2. Referenced Projects . . . 352
8.3. Files and Folders . . . 353
8.4. Path Variables . . . 354
8.5. ActiveConfiguration . . . 354
8.6. Configurations . . . 355
8.7. Packed Referenced Projects . . . 364
8.8. Important Information, Limitations . . . 368
9. XSD to TTCN-3 Converter . . . 370
9.1. Terminology . . . 370
9.2. Schema Component . . . 370
9.3. Command-line Syntax . . . 370
9.4. The Compilation Process for XML Schema . . . 372
9.5. Restrictions . . . 376
9.6. Extensions . . . 376
10. Code Coverage of TTCN-3 Modules . . . 378
10.1. Generating Code Coverage . . . 378
10.2. Converting Code Coverage Data from XML to HTML . . . 379
10.3. Command Line Syntax of tcov2lcov . . . 379
10.4. Limitations . . . 380
11. The TTCN-3 Debugger . . . 381
11.1. Gathered information . . . 381
11.2. Breakpoints . . . 382
11.3. User interface and list of commands . . . 383
11.4. Example . . . 392
12. Tips & Troubleshooting . . . 397
12.1. Type Aliasing . . . 397
12.2. Reusing Logged Values or Templates in TTCN-3 Code . . . 397
12.3. Using the TTCN-3 Preprocessing Functionality . . . 398
12.4. More Efficient Implementation of the Types record of and set of . . . 399
12.5. Workflow for Native XML Support . . . 399
12.6. Debug Memory Use of Record/set of Types . . . 406
12.7. Parsing limitations . . . 407

13. References 409
14. Abbreviations 411

Abstract

This document describes detailed information on writing components of executable test suites for the TITAN TTCN-3 Toolset.

Copyright

Copyright (c) 2000-2026 Ericsson Telecom AB.
All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v2.0 that accompanies this distribution, and is available at https://www.eclipse.org/org/documents/epl-2.0/EPL-2.0.html.

Disclaimer

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson should have no liability for any error or damage of any kind resulting from the use of this document.

Chapter 1. About the Document

1.1. Purpose

The purpose of this document is to provide detailed information on writing components, for example, test ports, and so on, for executable test suites.

1.2. Target Groups

This document is intended for programmers of TTCN-3 test suites with information in addition to that provided in the TITAN User Guide. It is recommended that the programmer reads the TITAN User Guide before reading this document.

1.3. Typographical Conventions

This document uses the following typographical conventions:

Bold is used to represent graphical user interface (GUI) components such as buttons, menus, menu items, dialog box options, fields and keywords, as well as menu commands. Bold is also used with '+' to represent key combinations. For example, **Ctrl+Click**

The character `/' is used to denote a menu and sub-menu sequence. For example, **File / Open**.

`Monospaced` font is used represent system elements such as command and parameter names, program names, path names, URLs, directory names and code examples.

`Bold monospaced` font is used for commands that must be entered at the Command Line Interface (CLI).

Chapter 2. TTCN-3 Limitations in this Version

The present Test Executor is an implementation of TTCN-3 Core Language standard ([1]) with support of ASN.1 ([3]). However, the following TTCN-3 language constructs are not supported in the current version of the Test Executor. When applicable, the relevant clause of the standard text ([1]) is given within parentheses after each limitation. The list of ASN.1 related limitations can be found in chapter **4.25**.

- C++ code generation for parameterized local templates is not supported.[1] (5.0, relevant cells of Table 1)
- Parameterized TTCN-3 `record`, `set` and `union types`. (5.4 in [1]))
- TTCN-3 sub-typing constraints are checked only at compilation time. In the run-time environment the restricted types are substituted with the corresponding base type and no run-time error is produced if the assigned value violates the subtype constraint.
- The special TTCN-3 type `anytype` is supported with restrictions. (6.2.6 in [1])
- Type compatibility of structured types.[2] (6.3 in [1])
- Two (non-empty) component types are considered to be compatible only if the compatibility relation is explicitly specified by the test suite writer. Details can be found in section 4.21. (6.3.3 and 9.3 in [1])
- Selective import statements. All TTCN-3 imports are treated as `import all`.[3] (8.2.3 and F.2 in [1])
- Type `address` must not be an external type specified outside TTCN-3. (9.6 in [1])
- The compiler does not check whether a TTCN-3 function invoked from within a `template`, Boolean guard expression of an alt construct, local variable initializer of an `altstep` or an `interleave` statement has side-effects. The run-time behavior is undefined if a function with side-effects (e.g. communication operations) is called while one of the above statements is being executed. (20 in [1])
- The `disconnect` and `unmap` operations cannot refer to multiple connections or mappings. (21.1.2, relevant parts in [1])
- Broadcast communication only works between internally connected ports. Components within the SUT will not be reached by the broadcast. (22.2.1 and 22.3.1 in [1])
- Attributes of type definitions cannot be changed when they are being imported. (27.1.2.1 in [1])
- Template instances cannot be used in the to clause of communication operations. Only values of `component` and `address` types are allowed. (stated only in BNF)
- The additional predefined function `decomp` is not implemented. (D.2 of [3])
- In `port type` definitions the list of incoming and outgoing message types or signatures must be explicitly specified, the `all` keyword is ignored by the compiler. (G.3 in [1])
- The TTCN-3 and ASN.1 modules are identified only by their names. Object identifiers in module headers are ignored. Module object identifiers in `import` statements and references are skipped without any checking, the semantic analyzer uses the module identifier only. (7.2.3 of [3], 8.1 in [1])

- The comparison operators do not work on `objid` values. Only the equality (==) and non-equality (!=) operators are allowed. (7.2.5.2 of [3], 7.1.3 in [1])
- Templates can not be used in the parameter of `encvalue` built-in function. (C.38 in [1])
- The declaration of object identifiers can only point to constant values and integer variables, references to `objid` variables are not supported.
- The Configuration and Deployment Support and the Advanced Parameterization packages of the TTCN-3 standard are not supported yet, except the Port with translation capability clause. ([21]).
- In contrast to the standard, TITAN does not allow applying the same name to a structured type and to an element of the same type.
- From version 1.8.pl3 (or R8D) the logging machinery uses an internal TTCN-3 module, named `TitanLoggerApi`, hence using this module name in user code is not allowed.
- Referencing into an omitted field of any non-const variable/template of record/set type is allowed and it will expand the structure to the level of reference. All the expanded fields under `omit` will be unbound. This behavior is TITAN specific. According to the TTCN-3 standard (15.6.2 of [1]), the proper behavior would be a dynamic test case error in this situation. In case of variable templates referencing into a matching mechanism will change the template regardless of it being a left hand side or a right hand side value. In case of non-variable templates referencing into a matching mechanism will cause an error. According to the TTCN-3 standard the proper behavior for right hand side templates would be to return an expanded value but not change it's own value in case of AnyValue matching mechanism or stop with an error in case of other matching mechanisms.
- According to the standard, before matching the tools have to make sure that the template being used is completely initialized, with no fields or elements left unbound. For performance reasons this check is not done before the matching is done. Instead the matching will report the error, when it tries to use an unbound field or elements.
- In case the compiler is not able to decide at compile time, if all possible execution branches contain a return statement, that is, in cases of alt statements, loops and branching statement like if-else, select case, and so on, it will report an error without generating code. For example:

```
function f_check() return boolean {
  for (var integer i:=0; i < some_variable; i := i + 1) {
    return true;
    }
  }
```

  In this case the compiler will report an error as it can not evaluate, if the loop will be executed at least once, and if the loop is not executed, the end of the function would be reached without a return statement. The workaround for this kind of problem is easy, the user needs to insert an extra return statement at the end of the function, like:

```
function f_check() return boolean {
  for (var integer i:=0; i < some_variable; i := i + 1) {
    return true;
    } return false
  }
```

- The language specification, after the "language" keyword, is ignored by the compiler.
- For record of/set of types of fixed size, which have a length restriction of one concrete value, and arrays the `sizeof()` and `lengthof()` predefined functions are not standard compliant: `sizeof()` returns the number of elements, `lengthof()` returns the index of the last initialized element plus one.
- IPv6 networking between the MC, HC and Parallel Test Components is supported only on Linux and Cygwin 1.7.
- The `optional "implicit omit"` attribute can be applied directly to global value and template definitions, but not to local value and template definitions.
- The `optional "implicit omit"` attribute can be applied to a module, in which case it will have effect on global value and template definitions in the module, and local value and template definitions in the module, with the exception of (local) variable definitions
- Templates using the `decmatch` (decoded content match, B.1.2.9 in [1]) matching mechanism cannot be sent through test ports (doing so will result in a dynamic test case error). Template module parameters using `decmatch` are also not supported.
- Since TITAN version R5B the matching symbol "*" (AnyValueOrNone, B.1.2.4 in [1]) causes a compile time error when assigned to a mandatory field of a record or set template, as it is stated in the standard. This breaks backwards compatibility because in the older versions of TITAN only a warning was emitted.
- When assigning a value to a structure using the value list notation, assignment notation or index notation (but not when assigning values to fields or elements one at a time), if the structure's old value (or part of it) is referenced on the right hand side, the structure's new value will only contain the fields or elements set in that assignment. All other fields or elements that may have been initialized in prior assignments will be set to unbound.

  If the structure's old value is not referenced on the right hand side of the assignment, then only the fields or elements mentioned in the assignment will be overwritten. All other fields or elements will retain their previous values. Example:

```
type record R {
  integer i1,
  integer i2,
  integer i3
}

...

var R x := { 1, 2, 3 };

x := { i2 := 3 }; // assignment notation with no self-reference (OK)
// result: x := { i1 := 1, i2 := 3, i3 := 3 }

x := { i1 := x.i2 }; // assignment notation with self-reference (not OK)
// result: x := { i1 := 3, i2 := <unbound>, i3 := <unbound> }

x.i3 := x.i1; // individual field assignment with self-reference (OK)
// result: x := { i1 := 3, i2 := <unbound>, i3 := 3 }
```

- Declaring multiple user ports (i.e. non-internal ports) with the same name is not fully supported. The generated headers of two modules containing user ports with the same name will cause C++ compilation errors, if one of the modules imports the other, or if it imports a module that imports the other, etc. It is advised to give all user ports unique names.

[1] The semantic analyzer is able to verify modules with such definitions, but the generated C++ code will be incomplete or erroneous.

[2] Type compatibility for structured types is enabled only in the function test run-time due to performance considerations (except record of/set of types for certain element types, see section 4.32.2). In the load test run-time aliased types and sub-types are treated to be equivalent to their unrestricted root types. Different structured types are incompatible to each other. Two array types are compatible if both have the same size and index offset and the element types are compatible according to the rules above.

[3] Recursive and non-recursive import means exactly the same when importing all definitions from a module.

Chapter 3. Clarifications to the TTCN-3 Standard

The TTCN-3 Core Language standard ([1]) and its Operational Semantics ([1]) give ambiguous description for some language constructs. This section specifies our resolution for these ambiguities that was followed during the implementation of our compiler and run-time environment.

3.1. Predefined Function Identifiers

The standard does not clarify the status of predefined function identifiers, that is, the names of functions defined in Annex C of [1]. In our interpretation these words cannot be used to identify userdefined TTCN3 entities because such a definition would hide the predefined function completely. Thus our compiler treats these identifiers in the same way as the normal keywords of the language. Therefore the inappropriate use of predefined functions, for example wrong number of arguments, will result in syntax errors rather than semantic errors.

3.2. Meaning of any and all

The meaning of the keywords is only loosely defined in the standard. The resulting equivocality concerns timer, port and component operations.

3.2.1. Timer and Port Operations

The meaning of keywords `any` and `all` in timer and port operations is unclear. These constructs might be resolved statically at compilation time by applying the operation on all visible timers and ports of the given scope unit. Our run-time environment, however, implements a dynamic resolution, that is, it walks through the list of active timers and ports and applies the respective operation. As a consequence of this, such operations are also applicable in scope units without visible timers and ports, for example in functions without `runs on` clause. Because of the run-time evaluation there is one limitation, which is verified by our semantic analyzer: the receiving port operations, that is, `receive`, `trigger`, `getcall`, `getreply`, `catch` and `check`) that refer to any port cannot have template parameter and `value` or `param` redirect. To avoid incompatibilities with future versions it is not recommended to use `any` or `all` in timer and port operations.

3.2.2. Component Operations

The standard does not specify explicitly the behavior of the component operations that refer to `all component` when only the MTC exists, that is, no PTC had been created during the testcase. In our implementation both `all component.running` and `all component.alive` return `true` and the operations `all component.done` and `all component.killed` succeed immediately in this situation. Operations `all component.stop` and `all component.kill` do nothing; instead, a warning is issued. The same rules are applied in single mode, when it is impossible to create PTCs, as well.

3.3. Response and Exception Handling Parts

The behavior of the response and exception handling part of a `call` operation is not clearly specified in the standard. The allowed `getreply` and `catch` operations can handle only the possible responses and exceptions of the previous signature call. In our implementation if any other event arrives into the port queue during the execution of the response and exception handling part it may block the execution forever. The runtime environment generates a dynamic test case error in such a situation. If the test suite writer expects any other event on the same port during the outstanding call, for example a simultaneous incoming call initiated by the other side, a non-blocking call operation with the keyword `nowait` should be used. The response and the possible incoming calls should be handled in a forthcoming regular alt construct using the appropriate `getreply` and `getcall` operations.

3.4. Variable Lists in param Redirect

In the standard, it is not clear that the `VariableList` notation in the `param` redirect of `getcall` and `getreply` operations should refer to **all** parameters of the respective signature or to the **relevant** parameters[4] only. Our compiler expects variable entries only for the relevant parameters and ignores the irrelevant ones. This is because otherwise the test writer should use `NotUsedSymbols` for all irrelevant parameters, which would be a redundant notation. For example, if a signature has one `in`, one `out` and one `inout` parameter the compiler expects two variable entries in both `getcall` and `getreply` operations.

3.5. References between Language Elements

The TTCN-3 standard does not specify clearly the permitted references between different kinds of language elements. The following table shows our interpretation.

Table 1. References between TTCN-3 elements

Referred element Referring element	**Literal value**	**Constant**	**External constant**	**Module parameter**	**Template**
Constant	Y	Y*	N	N	N
Array size	Y	Y	N	N	N
Subtype constraint	Y	Y	N	N	N
Default value of module parameter	Y	Y	Y	N	N

Referred element Referring element	Literal value	Constant	External constant	Module parameter	Template
Actual value of module parameter (in configuration file)	Y	N	N	N	N
Default duration of timer	Y	Y	Y	Y	N
Template (non-parameterized)	Y	Y	Y	Y	Y*

Legend:

- N Not allowed by the TTCN-3 language.
- Y Allowed and fully supported by the current version of this TTCN-3 tool.
- Y* Allowed and fully supported, but circular reference chains must be avoided.

NOTE

- The above table implies that the value of all constants and the attributes of all type constructs (type constraints, array sizes, etc.) shall be known at compilation time.
- ASN.1 value assignments are treated as TTCN-3 constants.
- The value of constants shall refer only to built-in operators or additional predefined functions.
- The body of non-parameterized templates and the default duration of timers shall be known at test startup (load) time when all module parameters are known.
- The actual parameters of templates or the actual duration of timers shall be determined run-time because the actual value of variables may be referred.
- The rules for a language element do not depend on its scope unit. For example the same rules apply on module, component and local (function, testcase, altstep) constants.

3.6. Encoding Rules

The standard does not specify clearly some of the encoding rules.

- The encoding of fields in `record`, `set` and `union` types is supported.
- The order of attributes of the same type in a `with` statement is important. The second variant might override the first, or an overriding attribute will override all the following attributes of

the same type.

- Encode attributes are an exception to this as they are not really attributes, but "contexts". It cannot be determined to which encode "contexts" the variants of the same `with` statement should belong if there are several. As having several encode "contexts" in the same `with` statement would be a bad coding practice, a warning is generated and the last encode is used as the statement's encode "contexts".
- As encodes are contexts, an encode is only overridden if the overriding context is not the same.
- The order of attributes of different type in a `with` statement is not important, they do not affect each other.
- In case of structured types, the encode context of the type is the encode context of its fields too, if the fields do not override this attribute. The other attribute types are handled separately for the structured type and its fields. Attributes inherited from higher level (module/group/structured type) might change the encoding of a record and that of its fields.
- Attributes with qualifiers referring to the same field are handled as if they were separate `with` statements. The same rules apply to them. For example, the last encode from the ones referring to the same field is taken as the encoding context of the field.
- Attributes belonging to a field of a structured type or a type alias have the following overwriting rules. A new `variant` attribute together with the directive `override` clears all current attributes defined for the type of the field. A new `variant` attribute without the directive override overwrites only the current `variant` attribute, all other attributes remain unchanged.

3.7. Address Type

The standard does not specify clearly the status of special TTCN-3 type `address`. Our implementation is based on the rules below.

The test suite writer can assign the name `address` to a regular data type. There can be at most one type named `address` in each TTCN-3 module. It is allowed that different modules of the test suite assign the name `address` to different types.

The name `address` cannot be assigned to the following TTCN-3 types:

- port types
- component types[5]
- signatures
- the built-in type default[6]

Whenever the word `address` is used as a type, it is assumed to be a reference to the type named `address` in the current module. The type named `address` cannot be imported into another TTCN-3 module, that is, it can be referenced using the name `address` only within its own module. If one wants to use this type in other modules a regular alternate name must be assigned to it with type aliasing.

Addressing the SUT in communication operations is allowed only if the `address` type is defined in the same module as the corresponding port type. In addition, the port type must have a special

`extension` attribute to support `address` values (See section "Support of address type" in [16] for more details).

Note that it is possible to use different address types on different ports in the same TTCN-3 module if the respective port types are imported from different modules, but neither address type may be referenced with name `address` by the importing module.

3.8. Importing import Statement from TTCN-3 Modules

See [18] standard for detailed description. Additional information for better understanding:

- Import (see following chapters of the [18] standard 8.2.3.1-8.2.3.6, and 8.2.5, only applies for global definitions (see [18] table 8. in 8.2.3.1), therefore import functionality is not interfered by import of import statement.
- Import statement can be imported by only import of import statement (chapter 8.2.5 and 8.2.3.7).
- Import statements are by default private, importing of import statement with public or friend visibility is recursively resolved, and thus importing of importing of import statement is possible.
- Importing of import statement - in case of friend visibility -recursive resolving is broken, if the import chain has a member that is not friend of the exporting module.
- Importing of import statement circular import chain causes error.
- Example for friend type and importing of import statement

```
B.ttcn // friend template
friend module C, E;

friend template integer t_B_i_fr := 0;

C.ttcn // public import and importing of import statement, friend of B
public import from B all;
public import from B {import all};

D.ttcn // public import and importing of import statement, NOT friend of B
public import from C all;
public import from C { import all };

E.ttcn // public import and importing of import statement, friend of B
public import from D { import all };
public import from D all;

testcase tc_B() runs on MTC {
var integer i:=valueof(t_B_i_fr); //Visible!
setverdict(pass);
}
```

3.9. Description of Behavior Types Syntax

TITAN supports the behaviour type package of the TTCN-3 standard, but with a different syntax. For details of the behaviour types see [5].

Table 2. Behaviour types - refers shows the different syntax of the function behaviour type.

Standard (6.2.13.2 in [5])	**Titan specific syntax**
type function MyFunc3 (in integer p1) return charstring;	var MyFunc3 myVar1 := refers(int2char);

NOTE The functionality is same as in the standard, only the syntax is different.

The syntax of the apply operation is different, Table 3 Behaviour types - apply and derefers

Standard:

Table 3. Behaviour types - apply and derefers

Standard (6.2.13.2 in [5])	**Titan specific syntax**
type function MyFuncType ();	v_func.apply(MyVar2)
type function t_functionstartTests();	vl_comp.start(derefers(vl_function2)());

3.10. Partially initialized structure values

According to the standard TTCN-3 variables and module parameters (of structured types) can be in 3 different states during their initialization:

- *uninitialized* (or unbound) - none of the value's fields or elements has been initialized - values in this state cannot be copied or used on the right hand side of an operation;
- *partially initialized* - some of the value's fields or elements have been initialized, but not all of them (or at least not enough to meet the minimum type restrictions) - these values can be copied, but cannot be used on the right hand side of an operation;
- *fully initialized* (or bound) - all of the value's fields or elements have been initialized - these values are ready to be used on the right hand side of an operation.

The `isbound` operation should only return `true` if the value is in the 3rd (fully initialized) state.

This isn't the case in the TITAN runtime. Values only have 2 states: *bound* and *unbound*, which is what the `isbound` operation returns. This can be any combination of the previously mentioned 3 states, depending on the type:

- `record` / `set`: unbound = uninitialized, bound = at least partially initialized, meaning that a `record` / `set` is bound if at least one of its fields is bound[7];
- `record of` / `set of`: unbound = uninitialized, bound = at least partially initialized, meaning that the record of is only unbound if it has never received an initial value (even initializing with {} creates a bound `record of` / `set of` value);

- `array`: unbound = uninitialized or partially initialized, bound = fully initialized, meaning that the array is only bound if all of its elements are bound;
- `unions` can't be partially initialized, so TITAN stores their bound state correctly (although it's still possible to create `union` values, where the selected alternative is unbound, with the legacy command line option `-B`; these values would be considered bound by TITAN).

There is a workaround in TITAN's implementation of `records` / `sets` to allow the copying of partially initialized values (`union` values with unbound selected alternatives can also be copied when the compiler option `-B` is set). In all other cases the user is responsible for making sure the value is usable on the right hand side of an operation. The `isbound` function is usually not enough to ensure, that the value is usable.

3.11. Concatenation of templates

TITAN supports the concatenation of templates and template variables of string types (`bitstring`, `hexstring`, `octetstring`, `charstring`, `universal charstring`) and list types (`record of`, `set of`) with the following limitations:

- templates can only be concatenated in the Function Test runtime;
- valid concatenation operands (for binary string and list types):
  - specific values (i.e. literal values),
  - any value ("?"") with no length restriction or with a fixed[8] length restriction,
  - any value or none ("*") with a fixed length restriction,
  - references to constants, templates, variables, or template variables;
- operands of `charstring` and `universal charstring` template concatenation cannot contain matching mechanisms (not even patterns), only specific values and references;
- reference operands of binary string (`bitstring`, `hexstring`, `octetstring`) template concatenation can also refer to binary string templates with wildcards in addition to the template types listed as valid operands (these cannot be used in template concatenations directly, because of parser limitations);
- similarly, reference operands of `record of` or `set of` template concatenation can also refer to template lists containing matching mechanisms (but these cannot appear in template concatenations directly due to parser limitations);
- the first operand of a `record of` or `set of` template concatenation can only be a reference (because of parser limitations);
- template module parameters cannot be concatenated in the configuration file.

3.12. The predefined function replace

In TITAN the predefined function `replace` cannot be used on arrays.

If the fourth parameter of `replace` is an empty string or sequence, then it acts as a delete function (the specified substring or subsequence is simply removed from the input value and nothing is

inserted in its stead).

Example:

```
type record of integer IntList;
...
var IntList vl_myList := { 1, 2, 3 };
var IntList vl_emptyList := {};
replace(vl_myList, 1, 2, vl_emptyList); // returns { 1 }
replace("abcdef", 2, 1, ""); // returns "abdef"
replace('12FFF'H, 3, 2, ''H); // returns '12F'H
```

3.13. ASN.1 extension additions

Extension addition fields in ASN.1 `SEQUENCE` and `SET` types are treated as `optional` fields, after they are imported into TTCN-3. The ASN.1-based codecs, `BER`, `OER` and `PER`, still treat these fields as they are declared in ASN.1.

3.14. Runs-on, mtc, and system compatibility

The TTCN-3 standard allows component types to be runs-on-compatible if the caller and the called component are different, but the caller component contains all definitions in the called component. In TITAN the two components are only considered runs-on-compatible, if both run on the same component, or if the caller component extends the called component.

Mtc and system compatiblity works the same way as runs-on compatibility, if both the caller and the called entity have their mtc/system component specified.

Mtc and system compatibility is not checked if the caller or the called entity do not have their mtc/system components specified. It's possible in these cases to call a function or altstep with the wrong mtc or system component. The variables and ports of these components will be uninitialized at runtime.

3.15. Embedded fields

An embedded field can refer to any type, but its fields will only be promoted if the embedded field type is of the same structured type as the parent type (i.e. a `record` type can only have promoted fields of embedded `record` fields, a `set` type can only have promoted fields of embedded `set` fields, and the same for embedded `union` fields).

Multiple embedded fields can have fields with the same name. The compiler determines which field to promote by going through the embedded fields in the order of their declaration.

If a referenced `union` field is not found, then looking for a promoted field takes precedence over checking the `@default` alternative.

Example:

```
type union InnerUni {
  integer a, // promoted
  integer b
}

type union MidUni1 {
  InnerUni,
  integer b // promoted
}

type union MidUni2 {
  integer a,
  integer b
}

type union OuterUni {
  MidUni1,
  @default MidUni2
}

...

var OuterUni x := ...
// x.a refers to x.MidUni1.InnerUni.a
// x.b refers to x.MidUni1.b
```

3.16. Handling of `out` parameters

There are currently three different ways the TTCN-3 compiler can handle `out` function parameters:

1. `out` parameters behave exactly like `inout` parameters. This is the oldest version, that can be activated with the legacy command line switch `--legacy-out`.
2. `out` parameters behave like `inout` parameters, but are set to `unbound` at the start of the function. This is the second/previous version, that can be activated with the legacy command line switch `--legacy-out2`.
3. `out` formal parameters start as unbound at the beginning of the function, but their content at the end of the function is only passed back to the actual parameter if it has been initialized (i.e. is bound). This version also allows `out` actual parameters to be omitted (like `in` parameters with default values), in which case the final value of the formal parameter is not passed back anywhere. This is the version described by the current TTCN-3 standard, and the version that's active by default.

Example:

```
function f(out integer p1, out integer p2) {
  log(p1); // version1: 1, version2: <unbound>, version3: <unbound>
  log(p2); // version1: 2, version2: <unbound>, version3: <unbound>

  p1 := 6;
}

function f2() {
  var integer x1 := 1;
  var integer x2 := 2;

  f(x1, x2);

  log(x1); // version1: 6, version2: 6, version3: 6
  log(x2); // version1: 2, version2: <unbound>, version3: 2

  f(x1, -); // only allowed in version3
}
```

3.17. Referencing elements of non-specific templates

Accessing fields and elements of non-specific templates is handled according to chapter 15.6 of the TTCN-3 standard, with the exception, that AnyElementsOrNone (*) can only have length restrictions in the Function Test runtime.

The command line option `--legacy-template-subrefs` can be used to enable legacy handling of fields/elements of non-specific templates, which results in the following changes:

- accessing a field of a non-specific `record`, `set` or `union` template on the right hand side is always successful, and sets the template to a specific value;
- accessing a field of a non-specific `record`, `set` or `union` template on the left hand side always causes a runtime error;
- accessing an element of a `record of`, `set of` or array template on the right hand side is always successful, if the template is a specific value, omit value, AnyValue (?), AnyValueOrNone (*) or uninitialized, and sets the template to a specific value containing only uninitialized elements;
- accessing an element of a non-specific `record of`, `set of` or array template on the left hand side always causes a runtime error;
- AnyElementsOrNone (*) is considered as one element inside a `record of`, `set of` or array template, and accessing it returns an AnyValueOrNone (*) template of the element's type;
- elements inside a permutation can be accessed as if they were regular elements.

[4] Relevant parameters are the in and inout parameters in case of getcall operation as well as out and inout ones in case of getreply.

[5] If component types were allowed for addressing the compiler would not be able to decide whether a component reference in the to or from clause of a communication operation denotes a test component, which is reachable through a port connection or an address inside the SUT, which is reachable through a port mapping.

[6] The values of type default (i.e. the TTCN-3 default references) cannot be passed outside the test component by any means.

[7] The bound state of fields or elements is also determined by using the isbound operation on the field or element.

[8] In this case a range length restriction, whose upper and lower bounds are equal, is also considered as a `fixed' length restrictione.g.: ? length(2..2) is a valid operand, but ? length(2..3) is not

Chapter 4. TTCN-3 Language Extensions

The Test Executor supports the following non-standard additions to TTCN-3 Core Language in order to improve its usability or provide backward compatibility with older versions.

4.1. Syntax Extensions

The compiler does not report an error or warning if the semi-colon is missing at the end of a TTCN-3 definition although the definition does not end with a closing bracket.

The statement block is optional after the guard operations of `altsteps`, `alt` and `interleave` constructs and in the response and exception handling part of `call` statements. A missing statement block has the same meaning as an empty statement block. If the statement block is omitted, a terminating semi-colon must be present after the guard statement.

The standard escape sequences of C/C++ programming languages are recognized and accepted in TTCN-3 character string values, that is, in literal values of `charstring` and `universal charstring` types, as well as in the arguments of built-in operations `log()` and `action()`.

NOTE	As a consequence of the extended escape sequences and in contrast with the TTCN-3 standard, the backslash character itself has to be always duplicated within character string values.

The following table summarizes all supported escape sequences of TTCN-3 character string values:

Table 4. Character string escape sequences

Escape sequence	Character code (decimal)	Meaning
	7	bell
	8	backspace
	12	new page
	10	line feed
	13	carriage return
	9	horizontal tabulator
11	vertical tabulator	
\	92	backslash
"	34	quotation mark
'	39	apostrophe
?	63	question mark
<newline>	nothing	line continuation
	NNN	octal notation (NNN is the character code in at most 3 octal digits)

Escape sequence	Character code (decimal)	Meaning
	NN	hexadecimal notation (NN is the character code in at most 2 hexadecimal digits)
""	34	quotation mark (standard notation of TTCN-3)

NOTE	Only the standardized escape sequences are recognized in matching patterns of character string templates because they have special meaning there. For example, inside string patterns `\n` denotes a set of characters rather than a single character.

Although the standard requires that characters of TTCN-3 `charstring` values must be between 0 and 127, TITAN allows characters between 0 and 255. The printable representation of characters with code 128 to 255 is undefined.

The compiler implements an ASN.1-like scoping for TTCN-3 enumerated types, which means it allows the re-use of the enumerated values as identifiers of other definitions. The enumerated values are recognized only in contexts where enumerated values are expected; otherwise the identifiers are treated as simple references. However, using identifiers this way may cause misleading error messages and complicated debugging.

The compiler allows the local definitions (constants, variables, timers) to be placed in the middle of statement blocks, that is, after other behavior statements. The scope of such definitions extends from the statement following the definition to the end of the statement block. Forward-referencing of local definitions and jumping forward across them using `goto` statements are not allowed.

The compiler accepts in-line compound values in the operands of TTCN-3 expressions although the BNF of the standard allows only single values. The only meaningful use of the compound operands is with the comparison operators, that is, == and !=. Two in-line compound values cannot be compared with each other because their types are unknown; at least one operand of the comparison must be a referenced value. This feature has a limitation: In the places where in-line compound templates are otherwise accepted by the syntax (e.g. in the right-hand side of a variable assignment or in the actual parameter of a function call) the referenced value shall be used as the left operand of the comparison. Otherwise the parser gets confused when seeing the comparison operator after the compound value.

Examples:

```
// invalid since neither of the operands is of known type
if ({ 1, 2 } == { 2, 1 }) { }

// both are valid
while (v_myRecord == { 1, omit }) { }
if ({ f1 :=1, f2 := omit } != v_mySet) {}

// rejected because cannot be parsed
v_myBooleanFlag := { 1, 2, 3 } == v_myRecordOf;
f_myFunctionTakingBoolean({ 1, 2, 3 } != v_mySetOf);

// in reverse order these are allowed
v_myBooleanFlag := v_myRecordOf == { 1, 2, 3 };
f_myFunctionTakingBoolean(v_mySetOf != { 1, 2, 3 });
```

4.2. Visibility Modifiers

TITAN defines 3 visibility modifiers for module level definitions, and component member definitions: public, private, friend (8.2.5 in [1]).

On module level definitions they mean the following:

- The public modifier means that the definition is visible in every module importing its module.
- The private modifier means that the definition is only visible within the same module.
- The friend modifier means that the definition is only visible within modules that the actual module declared as a friend module.

If no visibility modifier is provided, the default is the public modifier.

In component member definitions they mean the followings:

- The public modifier means that any function/testcase/altstep running on that component can access the member definition directly.
- The private modifier means that only those functions/testcases/altsteps can access the definition which runs on the component type directly. If they run on a component type extending the one containing the definition, it will not be directly visible.

The friend modifier is not available within component types.

Example:

```
module module1
{
import from module2 all;
import from module3 all;
import from module4 all;

const module2Type akarmi1 := 1; //OK, type is implicitly public
const module2TypePublic akarmi2 := 2; //OK, type is explicitly public
const module2TypeFriend akarmi3 := 3; //OK, module1 is friend of module2
const module2TypePrivate akarmi4 := 4; //NOK, module2TypePrivate is private to module2

const module3Type akarmi5 := 5; //OK, type is implicitly public
const module3TypePublic akarmi6 := 6; //OK, type is explicitly public
const module3TypeFriend akarmi7 := 7; //NOK, module1 is NOT a friend of module3
const module3TypePrivate akarmi8 := 8; //NOK, module2TypePrivate is private to module2

type component User_CT extends Lib4_CT {};
function f_set3_Lib4_1() runs on User_CT { v_Lib4_1 := 0 } //OK
function f_set3_Lib4_2() runs on User_CT { v_Lib4_2 := 0 } //OK
function f_set3_Lib4_3() runs on User_CT { v_Lib4_3 := 0 } //NOK, v_Lib4_3 is private
}

module module2
{

friend module module1;

type integer module2Type;
public type integer module2TypePublic;
friend type integer module2TypeFriend;
private type integer module2TypePrivate;
} // end of module

module module3
{
type integer module3Type;
public type integer module3TypePublic;
friend type integer module3TypeFriend;
private type integer module3TypePrivate;
} // end of module

module module4 {
type component Lib4_CT {
var integer v_Lib4_1;
public var integer v_Lib4_2;
private var integer v_Lib4_3;
}
```

4.3. The `anytype`

The special TTCN-3 type `anytype` is defined as shorthand for the union of all known data types and the address type (if defined) in a TTCN-3 module. This would result in a large amount of code having to be generated for the `anytype`, even if it is not actually used. For performance reasons, Titan only generates this code if a variable of `anytype` is declared or used, and does not create fields in the `anytype` for all data types. Instead, the user has to specify which types are needed as `anytype` fields with an extension attribute at module scope.

Examples:

```
module elsewhere {
  type float money;
  type charstring greeting;
}
module local {
  import from elsewhere all;
  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";
  // ischosen(v_any.charstring) == true

  v_any.greeting := "hello";
  // ischosen(v_any.charstring) == false
  // ischosen(v_any.greeting) == true

  v_any.MyRec := { i := 42, f := 0.5 }
  // ischosen(v_any.MyRec) == true

  v_any.integer := v_any.MyRec.i - 2;
  // back to ischosen(v_any.integer) == true v_any.money := 0;
  // local money i.e. integer
  // not elsewhere.money (float)
  // ischosen(v_any.integer) == false
  // ischosen(v_any.money) == true

  // error: no such field (not added explicitly)
  // v_any.float := 3.1;

  // error: v_any.elsewhere.money
 }
}

with {

extension "anytype integer, charstring" // adds two fields
extension "anytype MyRec" // adds a third field
extension "anytype money" // adds the local money type
//not allowed: extension "anytype elsewhere.money"
extension "anytype greeting" // adds the imported type}
```

In the above example, the `anytype` behaves as a union with five fields named "integer", "charstring",

"MyRec", "money" and "greeting". The anytype extension attributes are cumulative; the effect is the same as if a single extension attribute contained all five types.

NOTE	Field "greeting" of type charstring is distinct from the field "charstring" even though they have the same type (same for "integer" and "money").

Types imported from another module (elsewhere) can be added to the anytype of the importing module (local) if the type can be accessed with its unqualified name, which requires that it does not clash with any local type. In the example, the imported type "greeting" can be added to the anytype of module local, but "money" (a float) clashes with the local type "money" (an integer). To use the imported "money", it has to be qualified with its module name, for example a variable of type elsewhere.money can be declared, but elsewhere.money can not be used as an anytype field.

4.4. Ports and Test Configurations

If all instances of a TTCN-3 port type are intended to be used for internal communication only (i.e. between two TTCN-3 test components) the generation and linking of an empty Test Port skeleton can be avoided. If the attribute `with { extension "internal" }` is appended to the port type definition, all C++ code that is needed for this port will be included in the output modules.[9]

If the user wants to use `address` values in `to` and `from` clause and sender redirect of TTCN-3 port operations the `with { extension "address" }` attribute shall be used in the corresponding port type definition(s) to generate proper C++ code.

NOTE	When address is used in port operations the corresponding port must have an active mapping to a port of the test system interface, otherwise the operation will fail at runtime. Using of address values in to and from clauses implicitly means system as component reference. (See section "Support of address type" in [16] for more details).[10]

Unlike the latest TTCN-3 standard, our run time environment allows to connect a TTCN-3 port to more than one ports of the same remote test component. When these connections persist (usually in transient states), only receiving is allowed from that remote test component, because the destination cannot be specified unambiguously in the `to` clause of the `send` operation. Similarly, it is allowed to map a TTCN-3 port to more than one ports of the system, although it is not possible to send messages to the SUT.

4.5. Parameters of create Operation

The built-in TTCN-3 `create` operation can take a second, optional argument in the parentheses. The first argument, which is the part of the standard, can assign a name to the newly created test component. The optional, non-standard second argument specifies the location of the component. Also the second argument is a value or expression of type `charstring`.

According to the standard the component name is a user-defined attribute for a test component, which can be an arbitrary string value containing any kind of characters including whitespace. It is not necessary to assign a unique name for each test component; several active test components can

have the same name at the same time. The component name is not an identifier; it cannot be used to address test components in configuration operations as component references can. The name can be assigned only at component creation and it cannot be changed later.

Component name is useful for the following purposes:

- it appears in the printout when logging the corresponding component reference;
- it can be incorporated in the name of the log file (see the metacharacter `%n`);
- it can be used to identify the test component in the configuration file (when specifying test port parameters (see section `[LOGGING]`), component location constraints (see section [COMPONENTS] (Parallel mode)) and logging options (see sections `FileMask` and `ConsoleMask`).

Specifying the component location is useful when performing distributed test execution. The value used as location must be a host name, a fully qualified domain name, an IP address or the name of a host group defined in the configuration file (see section [GROUPS] (Parallel mode)). The explicit specification of the location overrides the location constraints given in the configuration file (see section [COMPONENTS] (Parallel mode) for detailed description). If no suitable and available host is found the `create` operation fails with a dynamic test case error.

If only the component name is to be specified, the second argument may be omitted. If only the component location is specified a `NotUsedSymbol` shall be given in the place of the component name.

Examples:

```
//create operation without arguments
var MyCompType v_myCompRef := MyCompType.create;

// component name is assigned
v_myCompRef := MyCompType.create("myCompName");

// component name is calculated dynamically
v_myCompArray[i] := MyCompType.create("myName" & int2str(i));

// both name and location are specified (non-standard notation)
v_myCompRef := MyCompType.create("myName", "heintel");

// only the location is specified (non-standard notation)
v_myCompRef := MyCompType.create(-, "159.107.198.97") alive;
```

4.6. Altsteps and Defaults

According to the TTCN-3 standard an `altstep` can be activated as `default` only if all of its value parameters are `in` parameters. However, our compiler and run-time environment allows the activation of altsteps with `out` or `inout` value or template parameters as well. In this case the actual parameters of the activated `default` shall be the references of variables or template variables that are defined in the respective component type. This restriction is in accordance with the rules of the standard about timer parameters of activated defaults.

NOTE	Passing local variables or timers to defaults is forbidden because the lifespan of local definitions might be shorter than the `default` itself, which might lead to unpredictable behavior if the `default` is called after leaving the statement block that the local variable is defined in. Since ports can be defined only in component types, there is no restriction about the `port` parameters of `altsteps`. These restrictions are not applicable to direct invocations of `altsteps` (e.g. in `alt` constructs).

The compiler allows using a statement block after `altstep` instances within `alt` statements. The statement block is executed if the corresponding `altstep` instance was chosen during the evaluation of the alt statement and the `altstep` has finished without reaching a `repeat` or `stop` statement. This language feature makes the conversion of TTCN-2 test suites easier.

NOTE	This construct is valid according to the TTCN-3 BNF syntax, but its semantics are not mentioned anywhere in the standard text.

The compiler accepts `altsteps` containing only an `[else]` branch. This is not allowed by the BNF as every `altstep` must have at least one regular branch (which can be either a guard statement or an `altstep` instance). This construct is practically useful if the corresponding `altstep` is instantiated as the last branch of the alternative.

4.7. Interleave Statements

The compiler realizes TTCN-3 `interleave` statements using a different approach than it is described in section 7.5 of [1]. The externally visible behavior of the generated code is equivalent to that of the canonical mapping, but our algorithm has the following advantages:

- Loop constructs `for`, `while` and `do-while` loops are accepted and supported without any restriction in `interleave` statements. The transformation of statements is done in a lower level than the TTCN-3 language, which does not restrict the embedded loops.
- Statements `activate`, `deactivate` and `stop` can also be used within `interleave`. The execution of these statements is atomic so we did not see the reason why the standard forbids them.
- The size of our generated code is linear in contrast to the exponential code growth of the canonical algorithm. In other words, the C++ equivalent of every embedded statement appears exactly once in the output.
- The run-time realization does not require any extra operating system resources, such as multi-threading.

4.8. Logging Disambiguation

The TTCN-3 log statement provides the means to write logging information to a file or display on console (standard error). Options `FileMask` and `ConsoleMask` determine which events will appear in the file and on the console, respectively. The generated logging messages are of type `USER_UNQUALIFIED`.

The `log` statement accepts among others fixed character strings TTCN-3 constants, variables, timers, functions, templates and expressions; for a complete list please refer to the table 18 in [1]. It is

allowed to pass multiple arguments to a single `log` statement, separated by commas.

The TTCN-3 standard does not specify how logging information should be presented. The following sections describe how TITAN implemented logging.

The arguments of the TTCN-3 statement `action` are handled according to the same rules as `log`.

4.8.1. Literal Free Text String

Strings entered between quotation marks (") [11] and the results of special macros given in section TTCN-3 Macros in the argument of the `log` statement are verbatim copied to the log. The escape sequences given in Table 4 are interpreted and the resulting non-printable characters (such as newlines, tabulators, etc.) will influence the printout.

Example:

```
log("foo");//The log printout will look like this:
 12:34:56.123456 foo
 bar
```

4.8.2. TTCN-3 Values and Templates

Literal values, referenced values or templates, wildcards, compound values, in-line (modified) templates, etc. (as long as the type of the expression is unambiguous) are discussed in this section.

These values are printed into the log using TTCN-3 Core Language syntax so that the printout can be simply copied into a TTCN-3 module to initialize an appropriate constant/variable/template, etc.

In case of (`universal`) `charstring` values the delimiter quotation marks ("") are printed and the embedded non-printable characters are substituted with the escape sequences in the first 9 rows of Table 4. All other non-printable characters are displayed in the TTCN-3 quadruple notation.

If the argument refers to a constant of type `charstring`, the actual value is not substituted to yield a literal string.

Example:

```
const charstring c_string := "foo\000";
log(c_string);
//The log printout will look like this:
12:34:56.123456 "foo" & char(0, 0, 0, 0)
```

4.8.3. Built-in Function match()

For the built-in `match()` function the printout will contain the detailed matching process field-by-field (similarly to the failed `receive` statements) instead of the Boolean result.

This rule is applied only if the` match()` operation is the top-level expression to be logged, see the

example below:

```
 // this will print the detailed matching process
log(match(v_myvalue, t_template));
 // this will print only a Boolean value (true or false)
log(not not match(v_myvalue, t_template));
```

All the other predefined and user-defined functions with actual arguments will print the return value of the function into the log according to the TTCN-3 standard.

4.8.4. Special TTCN-3 Objects

If the argument refers to a TTCN-3 `port`, `timer` or array (slice) of the above, then the actual properties of the TTCN-3 object is printed into the log.

For ports the name and the state of the port is printed.

In case of timers the name of the timer, the default duration, the current state (`inactive`, `started` or `expired`), the actual duration and the elapsed time (if applicable) is printed in a structured form.

4.9. Value Returning done

The compiler allows starting TTCN-3 functions having return type on PTCs. Those functions must have the appropriate `runs on` clause. If such a function terminates normally on the PTC, the returned value can be matched and retrieved in a `done` operation.

According to the TTCN-3 standard, the value redirect in a `done` operation can only be used to store the local verdict on the PTC that executed the behavior function. In TITAN the value redirect can also be used to store the behavior function's return value with the help of an optional template argument.

If this template argument is present, then the compiler treats it as a value returning done operation, otherwise it is treated as a verdict returning `done`.

The following rules apply to the optional template argument and the value redirect:

- The syntax of the template and value redirect is identical with that of the `receive` operation.
- If the template is present, then the type of the template and the variable used in the value redirect shall be identical. If the template is not present, then the type of the value redirect must be `verdicttype`.
- In case of a value returning done the return type shall be a TTCN-3 type marked with the following attribute: `with { extension "done" }`. It is allowed to mark and use several types in done statements within one test suite. If the type to be used is defined in ASN.1 then a type alias shall be added to one of the TTCN-3 modules with the above attribute.
- In case of a value returning done the type of the template or variable must be visible from the module where the `done` statement is used.
- Only those done statements can have a template or a value redirect that refer to a specific PTC

component reference. That is, it is not allowed to use this construct with `any component.done` or `all component.done`.

A value returning `done` statement is successful if all the conditions below are fulfilled:

- The corresponding PTC has terminated.
- The function that was started on the PTC has terminated normally. That is, the PTC was stopped neither by itself nor by other component and no dynamic test case error occurred.
- The return type of the function that was started on the PTC is identical to the type of the template used in the `done` statement.
- The value returned by the function on the PTC matches the given template.

If the `done` operation was successful and the value redirect is present the value returned by the PTC (if there was a matching template), or the local verdict on the PTC (if there was no matching template) is stored in the given variable or variable field.

The returned value can be retrieved from `alive` PTCs, too. In this case the `done` operation always refers to the return value of the lastly started behavior function of the PTC. Starting a new function on the PTC discards the return value of the previous function automatically (i.e. it cannot be retrieved or matched after the start component operation anymore).

Example:

```
type integer MyReturnType with { extension "done" };

function ptcBehavior() runs on MyCompType return MyReturnType
{
  setverdict(inconc);
  return 123;
}

// value returning 'done'
testcase myTestCase() runs on AnotherCompType
{
  var MyReturnType myVar;
  var MyCompType ptc := MyCompType.create;
  ptc.start(ptcBehavior());
  ptc.done(MyReturnType : ?) -> value myVar;
  // myVar will contain 123
}

// verdict returning 'done'
testcase myTestCase2() runs on AnotherCompType
{
  var verdicttype myVar;
  var MyCompType ptc := MyCompType.create;
  ptc.start(ptcBehavior());
  ptc.done -> value myVar;
  // myVar will contain inconc
}
```

4.10. Dynamic Templates

Dynamic templates (template variables, functions returning templates and passing template variables by reference) are now parts of the TTCN-3 Core Language standard ([1]). These constructs have been added to the standard with the same syntax and semantics as they were supported in this Test Executor. Thus dynamic templates are not considered language extensions anymore.

However, there is one extension compared to the supported version of Core Language. Unlike the standard, the compiler and the run-time environment allow the external functions to return templates.

Example:

```
// this is not valid according to the standard
external function MyExtFunction() return template octetstring;
```

4.11. Template Module Parameters

The compiler accepts template module parameters by inserting an optional "template" keyword

into the standard modulepar syntax construct between the modulepar keyword and the type reference. The extended BNF rule:

```
ModuleParDef ::= "modulepar" (ModulePar | ("{"MultiTypedModuleParList "}"))ModulePar
::= ["template"] Type ModuleParList
```

Example:

```
modulepar template charstring mp_tstr1 := ( "a" .. "f") ifpresent
modulepar template integer mp_tint := complement (1,2,3)
```

4.12. Predefined Functions

The built-in predefined functions `ispresent`, `ischosen`, `lengthof` and `sizeof` are applicable not only to value-like language elements (constants, variables, etc.), but template-like entities (templates, template variables, template parameters) as well. If the function is allowed to be called on a value of a given type it is also allowed to be called on a template of that type with the meaning described in the following subchapters.

NOTE	"dynamic test case error" does not necessarily denote here an error situation: it may well be a regular outcome of the function.

4.12.1. `sizeof`

The function `sizeof` is applicable to templates of `record`, `set`, `record` of, `set of` and `objid` types. The function is applicable only if the `sizeof` function gives the same result on all values that match the template.[12] In case of `record of` and `set of` types the length restrictions are also considered. Dynamic test case error occurs if the template can match values with different sizes or the length restriction contradicts the number of elements in the template body.

Examples:

```
type record of integer R;
type set S { integer f1, bitstring f2 optional, charstring f3 optional }
template R tr_1 := { 1, permutation(2, 3), ? }
template R tr_2 := {1, *, (2, 3) }
template R tr_3 := { 1, *, 10 } length(5)
template R tr_4 := { 1, 2, 3, * } length(1..2)
template S tr_5 := { f1 := (0..99), f2 := omit, f3 := ? }
template S tr_6 := { f3 := *, f1 := 1, f2 := '00'B ifpresent }
template S tr_7 := ({ f1 := 1, f2 := omit, f3 := "ABC" },
                    { f1 := 2, f3 := omit, f2 := '1'B })
template S tr_8 := ?

//sizeof(tr_1) → 4
//sizeof(tr_2) → error
//sizeof(tr_3) → 5
//sizeof(tr_4) → error
//sizeof(tr_5) → 2
//sizeof(tr_6) → error
//sizeof(tr_7) → 2
//sizeof(tr_8) → error
```

4.12.2. ispresent

The predefined function `ispresent` has been extended; its parameter can now be any valid TemplateInstance. It is working according to the following ETSI CRs: http://forge.etsi.org/mantis/view.php?id=5934 and http://forge.etsi.org/mantis/view.php?id=5936.

4.12.3. oct2unichar

The function `oct2unichar (in octetstring invalue, in charstring string_encoding := "UTF-8") return universal charstring` converts an octetstring `invalue` to a universal charstring by use of the given `string_encoding`. The octets are interpreted as mandated by the standardized mapping associated with the given `string_encoding` and the resulting characters are appended to the returned value. If the optional `string_encoding` parameter is omitted, the default value "UTF-8".

The following values are allowed as `string_encoding` actual parameters: `UTF8`, `UTF-16`, `UTF-16BE`, `UTF-16LE`, `UTF-32`, `UTF-32BE`, `UTF-32LE`.

DTE occurs if the `invalue` does not conform to UTF standards. The `oct2unichar` checks if the Byte Order Mark (BOM) is present. If not a warning will be appended to the log file. `oct2unichar` will `decode` the invalue even in absence of the BOM.

Any code unit greater than 0x10FFFF is ill-formed.

UTF-32 code units in the range of 0x0000D800 - 0x0000DFFF are ill-formed.

UTF-16 code units in the range of 0xD800 - 0xDFFF are ill-formed.

UTF-8 code units in the range of 0xD800 - 0xDFFF are ill-formed.

Example:

```
oct2unichar('C384C396C39CC3A4C3B6C3BC'O)="ÄÖÜäöü";oct2unichar('00C400D600DC00E400F600F
C'O,"UTF-16LE") = "ÄÖÜäöü";
```

4.12.4. unichar2oct

The function `unichar2oct (in universal charstring invalue, in charstring string_encoding := "UTF-8") return octetstring` converts a universal charstring `invalue` to an octetstring. Each octet of the octetstring will contain the octets mandated by mapping the characters of `invalue` using the standardized mapping associated with the given `string_encoding` in the same order as the characters appear in inpar. If the optional `string_encoding` parameter is omitted, the default encoding is "UTF-8".

The following values are allowed as `string_encoding` actual parameters: UTF-8, UTF-8 BOM, UTF-16, UTF-16BE, UTF-16LE, UTF-32, UTF-32BE, UTF-32LE.

The function `unichar2oct` adds the Byte Order Mark (BOM) to the beginning of the `octetstring` in case of `UTF-16` and `UTF-32` encodings. The `remove_bom` function helps to remove it, if it is not needed. The presence of the BOM is expected at the inverse function `oct2unichar` because the coding type (without the BOM) can be detected only in case of `UTF-8` encoded `octetstring`. By default UTF-8 encoding does not add the BOM to the `octetstring`, however `UTF-8 BOM` encoding can be used to add it.

DTE occurs if the `invalue` does not conform to UTF standards.

Any code unit greater than 0x10FFFF is ill-formed.

Example:

```
unichar2oct("ÄÖÜäöü") = 'EFBBBFC384C396C39CC3A4C3B6C3BC'O;
unichar2oct("ÄÖÜäöü","UTF-16LE") = 'FFFE00C400D600DC00E400F600FC'O;
```

4.12.5. get_stringencoding

The function `get_stringencoding (in octetstring encoded_value) return charstring` identifies the encoding of the `encoded_value`. The following return values are allowed as charstring: ASCII, UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE.

If the type of encoding could not been identified, it returns the value: <unknown>

Example:

```
var octetstring invalue := 'EFBBBFC384C396C39CC3A4C3B6C3BC'O;
var charstring codingtype := get_stringencoding(invalue);
the resulting codingtype is "UTF-8"
```

4.12.6. remove_bom

The function `remove_bom (in octetstring encoded_value) return octetstring` strips the BOM if it is present and returns the original octetstring otherwise.

Example:

```
var octetstring invalue := 'EFBBBFC384C396C39CC3A4C3B6C3BC'O;
var octetstring nobom := remove_bom(invalue);
the resulting nobom contains: 'C384C396C39CC3A4C3B6C3BC'O;
```

4.13. Additional Predefined Functions

In addition to standardized TTCN-3 predefined functions given in Annex C of [1] and Annex B of [3] the following built-in conversion functions are supported by our compiler and run-time environment:

4.13.1. str2bit

The function `str2bit (charstring value) return bitstring` converts a `charstring` value to a `bitstring`, where each character represents the value of one bit in the resulting bitstring. Its argument may contain the characters "0" or "1" only, otherwise the result is a dynamic test case error.

NOTE This function is the reverse of the standardized `bit2str`.

Example:

```
str2bit ("1011011100") = '1011011100'B
```

4.13.2. str2hex

The function `str2hex (charstring value) return hexstring` converts a `charstring` value to a `hexstring`, where each character in the character string represents the value of one hexadecimal digit in the resulting `hexstring`. The incoming character string may contain any number of characters. A dynamic test case error occurs if one or more characters of the charstring are outside the ranges "0" .. "9", "A" .. "F" and "a" .. "f".

NOTE This function is the reverse of the standardized `hex2str`.

Example:

```
str2hex ("1D7") = '1D7'H
```

4.13.3. float2str

The function `float2str (float value) return charstring` converts a `float` value to a `charstring`. If the input is zero or its absolute value is between 10^{-4} and 10^{10}, the decimal dot notation is used in the output with 6 digits in the fraction part. Otherwise the exponential notation is used with automatic (at most 6) digits precision in the mantissa.

Example:

```
float2str (3.14) = "3.140000"
```

4.13.4. unichar2char

The function `unichar2char (universal charstring value) return charstring` converts a` universal charstring` value to a `charstring`. The elements of the input string are converted one by one. The function only converts universal characters when the conversion result lies between 0 end 127 (that is, the result is an ISO 646 character).

NOTE	The inverse conversion is implicit, that is, the `charstring` values are converted to `universal charstring` values automatically, without the need for a conversion function.

Example:

```
unichar2char(char(0,0,0,64)) = "@"
```

4.13.5. log2str

The function `log2str` can be used to log into `charstring` instead of the log file.

Syntax:

```
log2str (⋯) return charstring
```

This function can be parameterized in the same way as the `log` function, it returns a charstring value which contains the log string for all the provided parameters, but it does not contain the timestamp, severity and call stack information, thus the output does not depend on the runtime configuration file. The parameters are interpreted the same way as they are in the log function: their string values are identical to what the log statement writes to the log file. The extra information (timestamp, severity, call stack) not included in the output can be obtained by writing external functions which use the runtime's Logger class to obtain the required data.

4.13.6. testcasename

The function `testcasename` returns the unqualified name of the actually executing test case. When it is called from the control part and no test case is being executed, it returns the empty string.

Syntax:

```
testcasename () return charstring
```

4.13.7. isbound

The function `isbound` behaves identically to the `isvalue` function with the following exception: it returns true for a record-of value which contains both initialized and uninitialized elements.

```
type record of integer rint;
var rint r_u; // uninitialized
isvalue(r_u); // returns false
isbound(r_u); // returns false also
//lengthof(r_u) would cause a dynamic testcase error

var rint r_0 := {} // zero length
isvalue(r_3); // returns true
isbound(r_3); // returns true
lengthof(r_3); // returns 0

var rint r_3 := { 0, -, 2 } // has a "hole"
isvalue(r_3); // returns false
isbound(r_3); // returns true
lengthof(r_3); // returns 3

var rint r_3full := { 0, 1, 2 }
isvalue(r_3full); // returns true
isbound(r_3full); // returns true
lengthof(r_3full); // returns 3
```

The introduction of `isbound` permits TTCN-3 code to distinguish between r_u and r_3; `isvalue` alone cannot do this (it returns false for both).

Syntax:

```
isbound (in template any_type i) return boolean;
```

4.13.8. ttcn2string

Syntax:

```
ttcn2string(in <TemplateInstance> ti) return charstring
```

This predefined function returns its parameter's value in a string which is in TTCN-3 syntax. The returned string has legal ttcn-3 with a few exceptions such as unbound values. Unbound values are

returned as "-", which can be used only as fields of assignment or value list notations, but not as top level assignments (e.g. `x:=-` is illegal). Differences between the output format of `ttcn2string()` and `log2str()`:

Value/template	`log2str()`	`ttcn2string()`
Unbound value	`"<unbound>"`	"-"
Uninitialized template	`"<uninitialized template>"`	"-"
Enumerated value	`name (number)`	name

4.13.9. `string2ttcn`

Syntax:

```
string2ttcn(in charstring ttcn_str, inout <reference> ref)
```

This predefined function does not have a return value, thus it is a statement. Any error in the input string will cause an exception that can be caught using @try - @catch blocks. The message string of the exception contains the exact cause of the error. There might be syntax and semantic errors. This function uses the module parameter parser of the TITAN runtime, it accepts the same syntax as the module parameters of the configuration file. Check the documentation chapters for the module parameters section. There are differences between the ttcn-3 syntax and the configuration file module parameters syntax, these are described in the documentation chapter of the module parameters. The second parameter must be a reference to a value or template variable.

Example code:

```
type record MyRecord { integer a, boolean b }
...
var template MyRecord my_rec
@try {
  string2ttcn("complement ({1,?},{(1,2,3),false}) ifpresent", my_rec)
  log(my_rec)
  }
  @catch (err_str) {
    log("string2ttcn() failed: ", err_str)
  }

The log output will look like this:
complement ({ a := 1, b := ? }, { a := (1, 2, 3), b := false }) ifpresent
```

4.13.10. `encode_base64`

Syntax:

```
encode_base64(in octetstring ostr, in boolean
  use_linebreaks := false) return charstring
```

The function `encode_base64 (in octetstring ostr, in boolean use_linebreaks := false) return charstring` `converts an octetstring` `ostr` to a charstring. The charstring will contain the Base64 representation of `ostr`. The `use_linebreaks` parameter adds newlines after every 76 output characters, according to the MIME specs, if it is omitted, the default value is false.

Example:

```
encode_base64('42617365363420656E636F64696E6720736368656D65'O) ==
"QmFzZTY0IGVuY29kaW5nIHNjaGVtZQ=="
```

4.13.11. decode_base64

Syntax:

```
decode_base64(in charstring str) return octetstring
```

The function `decode_base64 (in charstring str) return octetstring` converts a charstring `str` encoded in Base64 to an octetstring. The octetstring will contain the decoded Base64 string of `str`.

Example:

```
decode_base64("QmFzZTY0IGVuY29kaW5nIHNjaGVtZQ==") ==
'42617365363420656E636F64696E6720736368656D65'O
```

4.13.12. json2cbor

Syntax:

```
json2cbor(in universal charstring us) return octetstring
```

The function `json2cbor(in universal charstring us) return octetstring` converts a TITAN encoded json document into the binary representation of that json document using a binary coding called CBOR. The encoding follows the recommendations written in the CBOR standard [22] section 4.2.

Example:

```
json2cbor("{"a":1,"b":2}") == 'A2616101616202'O
```

4.13.13. cbor2json

Syntax:

```
cbor2json(in octetstring os) return universal charstring
```

The function `cbor2json(in octetstring os) return universal charstring` converts a CBOR encoded bytestream into a json document which can be decoded using the built in JSON decoder. The decoding follows the recommendations written in the CBOR standard [22] section 4.1 except that the indefinite-length items are not made definite before conversion and the decoding of indefinite-length items is not supported.

Example:

```
cbor2json('A2616101616202'O) == "{"a":1,"b":2}"
```

4.13.14. json2bson

Syntax:

```
json2bson(in universal charstring us) return octetstring
```

The function `json2bson(in universal charstring us) return octetstring` converts a TITAN encoded json document into the binary representation of that json document using a binary coding called BSON. Only top level json objects and arrays can be encoded. (Note that an encoded top level json array will be decoded as a json object) The encoding follows the rules written in the BSON standard [23]. The encoding handles the extension rules written in the MongoDB Extended JSON document [24]. The encoding of 128-bit float values is not supported.

Example:

```
json2bson("{"a":1,"b":2}") == '13000000106100010000000106200020000000 0'O
```

4.13.15. bson2json

Syntax:

```
bson2json(in octetstring os) return universal charstring
```

The function `bson2json(in octetstring os) return universal charstring` converts a BSON encoded bytestream into a json document which can be decoded using the built in JSON decoder. The decoding follows the extension rules written in the BSON standard [23]. The decoding handles the rules written in the MongoDB Extended JSON document [24]. The decoding of 128-bit float values is not supported.

Example:

```
bson2json('13000000106100010000001062000200000000'O) == "{"a":1,"b":2}"
```

4.14. Exclusive Boundaries in Range Subtypes

The boundary values used to specify range subtypes can be preceded by an exclamation mark. By using the exclamation mark the boundary value itself can be excluded from the specified range. For example integer range (!0..!10) is equivalent to range (1..9). In case of float type open intervals can be specified by using excluded boundaries, for example (0.0..!1.0) is an interval which contains 0.0 but does not contain 1.0.

4.15. Special Float Values Infinity and not_a_number

The keyword infinity (which is also used to specify value range and size limits) can be used to specify the special float values -infinity and +infinity, these are equivalent to MINUS-INFINITY and PLUS-INFINITY used in ASN.1. A new keyword not_a_number has been introduced which is equivalent to NOT-A-NUMBER used in ASN.1. The -infinity and +infinity and not_a_number special values can be used in arithmetic operations. If an arithmetic operation's operand is not_a_number then the result of the operation will also be not_a_number. The special value not_a_number cannot be used in a float range subtype because it's an unordered value, but can be added as a single value, for example subtype (0.0 .. infinity, not_a_number) contains all positive float values and the not_a_number value.

4.16. TTCN-3 Preprocessing

Preprocessing of the TTCN-3 files with a C preprocessor is supported by the compiler. External preprocessing is used: the Makefile Generator generates a `Makefile` which will invoke the C preprocessor to preprocess the TTCN-3 files with the suffix `."ttcnpp`. The output of the C preprocessor will be generated to an intermediate file with the suffix `."ttcn`. The intermediate files contain the TTCN-3 source code and line markers. The compiler can process these line markers along with TTCN-3. If the preprocessing is done with the `-P` option [13], the resulting code will not contain line markers; it will be compatible with any standard TTCN-3 compiler. The compiler will use the line markers to give almost [14] correct error or warning messages, which will point to the original locations in the `.ttcnpp` file. The C preprocessor directive `#"include` can be used in .ttcnpp files; the Makefile Generator will treat all files with suffix `."ttcnin` as TTCN-3 include files. The `."ttcnin` files will be added to the Makefile as special TTCN-3 include files which will not be translated by the compiler, but will be checked for modification when building the test suite.

Extract from the file:

```
Example.ttcnpp:
module Example {
function func()
{
#ifdef DEBUG
log("Example: DEBUG");
#else
log("Example: RELEASE");
#endif

}

...
```

The output is a preprocessed intermediate file `Example.ttcn`. The resulting output from the above code:

```
...
# 1 "Example.ttcnpp"
module Example {
function func()
{
log("Example: RELEASE");
}
```

The line marker (`# 1 "Example.ttcnpp"`) tells the compiler what the origin of the succeeding code is.

4.17. Parameter List Extensions

In addition to standardized TTCN-3 parameter handling described in 5.4.2 of [1] TITAN also supports the mixing of list notation and assignment notation in an actual parameter list.

4.17.1. Missing Named and Unnamed Actual Parameters

To facilitate handling of long actual parameter lists in the TITAN implementation, the actual parameter list consists of two optional parts: an unnamed part followed by a named part, in this order. In the actual parameter list a value must be assigned to every mandatory formal parameter either in the named part or in the unnamed part. (Mandatory parameter is one without default value assigned in the formal parameter list.) Consequently, the unnamed part, the named part or both may be omitted from the actual parameter list. Omitting the named part from the actual parameter lists provides backward compatibility with the standard notation.

The named and unnamed parts are separated by a comma as are the elements within both lists. It is not allowed to assign value to a given formal parameter in both the named and the unnamed part of the actual parameter list.

There can be at most one unnamed part, followed by at most one named part. Consequently, an

unnamed actual parameter may not follow a named parameter.

Named actual parameters must follow the same relative order as the formal parameters. It is not allowed to specify named actual parameters in an arbitrary order.

Examples

The resulting parameter values are indicated in brackets in the comments:

```
function myFunction(integer p_par1, boolean p_par2 := true) { … }
control {
*// the actual parameter list is syntactically correct below:*
myFunction(1, p_par2 := false); // (1, false)
myFunction(2); // (2, true)
myFunction(p_par1 := 3, p_par2 := false); // (3, false)
*// the actual parameter list is syntactically erroneous below:*
myFunction(0, true, -); // too many parameters
myFunction(1, p_par1 := 1); // p_par1 is given twice
myFunction(); // no value is assigned to mandatory p_par1
myFunction(p_par2 := false, p_par1 := 3); // out of order
myFunction(p_par2 := false, 1); // unnamed part cannot follow
// named part
}
```

4.18. `function`, `altstep` and `testcase` References

Although TITAN supports the behaviour type package ([5]) of the TTCN-3 standard, but this feature was included in the standard with a different syntax.

It is allowed to create TTCN-3 types of `functions`, `altsteps` and `testcases`. Values, for example variables, of such types can carry references to the respective TTCN-3 definitions. To facilitate reference using, three new operations (`refers`, `derefers` and `apply`) were introduced. This new language feature allows to create generic algorithms in TTCN-3 with late binding, (i.e. code in which the function to be executed is specified only at runtime).

4.19. Function Types with a RunsOn_self Clause

A function type or an altstep type, defined with a standard `runs on` clause, can use all constants, variables, timers and ports given in the component type definition referenced by the `runs on` clause (see chapter 16 of [1]).

A function type or an altstep type, defined with the TITAN-introduced `runs on self` clause, similarly, makes use of the resources of a component type; however, the component type in question is not given in advance. When an altstep or a function is called via a function variable, that is, a reference, using the `apply` operation, it can use the resources defined by the component type indicated in the `runs on` clause of the actually referenced function or altstep.

The "runs on self" construct is permitted only for `function` and `altstep` types. Any actual function or

altstep must refer to a given component type name in their `runs on` clause.

A variable with type of function type is called a **function variable**. Such variables can contain references to functions or altsteps. At function variable assignment, component type compatibility checking is performed with respect to the component context of the assignment statement and the "runs on" clause of the assigned function or altstep. When the `apply()` operator is applied to a function variable, no compatibility checking is performed.

The rationale for this distinction is the following: due to type compatibility checking at the time of value assignment to the function variable, the TTCN-3 environment can be sure that any non-`null` value of the variable is a function reference that is component-type-compatible with that component that is actually executing the code using the `apply()` operator.

As a consequence of this, it is forbidden to use values of function variables as arguments to the TTCN-3 operators `start()` or `send()`.

Example of using the clause `runs on self` in a library

A component type may be defined as an extension of another component type (using the standard `extends` keyword mentioned in chapter 6.2.10.2 of [1]). The effect of this definition is that the extended component type will implicitly contain all constant, variable, port and timer definitions from the parent type as well. In the example below, the component type `User_CT` aggregates its own constant, variable, port and timer definitions (resources) with those defined in the component type `Library_CT` (see line a).

The library developer writes a set of library functions that have a `runs on Library_CT` clause (see line h). Such library functions may offer optional references to other functions that are supposed to be specified by the user of the library (see line e). We say in this case that the library function may call user-provided **callback functions** via function variables. These function variables must have a type specified; optionally with a runs on clause. If this `runs on` clause refers to an actual component type name, then this actual type name must be known at the time of writing the library.

Library functions that runs on `Library_CT` can run on other component types as well, provided that the actual component type is compatible with `Library_CT` (see chapter 6.3.3 of [1]). An obvious design goal for the library writer is to permit references to any callback function that has a component-type-compatible `runs on` clause. However, the cardinality of compatible component types is infinitely large; therefore, they **cannot** be explicitly referenced by the function type definitions of the library.

The "runs on self" concept provides a remedy for this contradiction and allows conceiving library components prepared to take up user-written "plug-ins".

In the code excerpt below, function `f_LibraryFunction` (which has the clause `runs on Library_CT`) uses the function reference variable `v_callBackRef_self` (defined in `Library_CT`).The function `f_MyCallbackFunction` (see line b) has a `runs on User_CT` clause. `User_CT` (see line a) extends `Library_CT`, therefore it is suitable for running library function with runs on `Library_CT` clause, for example.

When the assignment to the function variable `v_CallbackRef_self` is performed (see line c) inside `f_MyUserFunction` (that is, inside the context `User_CT`), then compatibility checking is performed.

Since `User_CT` is compatible with `Library_CT`, the assignment is allowed.

Direct call to `f_MyCallbackFunction()` with `runs on User_CT` from a `runs on Library_CT` context (see line g) would cause semantic error according to the TTCN3 language. However, calling the function via `v_CallBackRef_self` is allowed (see line d).

```
module RunsOn_Self
{
//=========================================================================
// Function Types
//=========================================================================

//---- line f)
type function CallbackFunctionRefRunsonSelf_FT () runs on self;

//=========================================================================
//Component Types
//=========================================================================
type component Library_CT
{
//---- line e)
  var CallbackFunctionRefRunsonSelf_FT v_CallbackRef_self := null;
  var integer v_Lib;
}
//---- line a)
type component User_CT extends Library_CT
{
  var integer v_User;
}

//---- line h)
function f_LibraryFunction () runs on Library_CT
{
//---- line g)
  // Direct call of the callback function would cause semantic ERROR
//f_MyCallbackFunction();

  if (v_CallbackRef_self != null)
  {
    // Calling a function via reference that has a "runs on self" in its header
    // is always allowed with the exception of functions/altsteps without runs
    // on clause
//---- line d)
    v_CallbackRef_self.apply();
  }
}// end f_LibraryFunction

function f_MyUserFunction () runs on User_CT
{
  // This is allowed as f_MyCallbackFunction has runs on clause compatible
```

```
  // with the runs on clause of this function (f_MyUserFunction)
  // The use of function/altstep references with "runs on self" in their
  // headers is limited to call them on the given component instance; i.e.
  // allowed: assignments, parameterization and activate (the actual function's
  //          runs on is compared to the runs on of the function in which
  //          the operation is executed)
  // not allowed: start, sending and receiving
  // no check is needed for apply!
//---- line c)
  v_CallbackRef_self := refers (f_MyCallbackFunction);

  // This is allowed as Library_CT is a parent of User_CT
  // Pls. note, as the function is executing on a User_CT
  // instance, it shall never cause a problem of calling
  // a callback function with "runs on User_CT" from it.
  f_LibraryFunction();

}//end f_MyUserFunction

//---- line b)
function f_MyCallbackFunction () runs on User_CT
{/*application/dependent behaviour*/}

} // end of module RunsOn_Self
```

4.20. TTCN-3 Macros

The compiler and the run-time environment support the following non-standard macro notation in TTCN-3 modules. All TTCN-3 macros consist of a percent (%) character followed by the macro identifier. Macro identifiers are case sensitive. The table below summarizes the available macros and their meaning. Macro identifiers not listed here are reserved for future extension.

Table 5. TTCN-3 macros

Macro	Meaning
`%moduleId`	name of the TTCN-3 module
`%definitionId`	name of the top-level TTCN-3 definition
`%testcaseId`	name of the test case that is currently being executed
`%fileName`	name of the TTCN-3 source file
`%lineNumber`	number of line in the source file

The following rules apply to macros:

- All macros are substituted with a value of type `charstring`. They can be used as operands of complex expressions (concatenation, comparison, etc.).
- All macros except `%testcaseId` are evaluated during compilation and they can be used anywhere

in the TTCN-3 module.

- Macro `%testcaseId` is evaluated at runtime. It can be used only within functions and altsteps that are being run on test components (on the MTC or PTCs) and within testcases. It is not allowed to use macro `%testcaseId` in the module control part. If a function or altstep that contains macro `%testcaseId` is called directly from the control part the evaluation of the macro results in a dynamic test case error.
- The result of macro `%testcaseId` is not a constant thus it cannot be used in the value of TTCN-3 constants. It is allowed only in those contexts where TTCN-3 variable references are permitted.
- Macro `%definitionId` is always substituted with the name of the top-level module definition that it is used in. [15] For instance, if the macro appears in a constant that is defined within a function then the macro will be substituted with the function's name rather than the one of the constant. When used within the control part macro `%definitionId` is substituted with the word "control".
- Macro `%fileName` is substituted with the name of the source file in the same form as it was passed to the compiler. This can be a simple file name, a relative or an absolute path name.
- The result of macro `%lineNumber` is always a string that contains the current line number as a decimal number. Numbering of lines starts from 1. All lines of the input file (including comments and empty lines) are counted. When it needs to be used in an integer expression a conversion is necessary: `str2int(%lineNumber)`. The above expression is evaluated during compilation without any runtime performance penalty.
- Source line markers are considered when evaluating macros `%fileName` and `%lineNumber`. In preprocessed TTCN-3 modules the macros are substituted with the original file name and line number that the macro comes from provided that the preprocessor supports it.
- When macros are used in `log()` statements, they are treated like literal strings rather than charstring value references. That is, quotation marks around the strings are not used and special characters within them are not escaped in the log file.
- For compatibility with the C preprocessor the compiler also recognizes the following C style macros: __FILE__ is identical to `%fileName` and __LINE__ is identical to `str2int(%lineNumber)`.
- Macros are not substituted within quotation marks (i.e. within string literals and attributes).
- The full power of TTCN-3 macros can be exploited in combination with the C preprocessor.

Example:

```
module M {
// the value of c_MyConst will be "M"
const charstring c_MyConst := %moduleId;
// MyTemplate will contain 28
template integer t_MyTemplateWithVeryLongName := lengthof(%definitionId);
function f_MyFunction() {
// the value of c_MyLocalConst1 will be "f_MyFunction"
const charstring c_MyLocalConst1 := %definitionId;
// the value of c_MyLocalConst2 will be "%definitionId"
const charstring c_MyLocalConst2 := "%definitionId";
// the value of c_MyLocalConst3 will be "12"
const charstring c_MyLocalConst3 := %lineNumber; //This is line 12
// the value of c_MyLocalConst4 will be 14
const integer c_MyLocalConst4 := str2int(%lineNumber);//This is line 14
// the line below is invalid because %testcaseId is not a constant
const charstring c_MyInvalidConst := %testcaseId;
// this is valid, of course
var charstring v_MyLocalVar := %testcaseId;
// the two log commands below give different output in the log file
log("function:", %definitionId, " testcase: ", %testcaseId);
// printout: function: f_MyFunction testcase: tc_MyTestcase
log("function:", c_MyLocalConst1, " testcase: ", v_MyLocalVar);
// printout: function: "f_MyFunction" testcase: "tc_MyTestcase"
}
}
```

4.21. Component Type Compatibility

The ETSI standard defines type compatibility of component types for component reference values and for functions with “runs on” clause. In order to be compatible, both component types are required to have identical definitions (cf. [1], chapter 6.3.3).

NOTE	Compatibility is an asymmetric relation, if component type B is compatible with component type A, the opposite is not necessarily true. (E.g., component type B may contain definitions absent in component type A.)

All definitions from the parent type are implicitly contained when the keyword `extends` appears in the type definition (cf. [1], chapter 6.2.10.2) and so the required identity of the type definitions is ensured. The compiler considers component type B to be compatible with A if B has an `extends` clause, which contains A or a component type that is compatible with A.

Example:

```
type component A { var integer i; }
type component B extends A {
// extra definitions may be added here
}
```

In order to provide support for existing TTCN-3 code (e.g. standardized test suites) it is allowed to explicitly signal the compatibility relation between component types using a special `extension` attribute. Using such attributes shall be avoided in newly written TTCN-3 modules. Combining component type inheritance and the attribute `extension` is possible, but not recommended.

Thus, the compiler considers component type B to be compatible with A if B has an `extension` attribute that points to A as base component type and all definitions of A are present and identical in B.

```
type component A { var integer i; }
type component B {
var integer i; // definitions of A must be repeated
var octetstring o; // new definitions may be added
} with {
extension "extends A"
}
```

4.21.1. Implementation Restrictions

The list of definitions shared with different compatible component types shall be distinct. If component type Z is compatible with both X and Y and neither X is compatible with Y nor Y is compatible with X then X and Y shall not have definitions with identical names but different origin. If both X and Y are compatible with component type C then all definitions in X and Y which are originated from C are inherited by Z on two paths.

Example: According to the standard component type Z should be compatible with both X and Y, but the compatibility relation cannot be established because X and Y have a definition with the same name.

```
type component X { timer T1, T2; }
type component Y { timer T1, T3; }
type component Z { timer T1, T2, T3; }
with { extension "extends X, Y" }
// invalid because the origin of T1 is ambiguous
```

The situation can be resolved by introducing common ancestor C for X and Y, which holds the shared definition.

```
type component C { timer T1; }
type component X { timer T1, T2; } with { extension "extends C" }
type component Y { timer T1, T3; } with { extension "extends C" }
type component Z {
timer T1, // origin is C
T2, // origin is X
T3; // origin is Y
} with { extension "extends X, Y" }
```

Circular compatibility chains between component types are not allowed. If two component types need to be defined as identical, type aliasing must be used instead of compatibility.

The following code is invalid:

```
type component A {
…
// the same definitions as in B
} with { extension "extends B" }
type component B {
…
// the same definitions as in A
} with { extension "extends A" }
```

When using the non-standard extension attribute the initial values of the corresponding definitions of compatible components should be identical. The compiler does not enforce this for all cases; however, in the case of different initial values the resulting run-time behavior is undefined. If the initial values cannot be determined at compile time (module parameters) the compiler will remain silent. In other situations the compiler may report an error or a warning.

All component types are compatible with each empty component type. Empty components are components which have neither own nor inherited definitions.

4.22. Implicit Message Encoding

The TTCN-3 standard [1] does not specify a standard way of data encoding/decoding. TITAN has a common C++ API for encoding/decoding; to use this API external functions are usually needed. The common solution is to define a TTCN-3 external function and write the C++ code containing the API calls. In most cases the C++ code explicitly written to an auxiliary C++ file contains only simple code patterns which call the encoding/decoding API functions on the specified data. In TITAN there is a TTCN-3 language extension which automatically generates such external functions.

Based on this automatic encoding/decoding mechanism, dual-faced ports are introduced. Dual-faced ports have an external and an internal interface and can automatically transform messages passed through them based on mapping rules defined in TTCN-3 source files. These dual-faced ports eliminate the need for simple port mapping components and thus simplify the test configuration.

4.22.1. Dual-faced Ports

In the TTCN-3 standard ([1]), a port type is defined by listing the allowed incoming and outgoing message types. Dual-faced ports have on the other hand two different message lists: one for the external and one for the internal interface. External and internal interfaces are given in two distinct port type definitions. The dual-faced concept is applicable to message based ports and the message based part of mixed ports.

Dual-faced port types must have `user` attribute to designate its external interface. The internal interface is given by the port type itself. A port type can be the external interface of several different dual-faced port types.

The internal interface is involved in communication operations (`send`, `receive`, etc.) and the external interface is used when transferring messages to/from other test components or the system under test. The operations `connect` and `map` applied on dual-faced ports will consider the external port type when checking the consistency of the connection or mapping.[16]

Dual-faced Ports between Test Components

Dual-faced ports used for internal communication must have the attributes `internal` in addition to `user` (see section Visibility Modifiers). The referenced port type describing the external interface may have any attributes.

Dual-faced Ports between Test Components and the SUT

The port type used as external interface must have the attribute `provider`. These dual-faced port types do not have their own test port; instead, they use the test port belonging to the external interface when communicating to SUT. Using the attribute `provider` implies changes in the Test Port API of the external interface. For details see the section "Provider port types" in [16].

If there are several entities within the SUT to be addressed, the dual-faced port type must have the attribute `address` in addition to `user`. In this case the external interface must have the attribute `address` too. For more details see section Visibility Modifiers.

4.22.2. Type Mapping

Mapping is required between the internal and external interfaces of the dual-faced ports because the two faces are specified in different port type definitions, thus, enabling different sets of messages.

Messages passing through dual-faced ports will be transformed based on the mapping rules. Mapping rules must be specified for the outgoing and incoming directions separately. These rules are defined in the attribute `user` of the dual-faced port type.

An outgoing mapping is applied when a `send` operation is performed on the dual-faced port. The outcome of the mapping will be transmitted to the destination test component or SUT. The outgoing mappings transform each outgoing message of the internal interface to the outgoing messages of the external interface.

An incoming mapping is applied when a message arrives on a dual-faced port from a test component or the SUT. The outcome of the mapping will be inserted into the port queue and it will be extracted by the `receive` operation. The incoming mappings transform each incoming messages of the external interface to the incoming message of the internal interface.

Mapping Rules

A mapping rule is an elementary transformation step applied on a message type (source type) resulting in another message type (target type). Source type and target type are not necessarily different.

Mapping rules are applied locally in both directions, thus, an error caused by a mapping rule affects the test component owning the dual-faced port, not its communication partner.

Mappings are given for each source type separately. Several mapping targets may belong to the same source type; if this is the case, all targets must be listed immediately after each other (without repeating the source type).

The following transformation rules may apply to the automatic conversion between the messages of the external and internal interfaces of a dual-faced port:

- No conversion. Applicable to any message type, this is a type preserving mapping, no value conversion is performed. Source and target types must be identical. This mapping does not have any options. For example, control or status indication massages may transparently be conveyed between the external and the internal interfaces. Keyword used in attribute `user` of port type definition: `simple`.
- Message discarding. This rule means that messages of the given source type will not be forwarded to the opposite interface. Thus, there is no destination type, which must be indicated by the not used symbol (-). This mapping does not have any options. For example, incoming status indication massages of the external interface may be omitted on the internal interface. Keyword used in attribute `user` of port type definition: `discard`.
- Conversion using the built-in codecs. Here, a corresponding encoding or decoding subroutine of the built-in codecs (for example RAW, TEXT or BER) is invoked. The conversion and error handling options are specified with the same syntax as used for the encoding/decoding functions, see section Attribute Syntax. Here, source type corresponds to input type and target type corresponds to output type of the encoding. Keyword used in attribute `user` of port type definition: `encode` or `decode`; either followed by an optional `errorbehavior`.
- Function or external function. The transformation rule may be described by an (external) function referenced by the mapping. The function must have the attribute `extension` specifying one of the prototypes given in section Encoder/decoder Function Prototypes. The incoming and the outgoing type of the function must be equal to the source and target type of the mapping, respectively. The function may be written in TTCN-3, C++ or generated automatically by the compiler. This mapping does not have any options. Keyword used in attribute `user` of port type definition: `function`.

Mapping with One Target

Generally speaking, a source type may have one or more targets. Every mapping target can be used alone. However, only one target can be designated with the following rules if

- no conversion takes place (keyword `simple`);
- encoding a structured message (keyword `encode`) [17];
- an (external) function with prototype `convert` or `fast` is invoked

Mapping with More Targets

On the other hand, more than one target is needed, when the type of an encoded message must be reconstructed. An octetstring, for example, can be decoded to a value of more than one structured PDU type. It is not necessary to specify mutually exclusive decoder rules. It is possible and useful to define a catch-all rule at the end to handle invalid messages.

The following rules may be used with more than one target if

- an (external) function with prototype `backtrack` is invoked;
- decoding a structured message (keyword `decode`);
- (as a last alternative) the source message is `discarded`

The conversion rules are tried in the same order as given in the attribute until one of them succeeds, that is, the function returns `0 OK` or decoding is completed without any error. The outcome of the successful conversion will be the mapped result of the source message. If all conversion rules fail and the last alternative is `discard`, then the source message is discarded. Otherwise dynamic test case error occurs.

Mapping from Sliding Buffer

Using sliding buffers is necessary for example, if a stream-based transport, like TCP, is carrying the messages. A stream-based transport is destroying message boundaries: a message may be torn apart or subsequent messages may stick together.

The following rules may be used with more than one target when there is a sliding buffer on the source side if

- an (external) function with prototype `sliding` is invoked;
- decoding a structured message (keyword `decode`)

Above rules imply that the source type of this mapping be either `octetstring` or `charstring`. The run-time environment maintains a separate buffer for each connection of the dual-faced port. Whenever data arrives to the buffer, the conversion rules are applied on the buffer in the same order as given in the attribute. If one of the rules succeeds (that is, the function returns `0` or decoding is completed without any error) the outcome of the conversion will appear on the destination side. If the buffer still contains data after successful decoding, the conversion is attempted again to get the next message. If one of the rules indicates that the data in the buffer is insufficient to get an entire message (the function returns `2 INCOMPLETE_MESSAGE` or decoding fails with error code `ET_INCOMPL_MSG`), then the decoding is interrupted until the next fragment arrives in the buffer. If all conversion rules fail (the function returns `1 NOT_MY_TYPE` or decoding fails with any other error code than `ET_INCOMPL_MSG`), dynamic test case error occurs.

NOTE	Decoding with sliding should be the last decoding option in the list of decoding options and there should be only one decoding with sliding buffer. In other cases the first decoding with sliding buffer might disable reaching later decoding options.

4.22.3. Encoder/decoder Function Prototypes

Encoder/decoder functions are used to convert between different data (message) structures. We can consider e.g. an octet string received from the remote system that should be passed to the upper layer as a TCP message.

Prototypes are attributes governing data input/output rules and conversion result indication. In other words, prototypes are setting the data interface types. The compiler will verify that the

parameters and return value correspond to the given prototype. Any TTCN-3 function (even external ones) may be defined with a prototype. There are four prototypes defined as follows:

- prototype `convert`

  Functions of this prototype have one parameter (i.e. the data to be converted), which shall be an `in` value parameter, and the result is obtained in the return value of the function.

  Example:

```
external function f_convert(in A param_ex) return B
with { extension "prototype(convert)" }
```

+ The input data received in the parameter `param_ex` of type A is converted. The result returned is of type B.

- prototype `fast`

  Functions of this prototype have one input parameter (the same as above) but the result is obtained in an `out` value parameter rather than in return value. Hence, a faster operation is possible as there is no need to copy the result if the target variable is passed to the function. The order of the parameters is fixed: the first one is always the input parameter and the last one is the output parameter.

  Example:

```
external function f_fast(in A param_1, out B param_2)
with { extension "prototype(fast)" }
```

+ The input data received in the parameter `param_1` of type A is converted. The resulting data of type B is contained in the output parameter `param_2` of type B.

- prototype `backtrack`

  Functions of this prototype have the same data input/output structure as of prototype `fast`, but there is an additional integer value returned to indicate success or failure of the conversion process. In case of conversion failure the contents of the output parameter is undefined. These functions can only be used for decoding. The following return values are defined to indicate the outcome of the decoding operation:

  - 0 (`OK`). Decoding was successful; the result is stored in the out parameter.
  - 1 (`NOT_MY_TYPE`). Decoding was unsuccessful because the input parameter does not contain a valid message of type `B`. The content of the out parameter is undefined.

  Example:

```
external function f_backtrack(in A param_1, out B param_2) return integer
with { extension "prototype(backtrack)" }
```

The input data received in the parameter `param_1` of type A is converted. The resulting data of type B is contained in the output parameter `param_2` of type B. The function return value (an integer) indicates success or failure of the conversion process.

- prototype `sliding`

  Functions of this prototype have the same behavior as the one of prototype backtrack, consequently, these functions can only be used for decoding. The difference is that there is no need for the input parameter to contain exactly one message: it may contain a fragment of a message or several concatenated messages stored in a FIFO buffer. The first parameter of the function is an `inout` value parameter, which is a reference to a buffer of type `octetstring` or `charstring`. The function attempts to recognize an entire message. It if succeeds, the message is removed from the beginning of the FIFO buffer, hence the name of this prototype: sliding (buffer). In case of failure the contents of the buffer remains unchanged. The return value indicates success or failure of the conversion process or insufficiency of input data as follows:

  - 0 (`OK`). Decoding was successful; the result is stored in the out parameter. The decoded message was removed from the beginning of the inout parameter which is used as a sliding buffer.
  - 1 (`NOT_MY_TYPE`). Decoding was unsuccessful because the input parameter does not contain or start with a valid message of type B. The buffer (`inout` parameter) remains unchanged. The content of out parameter is undefined.
  - 2 (`INCOMPLETE_MESSAGE`). Decoding was unsuccessful because the input stream does not contain a complete message (i.e. the end of the message is missing). The input buffer (inout parameter) remains unchanged. The content of out parameter is undefined.

  Example:

```
external function f_sliding(inout A param_1, out B param_2) return integer
with { extension "prototype(sliding)" }
```

+ The first portion of the input data received in the parameter `param_1` of type `A` is converted. The resulting data of type B is contained in the output parameter `param_2` of type `B`. The return value indicates the outcome of the conversion process.

4.22.4. Automatic Generation of Encoder/decoder Functions

Encoding and decoding is performed by C++ external functions using the built-in codecs. These functions can be generated automatically by the complier. The present section deals with attributes governing the function generation.

Input and Output Types

Automatically generated encoder/decoder functions must have an attribute `prototype` assigned. If the encoder/decoder function has been written manually, only the attribute `prototype` may be given. Automatically generated encoder/decoder functions must have either the attribute `encode` or the attribute `decode`. In the case of encoding, the input type of the function must be the (structured) type to be encoded, which in turn must have the appropriate encoding attributes needed for the specified encoding method. The output type of the encoding procedure must be `octetstring` (BER, RAW, XER and JSON coding) or `charstring` (TEXT coding). In case of decoding the functions work the other way around: the input type is `octetstring` or `charstring` and the output type can be any (structured) type with appropriate encoding attributes.

Attribute Syntax

The syntax of the `encode` and `decode` attributes is the following:

```
("encode"|"decode") "("("RAW"|"BER"|"TEXT"|"XER"|"JSON"|"OER"|"PER") [":"
<codec_options>] ")"
```

BER encoding can be applied only for ASN.1 types.

The <`codec_options`> part specifies extra options for the particular codec. Currently it is applicable only in case of BER, XML and PER encoding/decoding. In case of BER and XER the `codec_options` are copied transparently to the parameter list of the C++ encoder/decoder function call in the generated function body without checking the existence or correctness of the referenced symbols.

Example of prototype `convert`, BER encoding and decoding (the PDU is an ASN.1 type):

```
external function encode_PDU(in PDU pdu) return octetstring
with { extension "prototype(convert) encode(BER:BER_ENCODE_DER)" }
external function decode_PDU(in octetstring os) return PDU
with { extension "prototype(convert) decode(BER:BER_ACCEPT_ALL)" }
```

Example of prototype `convert`, XML encoding and decoding (the PDU is a TTCN-3 type):

```
external function encode_PDU(in PDU pdu) return octetstring
with { extension "prototype(convert) encode(XER:XER_EXTENDED)" }
external function decode_PDU(in octetstring os) return PDU
with { extension "prototype(convert) decode(XER:XER_EXTENDED)" }
```

In case of PER the `codec_options` are analyzed during compilation. If they contain the text `ALIGNED` (except when that text is preceded by `UN`), then the encoder or decoder function will use the aligned version of PER, otherwise it will use the unaligned version. If the `codec_options` contain the text `CANONICAL`, then the encoder or decoder will use the canonical version of PER, otherwise it will use the basic version. All other text in the `codec_options` is ignored.

Example of prototype `convert`, PER encoding and decoding (the PDU is an ASN.1 type):

```
external function encode_PDU(in PDU pdu) return octetstring
with { extension "prototype(convert) encode(PER)" } // unaligned, basic
external function encode_PDU_a(in PDU pdu) return octetstring
with { extension "prototype(convert) encode(PER:ALIGNED)" } // aligned, basic
external function encode_PDU_c(in PDU pdu) return octetstring
with { extension "prototype(convert) encode(PER:CANONICAL)" } // unaligned, canonical
external function encode_PDU_a_c(in PDU pdu) return octetstring
with { extension "prototype(convert) encode(PER:ALIGNED CANONICAL)" } // aligned,
canonical

external function decode_PDU(in octetstring os) return PDU
with { extension "prototype(convert) decode(PER)" } // unaligned, basic
external function decode_PDU_a(in octetstring os) return PDU
with { extension "prototype(convert) decode(PER:ALIGNED)" } // aligned, basic
external function decode_PDU_c(in octetstring os) return PDU
with { extension "prototype(convert) decode(PER:CANONICAL)" } // unaligned, canonical
external function decode_PDU_a_c(in octetstring os) return PDU
with { extension "prototype(convert) decode(PER:ALIGNED CANONICAL)" } // aligned,
canonical
```

Codec Error Handling

The TITAN codec API has some well defined function calls that control the behavior of the codecs in various error situations during encoding and decoding. An error handling method is set for each possible error type. The default error handling method can be overridden by specifying the `errorbehavior` attribute:

```
"errorbehavior" "(" <error_type> ":" <error_handling>
{ "," <error_type> ":" <error_handling> } ")"
```

Possible error types and error handlings are defined in [16], section "The common API". The value of `<error_type>` shall be a value of type `error_type_t` without the prefix `ET_`. The value of `<error_handling>` shall be a value of type `error_behavior_t` without the prefix `EB_`.

The TTCN-3 attribute `errorbehavior(INCOMPL_ANY:ERROR)`, for example, will be mapped to the following C++ statement:

```
TTCN_EncDec::set_error_behavior(TTCN_EncDec::ET_INCOMPL_ANY,
  TTCN_EncDec::EB_ERROR);
```

When using the `backtrack` or `sliding` decoding functions, the default error behavior has to be changed in order to avoid a runtime error if the `in` or `inout` parameter does not contain a type we could decode. With this change an integer value is returned carrying the fault code. Without this change a dynamic test case error is generated. Example:

```
external function decode_PDU(in octetstring os, out PDU pdu) return integer
with {
extension "prototype(backtrack)"
extension "decode(BER:BER_ACCEPT_LONG|BER_ACCEPT_INDEFINITE)"
extension "errorbehavior(ALL:WARNING)"
}
```

4.22.5. Handling of encode and variant attributes

The TITAN compiler offers two different ways of handling encoding-related attributes:

- the new (standard compliant) handling method, and
- the legacy handling method, for backward compatibility.

New codec handling

This method of handling `encode` and `variant` attributes is active by default. It supports many of the newer encoding-related features added to the TTCN-3 standard.

Differences from the legacy method:

- `encode` and `variant` attributes can be defined for types as described in the TTCN-3 standard (although the type restrictions for built-in codecs still apply);
- a type can have multiple `encode` attributes (this provides the option to choose from multiple codecs, even user-defined ones, when encoding values of that type);
- ASN.1 types automatically have `BER`, `JSON`, `PER`, and XML (if the compiler option `-a` is set) encoding (they are treated as if they had the corresponding `encode` attributes);
- encoding-specific `variant` attributes are supported (e.g.: `variant "XML"."untagged"`);
- the parameters `encoding_info/decoding_info` and `dynamic_encoding` of predefined functions `encvalue`, `decvalue`, `encvalue_unichar` and `decvalue_unichar` are supported (the `encoding_info` /`decoding_info` parameters are currently ignored, see also the next point)
- the `dynamic_encoding` parameter can be used for choosing the codec to use for values of types with multiple encodings, the following values are available: `BER:2002`, `CER:2002`, `DER:2002`, `JSON`, `OER`, `RAW`, `TEXT`, `XER`, `XML`, `PER` (for unaligned basic PER), `PER:ALIGNED` (for aligned basic PER), `PER:CANONICAL` (for unaligned canonical PER) and `PER:CANONICAL ALIGNED`;
- the `self.setencode` version of the `setencode` operation is supported (it can be used for choosing the codec to use for types with multiple encodings within the scope of the current component);
- the `@local` modifier is supported for `encode` attributes;
- a type's the default codec (used by `decmatch` templates, the @decoded modifier, and the predefined functions `encvalue`, `decvalue`, `encvalue_unichar` and `decvalue_unichar` when no dynamic encoding parameter is given) is:
- its one defined codec, if it has exactly one codec defined; or
- unspecified, if it has multiple codecs defined (the mentioned methods of encoding/decoding can only be used in this case, if a codec was selected for the type using `self.setencode`).

Differences from the TTCN-3 standard:

- switching codecs during the encoding or decoding of a structure is currently not supported (the entire structure will be encoded or decoded using the codec used at top level);
- the port-specific versions of the `setencode` operation are not supported (since messages sent through ports are not automatically encoded; see also dual-faced ports in section Dual-faced Ports);
- the `@local` modifier only affects encode attributes, it does not affect the other attribute types;
- `encode` and `variant` attributes do not affect `constants`, `templates`, `variables`, `template variables` or `import` statements (these are accepted, but ignored by the compiler);
- references to multiple definitions in attribute qualifiers is not supported(e.g.: `encode (template all except (t1))` "RAW");
- retrieving attribute values is not supported (e.g.: `var universal charstring x := MyType.encode`).

Legacy codec handling

This is the method of handling encode and variant attributes that was used before version 6.3.0 (/6 R3A). It can be activated through the compiler command line option `-e`.

Differences from the new method:

- each codec has its own rules for defining `encode` and `variant` attributes;
- a type can only have one `encode` attribute (if more than one is defined, then only the last one is considered), however, it can have `variant` attributes that belong to other codecs (this can make determining the default codec tricky);
- ASN.1 types automatically have `BER`, `JSON`, `PER`, and `XML` (if the compiler option -a is set) encoding, however the method of setting a default codec (for the predefined functions `encvalue`, `decvalue`, `encvalue_unichar`, `decvalue_unichar`, for `decmatch` templates, and for the `@decoded` modifier) is different (see section Setting the default codec for ASN.1 types);
- encoding-specific `variant` attributes are not supported (e.g.: `variant "XML"."untagged"`);
- the parameters `encoding_info/decoding_info` and `dynamic_encoding` of predefined functions `encvalue`, `decvalue`, `encvalue_unichar` and `decvalue_unichar` are ignored;
- the `setencode` operation is not supported;
- the `@local` modifier is not supported.
- the TTCN-3 language elements that automatically encode or decode (i.e. predefined functions `encvalue`, `decvalue`, `encvalue_unichar` and `decvalue_unichar`, `decmatch` templates, and value and parameter redirects with the `@decoded` modifier) ignore the `encode` and `variant` attributes in reference types and encode/decode values as if they were values of the base type (only the base type's `encode` and `variant` attributes are in effect in these cases). Encoder and decoder external functions take all of the type's attributes into account. For example:

```
type record BaseType {
  integer field1,
  charstring field2
}
with {
  encode "XML";
  variant "name as uncapitalized";
}

type BaseType ReferenceType
with {
  encode "XML";
  variant "name as uncapitalized";
}

external function f_enc(in ReferenceType x) return octetstring
  with { extension "prototype(convert) encode(XER:XER_EXTENDED)" }

function f() {
  var ReferenceType val := { field1 := 3, field2 := "abc" };

  var charstring res1 := oct2char(bit2oct(encvalue(val)));
  // "<baseType>\n\t<field>3</field>\n</baseType>\n\n"
  // it's encoded as if it were a value of type 'BaseType',
  // the name and attributes of type 'ReferenceType' are ignored

  var charstring res2 := oct2char(f_enc(val));
  // "<referenceType>\n\t<field>3</field>\n</referenceType>\n\n"
  // it's encoded correctly, as a value of type 'ReferenceType'
}
```

The differences from the TTCN-3 standard listed in the previous section also apply to the legacy method.

Setting the default codec for ASN.1 types

Since ASN.1 types cannot have `encode` or `variant` attributes, the compiler determines their encoding type by checking external encoder or decoder functions (of built-in encoding types) declared for the type.

The TITAN runtime does not directly call these external functions, they simply indicate which encoding type to use when encoding or decoding the ASN.1 type in question through predefined functions `encvalue` and `decvalue`, decoded content matching (`decmatch` templates) and in value and parameter redirects with the `@decoded` modifier.

These external functions can be declared with any prototype, and with the regular stream type of either `octetstring` or `charstring` (even though `encvalue` and `decvalue` have `bitstring` streams).

The ASN.1 type cannot have several external encoder or decoded functions of different (built-in or legacy PER) encoding types, as this way the compiler won't know which encoding to use. Multiple

encoder or decoder functions of the same encoding type can be declared for one type.

NOTE	These requirements are only checked if there is at least one `encvalue`, `decvalue`, `decmatch` template or decoded parameter or value redirect in the compiled modules. They are also checked separately for encoding and decoding (meaning that, for example, multiple encoder functions do not cause an error if only `decvalues` are used in the modules and no `encvalues`). The compiler searches all modules when attempting to find the coder functions needed for a type (including those that are not imported to the module where the encvalue, decvalue, decmatch or @decoded is located).

Example:

```
external function f_enc_seq(in MyAsnSequenceType x) return octetstring
with { extension "prototype(convert) encode(JSON)" }

external function f_dec_seq(in octetstring x, out MyAsnSequenceType y)
with { extension "prototype(fast) decode(JSON)" }

...

var MyAsnSequenceType v_seq := { num := 10, str := "abc" };
var bitstring v_enc := encvalue(v_seq); // uses the JSON encoder

var MyAsnSequenceType v_seq2;
var integer v_result := decvalue(v_enc, v_seq2); // uses the JSON decoder
```

4.22.6. Calling user defined encoding functions with encvalue and decvalue

The predefined functions `encvalue` and `decvalue` can be used to encode and decode values with user defined external functions (custom encoding and decoding functions).

These functions must have the `encode`/`decode` and `prototype` extension attributes, similarly to built-in encoder and decoder functions, except the name of the encoding (the string specified in the `encode` or `decode` extension attribute) must not be equal to any of the built-in encoding names (e.g. BER, TEXT, XER, etc.).

The compiler generates calls to these functions whenever `encvalue` or `decvalue` is called, or whenever a matching operation is performed on a `decmatch` template, or whenever a redirected value or parameter is decoded (with the `@decoded` modifier), if the value's type has the same encoding as the encoder or decoder function (the string specified in the type's `encode` attribute is equivalent to the string in the external function's `encode` or `decode` extension attribute).

Restrictions:

- only one custom encoding and one custom decoding function can be declared per user-defined codec (only checked if `encvalue`, `decvalue`, `decmatch` or `@decoded` are used at least once on the type)
- the prototype of custom encoding functions must be `convert`

- the prototype of custom decoding functions must be `sliding`
- the stream type of custom encoding and decoding functions is `bitstring`

NOTE	Although theoretically variant attributes can be added for custom encoding types, their coding functions would not receive any information about them, so they would essentially be regarded as comments. If custom variant attributes are used, the variant attribute parser's error level must be lowered to warnings with the compiler option `-E`. The compiler searches all modules when attempting to find the coder functions needed for a type (including those that are not imported to the module where the `encvalue`, `decvalue`, `decmatch` or `@decoded` is located; if this is the case, then an extra include statement is added in the generated C++ code to the header generated for the coder function's module).

Example:

```
type union Value {
  integer intVal,
  octetstring byteVal,
  charstring strVal
  }
with {
  encode "abc";
}

external function f_enc_value(in Value x) return bitstring
 with { extension "prototype(convert) encode(abc)" }

external function f_dec_value(inout bitstring b, out Value x) return integer
with { extension "prototype(sliding) decode(abc)" }

...

var Value x := { intVal := 3 };
var bitstring bs := encvalue(x); // equivalent to f_enc_value(x)

var integer res := decvalue(bs, x); // equivalent to f_dec_value(bs, x)
```

4.22.7. Legacy PER encoding and decoding through user defined functions

TITAN now has a built-in PER codec, but it also provides the means to call user defined PER encoder and decoder external functions when using `encvalue`, `decvalue`, `decmatch` templates, and value and parameter redirects with the `@decoded` modifier. This requires the use of the compiler option `-H`, otherwise these external functions will have the built-in PER encoding/decoding code generated for them.

This can be achieved the same way as the custom encoder and decoder functions described in section Calling user defined encoding functions with encvalue and decvalue, except the name of the

encoding (the string specified in the encode or decode extension attribute) must be PER.

This can only be done for ASN.1 types, and has the same restrictions as the custom encoder and decoder functions. There is one extra restriction when using legacy codec handling (see section Setting the default codec for ASN.1 types): an ASN.1 type cannot have both a PER encoder/decoder function and an encoder/decoder function of a built-in type set (this is checked separately for encoding and decoding).

NOTE	The compiler searches all modules when attempting to find the coder functions needed for a type (including those that are not imported to the module where the `encvalue`, `decvalue`, `decmatch` or `@decoded` is located; if this is the case, then an extra include statement is added in the generated C++ code to the header generated for the coder function's module).

Example:

```
external function f_enc_per(in MyAsnSequenceType x) return bitstring
with { extension "prototype(convert) encode(PER)" }

external function f_dec_per(in bitstring x, out MyAsnSequenceType y)
with { extension "prototype(fast) decode(PER)" }

...

var MyAsnSequenceType x := { num := 10, str := "abc" };
var bitstring bs := encvalue(x); // equivalent to f_enc_per(x)

var MyAsnSequenceType y;
var integer res := decvalue(bs, y); // equivalent to f_dec_per(bs, y);
```

4.22.8. Common Syntax of Attributes

All information related to implicit message encoding shall be given as `extension` attributes of the relevant TTCN-3 definitions. The attributes have a common basic syntax, which is applicable to all attributes given in this section:

- Whitespace characters (spaces, tabulators, newlines, etc.) and TTCN-3 comments are allowed anywhere in the attribute text. Attributes containing only comments, whitespace or both are simply ignored
  Example:
  `with { extension "/* this is a comment */" }`
- When a definition has multiple attributes, the attributes can be given either in one attribute text separated by whitespace or in separate TTCN-3 attributes.
  Example:
  `with { extension "address provider" }` means exactly the same as
  `with { extension "address"; extension "provider" }`
- Settings for a single attribute, however, cannot be split in several TTCN-3 attributes.

Example of an invalid attribute:
`with { extension "prototype("; extension "convert)" }`

- Each kind of attribute can be given at most once for a definition.
  Example of an invalid attribute:
  `with { extension "internal internal" }`
- The order of attributes is not relevant.
  Example:
  `with { extension "prototype(fast) encode(RAW)" }` means exactly the same as
  `with { extension "encode(RAW) prototype(fast)" }`
- The keywords introduced in this section, which are not TTCN-3 keywords, are not reserved words. The compiler will recognize the word properly if it has a different meaning (e.g. the name of a type) in the given context.
  Example: the attribute
  `with { extension "user provider in(internal → simple: function(prototype))" }` can be a valid if there is a port type named `provider`; `internal` and `simple` are message types and `prototype` is the name of a function.

4.22.9. API describing External Interfaces

Since the default class hierarchy of test ports does not allow sharing of C++ code with other port types, an alternate internal API is introduced for port types describing external interfaces. This alternate internal API is selected by giving the appropriate TTCN-3 extension attribute to the port. The following extension attributes or attribute combinations can be used:

Table 6. Port extension attributes

Attribute(s)	Test Port	Communication with SUT allowed	Using of SUT addresses allowed	External interface	Notes
nothing	normal	yes	no	own	
internal	none	no	no	own	
address	see [16] "Support of address type"	yes	yes	own	
provider	see [16] "Provider port types"	yes	no	own	
internal provider	none	no	no	own	means the same as internal
address provider	see [16] "Support of address type" and "Provider port types"	yes	yes	own	

Attribute(s)	Test Port	Communication with SUT allowed	Using of SUT addresses allowed	External interface	Notes
user PT ...	none	yes	depends on PT	PT	PT must have attribute provider
internal user PT ...	none	no	no	PT	PT can have any attributes
address user PT ...	none	yes	yes	PT	PT must have attributes address and provider

4.22.10. BNF Syntax of Attributes

```
FunctionAttributes ::= {FunctionAttribute}
FunctionAttribute ::= PrototypeAttribute | TransparentAttribute

ExternalFunctionAttributes ::= {ExternalFunctionAttribute}
ExternalFunctionAttribute ::= PrototypeAttribute | EncodeAttribute | DecodeAttribute |
ErrorBehaviorAttribute

PortTypeAttributes ::= {PortTypeAttribute}
PortTypeAttribute ::= InternalAttribute | AddressAttribute | ProviderAttribute |
UserAttribute

PrototypeAttribute ::= "prototype" "(" PrototypeSetting ")"
PrototypeSetting ::= "convert" | "fast" | "backtrack" | "sliding"

TransparentAttribute ::= "transparent"

EncodeAttribute ::= "encode" "(" EncodingType [":" EncodingOptions] ")"
EncodingType ::= "BER" | "RAW" | "TEXT"| "XER" | "JSON" | "OER" | "PER"
EncodingOptions ::= {ExtendedAlphaNum}

DecodeAttribute ::= "decode" "(" EncodingType [":" EncodingOptions] ")"

ErrorBehaviorAttribute ::= "errorbehavior" "(" ErrorBehaviorSetting {","
ErrorBehaviorSetting} ")"
ErrorBehaviorSetting ::= ErrorType ":" ErrorHandling
ErrorType ::= ErrorTypeIdentifier | "ALL"
ErrorHandling ::= "DEFAULT" | "ERROR" | "WARNING" | "IGNORE"

InternalAttribute ::= "internal"

AddressAttribute ::= "address"

ProviderAttribute ::= "provider"
```

```
UserAttribute ::= "user" PortTypeReference {InOutTypeMapping}
PortTypeReference ::= [ModuleIdentifier "."] PortTypeIdentifier
InOutTypeMapping ::= ("in" | "out") "(" TypeMapping {";" TypeMapping} ")"
TypeMapping ::= MessageType "->" TypeMappingTarget {"," TypeMappingTarget}
TypeMappingTarget ::= (MessageType ":" (SimpleMapping | FunctionMapping |
EncodeMapping | DecodeMapping)) | ("-" ":" DiscardMapping)

MessageType ::= PredefinedType | ReferencedMessageType
ReferencedMessageType ::= [ModuleIdentifier "."] (StructTypeIdentifier |
EnumTypeIdentifier | SubTypeIdentifier | ComponentTypeIdentifier)

SimpleMapping ::= "simple"

FunctionMapping ::= "function" "(" FunctionReference ")"
FunctionReference ::= [ModuleIdentifier "."] (FunctionIdentifier |
ExtFunctionIdentifier)

EncodeMapping ::= EncodeAttribute [ErrorBehaviorAttribute]

DecodeMapping ::= DecodeAttribute [ErrorBehaviorAttribute]

DiscardMapping ::= "discard"
```

Non-terminal symbols in bold are references to the BNF of the TTCN-3 Core Language (Annex A, [1])

Example:

```
type record ControlRequest { }
type record ControlResponse { }
type record PDUType1 { }
type record PDUType2 { }
// the encoder/decoder functions are written in {cpp}
external function enc_PDUType1(in PDUType1 par) return octetstring
with { extension "prototype(convert)" }
external function dec_PDUType1(in octetstring stream,
out PDUType1 result) return integer
with { extension "prototype(backtrack)" }

// port type PT1 is the external interface of the dual-faced port
// with its own Test Port. See section "The purpose of Test Ports" in the API guide.

type port PT1 message {
out ControlRequest;
in ControlResponse;
inout octetstring;
} with { extension "provider" }

// port type PT2 is the internal interface of the dual-faced port
// This port is communicating directly with the SUT using the Test Port of PT1.
```

```
type port PT2 message {
out ControlRequest;
inout PDUType1, PDUType2;
} with { extension "user PT1

out(ControlRequest -> ControlRequest: simple;
PDUType1 -> octetstring: function(enc_PDUType1);
PDUType2 -> octetstring: encode(RAW))
in(ControlResponse -> - : discard;
octetstring -> PDUType1: function(dec_PDUType1),

PDUType2: decode(RAW),
* : discard)"
}

type component MTC_CT {
port PT2 MTC_PORT;
}

type component SYSTEM_SCT {
port PT1 SYSTEM_PORT;
}
testcase tc_DUALFACED () runs on MTC_CT system SYSTEM_SCT

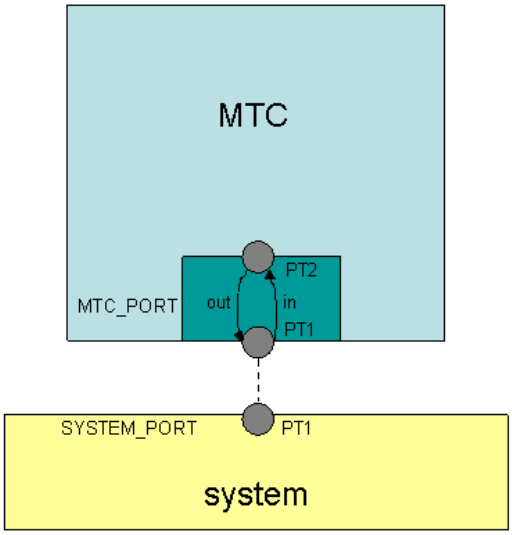
{
map(mtc:MTC_PORT, system:SYSTEM_PORT);
MTC_PORT.send(PDUType1:{…});
MTC_PORT.receive(PDUType1:?);
}
```

The external face of the dual-faced port (defined by `PT1`) sends and receives the protocol massages as octetstrings. On the internal face of the same dual-faced port (defined by `PT2`) the octetstring is converted to two message types (`PDUType1`, `PDUType2`). The conversion happens both when sending and when receiving messages.

When sending messages, messages of type `PDUType1` will be converted as defined by the function `enc_PDUType1`; whereas messages of type `PDUType2` will be converted using the built-in conversion rules RAW.

When a piece of octetstring is received, decoding will first be attempted using the function `dec_PDUType1`; in successful case the resulting structured type has `PDUType1`. When decoding using the function `dec_PDUType1` is unsuccessful, the octetstring is decoded using the built-in conversion rules RAW; the resulting message is of type `PDUType2`. When none of the above conversion succeeds, the octetstring will be discarded.

`ControlRequest` and `ControlResponse` will not be affected by a conversion in either direction.



4.23. RAW Encoder and Decoder

The RAW encoder and decoder are general purpose functionalities developed originally for handling legacy protocols.

The encoder converts abstract TTCN-3 structures (or types) into a bitstream suitable for serial transmission.

The decoder, on the contrary, converts the received bitstream into values of abstract TTCN-3 structures.

This section covers the coding rules in general, the attributes controlling them and the attributes allowed for a particular type.

You can use the encoding rules defined in this section to encode and decode the following TTCN-3 types:

- bitstring
- boolean
- charstring
- enumerated
- float

- hexstring
- integer
- octetstring
- record
- record of, set of
- set
- union
- universal charstring

The compiler will produce code capable of RAW encoding/decoding if

1. The module has attribute 'encode "RAW", in other words at the end of the module there is a text `with { encode "RAW" }`
2. Compound types have at least one `variant` attribute. When a compound type is only used internally or it is never RAW encoded/decoded then the attribute `variant` has to be omitted.

NOTE

When a type can be RAW encoded/decoded but with default specification then the empty variant specification can be used: variant "".
In order to reduce the code size the TITAN compiler only add the RAW encoding if

1. Either the type has a RAW variant attribute OR
2. The type is used by an upper level type definition with RAW variant attribute.

Example: In this minimal introductory example there are two types to be RAW encoded: OCT2 and CX_Frame but only the one of them has RAW variant attribute.

```
module Frame {
external function enc_CX_frame( in CX_Frame cx_message ) return octetstring
with { extension "prototype(convert) encode(RAW)" }

external function dec_CX_frame( in octetstring stream ) return CX_Frame
with { extension "prototype(convert) decode(RAW)" }

type octetstring OCT2 length(2);
type record CX_Frame

{
OCT2 data_length,
octetstring data_stream
} with { variant "" }
} with { encode "RAW" }
```

4.23.1. General Rules and Restrictions

The TTCN-3 standard defines a mechanism using `attributes` to define, among others, encoding

variants (see [1], chapter 27 "Specifying attributes"). However, the `attributes` to be defined are implementation specific. This and the following chapters describe each `attribute` available in TITAN.

General Rules

If an `attribute` can be assigned to a given type, it can also almost always be assigned to the same type of fields in a `record`, `set` or `union`. Attributes belonging to a `record` or `set` field overwrites the effect of the same attributes specified for the type of the field.

The location of an attribute is evaluated before the attribute itself. This means that if an attribute is overwritten thanks to its qualification or the overwriting rules, or both, its validity at the given location will not be checked.

It is not recommended to use the attributes `LENGTHTO`, `LENGTHINDEX`, `TAG`, `CROSSTAG`, `PRESENCE`, `UNIT`, `POINTERTO`, `PTROFFSET` with dotted qualifiers as it may lead to confusion.

Octetstrings and records with extension bit shall be octet aligned. That is, they should start and end in octet boundary.

Error encountered during the encoding or decoding process are handled as defined in section "Setting error behavior" in [16].

4.23.2. Rules Concerning the Encoder

The encoder doesn't modify the data to be encoded; instead, it substitutes the value of calculated fields (`length`, `pointer`, `tag`, `crosstag` and `presence` fields) with the calculated value in the encoded bitfield if necessary.

The value of the `pointer` and `length` fields are calculated during encoding and the resulting value will be used in sending operations. During decoding, the decoder uses the received length and pointer information to determine the length and the place of the fields.

During encoding, the encoder sets the value of the `presence`, `tag` and `crosstag` fields according to the presence of the `optional` and `union` fields.

4.23.3. Rule Concerning the Decoder

The decoder determines the presence of the optional fields on the basis of the value of the `tag`, `crosstag` and `presence` fields.

4.23.4. Attributes

An `attribute` determines coding and encoding rules. In this section the `attributes` are grouped according to their function.

Attributes Governing Conversion of TTCN-3 Types into Bitfields

This section defines the attributes describing how a TTCN-3 type is converted to a bitfield.

BITORDERINFIELD

Attribute syntax: `BITORDERINFIELD(<parameter>)`

Parameters allowed: `msb`, `lsb`

Default value: `lsb`

Can be used with: stand-alone types, or a field of a `record` or `set`.

Description: This attribute specifies the order of the bits within a field. When set to `msb`, the first bit sent will be the most significant bit of the original field. When set to `lsb`, the first bit sent will be the least significant bit of the original field.

Comment: The effect of `BITORDERINFIELD(msb)` is equal to the effect of `BITORDER(msb) BYTEORDER(last)`.

Example:

```
type bitstring BITn
with {
variant "BITORDERINFIELD(lsb)"
}

const BITn c_bits := '10010110'B
//Encoding of c_bits gives the following result: 10010110

type bitstring BITnreverse
with {
variant "BITORDERINFIELD(msb)"
}

const BITnreverse c_bitsrev := '10010110'B
//Encoding of c_bitsrev gives the following result: 01101001
```

COMP

Attribute syntax: `COMP(<parameter>)`

Parameters allowed: `nosign`, `2scompl`, `signbit`

Default value: `nosign`

Can be used with: stand-alone types or the field of a `record` or `set`.

Description: This attribute specifies the type of encoding of negative integer numbers as follows:
`nosign`: negative numbers are not allowed;
`2scompl`: 2's complement encoding;
`signbit`: sign bit and the absolute value is coded. (Only with integer and enumerated types.)

Examples:

```
//Example number 1): coding with sign bit
type integer INT1
with {
variant "COMP(signbit)";
variant "FIELDLENGTH(8)"
}

const INT1 c_i := -1
//Encoded c_i: 10000001 '81'O
// sign bit
//Example number 2): two's complement coding
type integer INT2 with {variant "COMP(2scompl)";
variant "FIELDLENGTH(8)"
}

const INT2 c_i2 := -1
//Encoded c_i2: 11111111 'FF'O
```

FIELDLENGTH

Attribute syntax: `FIELDLENGTH(<parameter>)`

Parameters allowed: `variable`, `null_terminated` (for `charstring` and universal `charstring` types only) positive integer

Default value: `variable`, 8 (for `integer` type only)

Can be used with:

- `integer`;
- `enumerated`;
- `octetstring`;
- `charstring`;
- `bitstring`;
- `hexstring`;
- `universal charstring`;
- `record` fields;
- `set` fields;
- `record of` types;
- `set of` types.

Description: `FIELDLENGTH` specifies the length of the encoded type. The units of the parameter value for specific types are the following:

- `integer, enumerated, bitstring:` bits;

- `octetstring, universal charstring:` octets;
- `charstring:` characters;
- `hexstring:` hex digits;
- `set of/record of:` elements.

The value 0 means variable length or, in case of the enumerated type, the minimum number of bits required to display the maximum `enumerated` value. `Integer` cannot be coded with variable length.

NOTE	If `FIELDLENGTH` is not specified, but a TTCN-3 length restriction with a fixed length is, then the restricted length will be used as `FIELDLENGTH`.

Examples:

```
//Example number 1): variable length octetstring
type octetstring OCTn
with {
variant "FIELDLENGTH(variable)"
}

//Example number 2): 22 bit length bitstrings
type bitstring BIT22
with {
variant "FIELDLENGTH(22)"
}

type record SomeRecord {
bitstring field
}

with {
variant (field) "FIELDLENGTH(22)"
}

// Null terminated strings
type charstring null_str with {variant "FIELDLENGTH(null_terminated)"}
type universal charstring null_ustr with { variant "FIELDLENGTH(null_terminated)"}
```

N bit / unsigned N bit

Attribute syntax: `[unsigned] <parameter> bit`

Parameters allowed: positive integer

Default value: -

Can be used with:

- `integer`;

- `enumerated`;
- `octetstring`;
- `charstring`;
- `bitstring`;
- `hexstring`;
- `record` fields;
- `set` fields.

Description: This attribute sets the `FIELDLENGTH`, `BYTEORDER` and `COMP` attributes to the following values:

- `BYTEORDER` is set to `last`.
- `N bit` sets `COMP` to `signbit`, while `unsigned N bit` sets `COMP` to `nosign` (its default value).
- Depending on the encoded value's type `FIELDLENGTH` is set to:
  `integer, enumerated, bitstring, boolean:` N;
  `octetstring, charstring:` N / 8;
  `hexstring:` N / 4.

NOTE	If `FIELDLENGTH` is not specified, but a TTCN-3 length restriction with a fixed length is, then the restricted length will be used as `FIELDLENGTH`.

The `[unsigned] <parameter> bits` syntax is also supported but the usage of `bit` keyword is preferred.

Examples:

```
//Example number 1): integer types
type integer Short (-32768 .. 32767)
with { variant "16 bit" };

// is equal to:
type integer ShortEq (-32768 .. 32767)
with { variant "FIELDLENGTH(16), COMP(signbit), BYTEORDER(last)" };

type integer UnsignedLong (0 .. 4294967295)
with { variant "unsigned 32 bit" };

// is equal to:
type integer UnsignedLongEq (0 .. 4294967295)
with { variant "FIELDLENGTH(32), COMP(nosign), BYTEORDER(last)" };

//Example number 2): string types
type hexstring HStr20
with { variant "unsigned 20 bit" };

// 20 bits = 5 hex nibbles, `unsigned' is ignored
type hexstring HStr20Eq
with { variant "FIELDLENGTH(5), BYTEORDER(last)" };

type octetstring OStr32
with { variant "32 bit" };

// 32 bits = 4 octets
type octetstring OStr32Eq
with { variant "FIELDLENGTH(4), BYTEORDER(last)" };

type charstring CStr64 with
{ variant "64 bit" };

// 64 bits = 8 characters
type charstring CStr64Eq
with { variant "FIELDLENGTH(8), BYTEORDER(last)" };
```

FORMAT

Attribute syntax: `FORMAT(<parameter>)`

Parameters allowed: `IEEE754 double`, `IEEE754 float`

Default value: `IEEE754 double`

Can be used with: `float` type.

Description: `FORMAT` specifies the encoding format of `float` values.
`IEEE754 double:` The `float` value is encoded as specified in standard IEEE754 using 1 sign bit, 11 exponent bits and 52 bits for mantissa.

`IEEE754 float:` The `float` value is encoded as specified in standard IEEE754 using 1 sign bit, 8 exponent bits and 23 bits for mantissa.

Examples:

```
//Example number 1): single precision float
type float Single_float
with {
variant "FORMAT(IEEE754 float)"
}

//Example number 2): double precision float
type float Double_float
with {
variant "FORMAT(IEEE754 double)"
}
```

Attributes Controlling Conversion of Bitfields into a Bitstream

This section defines the attributes describing how bits and octets are put into the buffer.

BITORDER

Attribute syntax: `BITORDER(<parameter>)`

Parameters allowed: `msb`, `lsb`

Default value: `lsb`

Can be used with: stand-alone types or the field of a `record` or `set`.

Description: This attribute specifies the order of the bits within an octet. When set to `lsb`, the first bit sent will be the least significant bit of the original byte. When set to `msb`, the first bit sent will be the most significant bit of the original byte. When applied to an `octetstring` using the extension bit mechanism, only the least significant 7 bits are reversed, the 8th bit is reserved for the extension bit.

Examples:

```
// Example number 1)
type octetstring OCT
with {
variant "BITORDER(lsb)"
}

const OCT c_oct := '123456'O

//The encoded bitfield: 01010110 00110100 00010010
// last octet^ ^first octet
// The buffer will have the following content:
```

```
// 00010010
// 00110100
// 01010110

//The encoding result in the octetstring '123456'O

//Example number 2)
type octetstring OCTrev
with {
variant "BITORDER(msb)"
}

const OCTrev c_octr := '123456'O

//The encoded bitfield: 01010110 00110100 00010010

// last octet^ ^first octet

//The buffer will have the following content:
// 01001000
// 00101100
// 01101010

//The encoding results in the octetstring '482C6A'O

//Example number 3)

type bitstring BIT12 with {
variant "BITORDER(lsb), FIELDLENGTH(12)"
}

const BIT12 c_bits:='101101101010'B
//The encoded bitfield: 1011 01101010

// last octet^ ^first octet

The buffer will have the following content:
// 01101010
// ….1011
// ^ next field

//The encoding will result in the octetstring '6A.9'O

//Example number 4)
type bitstring BIT12rev with {
variant "BITORDER(msb), FIELDLENGTH(12)"
}

const BIT12 c_BIT12rev:='101101101010'B
//The encoded bitfield: 1011 01101010
// last octet^ ^first octet
```

```
//The buffer will have the following content:
// 01010110
// ….1101
// ^ next field
//The encoding will result in the octetstring '56.D'O
```

BYTEORDER

Attribute syntax: `BYTEORDER(<parameter>)`

Parameters allowed: `first`, `last`

Default value: `first`

Can be used with: stand-alone types or the field of a `record` or `set`.

Description: The attribute determines the order of the bytes in the encoded data.

- `first`: The first octet placed first into the buffer.
- `last`: The last octet placed first into the buffer.

Comment: The attribute has no effect on a single octet field.

NOTE	The attribute works differently for `octetstring` and `integer` types. The ordering of bytes is counted from left-to-right (starting from the MSB) in an `octetstring` but right-to-left (starting from the LSB) in an `integer`. Thus, the attribute `BYTEORDER(first)` for an `octetstring` results the same encoded value than `BYTEORDER(last)` for an `integer` having the same value.

Examples:

```
//Example number 1)
type octetstring OCT
with {
variant "BYTEORDER(first)"
}

const OCT c_oct := '123456'O
//The encoded bitfield: 01010110 00110100 00010010
// last octet^ ^first octet

The buffer will have the following content:
// 00010010
// 00110100
// 01010110

//The encoding will result in the octetstring '123456'O

//Example number 2)
type octetstring OCTrev
```

```
with {variant "BYTEORDER(last)"
}

const OCTrev c_octr := '123456'O
//The encoded bitfield: 01010110 00110100 00010010
// last octet^ ^first octet

//The buffer will have the following content:

// 01010110

// 00110100

// 00010010

The encoding will result in the octetstring '563412'O
//Example number 3)
type bitstring BIT12 with {
variant "BYTEORDER(first), FIELDLENGTH(12)"
}

const BIT12 c_bits:='100101101010'B
//The encoded bitfield: 1001 01101010
// last octet^ ^first octet
The buffer will have the following content:
// 01101010
// ….1001
// ^ next field

//The encoding will result in the octetstring '6A.9'O
//Example number 4)
type bitstring BIT12rev with {
variant "BYTEORDER(last), FIELDLENGTH(12)"
}

const BIT12rev c_bits:='100101101010'B
//The encoded bitfield: 1001 01101010
// last octet^ ^first octet
//The buffer will have the following content:
// 10010110
// ….1010
// ^ next field
//The encoding will result in the octetstring '96.A'O
```

FIELDORDER

Attribute syntax: `FIELDORDER(<parameter>)`

Parameters allowed: `msb`, `lsb`

Default value: `lsb`

Can be used with: `record` or `set` types. It can also be assigned to a group of fields of a `record`.

Description: The attribute specifies the order in which consecutive fields of a structured type are placed into octets. * `msb:` coded bitfields are concatenated within an octet starting from MSB, when a field stretches the octet boundary, it continues at the MSB of next the octet. * `lsb:` coded bitfields are concatenated within an octet starting from LSB, when a field stretches the octet boundary, it continues at the LSB of next the octet.

Comment: Fields within an octet must be coded with the same `FIELDORDER`.
Fields are always concatenated in increasing octet number direction.
`FIELDORDER` has a slightly different effect than order attributes. While the `FIELDORDER` shifts the location of coded bitfields inside octets, the order attributes describes the order of the bits within a bitfield.
There is NO connection between the effect of the `FIELDORDER` and the effects of the other order attributes.

NOTE	The attribute does not extend to lower level structures. If the same field order is desired for the fields of a lower level `record`/`set`, then that `record`/`set` also needs a `FIELDORDER` attribute.

Examples:

```
//Example number 1)
type record MyRec_lsb {
BIT1 field1,
BIT2 field2,
BIT3 field3,
BIT4 field4,
BIT6 field5
}

with { variant "FIELDORDER(lsb)" }
const MyRec_lsb c_pdu := {
field1:='1'B // bits of field1: a
field2:='00'B // bits of field2: b
field3:='111'B // bits of field3: c
field4:='0000'B // bits of field4: d
field5:='111111'B // bits of field5: e
}

//Encoding of c_pdu will result in:
// 00111001 ddcccbba
// 11111100 eeeeeedd
//Example number 2)

type record MyRec_msb {
BIT1 field1,
BIT2 field2,
BIT3 field3,
BIT4 field4,
BIT6 field5
}

with { variant "FIELDORDER(msb)" }
const MyRec_msb c_pdu2 := {
field1:='1'B // bits of field1: a
field2:='00'B // bits of field2: b
field3:='111'B // bits of field3: c
field4:='0000'B // bits of field4: d
field5:='111111'B // bits of field5: e
}

//Encoding of c_pdu2 will result in:
// 10011100 abbcccdd
// 00111111 ddeeeeee
```

HEXORDER

Attribute syntax: `HEXORDER(<parameter>)`

Parameters allowed: `low`, `high`

Default value: `low`

Can be used with: `hexstring` or `octetstring` type.

Description: Order of the hexs in the encoded data. * `low:` The hex digit in the lower nibble of the octet is put in the lower nibble of the octet in the buffer. * `high:` The hex digit in the higher nibble of the octet is put in the lower nibble of the octet in the buffer. (The value is swapped)

NOTE	Only the whole octet is swapped if necessary. For more details see the example.

Examples:

```
//Example number 1)
type hexstring HEX_high
with {variant "HEXORDER(high)"}

const HEX_high c_hexs := '12345'H
//The encoded bitfield: 0101 00110100 00010010
// last octet^ ^first octet

//The buffer will have the following content:
// 00010010 12
// 00110100 34
// ….0101 .5
// ^ next field
//The encoding will result in the octetstring '1234.5'O

//Example number 2)
type hexstring HEX_low
with {variant "HEXORDER(low)"}
const HEX_low c_hexl := '12345'H

//The encoded bitfield: 0101 00110100 00010010
// last octet^ ^first octet
//The buffer will have the following content:
// 00100001 21
// 01000011 43
// ….0101 .5 ←not twisted!
// ^ next field
//The encoding will result in the octetstring '2143.5'O

//Example number 3)
type octetstring OCT
with {variant "HEXORDER(high)"}

const OCT c_hocts := '1234'O
//The encoded bitfield: 00110100 00010010
// last octet^ ^first octet
//The buffer will have the following content:
// 00100001 21
// 01000011 43
//The encoding will result in the octetstring '2143'O
```

CSN.1 L/H

Attribute syntax: `CSN.1 L/H`

Default value: unset

Can be used with: all basic types, `records`/`sets`/`unions` (in which case the attribute is set for all fields of the `record`/`set`/`union`)

Description: If set, the bits in the bitfield are treated as the relative values `L` and `H` from `CSN.1` instead of their absolute values (`0` is treated as `L` and `1` is treated as `H`). These values are encoded in terms of the default padding pattern '2B'O ('00101011'B), depending on their position in the bitstream.

Practically the bits in the bitfield are XOR-ed with the pattern '2B'O before being inserted into the stream.

Example:

```
type integer uint16_t
with { variant "FIELDLENGTH(16)" variant "CSN.1 L/H" }

const uint16_t c_val := 4080;
// Without the variant attribute "CSN.1 L/H" this would be encoded as '11110000
00001111'B
// With the variant attribute "CSN.1 L/H" this would be encoded as '11011011
00100100'B
```

Extension Bit Setting Attributes

This section defines the attributes describing the extension bit mechanism.

The extension bit mechanism allows the size of an Information Element (IE) to be increased by using the most significant bit (MSB, bit 7) of an octet as an extension bit. When an octet within an IE has bit 7 defined as an extension bit, then the value `0' in that bit position indicates that the following octet is an extension of the current octet. When the value is `1', the octet is not continued.

EXTENSION_BIT

Attribute syntax: `EXTENSION_BIT(<parameter>)`

Parameters allowed: `no`, `yes`, `reverse`

Default value: none

Can be used with:

- `octetstring`,
- (fields of a) `record`,
- `set`,
- `record of`,
- `set of`.

Description: When `EXTENSION_BIT` is set to `yes`, then each MSB is set to 0 except the last one which is set to 1. When `EXTENSION_BIT` is set to `reverse`, then each MSB is set to 1 and the MSB of the last octet is set to 0 to indicate the end of the Information Element. When `EXTENSION_BIT` is set to `no`, then no changes are made to the MSBs.

NOTE	In case of the types `record` of and `set of` the last bit of the element of the structured type will be used as `EXTENSION_BIT`. The data in the MSBs will be overwritten during the encoding. When `EXTENSION_BIT` belongs to a record, the field containing the `EXTENSION_BIT` must explicitly be declared in the type definition. Also the last bit of the element of `record of` and `set of` type shall be reserved for `EXTENSION_BIT` in the type definition.

Examples:

```
//Example number 1)
octetstring OCTn
with {variant "EXTENSION_BIT(reverse)"}
const OCTn c_octs:='586211'O

//The encoding will have the following result:
// 11011000
// 11100010
// 00010001
// □ the overwritten EXTENSION_BITs

//The encoding will result in the octetstring 'D8E211'O
//Example number 2)

type record Rec3 {
BIT7 field1,
BIT1 extbit1,
BIT7 field2 optional,
BIT1 extbit2 optional
}

with { variant "EXTENSION_BIT(yes)" }
const Rec3 c_MyRec{
field1:='1000001'B,
extbit1:='1'B,
field2:='1011101'B,
extbit2:='0'B
}

//The encoding will have the following result:
// 01000001
// 11011101
// □ the overwritten EXTENSION_BITs

The encoding will result in the octetstring '41DD'O

//Example number 3)
type record Rec4{
BIT11 field1,
BIT1 extbit
}
```

```
type record of Rec4 RecList
with { variant "EXTENSION_BIT(yes)"}
const RecList c_recs{
{ field1:='10010011011'B, extbit:='1'B}
{ field1:='11010111010'B, extbit:='0'B}
}

//The encoding will have the following result:
// 10011011
// 10100100
// 11101011
// □ the overwritten EXTENSION_BITs

//The encoding will result in the octetstring '9BA4EB'O
```

EXTENSION_BIT_GROUP

Attribute syntax: `EXTENSION_BIT_GROUP(<param1, param2, param3>)`

Parameters allowed: `param1: no, yes, reverse`
`param2: first_field_name`,
`param3: last_field_name`

Default value: none

Can be used with: a group of `record` fields

Description: The `EXTENSION_BIT_GROUP` limits the extension bit mechanism to a group of the fields of a `record` instead of the whole `record`.
`first_field_name`: the name of the first field in the
`grouplast_field_name`: the name of the last field in the group

NOTE	Multiple group definition is allowed to define more groups within one `record`. Every group must be octet aligned and the groups must not overlap.

Example:

```
type record MyPDU{
OCT1 header,
BIT7 octet2info,
BIT1 extbit1,
BIT7 octet2ainfo optional,
BIT1 extbit2 optional,
OCT1 octet3,
BIT7 octet4info,
BIT1 extbit3,
BIT7 octet4ainfo optional,
BIT1 extbit4 optional,
} with {
variant "EXTENSION_BIT_GROUP(yes,octet2info,extbit2)";
variant "EXTENSION_BIT_GROUP(yes,octet4info,extbit4)"
}

const MyPDU c_pdu:={
header:='0F'O,
octet2info:='1011011'B,
extbit1:= '0'B,
octet2ainfo:= omit,
extbit2:= omit,
octet3:='00'O,
octet4info:='0110001'B,
extbit3:='1'B,
octet4ainfo:='0011100'B,
extbit4:='0'B,
}

//The encoding will have the following result:
// 00001111
// **1**1011011
// 00000000
// **0**0110001
// **1**0011100
//  the overwritten extension bits
//The encoding will result in the octetstring: '0FDB00319C'O
```

Attributes Controlling Padding

This section defines the attributes that describe the padding of fields.

ALIGN

Attribute syntax: `ALIGN(<parameter>)`

Parameters allowed: `left`, `right`

Default value: `left` for `octetstrings`, `right` for all other types

Can be used with: stand-alone types or the field of a `record` or `set`.

Description: This attribute has meaning when the length of the actual value can be determined and is less than the specified `FIELDLENGTH`. In such cases the remaining bits/bytes will be padded with zeros. The attribute `ALIGN` specifies the sequence of the actual value and the padding within the encoded bitfield.
`right`: The LSB of the actual value is aligned to the LSB of coded bitfield
`left`: The MSB of the actual value is aligned to the MSB of coded bitfield

NOTE	It has no meaning during decoding except if the length of the actual value can be determined from the length restriction of the type. In this case the `ALIGN` also specifies the order of the actual value and the padding within the encoded bitfield.

Examples:

```
//Example number 1)
type octetstring OCT10
with {
variant "ALIGN(left)";
variant "FIELDLENGTH(10)"
}

const OCT10 c_oct := '0102030405'O
//Encoded value: '0102030405000000000'O
//The decoded value: '0102030405000000000'O
//Example number 2)
type octetstring OCT10length5 length(5)
with {
variant "ALIGN(left)";
variant "FIELDLENGTH(10)"
}

const OCT10length5 c_oct5 := '0102030405'O
//Encoded value: '0102030405000000000'O
//The decoded value: '0102030405'O
```

PADDING

Attribute syntax: `PADDING(<parameter>)`

Parameters allowed:

- `no`
- `yes`
- `octet`
- `nibble`
- `word16`

- dword32
- integer to specify the padding unit and allow padding.

Default value: none

Can be used with: This attribute can belong to any types.

Description: This attribute specifies that an encoded type shall **end** at a boundary fixed by a multiple of padding unit bits counted from the beginning of the message. The default padding unit is 8 bits. If PADDING is set to yes, then unused bits of the last octets of the encoded type will be filled with padding pattern. If PADDING is set to no, the next field will use the remaining bits of the last octet. If padding unit is specified, then the unused bits between the end of the field and the next padding position will be filled with padding pattern.

NOTE	It is possible to use different padding for every field of structured types. The padding unit defined by PADDING and PREPADDING attributes can be different for the same type.

Examples:

```
//Example number 1)
type BIT5 Bit5padded with { variant "PADDING(yes)"}

const Bit5padded c_bits:='10011'B

//The encoding will have the following result:
// 00010011
// □ the padding bits
//The encoding will result in the octetstring '13'O

//Example number 2)
type record Paddedrec{
BIT3 field1,
BIT7 field2
} with { variant "PADDING(yes)"}

const Paddedrec c_myrec:={
field1:='101'B,
field2:='0110100'B
}

//The encoding will have the following result:
// 10100101
// 00000001
// □ the padding bits

//The encoding will result in the octetstring 'A501'O

//Example number 3): padding to 32 bits
```

```
type BIT5 Bit5padded_dw with { variant "PADDING(dword32)"}
const Bit5padded_dw c_dword:='10011'B
//The encoding will have the following result:
// 00010011
// 00000000
// 00000000
// 00000000
// □ the padding bits

The encoding will result in the octetstring '13000000'O

//Example number 4)
type record Paddedrec_dw{
BIT3 field1,
BIT7 field2
} with { variant "PADDING(dword32)"}
const Paddedrec_dw c_dwords:={
field1:='101'B,
field2:='0110100'B
}

//The encoding will have the following result:
// 10100101
// 00000001
// 00000000
// 00000000
// □ the padding bits
The encoding will result in the octetstring 'A5010000'O

//Example number 5)
type record ChangeActiveMaskRes
{
   INT1                       vtfunction (173),
   INT2                       newActiveMaskObjectID
   //we want this padded to 64 bits
} with {variant "PADDING(88), PADDING_PATTERN('11111111'B)"}

type record Message
{
OCT3     pgn,
record {
    record
        {
        ChangeActiveMaskRes   changeActiveMaskRes
        }       vt2ecu
    }    pdu
} with { variant "" };


const Message c_message:={
     pgn := '00E600'O,
```

```
     pdu := {
         vt2ecu := {
             changeActiveMaskRes := {
                 vtfunction := 173,
                 newActiveMaskObjectID := 1005
             }
         }
     }
}

The encoding will result in the following octetstring: '00E600ADED03FFFFFFFFF'O
//3 bytes pgn followed by 3 bytes pdu padded to 88 bits from the start of the message
//64 bits from the start of pdu

//Example number 6)
type record ChangeActiveMaskRes
{
   INT1                      vtfunction (173),
   INT2                      newActiveMaskObjectID
   //we want this padded to 64 bits
} with {variant "PADDING(64), PADDING_PATTERN('11111111'B)"}

type record Message
{
record {
    record
        {
        ChangeActiveMaskRes   changeActiveMaskRes
        }       vt2ecu
    }    pdu,
OCT3     pgn
} with { variant "" };

const Message c_message:={
     pdu := {
         vt2ecu := {
             changeActiveMaskRes := {
                 vtfunction := 173,
                 newActiveMaskObjectID := 1005
             }
         }
     },
pgn := '00E600'O
}

The encoding will result in the following octetstring: 'ADED03FFFFFFFFFF00E600'O
//3 bytes pdu padded to 64 bits from the start of the message
//followed by 3 bytes of pgn
```

PADDING_PATTERN

Attribute syntax: `PADDING_PATTERN(<parameter>)`

Parameters allowed: bitstring

Default value: `'0'B`

Can be used with: any type with attributes `PADDING` or `PREPADDING`.

Description: This attribute specifies padding pattern used by padding mechanism. The default padding pattern is '0'B.If the specified padding pattern is shorter than the padding space, then the padding pattern is repeated.

Comment: For a particular field or type only one padding pattern can be specified for `PADDING` and `PREPADDING`.

Examples:

```
//Example number 1)
type BIT8 Bit8padded with {
variant "PREPADDING(yes), PADDING_PATTERN('1'B)"
}

type record PDU {
BIT3 field1,
Bit8padded field2
} with {variant ""}

const PDU c_myPDU:={
field1:='101'B,
field2:='10010011'B
}

//The encoding will have the following result:
// 11111101
// 10010011
//the padding bits are indicated in bold
//The encoding will result in the octetstring 'FD93'O
//Example number 2): padding to 32 bits

type BIT8 Bit8pdd with {
variant "PREPADDING(dword32), PADDING_PATTERN('10'B)"
}

type record PDU{
BIT3 field1,
Bit8pdd field2
} with {variant ""}
const PDU c_myPDUplus:={
field1:='101'B,
field2:='10010011'B
}

//The encoding will have the following result:
// 01010101
// 01010101
// 01010101
// 01010101
// 10010011
//The padding bits are indicated in bold

//The encoding will result in the octetstring '5555555593'O
```

PADDALL

Attribute syntax: PADDALL(<parameter>)

Can be used with: `record` or `set`.

Description: If `PADDALL` is specified, the padding parameter specified for a whole `record` or `set` will be valid for every field of the structured type in question.

NOTE	If a different padding parameter is specified for any fields it won't be overridden by the padding parameter specified for the record.

Examples:

```
//Example number 1)
type record Paddedrec{
BIT3 field1,
BIT7 field2
} with { variant "PADDING(yes)"}
const Paddedrec c_myrec:={
field1:='101'B,
field2:='0110100'B
}

//The encoding will have the following result:
// 10100101
// 00000001
// □ the padding bits
//The encoding will result in the octetstring 'A501'O

//Example number 2)

type record Padddd{
BIT3 field1,
BIT7 field2
} with { variant "PADDING(yes), PADDALL"}

const Padddd c_myrec:={
field1:='101'B,
field2:='0110100'B
}

//The encoding will have the following result:
// 00000101
// 00110100
// □ the padding bits

//The encoding will result in the octetstring '0534'O

//Example number 3)

type record Padded{
BIT3 field1,
BIT5 field2,
BIT7 field3
} with { variant "PADDING(yes), PADDALL"}
```

```
const Padded c_ourrec:={
field1:='101'B,
field2:='10011'B,
field3:='0110100'B
}

//The encoding will have the following result:
// 00000101
// 00010011
// 00110100
// □ the padding bits

//The encoding will result in the octetstring '051334'O

//Example number 4): field1 shouldn't be padded

type record Paddd{
BIT3 field1,
BIT5 field2,
BIT7 field3
} with { variant "PADDING(yes), PADDALL";
variant (field1) "PADDING(no)" }
const Paddd c_myrec:={
field1:='101'B,
field2:='10011'B,
field3:='0110100'B
}

//The encoding will have the following result:
// 10011101 < field1 is not padded!!!
// 00110100
// □ the padding bit
//The encoding will result in the octetstring '9D34'O
```

PREPADDING

Attribute syntax: `PREPADDING(<parameter>)`

Parameters allowed:

- `no`
- `yes`
- `octet`
- `nibble`
- `word16`
- `dword32`
- integer to specify the padding unit and allow padding.

Default value: none

Can be used with: any type.

Description: This attribute specifies that an encoded type shall **start** at a boundary fixed by a multiple of padding unit bits counted from the beginning of the message. The default padding unit is 8 bits. If `PREPADDING` is set to `yes`, then unused bits of the last octets of the previous encoded type will be filled with padding pattern and the actual field starts at octet boundary. If `PREPADDING` is set to `no`, the remaining bits of the last octet will be used by the field. If padding unit specified, then the unused bits between the end of the last field and the next padding position will be filled with padding pattern and the actual field starts at from this point.

NOTE	It is possible to use different padding for every field of structured types. The padding unit defined by `PADDING` and `PREPADDING` attributes can be different for the same type.

Examples:

```
//Example number 1)

type BIT8 bit8padded with { variant "PREPADDING(yes)"}
type record PDU{
BIT3 field1,
bit8padded field2
} with {variant ""}
const PDU c_myPDU:={
field1:='101'B,
field2:='10010011'B
}

//The encoding will have the following result:
// 00000101
// 10010011
//The padding bits are indicated in bold
//The encoding will result in the octetstring '0593'O
//Example number 2): padding to 32 bits

type BIT8 bit8padded_dw with { variant "PREPADDING(dword32)"}
type record PDU{
BIT3 field1,
bit8padded_dw field2
} with {variant ""}
const PDU myPDU:={
field1:='101'B,
field2:='10010011'B
}

//The encoding will have the following result:
// 00000101
// 00000000
// 00000000
// 00000000
// 10010011

//The padding bits are indicated in bold

//The encoding will result in the octetstring '0500000093'O
```

Attributes of Length and Pointer Field

This section describes the coding attributes of fields containing length information or serving as pointer within a `record`.

The length and pointer fields must be of TTCN-3 `integer` type and must have fixed length.

The attributes described in this section are applicable to fields of a `record`.

LENGTHTO

Attribute syntax: `LENGTHTO(<parameter>) [ (+' | -') <offset> ]`

Parameters allowed: list of TTCN-3 field identifiers

Parameter value: any field name

Offset value: positive integer

Default value: none

Can be used with: fields of a `record`.

Description: The encoder is able to calculate the encoded length of one or several fields and put the result in another field of the same record. Consider a record with the fields `lengthField`, `field1`, `field2` and `field3`. Here `lengthField` may contain the encoded length of either one field (for example, `field2`), or sum of the lengths of multiple fields ((for example, that of `field2` + `field3`). The parameter is the field identifier (or list of field identifiers) of the `record` to be encoded.

If the offset is present, it is added to or subtracted from (the operation specified in the attribute is performed) the calculated length during encoding. During decoding, the offset is subtracted from or added to (the opposite operation to the one specified in the attribute is performed) the decoded value of the length field.

NOTE The length is expressed in units defined by the attribute UNIT The default unit is octet. The length field should be a TTCN-3 `integer`, `bitstring`, `octetstring` or `union` type. Special union containing only `integer`, `bitstring` or `octetstring` fields can be used for variable length field. It must not be used with `LENGTHINDEX`. The length field can be included in to the sum of the lengths of multiple fields (e.g. `lengthField` + `field2` + `field3`). The `union` field is NOT selected by the encoder. So the suitable field must be selected before encoding! The fields included in the length computing need not be continuous. If `bitstring` or `octetstring` type is used, the length of the field must be specified either with `length()` subtyping or with the `FIELDLENGTH` variant.

Examples:

```
//Example number 1)
type record Rec {
INT1 len,
OCT3 field1,
octetstring field2
}

with {
variant (len) "LENGTHTO(field1)";
variant (len) "UNIT(bits)"
}

//Example number 2)

type record Rec2 {
```

```
INT1 len,
OCT3 field1,
octetstring field2
}

with {
variant (len) "LENGTHTO(len, field1, field2)"
}

//Example number 3)

type record Rec3 {
INT1 len,
OCT3 field1,
OCT1 field2,
octetstring field3
}

with {
variant (len) "LENGTHTO(field1, field3)"
// field2 is excluded!
}

//Example number 4): using union as length field
type union length_union{
integer short_length_field,
integer long_length_field
} with {
variant (short_length_field) "FIELDLENGTH(7)";
variant (long_length_field) "FIELDLENGTH(15)";
}

type record Rec4{
BIT1 flag,
length_union length_field,
octetstring data
} with {
variant (length_field)
"CROSSTAG(short_length_field, flag = '0'B;
long_length_field, flag = '1'B)";
variant (length_field) "LENGTHTO(data)"
}

//Const for short data. Data is shorter than 127 octets:

const Rec4(octetstring oc):={
flag :='0'B,
length_field:={short_length_field:=0},
data := oc
}
```

```
//Const for long data. Data is longer than 126 octets:

const Rec4(octetstring oc):={
flag :='1'B,
length_field:={long_length_field:=0},
data := oc
}

//Example number 5): with offset
type record Rec5 {
integer len,
octetstring field
}

with {
variant (len) "LENGTHTO(field) + 1"
}

// { len := 0, field := '12345678'O } would be encoded into '0512345678'O
// (1 is added to the length of `field')
// and '0512345678'O would be decoded into { len := 4, field := '12345678'O }
// (1 is subtracted from the decoded value of `len')

//Example number 6): with offset

type record Rec6 {
integer len,
octetstring field
}

with {
variant (len) "LENGTHTO(field) - 2"
}

// { len := 0, field := '12345678'O } would be encoded into '0212345678'O
// (2 is subtracted from the length of `field')
// and '0212345678'O would be decoded into { len := 4, field := '12345678'O }
// (2 is added to the decoded value of `len')

//Example number 7): octetstring
type record Rec7 {
octetstring len length(1), // length of this field in octet
OCT3 field1,
octetstring field2
}

with {
variant (len) "LENGTHTO(field1)";
variant (len) "UNIT(bits)"
}
```

```
//Example number 8): bitstring
type record Rec8 {
btstring len,
OCT3 field1,
octetstring field2
}

with {
variant (len) "FIELDLENGTH(8)"; // length of this field in bits
variant (len) "LENGTHTO(field1)";
variant (len) "UNIT(bits)"
}
```

LENGTHINDEX

Attribute syntax: `LENGTHINDEX(<parameter>)`

Parameters allowed: TTCN-3 field identifier

Allowed values: any nested field name

Default value: none

Can be used with: fields of a `record`.

Description: This attribute extends the `LENGTHTO` attribute with the identification of the nested field containing the length value within the field of the corresponding `LENGTHTO` attribute.

Comment: See also the description of the `LENGTHTO` attribute.

NOTE	The field named by `LENGTHINDEX` attribute should be a TTCN-3 `integer`, `bitstring` or `octetstring` type.

Example (see also example of `LENGTHTO` attribute).

```
type integer INT1
with {
variant "FIELDLENGTH(8)"
}

type record InnerRec {
INT1 length
}

with { variant "" }
type record OuterRec {
InnerRec lengthRec,
octetstring field
}

with {
variant (lengthRec) "LENGTHTO(field)";
variant (lengthRec) "LENGTHINDEX(length)"
}
```

POINTERTO

Attribute syntax: `POINTERTO(<parameter>)`

Parameters allowed: TTCN-3 field identifier

Default value: none

Can be used with: fields of a `record`.

Description: Some record fields contain the distance to another encoded field. Records can be encoded in the form of: `ptr1`, `ptr2`, `ptr3`, `field1`, `field2`, `field3`, where the position of fieldN within the encoded stream can be determined from the value and position of field ptrN. The distance of the pointed field from the base field will be `ptrN * UNIT + PTROFFSET`. The default base field is the pointer itself. The base field can be set by the `PTROFFSET` attribute. When the pointed field is optional, the pointer value 0 indicates the absence of the pointed field.

Comment: See also the description of `UNIT` (0) and `PTROFFSET` (0) attributes.

NOTE	Pointer fields should be TTCN-3 `integer` type.

Examples:

```
type record Rec {
INT1 ptr1,
INT1 ptr2,
INT1 ptr3,
OCT3 field1,
OCT3 field2,
OCT3 field3
}

with {
variant (ptr1) "POINTERTO(field1)";
variant (ptr2) "POINTERTO(field2)";
variant (ptr3) "POINTERTO(field3)"
}

const Rec c_rec := {
ptr1 := <any value>,
ptr2 := <any value>,
ptr3 := <any value>,
field1 := '010203'O,
field2 := '040506'O,
field3 := '070809'O
}

//Encoded c_rec: '030507010203040506070809'O//The value of ptr1: 03
//PTROFFSET and UNIT are not set, so the default (0) is being //using.
//The starting position of ptr1: 0th bit
//The starting position of field1= 3 * 8 + 0 = 24th bit.
```

PTROFFSET

Attribute syntax: `PTROFFSET(<parameter>)`

Parameters allowed: `integer`, TTCN-3 field identifier

Default value: 0

Can be used with: fields of a `record`.

Description: The value of the pointer field calculated as the difference between beginning of the pointer field and the beginning of the pointed field in bits, minus the pointer offset value, expressed in whole `UNIT` (which is 8 bit, an octet by default). The `PTROFFSET` is measured in bits.

Comment: It can be specified a base field and pointer offset for one field. See also the description of the attributes `POINTERTO` (0) and `UNIT` (0).

Examples:

```
type record Rec {
INT2 ptr1,
INT2 ptr2
OCT3 field1,
OCT3 field2
}

with {
variant (ptr1) "POINTERTO(field1)";
variant (ptr1) "PTROFFSET(ptr2)";
variant (ptr2) "POINTERTO(field2)";
variant (ptr2) "PTROFFSET(field1)"
}

//In the example above the distance will not include//the pointer itself.
```

UNIT

Attribute syntax: `UNIT(<parameter>)`

Parameters allowed:

- bits
- octets
- nibble
- word16
- dword32
- elements
- integer

Default value: octets

Can be used with: fields of a `record`.

Description: `UNIT` attribute is used in conjunction with the `LENGTHTO` (0) or `POINTERTO` (0) attributes. Length indicator fields may contain length expressed in indicated number of bits.

Comment: See also description of the `LENGTHTO` and `POINTERTO` attributes. The elements can be used with `LENGTHTO` only if the length field counts the number of elements in a `record`/`set` of field.

Examples:

```
//Example number 1): measuring length in 32 bit long units
type record Rec {
INT1 length,
octetstring field
}

with {
variant (length) "LENGTHTO(field)";
variant (length) "UNIT(dword32)"
}

//Example number 2): measuring length in 2 bit long units
type record Rec {
INT1 length,
octetstring field
}

with {
variant (length) "LENGTHTO(field)";
variant (length) "UNIT(2)"
}

//Example number 3): counting the number of elements of record of field
type record of BIT8 Bitrec
type record Rec{
integer length,
Bitrec data
}

with{
variant (length) "LENGTHTO(data)";
variant (length) "UNIT(elements)"
}
```

Attributes to Identify Fields in Structured Data Types

This section describes the coding attributes which during decoding identifies the fields within structured data types such as record, set or union.

PRESENCE

Attribute syntax: `PRESENCE(<parameter>)`

Parameters allowed: a `presence_indicator` expression (see Description)

Default value: none

Can be used with: `optional` fields of a `record` or `set`.

Description: Within records some fields may indicate the presence of another optional field. The

attribute PRESENCE is used to describe these cases. Each optional field can have a PRESENCE definition. The syntax of the PRESENCE attribute is the following: a PRESENCE definition is a presence_indicator expression. Presence_indicators are of form <key> = <constant> or {<key1> = <constant1>, <key2> = <constant2>, ⋯ <keyN> = <constantN>} where each key is a field(.nestedField) of the record, set or union and each constant is a TTCN-3 constant expression (for example, 22, '25␣␣O or '1001101␣␣B).

NOTE	The PRESENCE attribute can identify the presence of the whole record. In that case the field reference must be omitted.

Examples:

```
type record Rec {
BIT1 presence,
OCT3 field optional
}

with {
variant (field) "PRESENCE(presence = '1'B)"
}

type record R2{
OCT1 header,
OCT1 data
} with {variant "PRESENCE(header='11'O)"}
```

TAG

Attribute syntax: TAG(<parameter>)

Parameters allowed: list of field_identifications (see Description)

Default value: none

Can be used with: record or set.

Description: The purpose of the attribute TAG is to identify specific values in certain fields of the set, record elements or union choices. When the TAG is specified to a record or a set, the presence of the given element can be identified at decoding. When the TAG belongs to a union, the union member can be identified at decoding. The attribute is a list of field_identifications. Each field_identification consists of a record, set or union field name and a presence_indicator expression separated by a comma (,). Presence_indicators are of form <key> = <constant> or { <key1> = <constant1>, <key2> = <constant2>, ⋯ <keyN> = <constantN> } where each key is a field(.nestedField) of the record, set or union and each constant is a TTCN-3 constant expression (e.g. 22, '25␣␣O or '1001101␣␣B).There is a special presence_indicator: OTHERWISE. This indicates the default union member in a union when the TAG belongs to union.

NOTE	TAG works on non-optional fields of a record as well. It is recommended to use the attributes CROSSTAG or PRESENCE leading to more effective decoding.

Examples:

```
//Example number 1): set
type record InnerRec {
INT1 tag,
OCT3 field
}

with { variant "" }
type set SomeSet {
InnerRec field1 optional,
InnerRec field2 optional,
InnerRec field3 optional
}

with {
variant "TAG(field1, tag = 1;
field2, tag = 2;
field3, tag = 3)"
}

//Example number 2): union
type union SomeUnion {
InnerRec field1,
InnerRec field2,
InnerRec field3
}

with {
variant "TAG(field1, tag = 1;
field2, tag = 2;
field3, OTHERWISE)"
}

If neither tag=1 in field1 nor tag=2 in field2 are matched, field3 is selected.

//Example number 3): record
type record MyRec{
OCT1 header,
InnerRec field1 optional
}

with{
variant "TAG(field1, tag = 1)"
}

//field1 is present when in field1 tag equals 1.
```

CROSSTAG

Attribute syntax: CROSSTAG(<parameter>)

Parameters allowed: list of union "field_identifications" (see Description)

Default value: none

Can be used with: union fields of records.

Description: When one field of a record specifies the union member of another field of a record, CROSSTAG definition is used. The syntax of the CROSSTAG attribute is the following. Each union field can have a CROSSTAG definition. A CROSSTAG definition is a list of union field_identifications. Each field_identification consists of a union field name and a presence_indicator expression separated by a comma (,). Presence_indicators are of form <key> = <constant> or {<key1> = <constant1>, <key2> = <constant2>, ⋯ <keyN> = <constantN>} where each key is a field(.nestedField) of the record, set or union and each constant is a TTCN-3 constant expression (for example, 22, '25□0 or '1001101□B).There is a special presence_indicator: OTHERWISE. This indicates the default union member in union.

NOTE	The difference between the TAG and CROSSTAG concept is that in case of TAG the field identifier is inside the field to be identified. In case of CROSSTAG the field identifier can either precede or succeed the union field it refers to. If the field identifier succeeds the union, they must be in the same record, the union field must be mandatory and all of its embedded types must have the same fixed size.

Examples:

```
type union AnyPdu {
PduType1 type1,
PduType2 type2,
PduType3 type3
}

with { variant "" }
type record PduWithId {
INT1 protocolId,
AnyPdu pdu
}

with {
variant (pdu) "CROSSTAG( type1, { protocolId = 1,
protocolId = 11 };
type2, protocolId = 2;
type3, protocolId = 3)"
}
```

REPEATABLE

Attribute syntax: REPEATABLE(<parameter>)

Parameters allowed: `yes`, `no`

Default value: none

Can be used with: `record/set` of type fields of a `set`.

Description: The element of the set can be in any order. The `REPEATABLE` attribute controls whether the element of the `record` or `set of` can be mixed with other elements of the `set` or they are grouped together.

NOTE	It has no effect during encoding.

Examples:

```
// Three records and a set are defined as follows:

type record R1{
OCT1 header,
OCT1 data
} with {variant "PRESENCE(header='AA'O)"}

type record of R1 R1list;

type record R2{
OCT1 header,
OCT1 data
} with {variant "PRESENCE(header='11'O)"}

type record R3{
OCT1 header,
OCT1 data
} with {variant "PRESENCE(header='22'O)"}

type set S1 {
R2 field1,
R3 field2,
R1list field3
}

with {variant (field3) "REPEATABLE(yes)"}

//The following encoded values have equal meaning:
// (The value of R1 is indicated in bold.)
//example1: 1145**AA01AA02AA03**2267
//example2: 114**5AA01**2267**AA02AA03**
//example3: **AA01**2267**AA02**1145*AA03*

The decoded value of S1 type:

{
```

```
field1:={
header:='11'O,
data:='45'O
},

field2:={
header:='22'O,
data:='67'O
},

field3:={
{header:='AA'O,data:='01'O},
{header:='AA'O,data:='02'O},
{header:='AA'O,data:='03'O}
}
}

type set S2 {
R2 field1,
R3 field2,
R1list field3
}

with {variant (field3) "REPEATABLE(no)"}

//Only the example1 is a valid encoded value for S2, because
//the elements of the field3 must be groupped together.
```

FORCEOMIT

Attribute syntax: `FORCEOMIT(<parameter>)`

Parameters allowed: list of TTCN-3 field identifiers (can also be nested)

Default value: none

Can be used with: fields of a `record`/`set`.

Description: Forces the *lower-level* optional field(s) specified by the parameters to always be omitted. It does not affect the own fields of the defining `record`/`set`.

NOTE	It has no effect during encoding. It only affects the specified fields (which are probably in a different type definition) if they are decoded as part of the type this instruction is applied to.

Examples:

```
type record InnerRec {
  integer opt1 optional,
  integer opt2 optional,
```

```
  integer opt3 optional,
  integer mand
}

// Note: decoding a value of type InnerRec alone does not force any of the
// fields mentioned in the variants below to be omitted

type record OuterRec1 {
  integer f1,
  InnerRec f2,
  integer f3
}
with {
  variant (f2) "FORCEOMIT(opt1)"
}

// Decoding '0102030405'O into a value of type OuterRec1 results in:
// {
//   f1 := 1,
//   f2 := { opt1 := omit, opt2 := 2, opt3 := 3, mand := 4 },
//   f3 := 5
// }

type record OuterRec2 {
  OuterRec1 f
}
with {
  variant (f) "FORCEOMIT(f2.opt2)"
}

// Decoding '01020304'O into a value of type OuterRec2 results in:
// {
//   f := {
//     f1 := 1,
//     f2 := { opt1 := omit, opt2 := omit, opt3 := 2, mand := 3 },
//     f3 := 4
//   }
// }

type record OuterRec3 {
  OuterRec1 f1,
  OuterRec1 f2
}
with {
  variant (f1) "FORCEOMIT(f2.opt2, f2.opt3)"
  variant (f2) "FORCEOMIT(f2.opt2), FORCEOMIT(f2.opt3)"
}

// Decoding '010203040506'O into a value of type OuterRec3 results in:
// {
//   f1 := {
```

```
//     f1 := 1,
//     f2 := { opt1 := omit, opt2 := omit, opt3 := omit, mand := 2 },
//     f3 := 3
//   },
//   f2 := {
//     f1 := 4,
//     f2 := { opt1 := omit, opt2 := omit, opt3 := omit, mand := 5 },
//     f3 := 6
//   }
// }
```

Type-specific attributes

IntX

Attribute syntax: `IntX`

Default value: none

Can be used with: `integer` types

Description: Encodes an integer value as the IntX type in the ETSI Common Library (defined in ETSI TS 103 097).

This is a variable length encoding for integers. Its length depends on the encoded value (but is always a multiple of 8 bits).

The data starts with a series of ones followed by a zero. This represents the length of the encoded value: the number of ones is equal to the number of additional octets needed to encode the value besides those used (partially) to encode the length. The following bits contain the encoding of the integer value (as it would otherwise be encoded).

Comment: Since the length of the encoding is variable, attribute `FIELDLENGTH` is ignored. Furthermore, `IntX` also sets `BITORDER` and `BITORDERINFIELD` to `msb`, and `BYTEORDER` to first, overwriting any manual settings of these attributes.

Only attribute `COMP` can be used together with `IntX` (if it's set to `signbit`, then the sign bit will be the first bit after the length).

Restrictions: Using `IntX` in a `record` or `set` with `FIELDORDER` set to `lsb` is only supported if the `IntX` field starts at the beginning of a new octet. A compiler error is displayed otherwise. The `IntX` field may start anywhere if the parent `record`/`set`s `FIELDORDER` is set to `msb`.

Examples:

```
// Example 1: Standalone IntX integer type with no sign bit:
type integer IntX_unsigned with { variant "IntX" }

// Encoding integer 10:
// 00001010
// ^ length bit (there are no ones as no additional octets are needed)

// Encoding integer 2184:
// 10001000 10001000
// ^^ length bits (one extra octet is needed after the partial length octet)

// Example 2: Standalone IntX integer type with sign bit:
type integer IntX_signed with { variant "IntX, COMP(signbit)" }
// Encoding integer -2184:
// 10101000 10001000
// length bits ^^
// ^ sign bit

// Example 3: Standalone IntX integer type with 2's complement:
type integer IntX_compl with { variant "IntX, COMP(2scompl)" }
// Encoding integer -2184:
// 10110111 01111000
// ^^ length bits

// Example 4: IntX integer record field (starting in a partial octet):
type record RecIntXPartial {
integer i,
integer ix,
bitstring bs
}

with {
variant "FIELDORDER(msb)";
variant (i) "FIELDLENGTH(12), BITORDER(msb)";
variant (i) "BYTEORDER(first), BITORDERINFIELD(msb)";
variant (ix) "IntX";
variant (bs) "FIELDLENGTH(8)";
}

// Encoding record value { i := 716, ix := 716, bs := '10101010'B }:
// 00101100 11001000 00101100 11001010 10100000
// ^^^^^^^^ ^^^^ field `i' (same encoding as `ix', but with no length bits)
// field `ix' ^^^^ ^^^^^^^^ ^^^^ (the first 2 bits are the length bits)
// field `bs' ^^^^ ^^^^
// Note: setting the record's FIELDORDER to `lsb' in this case is not supported
// and would cause the mentioned compiler error.
```

Obsolete Attributes

This section describes the obsolete attributes. These attributes are kept for compatibility reason. The usage of the obsolete attributes is not recommended in new developments.

BITORDERINOCTET

The attribute has the same meaning and syntax as `BITORDER`. In new developments only the attribute `BITORDER` may be used.

TOPLEVEL BITORDER

Attribute syntax: `TOPLEVEL( BITORDER(<parameter>))`

Parameters allowed: `msb`, `lsb`

Default value: `msb`

Can be used with: a toplevel type.

Description: This attribute specifies the order of the bits within an octet. When set to `lsb`, the first bit sent will be the least significant bit of the original byte.

Comment:

Example:

```
type record WholePDU {
Field1 field1,
Field2 field2
}

with { variant "TOPLEVEL( BITORDER(lsb) )" }
const WholePDU c_pdu := {
'12'O,
'12'O
}

//Encoding of c_pdu will result in '4848'O.
```

4.23.5. TTCN-3 Types and Their Attributes

This section lists the TTCN-3 types along with the attributes allowed to be used with the types.

BITSTRING

Coding: The `bitstring` is represented by its binary value. The LSB of the binary form of a bitstring is concatenated to the LSB of the bitfield. If the length of the `bitstring` is shorter than the specified `FIELDLENGTH`, aligning is governed by the attribute `ALIGN`. The `FIELDLENGTH` default value for `bitstring` type is `variable`.

Attributes allowed:

- `ALIGN (0)`,
- `BITORDER (0)`,
- `BITORDERINFIELD (0)`,
- `BYTEORDER (0)`,
- `FIELDLENGTH (0)`,
- `N bit / unsigned N bit` (0).

Example:

```
*//Example number 1): variable length bitstring*
const bitstring c_mystring:='1011000101'B
//The resulting bitfield: 1011000101
//The encoding will have the following result:
// 11000101
// ·····10

*//Example number 2): fixed length bitstring*
type bitstring BIT7 with { variant "FIELDLENGTH(7)" }
const BIT7 c_ourstring:='0101'B
//The resulting bitfield: 0000101

*//Example number 3): left aligned bitstring*
type bitstring BIT7align with {
variant "FIELDLENGTH(7), ALIGN(left)" }
const BIT7align c_yourstring:='0101'B
//The resulting bitfield: 0101000
```

BOOLEAN

Coding: The `boolean` value `true` coded as '1'B,the `boolean` value `false` coded as '0'B.If `FIELDLENGTH` is specified, the given number of ones (`true`) or zeros (`false`) is coded. If the decoded bitfield is zero the decoded value will be false otherwise true. The default `FIELDLENGTH` for `boolean` type is 1.

Attributes allowed: `FIELDLENGTH (0)`, `N bit` (0).

Examples:

```
*//Example number 1): boolean coded with default length*
const boolean c_myvar:=true
//The resulting bitfield: 1
*//Example number 2): boolean coded with fixed length*
type boolean Mybool with { variant "FIELDLENGTH(8)"}
const Mybool c_ourvar:=true
//The resulting bitfield: 11111111
```

CHARSTRING

Coding: The characters are represented by their ASCII binary value. The first character is put into the first octet of the bitfield. The second character is put into the second octet of the bitfield and so on. Thus, the first character is put first into the buffer. If the actual value of `charstring` is shorter than the specified `FIELDLENGTH`, aligning is governed by the attribute `ALIGN`. The default `FIELDLENGTH` for bitstring type is variable. The `FIELDLENGTH` is measured in chars.

Attributes allowed:

- `ALIGN (0)`,
- `BITORDER (0)`,
- `BITORDERINFIELD (0)`,
- `BYTEORDER (0)`,
- `FIELDLENGTH (0)`,
- `N bit (0)`

Examples:

```
*//Example number 1): variable length charstring*
const charstring c_mystring:="Hello"
//The resulting bitfield: 01101111 01101100 01101100
// 01100101 01001000
//The encoding will have the following result:
// 01001000 "H"
// 01100101 "e"
// 01101100 "l"
// 01101100 "l"
// 01101111 "o"

*//Example number 2): fixed length charstring*
type charstring CHR6 with { variant "FIELDLENGTH(6)" }
const CHR6 c_yourstring:="Hello"
//The resulting bitfield: 00000000 01101111 01101100 01101100
// 01100101 01001000

//The encoding will have the following result:
// 01001000 "H"
// 01100101 "e"
// 01101100 "l"
// 01101100 "l"
// 01101111 "o"
// 00000000 " "

*//Example number 3): left aligned charstring*
type charstring CHR6align with {
variant "FIELDLENGTH(6), ALIGN(left)" }
const CHR6align c_string:="Hello"

//The resulting bitfield: 01101111 01101100 01101100 01100101
// 01001000 00000000
//The encoding will have the following result:
// 00000000 " "
// 01001000 "H"
// 01100101 "e"
// 01101100 "l"
// 01101100 "l"
// 01101111 "o"
```

ENUMERATED

Coding: The `enumerated` type is coded as an integer value. This numerical value is used during encoding. The default `FIELDLENGTH` for `enumerated` type is the minimum number of bits required to display the highest `enumerated` value.

Attributes allowed:

- `BITORDER (0)`,

- BITORDERINFIELD (0),
- BYTEORDER (0),
- COMP (0),
- FIELDLENGTH (0),
- N bit / unsigned N bit (0).

Example:

```
type enumerated Enumm {zero, one, two, three, four, five}

const Enumm myenum:=two

//The maximum enumerated value: 5 (five)
//Minimum 3 to represent 5.
//The FIELDLENGTH will be 3
//The resulting bitfield: 010

type enumerated Enum { zero(2), one(23), two(4), three(1), four(0), five(5) }
const Enum c_myenum:=two

//The maximum enumerated value: 23 (one)
//Minimum 5 bits are needed to represent 23.
//The FIELDLENGTH will be 5
//The resulting bitfield: 00010
```

FLOAT

Coding: The float value is represented according to IEEE 754 standard. The FORMAT attribute specifies the number of the bits used in exponent and mantissa. IEEE754 double: The float value is encoded as specified in IEEE754 standard using 1 sign bit, 11 exponent bits and 52 bits for mantissa. IEEE754 float: The float value is encoded as specified in IEEE754 standard using 1 sign bit, 8 exponent bits and 23 bits for mantissa. The default FORMAT for float is IEEE754 double.

Attributes allowed:

- BITORDER (0),
- BITORDERINFIELD (0),
- BYTEORDER (0),
- FORMAT (0)

Example:

```
//S - sign bit
//E - exponent bits
//M - mantissa bits
```

```
*//Example number 1): single precision float*
type float SingleFloat
with {
variant "FORMAT(IEEE754 float)"
}

const SingleFloat c_float:=1432432.125
//The resulting bitfield: 10000001 11011011 10101110 01001001
// MMMMMMMM MMMMMMMM EMMMMMMM SEEEEEEE

//The encoding will have the following result:
// 01001001 SEEEEEEE
// 10101110 EMMMMMMM
// 11011011 MMMMMMMM
// 10000001 MMMMMMMM

//The encoding will result in the octetstring '49AEDB81'O

*//Example number 2): double precision float*
type float DoubleFloat
with {
variant "FORMAT(IEEE754 double)"
}

const DoubleFloat c_floatd:=1432432.112232

//The resulting bitfield:
//82 3c bb 1c70 db 35 41
//10000010 00111100 10111011 00011100
//01110000 11011011 00110101 01000001
//MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM
//MMMMMMMM MMMMMMMM EEEEMMMM SEEEEEEE

//The encoding will have the following result:

// 01000001 SEEEEEEE
// 00110101 EEEEMMMM
// 11011011 MMMMMMMM
// 01110000 MMMMMMMM
// 00011100 MMMMMMMM
// 10111011 MMMMMMMM
// 00111100 MMMMMMMM
// 10000010 MMMMMMMM

//The encoding will result in the octetstring

// '4135DB701CBB3C82'O
```

HEXSTRING

Coding: The hexadecimal digit is represented by its binary value. The first hexadecimal digit is put

into the lower nibble of first octet of the bitfield. The second hexadecimal digit is put into the higher nibble of first octet of the bitfield. The 3rd hexadecimal digit is put into the lower nibble of second octet of bitfield and so on. Thus, the first hexadecimal digit is put first into the buffer. Is the actual length of hexstring shorter than the specified by `FIELDLENGTH`, aligning is governed by the attribute `ALIGN`. The default `FIELDLENGTH` value for `hexstring` type is `variable`. In this case, `FIELDLENGTH` is measured in hexdigits.

Attributes allowed:

- `ALIGN (0)`,
- `BITORDER (0)`,
- `BITORDERINFIELD (0)`,
- `BYTEORDER (0)`,
- `FIELDLENGTH (0)`,
- `N bit (0)`.

Example:

```
*//Example number 1): variable length hexstring*
const hexstring c_mystring:='5AF'H

//The resulting bitfield: 1111 10100101
//The encoding will have the following result:
// 10100101 A5
// ….1111 .F

*//Example number 2): fixed length hexstring*
type hexstring HEX4 with { variant "FIELDLENGTH(4)" }
const HEX4 c_yourstring:='5AF'H
//The resulting bitfield: 00001111 10100101
//The encoding will have the following result:
// 10100101 A5
// 00001111 0F

*//Example number 3): left aligned hexstring*
type hexstring HEX4align with {
variant "FIELDLENGTH(4), ALIGN(left)" }
const HEX4align c_ourstring:='5AF'H

//The resulting bitfield: 11111010 01010000
//The encoding will have the following result:
// 01010000 50
// 11111010 FA
```

INTEGER

Coding: The LSB of the binary form of an `integer` is concatenated to the LSB of the bitfield. The

value of the attribute `COMP` determines how the value of an `integer` type will be coded to binary form. The integer is always of fixed length and fills the space specified by `FIELDLENGTH`. The default value of `FIELDLENGTH` for integer is 8 bit. The `ALIGN` attribute has no meaning for `integer`.

Attributes allowed:

- `BITORDER (0)`,
- `BITORDERINFIELD (0)`,
- `BYTEORDER (0)`,
- `COMP (0)`,
- `FIELDLENGTH (0)`,
- `IntX (0)`,
- `N bit / unsigned N bit (0)`.

Example:

```
*//Example number 1)*

type integer Int12
with{ variant "FIELDLENGTH(12)"}
const Int12 c_myint:=1052

//The resulting bitfield is 010000011100
//The encoding will have the following result:
// 00011100
// ….0100

//The same result represented as octetstring: '1C.4'O

*//Example number 2)*

type integer Int12sg
with{ variant "FIELDLENGTH(12), COMP(signbit)"}
const Int12sg c_mysignedint:=-1052

//The resulting bitfield: 110000011100
//The encoding will have the following result:
// 00011100
// ….1100
//The same result represented as octetstring: '1C.C'O

*//Example number 3)*

type integer Int12c
with{ variant "FIELDLENGTH(12), COMP(2scompl)"}
const int12c c_hisint:=-1052
//The resulted bitfield: 101111100111
//The encoding will have the following result:
// 11100111
// ….1011
//The same result represented as octetstring: 'E7.B'O
```

OCTETSTRING

Coding: The octets are represented by their binary value. The first octet is put into first octet of bitfield. The second octet is put second octet of bitfield and so on. Thus, the first octet is put first into the buffer. If the length of the `octetstring` is shorter than the specified `FIELDLENGTH`, aligning is governed by the attribute `ALIGN`. The default `FIELDLENGTH` value for `octetstring` type is `variable`. In this case, `FIELDLENGTH` is measured in octets.

Attributes allowed:

- `ALIGN` (0),
- `BITORDER` (0),
- `BITORDERINFIELD` (0),

- BYTEORDER (0),
- FIELDLENGTH (0),
- N bit (0).

Example:

```
*//Example number 1): variable length octetstring*
const octetstring c_mystring:='25AF'O

//The resulting bitfield: 10101111 00100101
//The encoding will have the following result:
// 00100101 25
// 10101111 AF

*//Example number 2): fixed length octetstring*

type octetstring OCT3 with { variant "FIELDLENGTH(3)" }
const OCT3 c_yourstring:='25AF'H
//The resulting bitfield: 00000000 10101111 00100101
//The encoding will have the following result:
// 00100101 25
// 10101111 AF
// 00000000 00

*//Example number 3): left aligned octetstring*
type octetstring OCT3align with {
variant "FIELDLENGTH(3), ALIGN(left)" }
const OCT3align c_string:='25AF'H

//The resulting bitfield: 10101111 00100101 00000000
//The encoding will have the following result:
// 00000000 00
// 00100101 25
// 10101111 AF
```

SET

Encoding: During encoding the fields present are encoded one by one. If TAG is specified for one field, the value of the key field is checked for a valid value. If a valid value is not found, the value of the key field will be substituted with a valid key value.

Decoding: The fields can be received in any order. If TAG is specified, the value of the key field identifies the fields. If TAG is not specified for any field, the decoder tries to decode a field. If the decoding is successful, the decoder assumes that the field was received. The matching algorithm is the following: First try to identify the received fields by TAGs; if it fails, try to decode the fields; if it fails and OTHERWISE is specified in TAG, try that field; if it fails: unknown field is received. If all mandatory fields have already been decoded, then the set is successfully decoded, else the decoding of set has failed.

RECORD

Encoding: The fields present are encoded one by one. The value of length and pointer fields are calculated and substituted. If `TAG`, `CROSSTAG` or `PRESENCE` are specified for one field, the value of the key field is checked for a valid value. If a valid value is not found, the value of key field will be substituted with a valid key value. Finally, the extension bits are set.

Decoding: Fields are decoded one by one. The presence of optional fields is determined by the attributes `TAG`, `PRESENCE`, by extension bits and by the value of the length field. The chosen field of union is determined by `CROSSTAG`, if present. The value of pointer field is used to determine the beginning of the pointed field. Have all of the mandatory fields been received and successfully decoded, the decoding of the record is successful.

RECORD OF, SET OF

Encoding: The elements of `record` of or `set of` are encoded one by one. Finally, the extension bits are set, if needed.

Decoding: The items of `record` of or `set of` are decoded one by one. The number of items is determined by the attribute `FIELDLENGTH`, by extension bits or the number of available bits in buffer. The decoding of `record of` or `set of` is successful if at least one item has been decoded.

UNION

Encoding: The chosen field will be encoded according to its own encoding rules. If `TAG` is specified for this field, the value of the key field is checked for a valid value. If a valid value is not found, the value of the key field will be substituted with a valid key value.

Decoding: The decoder tries to identify the received union field. If `TAG` is present, the decoder uses the value of the key fields to identify the fields. If `TAG` is not present, the decoder tries to decode the fields and if it succeeds, the decoder assumes that field is received. If the decoding of field is not successful, the decoder checks the next field. The decoding of the union will be unsuccessful if none of the fields can be decoded.

Examples:

```
type record Rec{
OCT1 key,
OCT1 values
}

type union MyUnion{
Rec field1,
Rec field2,
Rec field3
} with { variant "TAG( field1,{key = '56'O, key='7A'}; field2, key = 'FF'; field3,{key
= 'A4'O, key = '99'O})"
}

*//Example number 1): successful encoding*
const MyUnion c_PDU:={
field1:={ key:='7A'O, values:='B2'O}
}

//Chosen field: field1
//Value of key field: '7A'O; valid
//No substitution will take place.
//The encoding will have the following result:
// 01111010 7A
// 10110010 B2

*//Example number 2): key field substituted*

const MyUnion c_PDU2:={
field1:={ key:='00'O, values:='B2'O}
}

//Chosen field: field1
//Value of key field: '00'O not valid
//The value of key field will be substituted with:'56'O
//The encoding will have the following result:
// 01010110 56
// 10110010 B2
```

UNIVERSAL CHARSTRING

Coding: The characters are first converted to UTF-8 format, and the resulting octets are encoded as if they were an `octetstring`. That is, the octets are represented by their binary value. The first octet is put into the first octet of the bit field. The second octet is put into the second octet of the bit field, and so on. Thus, the first octet is put first into the buffer.

The RAW encoding of a `universal charstring` value with no non-ASCII characters is equal to the RAW encoding of a `charstring` containing the same characters (with the same attributes).

If the length of the UTF-8 encoded `universal charstring` is shorter than the specified `FIELDLENGTH`, aligning is governed by the attribute `ALIGN`. The default `FIELDLENGTH` for the `universal charstring`

type is `variable`. The `FIELDLENGTH` is measured in UTF-8 octets.

Attributes allowed:

- `ALIGN` (0),
- `BITORDER` (0),
- `BITORDERINFIELD` (0),
- `BYTEORDER` (0),
- `FIELDLENGTH` (0),
- `N bit` (0).

Examples:

```
*//Example number 1): variable length universal charstring*

const universal charstring c_mystring := "sepr" & char(0, 0, 1, 113);

//The encoding will have the following result:
// 01110011 "s"
// 01100101 "e"
// 01110000 "p"
// 01110010 "r"
// 11000101 C5
// 10110001 B1 C5B1 is the UTF-8 encoding of char(0, 0, 1, 113)

*//Example number 2): fixed length universal charstring*
type universal charstring USTR8 with { variant "FIELDLENGTH(8)" }
const USTR8 c_yourstring := "sepr" & char(0, 0, 1, 113);

//The encoding will have the following result:
// 01110011 "s"
// 01100101 "e"
// 01110000 "p"
// 01110010 "r"
// 11000101 C5
// 10110001 B1 C5B1 is the UTF-8 encoding of char(0, 0, 1, 113)
// 00000000 " "
// 00000000 " "

*//Example number 3): left aligned universal charstring*
type universal charstring USTR8align with {
variant "FIELDLENGTH(8), ALIGN(left)" }
const USTR8align c_string := "sepr" & char(0, 0, 1, 113);
//The encoding will have the following result:
// 00000000 " "
// 00000000 " "
// 01110011 "s"
// 01100101 "e"
// 01110000 "p"
// 01110010 "r"
// 11000101 C5
// 10110001 B1 C5B1 is the UTF-8 encoding of char(0, 0, 1, 113)
```

4.24. TEXT Encoder and Decoder

The TEXT encoder and decoder are general purpose functionalities developed originally for handling verbose and tokenized protocols.

The encoder converts abstract TTCN-3 structures (or types) into a bitstream suitable for serial transmission. The decoder, on the contrary, converts the received bitstream into values of abstract TTCN-3 structures.

TITAN provides a special encoding scheme for coding elements into a textual representation. This is called TEXT and is used like `encoding "TEXT"`.

This section covers the attributes controlling the coding process and BNF specification of the attributes.

Error encountered during the encoding or decoding process are handled as defined in section "Setting error behavior" in [16].

4.24.1. Attributes

An `attribute` determines coding and encoding rules.

NOTE	the section 27.5 of the TTCN-3 standard ([1]) states that an `attribute` is used to refine the current encoding scheme defined with the keyword `encode`. Because of backward compatibility the presence of the `encode` attribute is not required, but might result in a compile time warning (which in the future might turn into an error).

BEGIN

Role: The `BEGIN` attribute defines the leading token of the type.

Format: `BEGIN(token_to_encode, <matching_exp>,<modifier>)`

Description: The attribute defines the leading token of the type. This token defines the beginning of the value of the type and will be written into the encoded string before the value of the type is encoded. `BEGIN` can be used with any type.

Parameters: `token_to_encode`
The token is used during encoding.

`Mandatory.matching_exp`
This TTCN-3 character pattern is used during decoding to identify the leading token of the type. The format of the parameter is described in clause B.1.5 of the TTCN-3 standard ([1]). This parameter is optional; when omitted, the parameter token_to_encode will be used as the matching pattern.

`modifier`
Modifies the behavior of the matching algorithm. Optional parameter. When omitted the default value will be used. The available modifiers:
* `case_sensitive` The matching is case sensitive. Default value.
* `case_insensitive` The matching is case insensitive.

Example:

```
//SIP header Subject header:

type record Subject{
charstring subject_value
}

with { variant "BEGIN('Subject: ','
(Subject[ ]#(,):[ ]#(,))|"
"(s[ ]#(,):[ ]#(,))',
case_insensitive)"
}

var Subject v_subj:= "the_subject";
//The encoded string will be: "Subject: the subject"
//The decoder will accept the long (Subject: the subject)
//and the short (s: the subject) format regardless
//of the case of the character of the header.
```

END

Role: The `END` attribute defines the closing token of the type.

Format: `END(token_to_encode, <matching exp>,<modifier>)`

Description: The attribute defines the closing token of the type. This token defines the end of the value of the type and will be written into the encoded string after the encoded value of the type. `END` can be used with any type.

Parameters: `token_to_encode`
The token used during encoding. Mandatory.

`matching_exp`
This TTCN-3 character pattern is used during decoding to identify the leading token of the type. The format of the parameter is described in clause B.1.5 of the TTCN-3 standard ([1]). This parameter is optional; when omitted, the `token_to_encode` will be used as matching pattern.

`modifier`
Modifies the behavior of the matching algorithm. Optional parameter. When omitted, the default value will be used. The available modifiers:
* `case_sensitive`: The matching is case sensitive. Default value.
* `case_insensitive`: The matching is case insensitive.

Example:

```
//SIP header Subject header:

type record Subject{
charstring subject_value
}

with { variant "BEGIN('Subject: ','
(Subject[ ]#(,):[ ]#(,))|"
"(s[ ]#(,):[ ]#(,))',
case_insensitive)";
variant "END('','([])|([])')"
}

var Subject v_subj:= "the_subject";
//The encoded string will be: "Subject: the_subject"
//The decoder will accept both "Subject: the_subject" and //"Subject: the_subject"
format.
```

SEPARATOR

Role: The attribute `SEPARATOR` defines the field separator token of the type.

Format: `SEPARATOR(token to encode, <matching exp>,<modifier>)`

Description: The attribute defines the field separator token of the type. This token separates the value of fields and will be written into the encoded string between the fields of the type. `SEPARATOR` can be used with any type.

Parameters: `token_to_encode`
The token used during encoding. Mandatory.

`matching_exp`
This TTCN-3 character pattern is used during decoding to identify the leading token of the type. The format of the parameter is described in clause B.1.5 of the TTCN-3 standard ([1]). Optional parameter. When omitted, the token to encode will be used as matching pattern.

`modifier`
Modifies the behavior of the matching algorithm. Optional parameter. When omitted, the default value will be used. The available modifiers:
* `case_sensitive` The matching is case sensitive. Default value.
* `case_insensitive` The matching is case insensitive.

Example:

```
type record Rec_1{
charstring field_1,
charstring field_2
}

with {
variant "BEGIN('Header: ')"
variant "SEPARATOR(';')"
}

var Rec_1 v_rec:={field1:="value_field1",
field2:="value_field2"}
//The encoded will result in the string:
//"Header: value_field1; value_field2"
```

TEXT_CODING

Role: The attribute TEXT_CODING defines the encoding and decoding rules of the value

Format: `TEXT_CODING(encoding_rule,<decoding_rule>,<matching_exp>,<modifier>)`

Description: The attribute controls the encoding and the decoding of the values.

Parameters: `encoding_rule`
Controls the encoding of the value. For syntax see the two tables below.

`decoding_rule`
Controls the decoding of the value. For syntax see the two tables below.

`matching_exp`
TTCN-3 character pattern, used during decoding to identify the value of the type. The format of the parameter is described in clause B.1.5 of the TTCN-3 standard ([1]). Optional parameter.

`modifier`
Modifies the behavior of the matching algorithm. Optional parameter. When omitted, the default value will be used. The available modifiers:
* `case_sensitive` The matching is case sensitive. Default value.
* `case_insensitive` The matching is case insensitive.

Table 7. Format of `encoding_rule` *and* `decoding_rule`

Type	encoding_rule	decoding_rule
`charstring`	The format of encoding_rule: `attr=value[;attr=value]` Usable attributes: `length`, `convert`, `just`	The format of decoding_rule: `attr=value[;attr=value]` Usable attributes: `length`, `convert`

Type	encoding_rule	decoding_rule
integer	The format of the encoding rule: attr=value[;attr=value] Usable attributes: length, leading0	The format of the decoding rule: attr=value[;attr=value] Usable attribute: length
boolean	The encoded value of true and false value: true:'token'[;false:'token'] The default encoded value of true is 'true'; the default encoded value of false is 'false'	The matching pattern of the value true and false: true:{'pattern'[,modifier]}[;false:{'pattern'[,modifier]}] The default decoding method is case sensitive
enumerated	The encoded value of enumeration: value:'token'[;value:'token'] The default encoded value of enumerations is the TTCN-3 identifier of the enumerations.	The matching pattern of enumerations: value:{'pattern'[,modifier]}[;value:{'pattern'[,modifier]}] The default decoding method is case sensitive.
set ofrecord of	Not applicable	The format of the decoding rule: attr=value[;attr=value] Usable attribute: repeatable
structured types	Not applicable	Not applicable

Table 8. Attributes used with encoding_rule and decoding_rule

attr	Description	Parameter	Default value
length	Determines the length of the coded value.	value	number of charactersof value
convert	Converts string values to lower or upper case during encoding or decoding.	lower_case, upper_case	no conversion
just	If the string is shorter than the value defined by the length attribute, just controls the justification of the value.	left, right, center	left
leading0	Controls the presence of the leading zeros of the coded integer value.	true, false	false

attr	Description	Parameter	Default value
`repeatable`	The attribute repeatable controls whether the element of the record of or set of can be mixed with other elements of the set or they are grouped together.	`true`, `false`	`false`

Example:

```
*//Example number 1): integer with leading zero*
type integer My_int with {
variant "TEXT_CODING(length=5;leading0=true)"
}

var My_int v_a:=4;
// The encoded value: '00004'
*//Example number 2): integer without leading zero*
type integer My_int2 with {
variant "TEXT_CODING(length=5)"
}

var My_int2 v_aa:=4;
// The encoded value: ' 4'
*//Example number 3): character string*
type charstring My_char with {
variant "TEXT_CODING(length=5)"
}

var My_char v_aaa:='str';
// The encoded value: ' str'
*//Example number 4): centered character string*

type charstring My_char2 with {
variant "TEXT_CODING(length=5;just=center)"
}

var My_char2 v_aaaa:='str';
// The encoded value: ' str '
*//Example number 5): character string converted to upper case*
type charstring My_char3 with {
variant "TEXT_CODING(length=5;convert=upper_case)"
}

var my_char3 v_b:='str';

// The encoded value: ' STR'
```

```
*//Example number 6): case converted character string*

type charstring My_char4 with {
variant "TEXT_CODING(convert=upper_case,convert=lower_case)"
}

var My_char4 v_bb:='str';
// The encoded value: 'STR'
// The decoded value: 'str'
*//Example number 6): boolean*
type boolean My_bool with {
variant "TEXT_CODING(true:'good';false:'bad')"
}

var my_bool v_bbb=false;
// The encoded value: 'bad'
```

4.24.2. BNF of the Attributes

```
COMMA = ","

SEMI = ";"

token = any valid character literal, "'" must be escaped

pattern = valid TTCN-3 character pattern, the reference is not supported

number = positive integer number

enumerated = the name of the enumerated value

attributes = attribute *(COMMA attribute)

attribute = begin-attr / end-attr / separator-attr / coding-attr

begin-attr = "BEGIN(" encode-token [ COMMA [ match-expr ] [COMMA modifier] ] ")"

end-attr = "END(" encode-token [ COMMA [ match-expr ] [COMMA modifier] ] ")"

separator-attr = "SEPARATOR(" encode-token [ COMMA [ match-expr ] [COMMA modifier] ]
")"

coding-attr = "TEXT_CODING(" [ [encoding-rules] [COMMA [decoding-rules] [ COMMA match-
expr [COMMA modifier] ] ] ] ")"

encode-token = "'" token "'"

match-expr = "'" pattern "'"

modifier = "case_sensitive" / "case_insensitive"
```

```
encoding-rules = encoding-rule *(SEMI encoding-rule)

encoding-rule = attr-def / enc-enum / enc-bool

decoding-rules = decoding-rule *(SEMI decoding-rule)

decoding-rule = attr-def / dec-enum / dec-bool

attr-def = ("length=" number )/ ("convert=" ("lower_case" / "upper_case") )/ ("just="
("left"/"right"/"center") )/ ("leading0=" ("true"/"false") )/ ("repeatable="
("true"/"false") )

enc-enu = enumerated ":" encode-token

enc-bool = ("true:" encode-token) / ("true:" encode-token)

dec-enum = enumerated ":" "{" [match-expr] [COMMA modifier] "}"

dec-bool = (true ":" "{" [match-expr] [COMMA modifier] "}")/(false ":" "{" [match-
expr] [COMMA modifier] "}")
```

4.25. XML Encoder and Decoder

The XML encoder and decoder are handling XML-based protocols. The encoder converts abstract TTCN-3 structures (or types) into an XML representation. The decoder converts the XML data into values of abstract TTCN-3 structures.

4.25.1. General Rules and Restrictions

The TTCN-3 standard defines a mechanism using attributes to define encoding variants. The attributes concerning the XML encoding are standardized in [4] (annex B of the standard lists the attributes and their effects).

Faults in the XML encoding/decoding process are set to error by default, but it can be modified with the `errorbehavior` TTCN-3 attribute. (Codec error handling)

4.25.2. Attributes

The following sections describe the TTCN-3 attributes that influence the XML coding.

Abstract

Attribute syntax: abstract

Applicable to (TTCN-3) Fields of unions

Description This attribute shall be generated for each field with the XSD attribute "abstract' set to true (usually during type substitution or element substitution). It can be used to distinguish XML messages with valid type or element substitutions from XML documents containing invalid

substitutions.

If the decoder finds an XML element or `xsi:type` attribute corresponding to an abstract union field, a coding error is displayed. The attribute has no effect on encoding.

Any element

Attribute syntax:

```
anyElement [ except ( 'freetext' | unqualified ) | from [unqualified ,] [ { 'freetext'
, } 'freetext' ] ]
```

Applicable to (TTCN-3) Fields of structured types generated for the XSD *any* element

Description One TTCN-3 attribute shall be generated for each field corresponding to an XSD any element. The freetext part(s) shall contain the URI(s) identified by the namespace attribute of the XSD any element. The namespace attribute may also contain wildcard. They shall be mapped as given in the following table:

Table 9. XSD namespace attributes

Value of the XSDnamespace attribute	**Except or from clause in the TTCN-3 attribute**	**Remark**
##any	*<nor except neither from clause present>*	
##local	from unqualified	
##other	except '*<target namespace of the ancestor schema element of the given any element>*'	Also disallows unqualified elements, i.e. elements without a target namespace
##other	except unqualified	In the case no target namespace is ancestor schema element of the given any element
##targetNamespace	from '*<target namespace of the ancestor schema element of the given any element >*'	
"http://www.w3.org/1999/xhtml ##targetNamespace"	from `http://www.w3.org/1999/xhtml' , '*<target namespace of the ancestor schema element of the given any element >*'	

The abstract value of the field will be encoded as an XML element in place of an XML element that would be generated for the field (ignoring the name of the field). During decoding, the abstract value of the field will contain the entire XML element.

Example:

```
type record AEProduct {
  charstring name,
  integer    price,
  universal charstring info
}
with {
  variant (info) "anyElement from 'http://www.example.com/A', "
  "'http://www.example.com/B', unqualified"
}
const AEProduct aep := {
  name  := "Trousers",
  price := 20,
  info  := "<xyz:color xmlns:xyz=""http://www.example.com/A""
available=""true"">red</xyz:color>"
}

/* XML encoding:
<AEProduct>
 <name>Trousers</name>
 <price>20</price>
 <xyz:color xmlns:xyz="http://www.example.com/A" available="true">red</xyz:color>
</AEProduct>
*/
```

Any attributes

Attribute syntax:

```
anyAttributes[ except 'freetext' | from [unqualified ,] { 'freetext', } 'freetext']
```

Applicable to (TTCN-3) Fields of structured types generated for the XSD *anyAttribute* element

Description This TTCN-3 attribute can be applied to a field which is of type `record of universal charstring`. Each component shall contain a valid XML attribute (name/value pair), optionally preceded by a namespace identifier (URI). The encoder shall remove the URI and insert it as a namespace declaration into the enclosing XML element, followed by the content of the `universal charstring` as an XML attribute. The decoder should recover each attribute into a component of the `record of`, preceded by its namespace URI if applicable. The mapping of namespaces behaves in the same way as the anyElement TTCN-3 attribute.

Example:

```
type record of universal charstring AttrList;
type record AAProduct {
  AttrList   info,
  charstring name,
  integer    price
}
with {
  variant (info) "anyAttributes from 'http://www.example.com/A', "
  "'http://www.example.com/B', unqualified"
}

const AAProduct aap := {
  info := {
    "http://www.example.com/A size=""small""",
    "http://www.example.com/B color=""red""",
    "available=""yes"""},
  name := "Trousers",
  price:= 20
}

/* XML encoding:
<AAProduct
 xmlns:b0="http://www.example.com/A" b0:size="small"
 xmlns:b1="http://www.example.com/B" b1:color="red" available="yes">
 <name>Trousers</name>
 <price>20</price>
</AAProduct>
*/
```

Attribute

Attribute syntax: attribute

Applicable to (TTCN-3) Top-level type definitions and fields of structured types generated for XSD *attribute* elements.

Description This encoding instruction causes the instances of the TTCN3 type or field to be encoded and decoded as attributes.

Comment Titan currently requires during decoding that attributes are present in the same order as they are declared in the enclosing record/set.

Example

```
type charstring Color
with {
  variant "attribute"
}
type record Product {
  Color      color,
  charstring material,
  charstring name,
  integer    price
}
with {
  variant (available) "attribute"
}

const Product shoes := {
  color := "blue",
  material := "suede",
  name := "Shoes",
  price:= 25
}
/* XML encoding
<Product color="blue" material="suede">
 <name>Shoes</name>
 <price>25</price>
</Product>
*/
```

AttributeFormQualified

Attribute syntax: `attributeFormQualified`

Applicable to (TTCN-3) Modules

Description This encoding instruction cause names of XML attributes that are instances of TTCN-3 definitions in the given module to be encoded as qualified names. At decoding time qualified names are expected as valid attribute names.

Control namespace identification

Attribute syntax: `controlNamespace` '*freetext*' `prefix` '*freetext*'

Applicable to (TTCN-3) Module

Description The control namespace is the namespace to be used for the type identification attributes and schema instances (e.g. in the special XML attribute value "xsi:nil". It shall be specified globally, with an encoding instruction attached to the TTCN-3 module.The first *freetext* component identifies the namespace (normally `http://www.w3.org/2001/XMLSchema-instance' is used), the second *freetext* component identifies the namespace prefix (normally `xsi' is used).

Please see the example for nillable elements, for example usage of `controlNamespace`.

Block

Attribute syntax: block

Applicable to (TTCN-3) Fields of unions

Description This attribute shall be generated for each field referred to by XSD `block` attributes (usually during type substitution or element substitution). It can be used to distinguish XML messages with valid type or element substitutions from XML documents containing invalid substitutions.

If the decoder finds an XML element or `xsi:type` attribute corresponding to a blocked union field, a coding error is displayed. The attribute has no effect on encoding.

Default for empty

Attribute syntax: defaultForEmpty as '*freetext*'

Applicable to (TTCN-3) TTCN-3 components generated for XSD *attribute* or *element* elements with a *fixed* or *default* attribute.

Description The '*freetext*' component shall designate a valid value of the type to which the encoding instruction is attached to. This encoding instruction has no effect on the encoding process and causes the decoder to insert the value specified by *freetext* if the corresponding attribute or element is omitted in the received XML document.

Example

```
type record DFEProduct {
charstring color,
charstring name,
float price,
charstring currency
}

with {
variant (color) "attribute";
variant (currency) "defaultForEmpty as `US Dollars"'";
}

const DFEProduct rval := {
color := "red",
name := "shirt",
price := 12.33,
currency := "US Dollars"
}

/* The following XML fragment will be decoded to the value of rval:

<DFEProduct color="red">
<name>shirt</name>
<price>12.33</price>
<currency/>
</DFEProduct>

*/
```

NOTE	TITAN allows the usage of constants and module parameters instead of the text value of the encoding instruction. The type of the field must be compatible with the type of the constant or module parameter. The form where constants and module parameters are allowed looks like this:

```
variant "defaultForEmpty as reference";
```

where reference is a constant or a module parameter. (Notice the missing apostrophe).

For example:

```
const integer c_int := 3;const charstring c_str := "abc";

type record MyRecord {
  integer i,
  charstring cs,
  float f
  }
  with {
    variant (i) "defaultForEmpty as c_int"; // allowed
    variant (cs) "defaultForEmpty as c_str"; // allowed
    variant (f) "defaultForEmpty as c_str"; // not allowed
    // incompatible types
  }
```

Element

Attribute syntax: element

Applicable to (TTCN-3): Top-level type definitions generated for XSD *element* elements that are direct children of a *schema* element.

Description: This encoding instruction causes the instances of the TTCN3 type to be encoded and decoded as XML elements.

Comment: This is the default behaviour. TTCN-3 types are encoded as elements unless altered by an encoding instruction. This encoding instruction can be used to cancel that effect.

ElementFormQualified

Attribute syntax: elementFormQualified

Applicable to (TTCN-3): Modules

Description: This encoding instruction causes tags of XML local elements and templates of XSD definitions in the given module to be encoded as qualified names, and inserts the namespace specification in the encoded XML. Tags of XML global elements are always encoded as qualified names, regardless of elementFormQualified. At decoding time only qualified names are accepted as valid element tag names.

Embed values

Attribute syntax: embedValues

Applicable to (TTCN-3): TTCN-3 record types generated for XSD *complexType*-s and *complexContent*-s with the value of the *mixed* attribute "true".

Description: The encoder shall encode the record type to which this attribute is applied in a way that produces the same result as the following procedure: first a partial encoding of the record is produced, ignoring the `embed_values` field. The first string of the `embed_values` field (the first record of element) shall be inserted at the beginning of the partial encoding, before the start-tag of the first

XML element (if any). Each subsequent string shall be inserted between the end-tag of the XML element and the start-tag of the next XML element (if any), until all strings are inserted. In the case the maximum allowed number of strings is present in the TTCN-3 value (the number of the XML elements in the partial encoding plus one) the last string will be inserted after end-tag of the last element (to the very end of the partial encoding). The following special cases apply:

1. At decoding, strings before, in-between and following the XML elements shall be collected as individual components of the `embed_values` field.If no XML elements are present, and there is also a defaultForEmptyencoding instruction on the sequence type, and the encoding is empty, a decoder shall interpret it as an encoding for the *freetext* part specified in the defaultForEmptyencoding instruction and assign this abstract value to the first (and only) component of the embed_values field.
2. If the type also has the useNilencoding instruction and the optional component is absent, then the embedValues encoding instruction has no effect.
3. If the type has a useNilencoding instruction and if a decoder determines, by the absence of a nil identification attribute (or its presence with the value false) that the optionalcomponent is present, then item a) above shall apply.

NOTE	Titan currently does not decode the values of the embed_values member. They will appear as empty strings.

Example

```
type record EmbProduct {
record of universal charstring embed_values,
universal charstring companyName,
universal charstring productColor,
universal charstring productName
}

with {
variant "embedValues"
}

const EmbProduct rval := {
embed_values := {"My Company", "produces", "", "which is very popular"},
ompanyName := "ABC",
productColor := "red",
productName := "shirt"
}

/* XML encoding

<EmbProduct>My
Company<companyName>ABC</companyName>produces<productColor>red</productColor>
<productName>shirt</productName>which is very popular</EmbProduct>

*/
```

Form

Attribute syntax: form as (qualified | unqualified)

Applicable to (TTCN-3): Top-level type definitions generated for XSD *attribute* elements and fields of structured type definitions generated for XSD *attribute* or *element* elements.

Description: This encoding instruction designates that names of XML attributes or tags of XML local elements corresponding to instances of the TTCN-3 type or field of type to which the form encoding instruction is attached, shall be encoded as qualified or unqualified names respectively and at decoding qualified or unqualified names shall be expected respectively as valid attribute names or element tags.

List

Attribute syntax: list

Applicable to (TTCN-3): Record-of types mapped from XSD *simpleType*-s derived as a list type.

Description: This encoding instruction designates that the record of type shall be handled as an XSD list type, namely, record of elements of instances shall be combined into a single XML list value using a single SP(32) (space) character as separator between the list elements. At decoding the XML list value shall be mapped to a TTCN-3 record of value by separating the list into its itemType elements (the whitespaces between the itemType elements shall not be part of the TTCN-3 value).

Example

```
type record of integer Pi;
with {
variant "list"
}

const Pi digits := {
3, 14, 15, 9, 26
}

/* XML encoding
<S>3 14 15 9 26</S>
*/
```

Name

Attribute syntax:

```
name (as ("freetext" | changeCase ) | all as changeCase ), wherechangeCase := (
capitalized | uncapitalized | lowercased | uppercased )
```

Applicable to (TTCN-3): Type or field of structured type. The form when *freetext* is empty shall be applied to fields of union types with the "useUnion" encoding instruction only

Description: The name encoding instruction is used when the name of the XML element or attribute differs from the name of the TTCN3 definition or field. The name resulted from applying the name encoding attribute shall be used as the non-qualified part of the name of the corresponding XML attribute or element tag.

When the "name as `*freetext*"' form is used, *freetext* shall be used as the attribute name or element tag, instead of the name of the related TTCN-3 definition (e.g. TTCN-3 type name or field name).

The "name as "" (i.e. freetext is empty) form designates that the TTCN-3 field corresponds to an XSD unnamed type, thus its name shall not be used when encoding and decoding XML documents.

The "name as capitalized" and "name as uncapitalized" forms identify that only the first character of the related TTCN3 type or field name shall be changed to lower case or upper case respectively.

The "name as lowercased" and "name as uppercased" forms identify that each character of the related TTCN3 type or field name shall be changed to lower case or upper case respectively.

The "name all as capitalized", "name all as uncapitalized", "name as lowercased" and "name as uppercased" forms has effect on all direct fields of the TTCN-3 definition to which the encoding instruction is applied (e.g. in case of a structured type definition to the names of its fields in a non-recursive way but not to the name of the definition itself and not to the name of fields embedded to other fields).

Example

```
type record S {
charstring r,
charstring blue,
charstring black
}

with {
variant (r) "name as `Red"';
variant (blue) "name as uppercased";
variant (black) "name as capitalized";
}

const NM outfit := { r := "shirt", blue := "trousers", black := "shoes" }

/* XML encoding

<S>

<Red>shirt</Red>
<BLUE>trousers</BLUE>
<Black>shoes</Black>
</S>

*/
```

Namespace identification

Attribute syntax: namespace as '*freetext*' [prefix "freetext"]

Applicable to (TTCN-3): Modules; fields of record types generated for *attribute_s of _complexTypes* taken in to *complexType* definitions by referencing *attributeGroup*(s), defined in *schema* elements with a different (but not absent) target namespace and imported into the *schema* element which is the ancestor of the *complexType.*

Description: The first *freetext* component identifies the namespace to be used in qualified XML attribute names and element tags at encoding, and to be expected in received XML documents. The second *freetext* component is optional and identifies the namespace prefix to be used at XML encoding. If the prefix is not specified, the encoder shall either identify the namespace as the default namespace (if all other namespaces involved in encoding the XML document have prefixes) or shall allocate a prefix to the namespace (if more than one namespace encoding instructions are missing the prefix part).

Example

```
type record S {
charstring firstName,
charstring lastName,
charstring middleInitial
}

with { variant "namespace as `http://www.example.org/test' prefix `tst"' }
const S jrh := { "John", "Doe", "M" }

/* XML encoding

<tst:S xmlns:tst="http://www.example.org/test">
<firstName>John</firstName>
<lastName>Doe</lastName>
<middleInitial>M</middleInitial>
</tst:S>

*/
```

Nillable elements

Attribute syntax: useNil

Applicable to (TTCN-3): Top-level record types or record fields generated for nillable XSD *element* elements.

Description: The encoding instruction designates that the encoder, when the optional field of the record (corresponding to the nillable element) is omitted, shall produce the XML element with the xsi:nil="true" attribute and no value. When the nillable XML element is present in the received XML document and carries the xsi:nil="true" attribute, the optional field of the record in the corresponding TTCN-3 value shall be omitted. If the nillable XML element carries the xsi:nil="true"

attribute and has children (either any character or element information item) at the same time, the decoder shall initiate a test case error.

Example

```
module UseNil {
type record Size {
  integer sizeval optional
}
with { variant "useNil" }

type record NilProduct {
  charstring name,
  ProductColor color,
  Size size
}

const NilProduct absent := {
  name  := "no shirt",
  color := red,
  size  := { omit }
}

const NilProduct present := {
  name  := "shirt",
  color := red,
  size  := { 10 }
}

}
with {
  encode "XML";
  variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'"
}
/* XML encoding of absent:
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<name>no shirt</name>
 <color>red</color>
 <size xsi:nil="true"/>
</Product>

XML encoding of present:
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<name>shirt</name>
 <color>red</color>
 <size xsi:nil="false">10</size>
</Product>

Another possible XML encoding of present:
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<name>shirt</name>
 <color>red</color>
 <size>10</size>
</Product>
*/
```

Text

Attribute syntax:

```
text ("name" as ("freetext"|) | all as changeCase)
```

where `changeCase` has been defined as seen here.

Applicable to (TTCN-3) Enumeration types generated for XSD enumeration facets where the enumeration base is a string type, and the name(s) of one or more TTCN-3 enumeration values are different from the related XSD enumeration item. Also applies to XSD.Boolean types, instances of XSD.Boolean types.

Description When *name* is used, it shall be generated for the differing enumerated values only. The *name* shall be the identifier of the TTCN-3 enumerated value the given instruction relates to. If the difference is that the first character of the XSD enumeration item value is a capital letter while the identifier of the related TTCN-3 enumeration value starts with a small letter, the "text `*name*' as capitalized" form shall be used. Otherwise, *freetext* shall contain the value of the related XSD enumeration item. If the first characters of all XSD enumeration items are capital letters, while the names of all related TTCN-3 enumeration values are identical to them except the case of their first characters, the "text all as capitalized" form shall be used. The encoding instruction designates that the encoder shall use *freetext* or the capitalized name(s) when encoding the TTCN-3 enumeration value(s) and vice versa. When the text encoding attribute is used with XSD.Boolean types, the decoder shall accept all four possible XSD boolean values and map the received value 1 to the TTCN-3 boolean value **true** and the received value 0 to the TTCN-3 boolean value **false**. When the text encoding attribute is used on the instances of the XSD.Boolean type, the encoder shall encode the TTCN3 values according to the encoding attribute (i.e. **true** as 1 and **false** as 0).

Comment For XSD.Boolean types, either of the forms "text 'true' as "1" and "text 'false' as '0' implies the other, i.e. Titan considers that both have been specified. Together, these two forms have the same effect as "text" (detailed in the last paragraph of Description).

Example

```
type enumerated ProductColor { red(0), light_green(1), blue(2) }
with {
variant "text `red' as uppercased";
variant "text `light_green' as `Light Green"'
variant "text `blue' as capitalized"
};

type boolean Availability
with {
variant "text"
}

type record T {
ProductColor color,
Availability available
}

const T prod := {
color := light_green,
available := true
}

/* XML encoding

<S>
<color>Light Green</color>
<available>1</available>
</S>

*/
```

Untagged

Attribute syntax: untagged

Applicable to (TCN-3): Type; or field of a record, set, union; or the embedded type of a record-of or set-of. This encoding instruction is ignored if applied to a type which is encoded as the top-level type, unless the top-level type is a union or anytype. It will take effect when a field of this type is encoded as a component of the enclosing structured type.

Description: The encoding instruction designates that the encoder shall omit the tag.

Example: "untagged" applied to a field.

```
*type* *record* Prod {
*charstring* name,
*float* price,
*charstring* description
}

*with* {
*variant* (description) "untagged"
}

*const* Prod prod := {
name := "Danish Blue",
price := 3.49,
description := "Soft blue cheese"
}

/* generated XML:
<Prod>
<name>Danish Blue</name>
<price>3.490000</price>
Soft blue cheese</Prod>
*/

Example: "untagged" applied to a union type.
*type* *union* ProdUnion {
*Prod* prod1,
*OtherProd* prod2
}

*with* {
*variant* "untagged"
}*const* ProdUnion produnion := { prod1 := {
name := "ProdName",
price := 66,
description := "The best product" }
}

/* generated XML:
<Prod>
<name>ProdName</name>
<price>66</price>
The best product</Prod>
*/
```

Use number

Attribute syntax: useNumber

Applicable to (TTCN-3) Enumeration types generated for XSD enumeration facets where the enumeration base is integer

Description The encoding instruction designates that the encoder shall use the integer values associated to the TTCN-3 enumeration values to produce the value of the corresponding XML attribute or element (as opposed to the names of the TTCN-3 enumeration values) and the decoder shall map the integer values in the received XML attribute or element to the appropriate TTCN-3 enumeration values.

Example

```
type enumerated ProductColor { red(0), green(1), blue(2) }
with {
variant "useNumber"
}

type record NrProduct {
charstring name,
ProductColor color,
integer size
}

const NrProduct rval := {
name := "shirt",
color := red,
size := { sizeval := 10 }
}

/* XML encoding:
<NrProduct>
<name>shirt</name>
<color>0</color>
<size>10</size>
</NrProduct>
*/
```

Use order

Attribute syntax: useOrder

Applicable to (TTCN-3) Record type definition, generated for XSD *complexType*-s with *all* constructor

Description The encoding instruction designates that the encoder shall encode the values of the fields corresponding to the children elements of the *all* constructor according to the order identified by the elements of the **order** field. At decoding, the received values of the XML elements shall be placed in their corresponding record fields and a new record of element shall be inserted into the **order** field for each XML element processed (the final order of the record of elements shall reflect the order of the XML elements in the encoded XML document).

Example

```
type record UOProduct {
record of enumerated { id, name, price, color } order,
integer id,
charstring name,
float price,
charstring color
}

with {
variant "useOrder";
}

const UOProduct rval := {
order := { id, color, price, name },
id := 100,
name := "shirt",
price := 12.23,
color := "red"
}

/* XML encoding:
<UOProduct>
<id>100</id>
<color>red</color>
<price>12.230000</price>
<name>shirt</name>
</UOProduct>
*/
```

Use union

Attribute syntax: useUnion

Applicable to (TTCN-3) unions (all alternatives of the union must be character-encodable)

Description The encoding instruction designates that the encoder shall not use the start-tag and the end-tag around the encoding of the selected alternative (field of the TTCN-3 union type). A type identification attribute (`xsi:type`, where `xsi` is the prefix of the control namespace) can be used to identify the selected alternative, or the encoding of the alternatives must be distinct enough that the decoder can determine the selected field based solely on its value. The decoder shall place the received XML value into the corresponding alternative of the TTCN-3 `union` type, based on the received value and the type identification attribute, if present. The encoder will always use the type identification `attribute` to identify the selected field whenever possible. If the union has the attribute or `untagged` encoding instructions, or is the component of a `record of` or set of with the `list` encoding instruction, then the insertion of the type identification attribute is not possible.

Comment There is no check implemented to ensure the fields are sufficiently distinct. If no type identification attribute is present, the first field (in the order of declaration) that manages to successfully decode the XML value will be the selected alternative.

Restrictions The use of the XSD type QName or other unions with the useType or useUnion coding instructions as alternatives are not supported. The useType or useUnion coding instructions cannot be applied to anytype.

Example 1

```
type union ProductId {
integer c1,
integer c2,
integer c3
}

with {
variant "useUnion"
}

const Product rval := {
id := { c2 := 100 },
price := 25.34,
color := "green"
}

/*
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<id xsi:type="c2">100</id>
<price>2.534E1</price>
<color>green</color>
</Product>

*/
Example 2
type union IntStr {
integer int,
charstring str
}

with {
variant "useUnion"
}

type record Data {
IntStr atr,
record of IntStr values
}

with {
variant(atr) "attribute";
variant(values) "list";
}
```

```
const Data d := {
atr := { int := 26 },
values := { { str := "abc" }, { str := "x" }, { int := -7 } }
}

/*

<Data xmlns:xsi=`http://www.w3.org/2001/XMLSchema-instance' atr=`26'>
<values>abc x -7</values>
</Data>
*/
```

Use type

Attribute syntax: useType

Applicable to (TTCN-3) unions

Description The encoding instruction designates that the encoder shall not use the start-tag and the end-tag around the encoding of the selected alternative (field of the TTCN-3 union type), a type identification attribute (`xsi:type`, where `xsi` is the prefix of the control namespace) will be used to identify the selected alternative. This attribute may be omitted in the case of the first alternative. The decoder shall place the received XML value into the corresponding alternative of the TTCN-3 `union` type, based on the received value and the type identification attribute. The first alternative will be selected if this attribute is not present. The encoder will never insert the type identification attribute for the first alternative. Any attributes the selected alternative might have will be inserted to the union's XML tag instead (after the type identification attribute, if it exists).

The `useType` or `useUnion` coding instructions cannot be applied to anytype.

Example

```
type record Shirt {
charstring color,
charstring make,
integer size
}

type record Trousers {
boolean available,
charstring color,
charstring make
} with {
variant(available) "attribute"
}

type record Shoes {
boolean available,
string color,
integer size
```

```
} with {
variant(available) "attribute"
}

type union Product {
Shirt shirt,
Trousers trousers,
Shoes shoes
} with {
variant "useType"
}

const Product pr1 := {
shoes := {
available := false,
color := "red",
size := 9
}
}
/*

<Product xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:type='shoes'
available='false'>
<color>red</color>
<size>9</size>
</Product>

*/
const Product pr2 := {
shirt := {
color := "red",
make := "ABC Company",
size := 9
}
}

/*

<Product xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
<color>red</color>
<make>ABC Company</make>
<size>9</size>
</Product>
*/
```

Whitespace control

Attribute syntax: whitespace (preserve|replace|collapse)

Applicable to (TTCN-3) String types or fields of structured types generated for XSD components with the *whitespace* facet.

Description The encoding instruction designates that the encoder shall normalize the encoded XML values corresponding to the TTCN-3 construct with the whitespace encoding instruction, and the received XML value shall be normalized before decoding as below.

- preserve: no normalization shall be done, the value is not changed.
- replace: all occurrences of HT(9) (horizontal tabulation), LF(10) (line feed) and CR(13) (carriage return) shall be replaced with an SP(32) (space) character.
- collapse: after the processing implied by replace, contiguous sequences of SP(32) (space) characters are collapsed to a single SP(32) (space) character, and leading and trailing SP(32) (space) characters are removed.

Example 1

```
type charstring R
with {
variant "whiteSpace replace"
}

const R rval := "First Line Second Line";
/* The following is a possible XML encoding of `rval'. During decoding it will be
normalized to the value of `rval'.
<R>First
Line
Second
Line</R>
*/
```

Example 2

```
type charstring C
with {
variant "whiteSpace collapse"
}

const C cval := "First Line Second Line";
/* The follwing is a possible XML encoding of `cval'. During decoding it will be
normalized to the value of `cval'.
<C>
First Line
Second Line
</C>
*/
```

4.25.3. External functions

XML encoder / decoder functions must have the “encode(XER)” / “decode(XER)” attribute set.

The following XML coding options can be specified: `XER_BASIC`, `XER_EXTENDED`, `XER_CANONICAL`. These can be used by writing for example: “encode(XER:XER_EXTENDED)” / “decode(XER:XER_EXTENDED)”.

Faults in the XML encoding/decoding process produce errors by default, but this can be modified with the `errorbehavior` attribute. (Codec error handling)

XML encoder functions can also have the “printing(compact)” or “printing(pretty)” attributes. This specifies whether the encoder should add extra white spaces to the XML code or not. This attribute cannot be set at module level.

If compact printing is selected no white spaces are added to the XML code, making it as short as possible, except at the end of the XML code there will always be a new-line character.

Pretty printing makes the code easier to read by adding spaces, new lines and indenting.

For example:

```
external function f_enc_MyRecord(in MyRecord par) return octetstring with { extension
"prototype(convert) encode(XER:XER_EXTENDED) printing(pretty)" }

external function f_dec_MyRecord(in MyRecord par) return octetstring with { extension
"prototype(convert) decode(XER:XER_EXTENDED) printing(pretty)" }
```

4.26. JSON Encoder and Decoder

The JSON encoder and decoder handles JSON-based protocols. The encoder converts abstract TTCN-3 structures (or types) into a JSON representation (see RFC 7159). The decoder converts JSON data into values of abstract TTCN-3 structures.

This section covers the coding rules in general, the attributes controlling them and the encoder / decoder external functions.

4.26.1. General rules and restrictions

You can use the encoding rules defined in this section to encode and decode the following TTCN-3 types:

- anytype
- array
- bitstring
- boolean
- charstring
- enumerated
- float
- hexstring

- integer
- objid
- octetstring
- record, set
- record of, set of
- union
- universal charstring
- verdicttype

The rules also apply to the following ASN.1 types (if imported to a TTCN-3 module):

- ANY
- BIT STRING
- BOOLEAN
- BMPString
- CHOICE, open type (in instances of parameterized types)
- ENUMERATED
- GeneralString
- GraphicString
- IA5String
- INTEGER
- NULL
- NumericString
- OBJECT IDENTIFIER
- OCTET STRING
- PrintableString
- RELATIVE-OID
- SEQUENCE, SET
- SEQUENCE OF, SET OF
- TeletexString
- UniversalString
- UTF8String
- VideotexString
- VisibleString

JSON encoding and decoding is allowed for types with the attribute `encode "JSON"`. The basic types specified in the list above support JSON encoding and decoding by default.

The attribute `encode "JSON"` can also be set globally (at module level), allowing JSON coding for all types defined in that module.

Types imported from ASN.1 modules (from the list above) automatically have JSON coding allowed and cannot have JSON variant attributes.

When using legacy codec handling the encode attribute can be omitted if the type has at least one JSON variant attribute (see here).

Additional requirements for JSON encoding and decoding when using legacy codec handling:

- in case of records, sets, unions and ASN.1 open types every field must also support JSON coding;
- in case of array, record of and set of structures the element type must also support JSON coding.

Basic types

The basic TTCN-3 types are encoded as JSON values.

All integer values and most float values are encoded as JSON numbers. The special float values `infinity`, `-infinity` and `not_a_number` are encoded as JSON strings.

Boolean values are encoded with the JSON literals `true` and `false`.

Bitstring, hexstring and octetstring values (and values of the ASN.1 ANY type) appear as JSON strings containing the bits or hex digits as characters.

Charstrings, universal charstrings and values of ASN.1 string types are encoded as JSON strings. Charstrings appear exactly like in TTCN-3. Universal charstrings will appear in UTF-8 encoding. JSON strings may contain the escaped character `\u` followed by 4 hex digit characters, the decoder will convert this into the character represented by the hex digits.

Object identifiers are encoded as JSON strings containing the components (in number form) separated by dots.

Verdicttype and enumerated types are encoded as JSON strings. The string contains the name of the verdict type or enumerated value.

The ASN.1 NULL value is encoded with the JSON literal `null`.

NOTE	the JSON decoder ignores all type restrictions and will successfully decode values that contradict them (e.g.: a `record of/set of` type with the `length (3..5)` restriction will successfully decode an array of 8 elements), with the exception of arrays. The restrictions of ASN.1 string types are ignored aswell (e.g.: `NumericStrings` can decode strings containing letters).

Structured types

Array, record of and set of structures are encoded as JSON arrays. Their elements will appear in the order they appear in TITAN.

Records and sets are encoded as JSON objects. The JSON object will contain the field name and value pairs of each field in the order they are declared in. Omitted optional fields will be skipped.

The decoder will accept the record / set field name and value pairs in any order, but every non-optional field must be present. Optional fields that do not appear in the JSON object will be omitted.

Unions, anytypes and ASN.1 open types are encoded as JSON objects. The object will contain one name-value pair: the name of the selected field and its value.

4.26.2. Attributes

The following sections describe the TTCN-3 attributes that influence JSON coding (only affects TTCN-3 types, ASN.1 types cannot have attributes that influence JSON coding).

All JSON attributes begin with the word `JSON` followed by a colon (`JSON:<attribute>`). Any number of white spaces (spaces and tabs only) can be added between each word or identifier in the attribute syntax, but at least one is necessary if the syntax does not specify a separator (a comma or a colon). The attribute can also start and end with white spaces.

Alternatively the syntaxes defined in [25] can also be used, for the supported attributes (without the need for the `JSON`: prefix).

Example:

```
variant(field1) "JSON:omit as null";           // ok

variant(field2) " JSON : omit as null ";           // ok (extra spaces)

variant(field3) "JSON   :   omit    as  null";  // ok (with tabs)

variant(field4) "JSON:omitasnull";          // not ok
```

Omit as null

Attribute syntax: omit as null

Applicable to (TTCN-3): Optional fields of records and sets

Description: If set, the value of the specified optional field will be encoded with the JSON literal `null` if the value is omitted. By default omitted fields (both their name and value) are skipped entirely. The decoder ignores this attribute and accepts both versions.

Example:

```
type record PhoneNumber {
  integer countryPrefix optional,
  integer networkPrefix,
  integer localNumber
} with {
  variant(countryPrefix) "JSON:omit as null"
}
var PhoneNumber pn := { omit, 20, 1234567 }
// JSON code with the attribute:
// {"countryPrefix":null,"networkPrefix":20, "localNumber":1234567}
// JSON code without the attribute:
// {"networkPrefix":20, "localNumber":1234567}
```

Name as ...

Attribute syntax: name as <alias>

Applicable to (TTCN-3): Fields of records, sets and unions

Description: Gives the specified field a different name in the JSON code. The encoder will use this alias instead of the field's name in TTCN-3, and the decoder will look for this alias when decoding this field. The syntax of the alias is the same as the syntax of an identifier in TITAN (regex: [A-Za-z][A-Za-z0-9_]*).

Example:

```
type union PersionID {
  integer numericID,
  charstring email,
  charstring name
} with {
  variant(numericID) "JSON:name as ID";
  variant(email) "JSON:name as Email";
  variant(name) "JSON:name as Name";
}
type record of PersionID PersionIDs;
var persionIDs pids := { { numericID := 189249214 }, { email := "jdoe@mail.com" }, {
name := "John Doe" } };
// JSON code:
// [{"ID":189249214},{"Email":"jdoe@mail.com"},{"Name":"John Doe"}]
```

As value

Attribute syntax: as value

Applicable to (TTCN-3): Unions, the anytype, or records or sets with one mandatory field

Description: The union, record, set or anytype will be encoded as a JSON value instead of as a JSON object with one name-value pair (the name of the selected field in case of unions and the anytype,

or the name of the one field in case of records and sets will be omitted, as well as the surrounding braces). This allows the creation of heterogenous arrays in the JSON document (e.g. ["text",10,true,null]).Since the field name no longer appears in the JSON document, the decoder will determine the selected field (in case of unions and the anytype) based on the type of the value. The first field (in the order of declaration) that can successfully decode the value will be the selected one.

This attribute can also be applied to fields of records, sets or unions, or to the element types of records of, sets of or arrays, if they meet the mentioned restrictions. In this case these fields or elements are encoded as JSON values when they are encoded as part of their larger structure (but the types of these fields or elements might be encoded as JSON objects when encoded alone, or as parts of other structures).

NOTE	Pay close attention to the order of the fields when using this attribute on unions and the anytype. It's a good idea to declare more restrictive fields before less restrictive ones (e.g.: hexstring is more restrictive than universal charstring, because hexstring can only decode hex digits, whereas universal charstring can decode any character; see also examples below).

Examples:

```
// Example 1: unions
type union U1 { // good order of fields
  integer i,
  float f,
  octetstring os,
  charstring cs
} with {
  variant "JSON : as value"
}

type union U2 { // bad order of fields
  float f,
  integer i,
  charstring cs,
  octetstring os
} with {
  variant "JSON : as value"
}

type record of U1 RoU1;
type record of U2 RoU2;

var RoU1 v_rou1 := { { i := 10 }, { f := 6.4 }, { os := '1ED5'O }, { cs := "hello" }
};
var RoU2 v_rou2 := { { i := 10 }, { f := 6.4 }, { os := '1ED5'O }, { cs := "hello" }
};

// Both v_rou1 and v_rou2 will be encoded into:
```

```
// [10,6.4,"1ED5","hello"]
// This JSON document will be decoded into v_rou1, when decoding as type RoU1,
// however it will not be decoded into v_rou2, when decoding as RoU2, instead // the
float field will decode both numbers and the charstring field will
// decode both strings: { { f := 10.0 }, { f := 6.4 }, { cs := "1ED5" },
// { cs := "hello" } };

// Example 2: record with one field
type record R {
  integer field
}
with {
  variant "JSON: as value"
}
type record of R RoR;
const RoR c_recs := { { field := 3 }, { field := 6 } };
// is encoded into: [3,6]

// Example 3: anytype (this can only be done as a field or element of a
// structure, since coding instructions cannot be defined for the anytype)
module MyModule {
type record AnyRec {
  anytype val
}
with {
  variant (val) "JSON: as value";
  variant (val) "JSON: name as value";
}
const AnyRec c_val := { val := { charstring := "abc" } };
// is encoded into: {"value":"abc"}
...
} // end of module
with {
  extension "anytype integer, charstring"
}
```

Default

Attribute syntax: default(<value>)

Applicable to (TTCN-3): Fields of records and sets

Description: The decoder will set the given value to the field if the field does not appear in the JSON document.

Legacy version: If the attribute has the `JSON: default(<value>)` format, the <value> contains the JSON encoding of a value of the field's type (only basic types are allowed). String types don't need the starting and ending quotes. The only allowed structured value is the empty structure value `{}`, which can be set for `record of` and `set of` types, as well as empty `record` and `set` types.

New (standard-compliant) version: If the attribute has the `default(<value>)` format (i.e. no `JSON:`

prefix), the <value> contains a TTCN-3 value of the field's type (all JSON-encodable types are allowed, and the value can also contain references to global values).

In both cases all JSON escaped characters can be used in <value>, plus the escape sequence `\)` will add a `)` (right round bracket) character. An un-escaped `)` character in the <value> is interpreted as the end of the attribute.

Optional fields with a default value will be set to `omit` if the field is set to `null` in JSON code, and will use the default value if the field does not appear in the JSON document.

Example (legacy version):

```
type record Product_legacy {
  charstring name,
  float price,
  octetstring id optional,
  charstring from
} with {
  variant(id) "JSON : default (FFFF)"
  variant(from) "JSON:default(Hungary)"
}

// { "name" : "Shoe", "price" : 29.50 } will be decoded into:
// { name := "Shoe", price := 29.5, id := 'FFFF'O, from := "Hungary" }

// { "name" : "Shirt", "price" : 12.99, "id" : null } will be decoded into:
// { name := "Shirt", price := 12.99, id := omit, from := "Hungary" }
```

Example (new version):

```
type record Product {
  charstring name,
  float price,
  octetstring id optional,
  charstring origin,
  universal charstring text
}
with {
  variant(id) "default ('FFFF'O)"
  variant(origin) "default(""Hungary"")"
  variant(text) "default (char(1,2,3,4\) & char(5,6,7,8\) & ""?"")"
}

// { "name" : "Shirt", "price" : 12.99, "id" : null } will be decoded into:
// { name := "Shirt", price := 12.990000, id := omit, origin := "Hungary",
//   text := char(1, 2, 3, 4) & char(5, 6, 7, 8) & "?" }

type record Shopping_cart {
  charstring name,
  Product product
} with {
  variant(product) "default ({""Shirt"", 12.99, omit, ""Hungary"", ""available"" })"
}

// { "name" : "test shopper" } will be decoded into:
// { name := "test shopper", product := { name := "Shirt", price := 12.990000,
//   id := omit, origin := "Hungary", text := "available" } }

const Product c_defaultProduct := {
  name := "Size ""M"" Shirt",
  price := 12.99,
  id := omit,
  origin := "Hungary",
  text := "available"
}

type record Shopping_cart_2 {
  charstring name,
  Product product
} with {
  variant(product) "default (c_defaultProduct)"
}

// { "name" : "test shopper" } will be decoded into:
// { name := "test shopper", product := { name := "Size \"M\" Shirt", price :=
12.990000,
//   id := omit, origin := "Hungary", text := "available" } }
```

Extend

Attribute syntax: extend(<key>):(<value>)

Applicable to (TTCN-3): Any type

Description: Extends the JSON schema segment generated for this type with the specified key-value pair. The <value> is inserted as a string value.

Both <key> and <value> are strings that can contain any character of a JSON string, plus the escape sequence `)' can be used to add a `)' (right round bracket) character.

This attribute can be set multiple times for a type, each key-value pair is inserted as a field to the end of the type's schema. Extending a schema with multiple fields with the same key produces a warning. Using one of the keywords used in the generated schema also produces a warning.

This attribute only influences schema generation. It has no effect on encoding or decoding values.

Metainfo for unbound

Attribute syntax metainfo for unbound

Applicable to (TTCN-3) Records, sets and fields of records and sets

Description Allows the encoding and decoding of unbound fields with the help of a meta info field. The attribute can be set to fields individually, or to the whole `record/set` (which is equal to setting the attribute for each of its fields).

The encoder sets the field's value in JSON to `null` and inserts an extra (meta info) field into the JSON object. The meta info field's name is `metainfo <fieldname>`, where <fieldname> is the name of the unbound field (or its alias, if the `name as ⋯` attribute is set). Its value is `unbound` (as a JSON string).

The decoder accepts the meta info field regardless of its position in the JSON object (it can even appear before the field it refers to). If the meta info field's value is not `unbound`, or it refers to a field that does not exist or does not have this attribute set, then an encoding error is displayed. The referenced field must either be `null` or a valid JSON value decodable by the field.

Example:

```
// Example 1: meta info for a single field
type record Rec {
  integer num,
  charstring str
}
with {
  variant(str) "JSON: metainfo for unbound";
}

// { num := 6, str := <unbound> } is encoded into:
// {"num":6,"str":null,"metainfo str":"unbound"}

// Example 2: meta info for the whole set (with "name as" and optional field)
type set Set {
  integer num,
  charstring str,
  octetstring octets optional
}
with {
  variant " JSON : metainfo for unbound ";
  variant (num) " JSON : name as int ";
}

// { num := <unbound>, str := "abc", octets := <unbound> } is encoded into:
// {"int":null,"metainfo int":"unbound","str":"abc","octets":null,
// "metainfo octets":"unbound"}

// Example 3: other values accepted by the decoder
// (these cannot be produced by the encoder)

// { "int" : 3, "str" : "abc", "octets" : "1234", "metainfo int" : "unbound" }
// is decoded into: { num := <unbound>, str := "abc", octets := '1234'O }

// {"metainfo int" : "unbound", "int" : null, "str" : "abc", "octets" : "1234"}
// is decoded into: { num := <unbound>, str := "abc", octets := '1234'O }
```

As number

Attribute syntax: as number

Applicable to (TTCN-3): Enumerated types

Description: If set, the enumerated value's numeric form will be encoded as a JSON number, instead of its name form as a JSON string.

Similarly, the decoder will only accept JSON numbers equal to an enumerated value, if this attribute is set.

Example:

```
type enumerated Length { Short (0), Medium, Long(10) }
with {
  variant "JSON: as number"
}
type record of Length Lengths;
const Lengths c_len := { Short, Medium, Long };
// is encoded into: [ 0, 1, 10 ]
```

Chosen

Attribute syntax: chosen (<parameters>)

Applicable to (TTCN-3): Union fields of records and sets

Description: This attribute indicates that the fields of the target `union` will be encoded without field names (as if the `union` had the attribute as `value`), and that the selected field in the `union` will be determined by the values of other fields in the parent `record`/`set`, as described by the rules in the attribute's parameters.

The attribute's parameters are a list of rules, separated by semicolons (;). Each rule consists of a field name from the `union` (or `omit`, if the `union` is an optional field in the parent `record`/`set`), and a condition (or list of conditions). If the condition is true, then the specified field will be selected (or the field will be omitted). If there are multiple conditions, then only one of them needs to be true for the specified field to be selected.

The rules have the following syntax:

<field or omit>, <condition>;

if there's only one condition, **or**

<field or omit>, { <condition1>, <condition2>, ... };

if there are multiple conditions.

The syntax of a condition is

<field reference> = *<value>*

or the keyword `otherwise` (which is true if all other conditions are false).

The <field reference> is a reference to a field within the record/set. It can contain multiple field names to indicate an embedded field, but it cannot contain array indexes.

The <value> can be any value of a built-in type.

The rules do not affect JSON encoding, only decoding (i.e. this attribute is equivalent to the attribute `as value`, when encoding).

Example:

```
type record PduWithId {
  integer protocolId,
  Choices field optional
}
with {
  variant (field) "chosen ( type1, { protocolId = 1, protocolId = 11 };
                            type2, protocolId = 2;
                            type3, protocolId = 3;
                            omit, otherwise)";
  // variant (protocolId) "default (2)";
}
type union Choices {
  StructType1 type1,
  StructType2 type2,
  StructType3 type3
}
// When decoding a value of type PduWithId, type1 will be selected if
// protocolId is 1 or 11, type2 if protocolId is 2, type3 if protocolId is 3,
// and the field will be omitted in all other cases.
// For example { "protocolId" : 2, "field" : { ... } } is decoded into:
// { protocolId := 2, field := { type2 := { ... } } }
// Note: the conditions in the attribute are evaluated when the decoder reaches
// the union field, so the protocolId field must precede the union field in the
// JSON document. Otherwise the decoder will use whatever value the protocolId
// field had before decoding began (likely <unbound>, which will cause a DTE).

// Note: If the protocolId field had the attribute 'default' (see commented
// line in the example), then the default value would be used to determine the
// selected field in the union, if the protocolId field is not decoded before
// the union field.
```

As map

Attribute syntax: as map

Applicable to (TTCN-3): Record of/set of with a record/set element type, that has 2 fields, the first of which is a non-optional universal charstring

Description: If set, the mentioned structure is encoded as a JSON object containing key-value pairs. The universal charstrings in the element records/sets are the keys, and the second field in each record/set contains the value. This allows the creation of heterogenous objects in the JSON document (i.e. JSON objects with any combination of field names and values).

Affects both encoding and decoding.

Example:

```
type record MapItem {
  universal charstring key,
  integer value_ optional
}

type set of MapItem Map
with { variant "as map" }

const Map c_map := { { "one", 1 }, { "two", 2 }, { "three", 3 }, { "zero", omit } };
// is encoded into: { "one" : 1, "two" : 2, "three" : 3, "zero" : null }
```

Text ... as ...

Attribute syntax: text '<enum text>' as '<new text>'

Applicable to (TTCN-3): Enumerated types

Description: This attribute can be used to change the encoding of certain enumerated values. Each attribute changes the encoding of one enumerated option.

Affects both encoding and decoding.

Example:

```
type enumerated EnumNumber { One, Two, Three }
with {
  variant "text 'One' as '1'";
  variant "text 'Two' as '2'";
  variant "text 'Three' as '3'";
}
type record of EnumNumber EnumNumberList;

const EnumNumberList c_numbers := { One, Two, Three };
// is encoded into: [ "1", "2", "3" ]
```

Escape as

Attribute syntax: escape as (short | usi | transparent)

Default value: short

Applicable to (TTCN-3): charstrings and universal charstrings

Description: This attribute changes the method of escaping characters when encoding charstrings and universal charstrings.

- `short` - Uses the JSON short escape sequences for any characters that have them (i.e. '\n', '\t', '\r', '\f', '\b', '\"', '\\' and '\/'), and uses USI-like escape sequences (i.e. '\u' followed by 4 hex digits containing the character's ASCII code) for all other characters between char(U0) and char(U1F),

and for char(U7F).

- `usi` - Uses USI-like escape sequences for all characters between char(U0) and char(U20), and for the characters '\"', '\\' and char(U7F). Does not escape the character '/'.
- `transparent` - Identical to the `short` escaping method, except that the characters '\\' and '/' are not escaped.

Example: The universal charstring "a\\b" & char(U7) & "c\td/e" is encoded as:

- `short`: "\"a\\\\b\\u0007c\\td\\/e\"",
- `usi`: "\"a\\u005Cb\\u0007c\\u0009d/e\"",
- `transparent`: "\"a\\b\\u0007c\\td/e\"".

IMPORTANT	Certain specifics of how the `usi` escaping method should work are not defined clearly in the TTCN-3 standard for using JSON in TTCN-3 ([25]), depending on how these issue get resolved in the standard we might be forced to change our implementation. (See http://oldforge.etsi.org/mantis/view.php?id=7913 and http://oldforge.etsi.org/mantis/view.php?id=7914 for more details.) TITAN currently does not escape the solidus character ('/'), and only uses the USI-like escape sequences for the characters listed in the JSON module (in Annex A of [25]), with the exception of the solidus character.

Type indicators

Attribute syntax: JSON: (integer | number | string | array | object | objectMember | literal)

Applicable to (TTCN-3):

- `JSON:integer`: integers,
- `JSON:number`: floats,
- `JSON:string`: universal charstrings,
- `JSON:array`: record ofs
- `JSON:object`: records or sets with one optional field of record of/set of type, whose element type has (or is valid for) the `JSON:objectMember` attribute;
- `JSON:objectMember`: records with two fields, the first one being a universal charstring;
- `JSON:literal`: booleans or enumerated types with one enumerated item.

Description: These attributes indicate which JSON schema types are represented by the TTCN-3 types they are set for. They are only meant to be used in the JSON module from Annex A of the standard for using JSON in TTCN-3 ([25]). Most of them don't change anything in the encoding or decoding of values, with the exceptions of `JSON:object` and `JSON:literal` (for enumerated types).

- `JSON:object` allows the creation of any valid JSON object, with user-defined field names and values. The TTCN-3 value is a `record` or `set` with one optional field of `record of` or `set of` type. Each of its elements represents one field in the JSON object. The empty JSON object (`{}`) is represented by the omitted field of the top-level record/set. The encoding and decoding of every

other value functions as if the top-level record/set had the attribute `as value`, and its field had the attribute `as map`.

- `JSON:literal`, when applied to an enumerated type with one enumerated item, changes the encoding of its one value to the JSON literal 'null'. Similarly, the decoder will only accept the JSON value 'null'. This attribute does not change the encoding or decoding of booleans.

4.26.3. External functions

JSON encoder / decoder functions must have the `encode(JSON)` / `decode(JSON)` attribute set.

Faults in the JSON encoding/decoding process produce errors by default, but this can be modified with the `errorbehavior` attribute. (Codec error handling)

JSON encoder functions can also have the `printing(compact)` or `printing(pretty)` attributes. This specifies whether the encoder should add extra white spaces to the JSON code or not. This attribute cannot be set at module level.

If compact printing is selected (or if the printing attribute is not set) no white spaces are added to the JSON code, making it as short as possible.

Pretty printing makes the code easier to read by adding spaces, new lines and indenting.

Example:

```
type enumerated Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
};
type record Date {
  charstring month,
  integer dayIdx,
  Day dayName
}
type record of Date Dates;
type record PhoneNumber {
  integer countryPrefix optional,
  integer networkPrefix,
  integer localNumber
} with {
  variant(countryPrefix) "JSON:omit as null"
}
type record Profile {
  charstring name,
  PhoneNumber phoneNo,
  charstring emailAddr,
  Dates meetings
} with {
  variant(phoneNo) "JSON: name as phone";
  variant(emailAddr) "JSON: name as email";
}
external function f_enc_profile(in Profile par) return octetstring
  with { extension "prototype(convert) encode(JSON) printing(pretty)" }
```

```
...
var Profile prof := { "John Doe", { omit, 20, 1234567 }, "jdoe@mail.com", { {
"December", 31, Saturday }, { "February", 7, Friday } } };
log(f_enc_profile(prof));
// JSON code:
// {
//     "name" : "John Doe",
//     "phone" : {
//         "countryPrefix" : null,
//         "networkPrefix" : 20,
//         "localNumber" : 1234567
//     },
//     "email" : "jdoe@mail.com",
//     "meetings" : [
//         {
//             "month" : "December",
//             "dayIdx" : 31,
//             "dayName" : "Saturday"
//         },
//         {
//             "month" : "February",
//             "dayIdx" : 7,
//             "dayName" : "Friday"
//         }
//     ]
// }
```

4.26.4. Converting TTCN-3 and ASN.1 types to a JSON schema

The TITAN compiler can convert type definitions in TTCN-3 and ASN.1 modules into a JSON schema that validates the JSON encoding of these types.

NOTE	the names of ASN.1 modules, types and fields will appear in TTCN-3 form (as if they were imported into a TTCN-3 module). E.g.: the ASN.1 names `Protocol-Elem` and `value` will appear as `Protocol_Elem` and `value_` respectively.

Usage

The compiler option `--ttcn2json` shall be used for the conversion, followed by JSON schema generator specific options and the list of TTCN-3 and ASN.1 file names.

The option `-j` restricts the TTCN-3 types used in the conversion: only those that have JSON coding enabled will be converted. By default all TTCN-3 types that can be converted will be used. This option does not affect ASN.1 types, these will always be converted.

If option `-f` is set, then the schema will only validate types that have JSON encoding and/or decoding functions, otherwise all types it will validate all types included in the schema.

The options `-A` and `-T` can be used before each input file to specify its type (`-A` for ASN.1 files and `-T` for TTCN-3 files). If a file is not preceeded by either of these option, then the compiler will attempt

to determine its type based on its contents.

The last parameter specifies the name of the JSON schema file if it is preceded by a dash (-). Otherwise the name of the schema will be created using the first input file name (its `.asn` or `.ttcn` extension will be replaced by `.json`, or, if it doesn't have either of these extension, then `.json` will simply be appended to its end).

Usage examples:compiler -ttcn2json -T module1.ttcn -A module2.asn - schema.jsoncompiler --ttcn2json -j module1.ttcn module2.asn

The first example will generate the `schema.json` JSON document containing the schema, the second one will generate `module1.json` (and only JSON-encodable types will be included). These documents will have the "pretty" printing format mentioned in 4.26.3.

Top level

On the top level the schema contains a JSON object with 2 properties.

The first property, "definitions", has the schema segments of the type definitions in the TTCN-3 and ASN.1 modules as its value. This value is a JSON object with one property (key-value pair) for each module. Each property has the module name as its key and an object containing the schema segments for the types definied in that module as its key. Similarly, each type definition's key is the type name and its value is the type's schema segment (these will be described in the next sections).

The second top level property is an "anyOf" structure, which contains references to the TTCN-3 and ASN.1 types' schema segments under "definitions". The types listed here are the ones validated by the schema. If the compiler option `-f` is set, then only the schema segments of types that have either a JSON encoding or decoding function (or both) will be referenced (ASN.1 types can have JSON encoding/decoding functions declared in TTCN-3 modules that import them). Extra information related to the encoding/decoding function(s) is stored after each reference.

Example:

```
module MyModule {
  type enumerated Height { Short, Medium, Tall };
  type set Num {
    integer num
  }
  external function f_enc_h(in Height h) return octetstring
    with { extension "prototype(convert) encode(JSON)" }
  external function f_dec_n(in octetstring o) return Num
    with { extension "prototype(convert) decode(JSON)" }
} with {
  encode "JSON"
}
// Generated JSON schema:
// {
//     "definitions" : {
//         "MyModule" : {
//             "Height" : {
```

```
//                 "enum" : [
//                     "Short",
//                     "Medium",
//                     "Tall"
//                 ],
//                 "numericValues" : [
//                     0,
//                     1,
//                     2
//                 ]
//             },
//             "Num" : {
//                 "type" : "object",
//                 "subType" : "set",
//                 "properties" : {
//                     "num" : {
//                         "type" : "integer"
//                     }
//                 },
//                 "additionalProperties" : false,
//                 "required" : [
//                     "num"
//                 ]
//             }
//         }
//     },
//     "anyOf" : [
//         {
//             "$ref" : "#/definitions/MyModule/Height",
//             "encoding" : {
//                 "prototype" : [
//                     "convert",
//                     "f_enc_h",
//                     "h"
//                 ]
//             }
//         },
//         {
//             "$ref" : "#/definitions/MyModule/Num",
//             "decoding" : {
//                 "prototype" : [
//                     "convert",
//                     "f_dec_n",
//                     "o"
//                 ]
//             }
//         }
//     ]
// }
```

Rules and extra keywords

The JSON schema will be generated according to the rules of the IETF draft v4 (see http://json-schema.org/documentation.html).

In addition to the "definitions" keyword specified above, the schema segments of the type definitions can use the following extra keywords:

- `"subType"`: distinguishes 2 or more types from each other, that otherwise have no other differences in their schema segments (such as: charstring and universal charstring; record and set; record of and set of)
- `"fieldOrder"`: stores the order of the fields of a record or set (value: an array containing the field names) - only needed if there are at least 2 fields
- `"originalName"`: stores the original name of a record/set field (see here)
- `"unusedAlias"`: stores the alias of a record/set/union field name, if it doesn't appear under a "properties" keyword (see here)
- `"omitAsNull"`: specifies if the "omit as null" JSON encoding instruction is present for an optional field of a record or set (see here and here)
- `"numericValues"`: lists the numeric values of the enumerated items (in the same order as the items themselves)

A schema segment is generated for each type that has its own definition in TTCN-3. References to other types in TTCN-3 type definitions are converted into references in the JSON schema. Schema segments for embedded TTCN-3 type definitions are defined inside their parent type's schema segment (see here and here for examples).

The examples in the following sections will only contain JSON schema segments, not complete schemas (generated for one or more TTCN-3/ASN.1 type definitions, not the whole module). These schema segments contain the type name and the schema that validates the type. In a complete JSON schema these segments would be directly under the module's property, which is under "definitions" (for examples see section Top Level, types "Height" and "Num").

Schema segments for basic types

The JSON encoding of basic types is detailed in section Basic Types. Here are their schema segments:

```
// integer(TTCN-3) and INTEGER(ANS.1):
// {
//     "type" : "integer"
// }
// float(TTCN-3) and REAL(ASN.1):
// {
//     "anyOf" : [
//         {
//             "type" : "number"
//         },
//         {
```

```
//             "enum" : [
//                 "not_a_number",
//                 "infinity",
//                 "-infinity"
//             ]
//         }
//     ]
// }
// boolean(TTCN-3) and BOOLEAN(ASN.1):
// {
//     "type" : "boolean"
// }
// charstring(TTCN-3), NumericString(ASN.1), PrintableString(ASN.1),
// IA5String(ASN.1) and VisibleString(ASN.1):
// {
//     "type" : "string",
//     "subType" : "charstring"
// }
// universal charstring(TTCN-3), GeneralString(ASN.1), UTF8String(ASN.1),
// UniversalString(ASN.1), BMPString(ASN.1), GraphicString(ASN.1),
// TeletexString(ASN.1) and VideotexString(ASN.1):
// {
//     "type" : "string",
//     "subType" : "universal charstring"
// }
// bitstring(TTCN-3) and BIT STRING(ASN.1):
// {
//     "type" : "string",
//     "subType" : "bitstring",
//     "pattern" : "^[01]*$"
// }
// hexstring(TTCN-3):
// {
//     "type" : "string",
//     "subType" : "hexstring",
//     "pattern" : "^[0-9A-Fa-f]*$"
// }
// octetstring(TTCN-3), OCTET STRING(ASN.1) and ANY(ASN.1):
// {
//     "type" : "string",
//     "subType" : "octetstring",
//     "pattern" : "^([0-9A-Fa-f][0-9A-Fa-f])*$"
// }
// NULL(ASN.1):
// {
//     "type" : "null"
// }
// objid(TTCN-3), OBJECT IDENTIFIER(ASN.1) and RELATIVE-OID(ASN.1):
// {
//     "type" : "string",
//     "subType" : "objid",
```

```
//     "pattern" : "^[0-2][.][1-3]?[0-9]([.][0-9]|([1-9][0-9]+))*$"
// }
// verdicttype:
// {
//     "enum" : [
//         "none",
//         "pass",
//         "inconc",
//         "fail",
//         "error"
//     ]
// }
// Enumerated types are converted the same way as the verdicttype with the
// addition of the numeric values. Example:
// TTCN-3:
type enumerated Season {
  spring (1), summer (2), fall (3), winter (4)
}
// ASN.1:
Season ::= ENUMERATED {
  spring (1), summer (2), fall (3), winter (4)
}
// JSON schema segment for type "Season":
// "Season" : {
//     "enum" : [
//         "spring",
//         "summer",
//         "fall",
//         "winter"
//     ],
//     "numericValues" : [
//         1,
//         2,
//         3,
//         4
// }
```

Schema segments for records and sets

The JSON object type is used for records and sets. The "properties" keyword specifies the fields of the record (each property's key is the field name, and the value is the field's schema segment). Additional properties are not accepted ("additionalProperties" : false). The "required" keyword determines which fields are mandatory (the names of all non-optional fields are listed here).

Optional fields have an "anyOf" structure directly under "properties" (instead of the field's schema segment). The "anyOf" structure contains the JSON null value and the field's schema segment. The "omitAsNull" keyword is used to specify how omitted optional values are encoded (after the "anyOf" structure).

Examples:

```
// Example 1:
// TTCN-3:
type record Product {
  charstring name,
  float price,
  octetstring id optional,
  charstring from
}
// ASN.1:
Product ::= SEQUENCE {
  name VisibleString,
  price REAL,
  id OCTET STRING OPTIONAL,
  from VisibleString
}
// Schema segment for type "Product":
// "Product" : {
//     "type" : "object",
//     "subType" : "record",
//     "properties" : {
//         "name" : {
//             "type" : "string",
//             "subType" : "charstring"
//         },
//         "price" : {
//             "anyOf" : [
//                 {
//                     "type" : "number"
//                 },
//                 {
//                     "enum" : [
//                     "not_a_number",
//                     "infinity",
//                     "-infinity"
//                 }
//             ],
//         }
//         "id" : {
//             "anyOf" : [
//                 {
//                     "type" : "null"
//                 },
//                 {
//                     "type" : "string",
//                     "subType" : "octetstring",
//                     "pattern" : "^([0-9A-Fa-f][0-9A-Fa-f])*$"
//                 },
//             ],
//             "omitAsNull" : false
//         },
```

```
//         "from" : {
//             "type" : "string",
//             "subType" : "charstring"
//         }
//     },
//     "additionalProperties" : false,
//     "fieldOrder" : [
//         "name",
//         "price",
//         "id",
//         "from"
//     ],
//     "required" : [
//         "name",
//         "price",
//         "from"
//     ]
// }
// Example 2: embedded type definition
// TTCN-3:
type set Barrels {
  integer numBarrels,
  record {
    enumerated { Small, Medium, Large } size,
    boolean filled
  } barrelType
}
// ASN.1:
Barrels ::= SET {
  numBarrels INTEGER,
  barrelType SEQUENCE {
    size ENUMERATED { Small, Medium, Large },
    filled BOOLEAN
  }
}
// JSON schema segment for type "Barrels":
// "Barrels" : {
//     "type" : "object",
//     "subType" : "set",
//     "properties" : {
//         "numBarrels" : {
//             "type" : "integer"
//         },
//         "barrelType" : {
//             "type" : "object",
//             "subType" : "record",
//             "properties" : {
//                 "size" : {
//                     "enum" : [
//                         "Small",
//                         "Medium",
```

```
//                         "Large"
//                     ],
//                     "numericValues" : [
//                         0,
//                         1,
//                         2
//                     ]
//                 },
//                 "filled" : {
//                     "type" : "boolean"
//                 }
//             },
//             "additionalProperties" : false,
//             "fieldOrder" : [
//                 "size",
//                 "filled"
//             ],
//             "required" : [
//                 "size",
//                 "filled"
//             ]
//         }
//     },
//     "additionalProperties" : false,
//     "fieldOrder" : [
//         "numBarrels",
//         "barrelType"
//     ],
//     "required" : [
//         "numBarrels",
//         "barrelType"
//     ]
// }
// Example 3: separate type definitions and references
// (the module name is "MyModule")
// TTCN-3:
type enumerated Size { Small, Medium, Large };
type record BarrelType {
  Size size,
  boolean filled
}
type set Barrels {
  integer numBarrels,
  BarrelType barrelType
}
// ASN.1:
Size ::= ENUMERATED { Small, Medium, Large }
BarrelType ::= SEQUENCE {
  size Size,
  filled BOOLEAN
}
```

```
Barrels ::= SET {
  numBarrels INTEGER,
  barrelType BarrelType
}
// Schema segments for types "Size", "BarrelType" and "Barrels":
// "Size" : {
//     "enum" : [
//         "Small",
//         "Medium",
//         "Large"
//     ],
//     "numericValues" : [
//         0,
//         1,
//         2
//     ]
// }
// "BarrelType" : {
//     "type" : "object",
//     "subType" : "record",
//     "properties" : {
//         "size" : {
//             "$ref" : "#/definitions/MyModule/Size"
//         },
//         "filled" : {
//             "type" : "boolean"
//         }
//     },
//     "additionalProperties" : false,
//     "fieldOrder" : [
//         "size",
//         "filled"
//     ],
//     "required" : [
//         "size",
//         "filled"
//     ]
// },
// "Barrels" : {
//     "type" : "object",
//     "subType" : "set",
//     "properties" : {
//         "numBarrels" : {
//             "type" : "integer"
//         },
//         "barrelType" : {
//             "$ref" : "#/definitions/MyModule/BarrelType"
//         }
//     },
//     "additionalProperties" : false,
//     "fieldOrder" : [
```

```
//         "numBarrels",
//         "barrelType"
//     ],
//     "required" : [
//         "numBarrels",
//         "barrelType"
//     ]
// }
```

Schema segments for records of, sets of and arrays

The JSON array type is used for records of, sets of and arrays. The "items" keyword specifies the schema segment of the items. In case of arrays, the "minItems" and "maxItems" properties are set to the array length.

Arrays are distinguishable from records of and sets of by the "minItems" and "maxItems" keywords, so there is no need for them to have the "subType" property.

Examples:

```
// Example 1:
// TTCN-3:
type record of bitstring Bits;
// ASN.1:
Bits ::= SEQUENCE OF BIT STRING
// Schema segment for type "Bits":
// "Bits" : {
//     "type" : "array",
//     "subType" : "record of",
//     "items" : {
//         "type" : "string",
//         "subType" : "bitstring",
//         "pattern" : "^[01]*$"
//     }
// }
// Example 2 (TTCN-3 only):
type integer Ints[4];
// Schema segment for type "Ints":
// "Ints" : {
//     "type" : "array",
//     "minItems" : 4,
//     "maxItems" : 4,
//     "items" : {
//         "type" : "integer"
//     }
// }
// Example 3:
// reference to record type Num defined in section Top Level.
// TTCN-3:
type set of Num Nums;
```

```
// ASN.1:
Nums ::= SET OF Num
// JSON schema segment for type "Nums":
// "Nums" : {
//     "type" : "array",
//     "subType" : "set of",
//     "items" : {
//         "$ref" : "#/definitions/MyModule/Num"
//     }
// }
// Example 4:
// the same thing with Num as an embedded type
// TTCN-3:
type set of set { integer num } Nums;
// ASN.1:
Nums ::= SET OF SET { num INTEGER }
// JSON schema segment for type "Nums":
// "Nums" : {
//     "type" : "array",
//     "subType" : "set of",
//     "items" : {
//         "type" : "object",
//         "subType" : "set",
//         "properties" : {
//             "num" : {
//                 "type" : "integer"
//             }
//         },
//         "additionalProperties" : false,
//         "required" : [
//             "num"
//         ]
//     }
// }
```

Schema segments for unions, anytype, selection type and open type

The "anyOf" structure is used for unions, open types and the anytype (if they have at least 2 fields). Each item in the "anyOf" structure represents one field of the union; they are each a JSON object with one key-value pair (one property). Same as with records, the "additionalProperties" keyword is set to false, and the one property is marked as required.

Examples:

```
// Example 1: union
// TTCN-3:
type union Thing {
  boolean b,
  integer i,
  charstring cs,
```

```
  record { integer num } rec
}
// ASN.1:
Thing ::= CHOICE {
  b BOOLEAN,
  i INTEGER,
  cs VisibleString,
  rec SEQUENCE { num INTEGER }
}
// Schema segment for type "Thing":
// "Thing" : {
//     "anyOf" : [
//         {
//             "type" : "object",
//             "properties" : {
//                 "b" : {
//                     "type" : "boolean"
//                 }
//             },
//             "additionalProperties" : false,
//             "required" : [
//                 "b"
//             ]
//         },
//         {
//             "type" : "object",
//             "properties" : {
//                 "i" : {
//                     "type" : "integer"
//                 }
//             },
//             "additionalProperties" : false,
//             "required" : [
//                 "i"
//             ]
//         },
//         {
//             "type" : "object",
//             "properties" : {
//                 "cs" : {
//                     "type" : "string",
//                     "subType" : "charstring"
//                 }
//             },
//             "additionalProperties" : false,
//             "required" : [
//                 "cs"
//             ]
//         },
//         {
//             "type" : "object",
```

```
//             "properties" : {
//                 "rec" : {
//                     "type" : "object",
//                     "subType" : "record",
//                     "properties" : {
//                         "num" : {
//                             "type" : "integer"
//                         }
//                     },
//                     "additionalProperties" : false,
//                     "required" : [
//                         "num"
//                     ]
//                 }
//             },
//             "additionalProperties" : false,
//             "required" : [
//                 "rec"
//             ]
//         }
//     ]
// }
// Example 2: anytype (TTCN-3 only)
module … {
  …
} with {
  extension "anytype integer,charstring"
  // the anytype must be referenced at least one,
  // otherwise its schema segment won't be generated
}
// JSON schema segment for the anytype:
// "anytype" : {
//     "anyOf" : [
//         {
//             "type" : "object",
//             "properties" : {
//                 "integer" : {
//                     "type" : "integer"
//                 }
//             },
//             "additionalProperties" : false,
//             "required" : [
//                 "integer"
//             ]
//         },
//         {
//             "type" : "object",
//             "properties" : {
//                 "charstring" : {
//                     "type" : "string",
//                     "subType" : "charstring"
```

```
//                 }
//             },
//             "additionalProperties" : false,
//             "required" : [
//                 "charstring"
//             ]
//         }
//     ]
// }
```

The ASN.1 selection type generates the same schema segment as the specified alternative of the CHOICE would.

Example:

```
// Continuing example 1 (ASN.1 only):
NumRec ::= rec < Thing
// JSON schema segment for type NumRec:
// "NumRec" : {
//     "type" : "object",
//     "subType" : "record",
//     "properties" : {
//         "num" : {
//             "type" : "integer"
//         }
//     },
//     "additionalProperties" : false,
//     "required" : [
//         "num"
//     ]
// }
```

Effect of coding instructions on the schema

For the list of JSON coding instructions see here. As mentioned before, only TTCN-3 types can have coding instructions, ASN.1 types can't.

- *omit as null* - its presence is indicated by the "omitAsNull" keyword (true, if it's present)
- *name as ...* - the alias is used under "properties" instead of the field's name in TTCN-3; the original name is stored under the "originalName" key
- *as value* - the union's "anyOf" structure contains the fields' schema segments instead of the JSON objects with one property; the field's name is stored under the "originalName" key
- *default* - specified by the "default" keyword
- *extend* - adds a custom key-value pair to the type's schema segment
- *as value* + *name as ...* - the field name aliases are stored under the "unusedAlias" keyword, as there are no more JSON objects with one property to store them in (they are also ignored by both the schema and the encoder/decoder, and a compiler warning is displayed in this case)

- *metainfo for unbound* - is ignored by the schema generator

Examples:

```
// Example 1: omit as null
type record Rec {
  integer num optional,
  universal charstring str optional
} with {
  variant(num) "JSON : omit as null"
}
// Schema segment for type "Rec":
// "Rec" : {
//     "type" : "object",
//     "subType" : "record",
//     "properties" : {
//         "num" : {
//             "anyOf" : [
//                 {
//                     "type" : "null"
//                 },
//                 {
//                     "type" : "integer"
//                 }
//             ],
//             "omitAsNull" : true
//         },
//         "str" : {
//             "anyOf" : [
//                 {
//                     "type" : "null"
//                 },
//                 {
//                     "type" : "string",
//                     "subType" : "universal charstring"
//                 }
//             ],
//             "omitAsNull" : false
//         }
//     },
//     "additionalProperties" : false,
//     "fieldOrder" : [
//         "num",
//         "str"
//     ]
// }
// Example 2: name as ...
type set Num {
  integer num
} with {
```

```
    variant(num) "JSON : name as number"
}
// Schema segment for type "Num":
// "Num" : {
//     "type" : "object",
//     "subType" : "set",
//     "properties" : {
//         "number" : {
//             "originalName" : "num",
//             "type" : "integer"
//         }
//     },
//     "additionalProperties" : false,
//     "required" : [
//         "number"
//     ]
// }
// Example 3: as value and name as ...
type union Thing {
  boolean b,
  integer i,
  charstring cs,
  record { integer num } rec
} with {
  variant "JSON : as value";
  variant(i) "JSON : name as int";
  variant(cs) "JSON : name as str";
}
// Schema segment for type "Thing":
// "Thing" : {
//     "anyOf" : [
//         {
//             "originalName" : "b",
//             "type" : "boolean"
//         },
//         {
//             "originalName" : "i",
//             "unusedAlias" : "int",
//             "type" : "integer"
//         },
//         {
//             "originalName" : "cs",
//             "unusedAlias" : "str",
//             "type" : "string",
//             "subType" : "charstring"
//         },
//         {
//             "originalName" : "rec",
//             "type" : "object",
//             "subType" : "record",
//             "properties" : {
```

```
//                 "num" : {
//                     "type" : "integer"
//                 }
//             },
//             "additionalProperties" : false,
//             "required" : [
//                 "num"
//             ]
//         }
//     ]
// }
// Example 4: default
type record Rec {
  integer num,
  universal charstring str
} with {
  variant(num) "JSON : default(0)";
  variant(str) "JSON : default(empty)";
}
// JSON schema segment for type "Rec":
// "Rec" : {
//     "type" : "object",
//     "subType" : "record",
//     "properties" : {
//         "num" : {
//             "type" : "integer",
//             "default" : 0
//         },
//         "str" : {
//             "type" : "string",
//             "subType" : "universal charstring",
//             "default" : "empty"
//         }
//     },
//     "additionalProperties" : false,
//     "fieldOrder" : [
//         "num",
//         "str"
//     ],
//     "required" : [
//         "num",
//         "str"
//     ]
// }
// Example 5: extend
type record Number {
  integer val
} with {
  variant "JSON:extend(comment):(first)";
  variant " JSON : extend (comment) : (second (todo: add more fields\)) ";
  variant "JSON: extend(description):(a record housing an integer)";
```

```
  variant(val) "JSON: extend(description):(an integer)";
  variant(val) "JSON: extend(subType):(positive integer)";
}

// Schema segment for type "Number":
// "Number" : {
//     "type" : "object",
//     "subType" : "record",
//     "properties" : {
//         "val" : {
//             "type" : "integer",
//             "description" : "an integer",
//             "subType" : "positive integer"
//         }
//     },
//     "additionalProperties" : false,
//     "required" : [
//         "val"
//     ],
//     "comment" : "first",
//     "comment" : "second (todo: add more fields)",
//     "description" : "a record housing an integer"
// }

// Displayed warnings:
// warning: JSON schema keyword 'subType' should not be used as the key of
// attribute 'extend'
// warning: Key 'comment' is used multiple times in 'extend' attributes of type
// '@MyModule.Number'
// (The multiple uses of 'description' don't generate a warning, since these
// belong to different types.)
```

External function properties in the schema

JSON encoding/decoding functions can only be declared in TTCN-3 modules, however they can be defined for both TTCN-3 types and imported ASN.1 types.

Information related to a type's JSON encoding/decoding external function is stored after the reference to the type's schema segment in the top level "anyOf" structure.

Extra JSON schema keywords for the external function properties:

- `"encoding"` and `"decoding"`: stores the specifics of the encoding or decoding function as properties (directly under the top level `"anyOf"`, after the reference to the type's schema segment)
- `"prototype"`: an array containing the prototype of the encoding function (as a string), the function name, and the parameter names used in its declaration (directly under `"encoding"` or `"decoding"`)
- `"printing"`: stores the printing settings (values: `"compact"` or `"pretty"`; directly under `"encoding"`)

- `"errorBehavior"`: an object containing the error behavior modifications as its properties, each modification has the error type as key and the error handling as value (directly under `"encoding"` or `"decoding"`)

Example:

```
module Mod {
  type record Rec {
    integer num,
    boolean b
  }
  external function f_enc(in Rec x) return octetstring with {
    extension "prototype(convert) encode(JSON) printing(pretty)"
  }
  external function f_dec(in octetstring o, out Rec x) with {
    extension "prototype(fast) decode(JSON)"
    extension "errorbehavior(ALL:WARNING,INVAL_MSG:ERROR)"
  }

} with {
  encode "JSON"
}
// JSON schema:
// {
//     "definitions" : {
//         "Mod" : {
//             "Rec" : {
//                 "type" : "object",
//                 "subType" : "record",
//                 "properties" : {
//                     "num" : {
//                         "type" : "integer"
//                     },
//                     "b" : {
//                         "type" : "boolean"
//                     }
//                 },
//                 "additionalProperties" : false,
//                 "fieldOrder" : [
//                     "num",
//                     "b"
//                 ],
//                 "required" : [
//                     "num",
//                     "b"
//                 ]
//             }
//         }
//     },
//     "anyOf" : [
```

```
//         {
//             "$ref" : "#/definitions/Mod/Rec",
//             "encoding" : {
//                 "prototype" : [
//                     "convert",
//                     "f_enc",
//                     "x"
//                 ],
//                 "printing" : "pretty"
//             },
//             "decoding" : {
//                 "prototype" : [
//                     "fast",
//                     "f_dec",
//                     "o",
//                     "x"
//                 ],
//                 "errorBehavior" : {
//                     "ALL" : "WARNING",
//                     "INVAL_MSG" : "ERROR"
//                 }
//             }
//         }
//     ]
// }
```

Schema segments for type restrictions

The compiler's `-ttcn2json` option also generates schema segments for type restrictions (subtyping constraints), even though these are ignored by the JSON encoder and decoder. Only restrictions of TTCN-3 types are converted to JSON schema format, ASN.1 type restrictions are ignored.

The generated schema segments only contain basic JSON schema keywords, no extra keywords are needed.

Table 10. Converting TTCN-3 type constraints to JSON schema segments

TTCN-3 type restriction	JSON schema segment
Length restrictions of string types	Keywords `minLength` and `maxLength` are used.
Length restrictions of array types	Keywords `minItems` and `maxItems` are used.
Single values	All single values (more specifically their JSON encodings) are gathered into one JSON `enum`. Keyword valueList is used to store single values of unions with the as value coding instruction (encoded as if they did not have this coding instruction).

TTCN-3 type restriction	JSON schema segment
Value range restrictions of `integers` and `floats`	The keywords minimum and maximum are used to specify the range, and keywords `exclusiveMinimum` and `exclusiveMaximum` indicate whether the limits are exclusive or not. All value range and single value restrictions are placed in an `anyOf` structure, if there are at least two value ranges, or if there is one value range and at least one single value.
Value range restrictions of charstrings and universal charstrings	All value range restrictions are gathered into a set expression in a JSON schema `pattern`.
String pattern restrictions	The TTCN-3 pattern is converted into an extended regular expression and inserted into the schema as a JSON `pattern`. Since the pattern is a JSON string, it cannot contain control characters. These are replaced with the corresponding JSON escape sequences, if available, or with the escape sequence `\u`, followed by the character's ASCII code in 4 hexadecimal digits. Furthermore all backslashes in the string are doubled.

These schema elements are inserted after the type's schema segment. If the type's schema segment only contains a reference to another type (in case it's a `record`/`set`/`union` field of a type with its own definition or it's an alias to a type with its own definition), then an `allOf` structure is inserted, which contains the reference as its first element and the restrictions as its second element (since the referenced type may contain some of the schema elements used in this type's restrictions).

If the value list restriction contains references to other subtypes, then the schema segments of their restrictions are inserted, too.

The JSON coding instructions `as value` (for unions) and `name as···` (for `records`, `sets` and `unions`) are taken into consideration when generating the schema elements for the single values.

All non-ASCII characters in `universal charstring` single values and patterns are inserted into the schema in UTF-8 encoding.

Special cases:

1. The restrictions of `floats` are inserted at the end of the first element in the `anyOf` structure, except those that are related to the special values (`infinity`, `-infinity` and `not_a_number`). The `enum` containing the special values is changed, if any of the special values is not allowed by the type's restrictions. If neither of the special values are allowed, then the `anyOf` structure is omitted, and the type's schema only contains `type : number`, followed by the rest of the restrictions. Similarly, if only special values are allowed by the restrictions, then the type's schema only contains the `enum` with the valid values.
2. If a verdicttype is restricted (with single values), then only the `enum` containing the list of single

values is generated (since it would conflict with the type's schema segment, which is also an `enum`).

3. If a single value restriction contains one or more `omit` values, then all possible JSON encodings of the single value are inserted into the `enum`. There are 2^N different encodings, where N is the number of `omits` in the single value, since each omitted field can be encoded in 2 ways (by not adding the field to the JSON object, or by adding the field with a `null` value).
4. Single value restrictions of unions with the `as value` coding instruction do not specify which alternative the value was encoded from. Thus, the single values are generated a second time, under the extra keyword `valueList`, as if they belonged to a union without `as value` (with alternative names). This second list does not contain all the combinations of omitted field encodings (mentioned in the previous point), only the one, where omitted fields are not added to their JSON objects.

Examples:

```
// Example 1: Type definition with value range restriction and its subtype
// with value list restriction
type integer PosInt (!0..infinity);
type PosInt PosIntValues (1, 5, 7, 10);

// Schema segment generated for type "PosInt":
// "PosInt" : {
//     "type" : "integer",
//     "minimum" : 0,
//     "exclusiveMinimum" : true
// }

// Schema segment generated for type "PosIntValues":
// "PosIntValues" : {
//     "allOf" : [
//         {
//             "$ref" : "#/definitions/MyModule/PosInt"
//         },
//         {
//             "enum" : [
//                 1,
//                 5,
//                 7,
//                 10
//             ]
//         }
//     ]
// }

// Example 2: String type definitions with length, value range and pattern
// constraints
type charstring CapitalLetters ("A".."Z") length (1..6);
type charstring CharstringPattern
  (pattern "*ab?\*\?\(\+[0-9a-fA-F*?\n]#(2,4)\d\w\n\r\s\"x"\\d);
```

```
type universal charstring UnicodeStringRanges
  ("a".. "z", char(0, 0, 1, 81)..char(0, 0, 1, 113));
type universal charstring UnicodePattern
  (pattern "abc?\q{ 0, 0, 1, 113 }z\\q1\q{0,0,0,2}");

// Schema segment generated for type "CapitalLetters":
// "CapitalLetters" : {
//     "type" : "string",
//     "subType" : "charstring",
//     "minLength" : 1,
//     "maxLength" : 6,
//     "pattern" : "^[A-Z]*$"
// }

// Schema segment generated for type "CharstringPattern":
// "CharstringPattern" : {
//     "type" : "string",
//     "subType" : "charstring",
//     "pattern" : "^.*ab.\\*\\?\\(\\+[\n-\r*0-9?A-Fa-f]{2,4}[0-9][0-9A-Za-z]
//[\n-\r]\r[\t-\r ]\"x\"\\\\d$"
// }

// Schema segment generated for type "UnicodeStringRanges":
// "UnicodeStringRanges" : {
//     "type" : "string",
//     "subType" : "universal charstring",
//     "pattern" : "^[a-ző-ű]*$"
// }

// Schema segment generated for type "UnicodePattern":
// "UnicodePattern" : {
//     "type" : "string",
//     "subType" : "universal charstring",
//     "pattern" : "^abc.űz\\\\q1\u0002$"
// }

// Example 3: Array type definitions with length restrictions and
// restrictions for the element type
type record length (3..infinity) of PosInt PosIntList;
type set length (2) of integer OnesAndTwos (1, 2);

// Schema segment generated for type "PosIntList":
// "PosIntList" : {
//     "type" : "array",
//     "subType" : "record of",
//     "items" : {
//         "$ref" : "#/definitions/MyModule/PosInt"
//     },
//     "minItems" : 3
// }
```

```
// Schema segment generated for type "OnesAndTwos":
// "OnesAndTwos" : {
//     "type" : "array",
//     "subType" : "set of",
//     "items" : {
//         "type" : "integer",
//         "enum" : [
//             1,
//             2
//         ]
//     },
//     "minItems" : 2,
//     "maxItems" : 2
// }

// Example 4: Float type definitions with all kinds of restrictions
type float RestrictedFloat (-infinity..-1.0, 0.0, 0.5, 1.0, not_a_number);
type float NegativeFloat (!-infinity..!0.0);
type float InfiniteFloat (-infinity, infinity);

// Schema segment generated for type "RestrictedFloat":
// "RestrictedFloat" : {
//     "anyOf" : [
//         {
//             "type" : "number",
//             "anyOf" : [
//                 {
//                     "enum" : [
//                         0.000000,
//                         0.500000,
//                         1.000000,
//                     ]
//                 },
//                 {
//                     "maximum" : -1.000000,
//                     "exclusiveMaximum" : false
//                 }
//             ]
//         },
//         {
//             "enum" : [
//                 "not_a_number",
//                 "-infinity"
//             ]
//         }
//     ]
// }

// Schema segment generated for type "NegativeFloat":
// "NegativeFloat" : {
```

```
//     "type" : "number",
//     "maximum" : 0.000000,
//     "exclusiveMaximum" : true
// }

// Schema segment generated for type "InfiniteFloat":
// "InfiniteFloat" : {
//     "enum" : [
//         "infinity",
//         "-infinity"
//     ]
// }

// Example 5: verdicttype definition with restrictions (single values)
type verdicttype SimpleVerdict (pass, fail, error);

// Schema segment generated for type "SimpleVerdict":
// "SimpleVerdict" : {
//     "enum" : [
//         "pass",
//         "fail",
//         "error"
//     ]
// }

// Example 6: Union type definition with the "as value" coding instruction and
// its subtypes (one of which references the other)
type union AsValueUnion {
  integer i,
  charstring str
}
with {
  variant "JSON: as value"
}

type AsValueUnion AsValueUnionValues (
  { i := 3 },
  { str := "abc" }
);

type AsValueUnion MoreAsValueUnionValues (
  AsValueUnionValues,
  { i := 6 }
);

// Schema segment generated for type "AsValueUnion":
// "AsValueUnion" : {
//     "anyOf" : [
//         {
//             "originalName" : "i",
//             "type" : "integer"
```

```
//         },
//         {
//             "originalName" : "str",
//             "type" : "string",
//             "subType" : "charstring"
//         }
//     ]
// }

// Schema segment generated for type "AsValueUnionValues":
// "AsValueUnionValues" : {
//     "allOf" : [
//         {
//             "$ref" : "#/definitions/MyModule/AsValueUnion"
//         },
//         {
//             "enum" : [
//                 3,
//                 "abc"
//             ],
//             "valueList" : [
//                 {
//                     "i" : 3
//                 },
//                 {
//                     "str" : "abc"
//                 }
//             ]
//         }
//     ]
// }

// Schema segment generated for type "MoreAsValueUnionValues":
// "MoreAsValueUnionValues" : {
//     "allOf" : [
//         {
//             "$ref" : "#/definitions/MyModule/AsValueUnion"
//         },
//         {
//             "enum" : [
//                 3,
//                 "abc",
//                 6
//             ],
//             "valueList" : [
//                 {
//                     "i" : 3
//                 },
//                 {
//                     "str" : "abc"
//                 },
```

```
//                 {
//                     "i" : 6
//                 }
//             ]
//         }
//     ]
// }

// Example 7: Record definition with field name aliases and extra restrictions
// to its fields, plus its subtype, which contains omit values
type record Rec {
  PosIntValues val optional,
  integer i (0..6-3),
  octetstring os ('1010'O, '1001'O, '1100'O) optional
}
with {
  variant(val) "JSON: name as posInt";
  variant(i) "JSON: name as int";
}

type Rec RecValues (
  { 1, 0, '1010'O },
  { 5, 0, '1001'O },
  { 7, 2, omit },
  { omit, 1, omit }
);

// Schema segment generated for type "Rec":
// "Rec" : {
//     "type" : "object",
//     "subType" : "record",
//     "properties" : {
//         "posInt" : {
//             "anyOf" : [
//                 {
//                     "type" : "null"
//                 },
//                     "originalName" : "val",
//                     "#ref" : "#/definitions/MyModule/PosIntValues"
//                 }
//             ],
//             "omitAsNull" : false
//         },
//         "int" : {
//             "originalName" : "i",
//             "type" : "integer",
//             "minimum" : 0,
//             "exclusiveMinimum" : false,
//             "maximum" : 3,
//             "exclusiveMaximum" : false
//         },
```

```
//         "os" : {
//             "anyOf" : [
//                 {
//                     "type" : "null",
//                 },
//                 {
//                     "type" : "string",
//                     "subType" : "octetstring",
//                     "pattern" : "^([0-9A-Fa-f][0-9A-Fa-f])*$",
//                     "enum" : [
//                         "1010",
//                         "1001",
//                         "1100"
//                     ]
//                 }
//             ],
//             "omitAsNull" : false
//         }
//     },
//     "additionalProperties" : false,
//     "fieldOrder" : [
//         "posInt",
//         "int",
//         "os"
//     ],
//     "required" : [
//         "int"
//     ]
// }

// Schema segment for type "RecValues":
// "RecValues" : {
//     "allOf" : [
//         {
//             "$ref" : "#/definitions/MyModule/Rec"
//         },
//         {
//             "enum" : [
//                 {
//                     "posInt" : 1,
//                     "int" : 0,
//                     "os" : "1010"
//                 },
//                 {
//                     "posInt" : 5,
//                     "int" : 0,
//                     "os" : "1001"
//                 },
//                 {
//                     "posInt" : 7,
//                     "int" : 2
```

```
//                 },
//                 {
//                     "posInt" : 7,
//                     "int" : 2,
//                     "os" : null
//                 },
//                 {
//                     "int" : 1,
//                 },
//                 {
//                     "posInt" : null,
//                     "int" : 1
//                 },
//                 {
//                     "int" : 1,
//                     "os" : null
//                 },
//                 {
//                     "posInt" : null,
//                     "int" : 1,
//                     "os" : null
//                 }
//             ]
//         }
//     ]
// }
```

4.26.5. Differences from the TTCN-3 standard

The JSON encoder and decoder work according to the rules defined in the JSON part of the TTCN-3 standard [25] with the following differences:

- No wrapper JSON object is added around the JSON representation of the encoded value, i.e. all values are encoded as if they had the JSON variant attribute `noType` (from the standard). Similarly, the decoder expects the JSON document to only contain the value's JSON representation (without the wrapper). If a wrapper object is desired, then the type in question should be placed in a `record`, `set` or `union`.
- The JSON encoder and decoder only accept the variant attributes listed here. Some of these have the same effect as variant attributes (with similar names) from the standard. The rest of the variant attributes from the standard are not supported. See here regarding the variant attributes `normalize` and `errorbehavior` (from the standard).
- The syntax of the JSON encode attribute is `encode JSON`. The attribute `encode JSON RFC7159` is not supported.
- The decoder converts the JSON number `-0.0` `(in any form) to the TTCN-3 float `-0.0`, i.e. float values are decoded as if they had the JSON variant attribute `useMinus` (from the standard).The same is not true for integers, since there is no integer value `-0` in TITAN.

4.27. OER Encoder and Decoder

The OER (Octet Encoding Rules) encoder and decoder handles OER-based protocols. The encoder converts abstract ASN.1 structures (or types) into an octetstring representation. The decoder converts octetstring data into values of abstract ASN.1 structures. The encoding and decoding rules of the structures can be found in the [20] standard.

This section covers the not supported parts of the standard and the encoder / decoder external functions.

4.27.1. Not supported parts of the standard

Generally, TITAN does not have full ASN.1 support, therefore some parts of the OER coding are not supported.

The following parts of the standard are not supported:

- In clause 12 (Encoding of real values) of the standard: the coding of real values, whether there are any constraints or not on a REAL ASN.1 type, is handled as it is declared in the clause 12.4 of the standard.
- Clause 23 and 24 are not supported.
- In clause 25 (Encoding of values of the embedded-pdv type): only the "general" case (sub clause 25.3) is supported. The "predefined" case (sub clause 25.2) will be handled as the "general" case.
- In clause 28 (Encoding of the unrestricted character string type): only the "general" case (sub clause 28.3) is supported. The "predefined" case (sub clause 28.2) will be handled as the "general" case.
- Clause 29 (Encoding of values of the time types) is not supported.
- Clause 31 (Canonical Octet Encoding Rules) is not fully supported, as currently there is no way to choose BASIC-OER or CANONICAL-OER coding.

4.27.2. External functions

OER encoder / decoder functions must have the `encode(OER)` / `decode(OER)` attribute set.

Faults in the OER encoding/decoding process produce errors by default, but this can be modified with the `errorbehavior` attribute. (Codec error handling)

4.28. PER Encoder and Decoder

The PER (Packed Encoding Rules) encoder and decoder handles PER-based protocols. The encoder converts abstract ASN.1 structures (or types) into an octetstring representation. The decoder converts octetstring data into values of abstract ASN.1 structures. The encoding and decoding rules are described in [29].

This section covers clarifications and limitations compared to the standard and the encoder / decoder external functions.

4.28.1. Clarifications and limitations compared to the standard

Generally, TITAN does not have full ASN.1 support, therefore some parts of the PER codec are not supported.

The PER codec also uses the result of BER encoding for certain types, so certain limitations in the BER codec will affect PER encoding, too. Most notably the BER codec doesn't support the binary encoding of `REAL` values, only the string-based encoding, because of this neither does the PER codec.

If an extension addition field (in a `SEQUENCE` or `SET`) is absent during decoding, then it will be set to `omit` (if the field is `OPTIONAL`), or to its `DEFAULT` value (if it has one), otherwise it will be left unbound.

The `data-value` field of an `EXTERNAL` value is always encoded as an `OCTET STRING` (i.e. in the `SEQUENCE` described in clause 29 of the standard the `octet-aligned` alternative is selected in all cases).

The `ANY` type is treated as an `OCTET STRING` in TITAN, so the PER encoding of its values uses the encoding rules described for the `OCTET STRING` type.

While decoding an `INTEGER`, if the length determinant is zero, then the decoded `INTEGER` value will also be zero.

If the decoder encounters an unknown extension addition in an `ENUMERATED`, `SEQUENCE`, `SET` or `CHOICE`, then it will skip the extension addition (without producing an error) and continue decoding the structure.

In the following example the encoding of a `Seq2` value can be successfully decoded into a `Seq1` value (the extra extension addition fields are ignored):

```
Seq1 ::= SEQUENCE {
  f1 [1] OCTET STRING,
  f2 [0] OCTET STRING,
  f3 [16] OCTET STRING OPTIONAL,
  ...,
  e1 OCTET STRING,
  e2 [2] OCTET STRING,
  ...,
  f4 [15] OCTET STRING DEFAULT ''H
}

Seq2 ::= SEQUENCE {
  f1 [1] OCTET STRING,
  f2 [0] OCTET STRING,
  f3 [16] OCTET STRING OPTIONAL,
  ...,
  e1 OCTET STRING,
  e2 [2] OCTET STRING,
  [[
    g1 [3] OCTET STRING,
    g2 [4] OCTET STRING OPTIONAL
  ]],
  [[
    g3 [5] OCTET STRING,
    g4 [6] OCTET STRING DEFAULT ''H
  ]],
  e3 [7] INTEGER (2),
  ...,
  f4 [15] OCTET STRING DEFAULT ''H
}
```

4.28.2. External functions

PER encoder / decoder functions must have the `encode(PER)` / `decode(PER)` attribute set.

The following PER coding options can be specified: `ALIGNED`, `UNALIGNED`, `BASIC`, `CANONICAL`. These can be used to set which version of PER the function should use: aligned / unaligned and basic / canonical. If no coding options are specified, then the function will use unaligned, basic PER.

Most faults in the PER encoding / decoding process produce errors by default, but this can be modified with the `errorbehavior` attribute. (Codec error handling)

If a value being encoded or the result of a decoding doesn't comply to subtype constraints, then a warning is produced by default.

4.29. Build Consistency Checks

Executable test suites are typically put together from many sources, some of which (test ports, function libraries, etc.) are not written by the test writers themselves, but are developed

independently. Sometimes, a test suite requires an external component with a certain feature or bug fix, or a certain minimum TITAN version. Building with a component which does not meet a requirement, or an old TITAN version, typically results in malfunction during execution or cryptic error messages during build. If version dependencies are specified explicitly, they can be checked during build and the mismatches can be reported.

4.29.1. Version Information in TTCN-3 Files

TITAN allows test writers to specify that a certain TTCN-3 module requires a minimum version of another TTCN-3 module or a minimum version of TITAN.

Format of Version Information

The format of the version information follows the format of Product Identity (Ericsson standard version information [19]); a combination of letters and digits according to the template pruductNumber/suffix RmXnn, where

- Product number identifies the product. It is 3 characters representing the ABC class, 3 digits called the type number and 2 to 4 digits called the sequence number. This part is optional for backward compatibility reasons.
- Suffix indicates new major versions, which are not backward compatible with previous versions ("Revision suffix"). This part is optional for backward compatibility reasons.
- R is the literal character `R'
- m is a single digit ("Revision digit"). It changes when the product (module) functionality is extended with new features (switching to this version is possible, but switching back might not be).
- X is an uppercase letter of the English alphabet (between A and Z inclusive) which changes when the product (module) realization changes ("Revision letter"). The following letters are not allowed: IOPQRW. Versions of a product where only this letter changes can be switched without side effect.
- nn (optional) is a two-digit number ("Verification step") which specifies a prerelease, a version made available during development.

If the final digits are not present, the version is considered a full release, which is a higher version than any prerelease.

Example accepted formats: CRL 113 200/1 R9A; CRL 113 200 R9A; R9APlease note, that only these are supported from the Ericsson Naming Scheme.

Here is a possible progression of release numbers, in strictly ascending order:

R1A01, R1A02..., R1A (first full release), R1B01, R1B02..., R1B, R1C, R2A, R2B01, R2B02..., R2B, R2C, R3A, etc.

Required TITAN Version

A TTCN-3 module can specify the minimum required version of TITAN which can be used to compile it. The format of the extension attribute is"requiresTITAN <version>". For example, the

following snippet:

```
module X {
  // ⋯
}
with {
  extension "requiresTITAN R8C";
}
```

specifies that module X has to be compiled with TITAN R8D (1.8.pl3) or later. Compiling the module with a TITAN which does not satisfy the requirement will cause a compilation error, stating that the version of the compiler is too low.

Compiling this module with TITAN R8B or below may result in a different compiler error, because the syntax of the attribute is not understood by earlier versions.

Specifying the Version of a TTCN-3 Module

A module's own version information can be specified in an extension attribute. The format of the extension attribute is "version <version data>" that is, the literal string "version" followed by the version information (R-state).

Example:

```
module supplier {
  // ⋯
}
with {
  extension "version R1A";
}
```

The version of the module should be set to match the R-state of the product it belongs to.

For backward compatibility, the lack of version information (no extension attribute with "version" in the module's "with" block) is equivalent to the highest possible version and satisfies any version requirement.

Required Version of an Imported Module

The minimum version of an imported module can be specified with an extension attribute. The format of the extension attribute is "requires <module name> <required version>" that is, the literal string "requires" followed by the actual module name and required version.

Example:

```
module importer {
  import from supplier all;
}
with {
  extension "requires supplier R2A"
}
```

The module name must be one that is imported into the module. Specifying a module which is not imported is flagged as an error.

In general, a module should require the full version of another module or TITAN (the R1A format). Depending on a prerelease version should be avoided whenever possible.

4.29.2. Consistency Check in the Generated Code

A number of checks are performed during the build to ensure consistency of the TITAN compiler, TITAN runtime, C++ compiler used during the build. The compiler generates checking code that verifies:

- The version of the TITAN compiler matches the version of the TITAN runtime
- The platform on which the build is being performed matches the platform of the TITAN compiler
- The compiler used to build the TITAN compiler matches the compiler used to build the TITAN runtime
- Some of this information (in form of C++ preprocessor macros definitions and instructions) is available to test port writers to express dependency on a particular TITAN version. When a C++ file includes a header generated by the TITAN compiler, that header includes the definitions for the TITAN runtime, including version information. These macro dependencies can be used in user-written C++ code.
- TTCN3_VERSION is a C/C++ macro defined by the TITAN runtime headers. It contains an aggregated value of the TITAN major version, minor version and patch level. So, to express that a certain C++ file must be compiled together with TITAN R8C, the following code can be used:

  ```
  #if TTCN3_VERSION < 10802
  #error This file requires TITAN 1.8.2
  #endif
  ```

- There is a preprocessor macro defined in the makefile which identifies the platform (operating system). It can be one of SOLARIS (for Solaris 6), SOLARIS8 (for Solaris 8 and above), LINUX, WIN32. Platform-dependent code can be isolated using conditional compilation based on these macro definitions.
- If the TITAN runtime was compiled with the GNU Compiler Collection (GCC), the macro GCC_VERSION is defined by the TITAN runtime headers. Its value is 10000 * (GCC major version) + 100 * (GCC minor version). For example, for GCC 3.4.6, GCC_VERSION will be defined to the value 30400; for GCC 4.1.2 it will be 40100. The value of this macro is compared during C++

compilation to the version of the compiler that was used to build TITAN itself to ensure consistency of the build. The GCC patch level is ignored for this comparison; code generated by a compiler with the same major and minor version is considered compatible. User-written code can use this value if it requires a certain version of the compiler. Alternatively, the predefined macros of the GNU compiler (**GNUC** and **GNUC_MINOR**) can be used for this purpose.

- If the TITAN runtime was built with the SunPro compiler, the compiler itself defines the __SUNPRO_CC macro. Please consult the compiler documentation for the possible values.

4.30. Negative Testing

4.30.1. Overview

As a TTCN-3 language extension Titan can generate invalid messages for the purpose of negative testing. The purpose is to generate wrong messages that do not conform to a given type that the SUT is expecting, and send them to the SUT and observe the SUT's reaction. In Titan only the encoding is implemented, the decoding of wrong messages is not in the scope of this feature.

In protocol testing the term of abstract syntax and transport syntax can be distinguished. In TTCN-3 abstract syntaxes are the data type definitions, while transport syntax is defined using with attributes (encode, variant) that are attached to type definitions. The negative testing feature defines modifications in the transport syntax, thus it does not affect TTCN-3 type definitions. This means that the content of the values, which shall be called **erroneous values** and **erroneous templates**, will not be modified; only their encoding will be. This encoding (transport syntax) is determined by the with attributes attached to the type definition, in case of negative testing the encoding of a value is modified by attaching special with attributes to the value which is to be encoded. TTCN-3 with attributes can be attached only to module level constants and templates; this is a limitation of the TTCN-3 standard.

Values and templates of the following structured types can be made erroneous:

- record
- set
- record of
- set of
- union

The corresponding ASN.1 types can also be used when imported from an ASN.1 module.

The following **erroneous** behaviors can be defined for the encoding of an **erroneous value** or **template**:

- omit specified fields
- change the specified field's value or both type and value
- omit all fields before or after the specified field
- insert a new field before or after the specified field

The inserted data can be either the value of a given constant or any "raw" binary data.

All encoding types (RAW, TEXT, BER, XER, JSON, OER) supported by TITAN can be used in negative testing.

4.30.2. Syntax

Erroneous attributes follow the syntax laid out in section A.1.6.6 (with statement) of the TTCN-3 standard with the following modifications:

```
AttribKeyword ::= EncodeKeyword | VariantKeyword | DisplayKeyword | ExtensionKeyword |
OptionalKeyword |
```

```
ErroneousKeywordErroneousKeyword ::= "erroneous"
```

For an erroneous attribute the syntax of the AttribSpec, a free text within double quotes, is as follows:

```
AttribSpecForErroneous := IndicatorKeyword [ "(" RawKeyword ")" ] ":="
TemplateInstance [ AllKeyword ]
```

```
IndicatorKeyword := "before" | "value" | "after"
```

```
RawKeyword := "raw"
```

Example (the meaning of this code will be explained in the next chapter):

```
type record MyRec {
  integer i,
  boolean b
}
const MyRec c_myrec := {i:=1,b:=true}
with {
  erroneous (i) "before := 123"
  erroneous (b) "value := omit"
}
```

4.30.3. Semantics

The TemplateInstance is defined in the TTCN-3 standard, however the compiler will accept only constant values that have no matching symbols. The reason for using the TemplateInstance syntax is that it can contain also a type reference, allowing to define both the value and its type.

For example:

```
template MyRec t_myrec := {i:=2,b:=false}
with {
  erroneous (i) "after := MyRec.i:123"
  erroneous (i) "before := MyInteger:123"
}
```

It is important to be able to specify the type of the inserted value because the encoding attributes are attached to the type. In the example above two integer values were inserted, both integers have the same value, however one has type MyRec.i and the other has type MyInteger, this will result in different encodings of the same value if the encoding attributes for the two types are different. In TTCN-3 the encoding attributes are specified using the with attribute syntax, in ASN.1 BER encoding the tagging specifies the encoding attributes. If no type is given then the compiler will use the default type if it can be determined.

For example:

```
erroneous (i) "value := 123"
```

NOTE The compiler will use the integer type and NOT the MyRec.i type.

Both references to constant values and literal values can be used:

```
const MyRec c_myrec := {i:=3,b:=true}
template MyRec t_myrec := {i:=2,b:=false}
with {
  erroneous (i) "after := c_myrec" // type determined by the definition of c_myrec
  erroneous (i) "before := MyRec: {i:=4,b:=true}" // type must be specified
}
```

One or more field qualifiers must be used in the AttribQualifier part. If more than one field is specified, then the erroneous behavior will be attached to all specified fields, for example:

```
erroneous (i,b) "after := MyInteger:123"
```

In this case the value of 123 which has type MyInteger will be inserted both after field i and after field b.

The field qualifiers may reference any field at any depth inside a structured type that can have embedded other structured types. An example for ASN.1 types:

```
MyUnion ::= CHOICE { sof MySeqOf }
MySeqOf ::= SEQUENCE OF MySeq
MySeq ::= SEQUENCE { i INTEGER }
const MyUnion c_myunion := { ⋯ }
with { erroneous (sof[5].i) "value := 3.14" }
This also works in case of recursive types:
type record MyRRec { MyRRec r optional }
const MyRRec c_myrrec := { ⋯ }
with { erroneous (r.r.r.r.r) "value := omit" }
```

If the erroneous value does not contain a field which was referred by the erroneous qualifier then the erroneous behavior specified for that field will have no effect. For example:

```
type union MyUni { integer i, boolean b }
const MyUni c_myuni := { i:=11}
with {
  erroneous (i) "value := MyUni.i:22"
  erroneous (b) "value := MyUni.b:false" // this rule has no effect
}
```

The reason for allowing the second rule is that the erroneous information can be transferred by using assignment. By assigning an erroneous constant to a local variable in a testcase or function it can be used with variables too. For example:

```
function func() {
  var MyUni vl_myuni := c_myuni;
  vl_myuni.b := true;
  // now field b is selected in vl_myuni, therefore the erroneous rule on
  // field b will be active, the rule on field i will have no effect
}
```

The erroneous attribute data is attached to the defined constant or template and not to its fields. The fields of this erroneous constant or template do not contain any information on how they are erroneous; this information is attached to the top level. If a field is encoded on its own or is assigned to some other variable it will not contain any erroneous information. Example:

```
module Example1
{
type record R {
  integer i,
  R r optional
} with { encode "TEXT" variant "BEGIN('[BEGIN]')"; variant "END('[END]')"; variant
"SEPARATOR('[*]')" }
external function encode_R( in R pdu) return charstring with { extension
"prototype(convert) encode(TEXT)" }
const R r1 := { i:=1, r:={ i:=2, r:=omit } }
with { erroneous (r.i) "value:=3" }
control {
  log(encode_R(r1)); // output: "[BEGIN]1[*][BEGIN]3[END][END]"
  log(encode_R(r1.r)); // output: "[BEGIN]2[END]"
  // r1.r is not erroneous if used on its own!
}
}
```

Erroneous constants can be assigned to fields of other erroneous constants and templates, however if the original field or any field embedded in that field was made erroneous then the top level erroneous data will be used and the referenced constant's erroneous data ignored. Erroneous data can be visualized as a tree that is a sub-tree of the tree of a type (in the examples the R type, which is recursive). If two erroneous sub-trees overlap then the one which was attached to the constant used as the value of that field where the overlapping happens will be ignored.

Example:

```
module Example2
{
type record R {
  integer i,
  R r optional
} with { encode "TEXT" variant "BEGIN('[BEGIN]')"; variant "END('[END]')"; variant
"SEPARATOR('[*]')" }
external function encode_R( in R pdu) return charstring with { extension
"prototype(convert) encode(TEXT)" }
const R r0 := { i:=0, r:=omit } with { erroneous (i) "value:=4" }
const R r1 := { i:=1, r:=r0 } with { erroneous (r.i) "value:=3" }
const R r2 := { i:=1, r:=r0 }
const R r3 := { i:=1, r:=r0 } with { erroneous (r.r) "value:=R:{i:=5,r:=omit}" }
control {
  log(encode_R(r0)); // output: "[BEGIN]4[END]"

  log(encode_R(r1)); // output: "[BEGIN]1[*][BEGIN]3[END][END]"
  // the value of r1.r.i is determined by the erroneous attribute of r1!

  log(encode_R(r2)); // output: "[BEGIN]1[*][BEGIN]4[END][END]"
  // the value of r2.r.i is determined by the erroneous attribute of r0

  log(encode_R(r3)); // output: "[BEGIN]1[*][BEGIN]0[*][BEGIN]5[END][END][END]"
  // the value of r3.r.i is 0, the erroneous attribute on r0.i was dropped because
  // when r0 is used as field r3.r then this r3.r field has embedded erroneous data
}
}
```

Meaning of IndicatorKeyword:

- `"before"`: the specified value will be inserted before the specified field(s)
- `"value"`: the specified value will be inserted instead of the value of the specified field(s)
- `"after"`: the specified value will be inserted after the specified field(s)

In case of unions only the "value" keyword can be used.

The optional "raw" keyword that can follow the IndicatorKeyword should be used when raw binary data has to be inserted instead of a value. The specified binary data will be inserted into the encoder's output stream at the specified position. The specified data will not be checked in any way for correctness. For convenience this binary data can be specified using TTCN-3 constants as containers. For different encoding types the different containers are as follows:

	RAW	**TEXT**	**XER**	**BER**	**JSON**	**PER (encoding not yet supported)**	**OER**
octetstring	X	X	X	X	X	X	X

	RAW	TEXT	XER	BER	JSON	PER (encoding not yet supported)	OER
bitstring	X					X	
charstring		X	X		X		
universal charstring			X		X		

Bitstrings can be used for encoding types that support the insertion of not only whole octets but also bits. For example to insert one zero bit between two fields:

```
erroneous (i) "after(raw) := '0'B"
replace a field with bits 101:
erroneous (b) "value(raw) := '101'B"
```

Charstring types can be used in case of text based encodings. For example insert some XML string between two fields:

```
erroneous (i) "after(raw) := ""<ERROR>erroneous element</ERROR>"""
```

Notice that the double quotes surrounding the charstring must be doubled because it's inside another string.

The optional "all" keyword after the TemplateInstance must be used when omitting all fields before or after a specified field, in all other cases it must not be used.

4.30.4. Typical Use Cases

Types used in the examples:

```
type record MyRec {
  integer i,
  boolean b,
  charstring s length (3),
  MyRec r optional
} with { encode "RAW" variant " ⋯.. " }
type record of integer MyRecOf;
type MyRec.i MyInteger with { encode "RAW" variant " ⋯.. " }
```

Discard Mandatory Fields

```
type record of integer IntList;
...
var IntList vl_myList := { 1, 2, 3 };
var IntList vl_emptyList := {};
replace(vl_myList, 1, 2, vl_emptyList); // returns { 1 }
replace("abcdef", 2, 1, ""); // returns "abdef"
replace('12FFF'H, 3, 2, ''H); // returns '12F'H
```

Insert New Fields

```
const MyRec c_myrec3 := { i:=1, b:=true, s:="str", r:=omit }
with {
  erroneous (i) "before := MyRec.i:3" // has same type as field i
  erroneous (b) "after := MyInteger:4"
}
const MyRecOf c_myrecof2 := { 1, 2, 3 }
with { erroneous ([1]) "after := MyRecOf[-]:99" }
```

Ignore Subtype Restrictions

```
const MyRec c_myrec4 := { i:=1, b:=true, s:="str", r:=omit }
with { erroneous (s) "value :=""too long string""" }
```

Change the Encoding of a Field

Here the TTCN-3 root type and value of field i are not changed but the encoding is changed:

```
const MyRec c_myrec5 := { i:=1, b:=true, s:="str", r:=omit }
with { erroneous (i) "value := MyInteger:1" }
```

Completely Change a Field to a Different Type and Value

The second field is changed from a boolean to an integer:

```
const MyRec c_myrec6 := { i:=1, b:=true, s:="str", r:=omit }
with { erroneous (b) "value := MyInteger:1" }
```

4.30.5. Summary

Main features of negative testing in TITAN:

- This feature is supported only by the Function Test runtime of TITAN; when doing negative testing this feature must be turned on using the **-R** switch to switch from the default Load Test runtime to the Function Test runtime

- Performance and functionality in case of positive testing is not affected by this feature
- Existing types can be used without modifications (both TTCN-3 and ASN.1 types)
- The erroneous attribute of a value or template does not modify its content, the erroneous feature of that value or template can be seen only when encoding or logging
- `ErroneousKeyword`, `IndicatorKeyword`, `RawKeyword` were not introduced as new keywords in TTCN-3, thus these can be used as identifiers, the compiler is backward compatible
- The erroneousness of a value is lost when sending it between components or using it as parameter of the start() function. In TTCN-3 sending and receiving of values is done by specifying the type of data, but the erroneous information is attached to a value and not the type, thus the receiving side cannot handle erroneous information.

4.30.6. Special Considerations for XML Encoding

There are a number of particularities related to negative testing of XML encoding.

- Inserted and replaced values are encoded using the normal XML encoding functions. These values are encoded as if they were top-level types: the name of the XML element is derived from the TTCN-3 or ASN.1 type name. For built-in types (e.g. integer, boolean, universal charstring) the XML name will be according to Table 4 at the end of clause 11.25 in X.680 ([6]), usually the uppercased name of the type (e.g. INTEGER, BOOLEAN, UNIVERSAL_CHARSTRING). If a particular XML name is desired, an appropriate type alias can be defined.

For example, encoding the following value:

```
type record R { integer i }
const R c_r := { 42 } with { erroneous (i) "value := \"fourty-two\" " }
```

will result in the following XML:

```
<R>
  <CHARSTRING>fourty-two</CHARSTRING>
</R>
```

To generate an XML element with a given name, e.g. "s", the following code can be used:

```
type record R { integer i }
type charstring s; // a type alias
const R c_r := { 42 } with { erroneous (i) "value := s : \"fourty-two\" " }
```

The resulting XML will be (erroneous values highlighted in yellow):

```
<R>
[yellow-background]#  <s>fourty-two</s>#
</R>
```

A `name as "…"` TTCN-3 attribute could also be used, but that also requires a separate type.

- By default, fields of ASN.1 SEQUENCE /TTCN-3 record are encoded as XML elements. Only those fields which have a `with { variant "attribute" }` TTCN-3 attribute applied are encoded as XML attributes. If a field having a `with { variant "attribute" }` has an erroneous value (`before/value/after`), this erroneous value will also be encoded amongst the XML attributes. However, by default the erroneous value will be encoded as an XML element; the resulting XML will not be well-formed:

```
type record R2 {
  charstring at,
  charstring el
}
with { variant (at) "attribute" }

const R2 c_r2 := {
  at := "tack", el := "le"
} with { erroneous (at) "before := 13 " }
results in:

<R2<INTEGER>13</INTEGER> at='tack'>
  <el>le</el>
</R2>
```

To ensure the erroneous value is encoded as an XML attribute, a TTCN-3 type alias can be created which also has a `with { variant "attribute" }` TTCN-3 attribute. The name of the XML attribute can also be controlled either with the name of the type or a name as `"…"` TTCN-3 attribute.

```
// type record R2 as above
type integer newatt with { variant "attribute" } // type alias for integer

const R2 c_r2a := {
  at := "tack", el := "le"
} with { erroneous (at) "before := newatt : 13 " }

<R2 newatt='13' at='tack'>
  <el>le</el>
</R2>
```

- One particularity of the Titan XML encoder is that the space before the name of an XML attribute "belongs" to that attribute (it is written together with the attribute name). If the field (encoded as an XML attribute) is omitted or replaced, the space will also be removed. If a well-formed XML output is desired, the loss of the space must be compensated when using raw

erroneous values (non-raw erroneous values encoded as attributes will supply the space, as can be seen in the previous example).

```
// type record R2 as above
const R2 c_r2r := {
  at := "tack", el := "le"
} with { erroneous (at) "before(raw) := ""ax='xx'"" " } // not compensated

<R2ax='xx' at='tack'>
  <el>le</el>
</R2>
```

The resulting XML is not well formed.

```
// type record R2 as above
const R2 c_r2r := {
  at := "tack", el := "le"
} with { erroneous (at) "before(raw) := "" ax='xx'"" " }
// compensated, note space here-----------^

<R2 #ax='xx' at='tack'>
  <el>le</el>
</R2>
```

Now the XML is well-formed.

- When using `"before := omit all"` or `"after := omit all"` on a member of a record which has a `with { variant "useOrder" }` TTCN-3 attribute, omit-before/omit-after refers to the order of the fields in the record, not the order in which they appear in the XML. In other words, useOrder has no effect on `omit-before/omit-after`.

```
type record UO {
  record of enumerated { id, name, price, color } order,

  integer    id,
  charstring name,
  float      price,
  charstring color
}
with {
  variant "element";
  variant "useOrder";
  variant "namespace as 'http://www.example.com' prefix 'exm'";
}

const UO c_uo_omit_after_price := {
  order := { id, name, color, price }, // color before price
  id    := 1,
  name  := "shoes",
  price :=  9.99,
  color := "brown"
}
with {
  erroneous (price) "after := omit all" // after price: that's just color
}
```

This will result in

```
<exm:UO xmlns:exm='http://www.example.com'>
        <id>1</id>
        <name>shoes</name>
        <!- color omitted here -->
        <price>9.990000</price>
</exm:UO>
```

In the XML, `color` comes before `price`. However, in record UO, `color` comes after `price`; therefore "omit all after price" does affect `color` even though `price` is the last element in the XML.

In a record type `with { variant "useOrder" }`, the first field (a record-of enumerated which controls the order of XML elements) does not appear in the generated XML. Therefore, omitting the first field has no effect.

4.30.7. Special considerations for RAW encoding

There are some RAW encoding attributes (e.g. `LENGTHTO, POINTERTO`) that can invalidate negative testing attributes of a given constant or template. These RAW encoding attributes instruct the RAW encoder to fill some specific fields of a given constant or template being encoded during the encoding process depending on some other specific fields of a given constant or template. In this case the RAW encoding attributes and the negative testing attributes can be in conflict. If a conflict

is detected by the encoder its behavior is user defined. Depending on the `errorbehavior` attribute of the given encoder function (see 4.22.4.3) the encoder can give the user an error (`errorbehavior(NEGTEST_CONFL:ERROR)`), a warning (`errorbehavior(NEGTEST_CONFL:WARNING)`) or it can ignore it as its default behavior (`errorbehavior(NEGTEST_CONFL:IGNORE)`).

The affected RAW encoding attributes and their behaviors used together with negative testing attributes are described in the following list. For detailed information about these RAW encoding attributes please check here.

1. `EXTENSION_BIT(<param>)`

   It is applied even on fields added or modified by negative testing attributes.

```
type record extbitrec {
                      integer f1,
                      integer f2
                    } with { variant "EXTENSION_BIT(yes)" encode "RAW" }
                    const extbitrec cr := { 1, 2 } with
                    { erroneous(f1) "before := 1" erroneous(f2) "value := 1" }
                    // The result will be '010181'O.
```

2. E`XTENSION_BIT_GROUP(<param1, param2, param3>)`

   If a specific extension bit group is somehow affected by negative testing attributes (e.g. some of the elements of a given extension bit group were modified, new fields were added into it) a warning [10] will be given and the extension bit group will be ignored.

```
type record extbitgrouprec {
  integer f1,
  integer f2,
  integer f3,
  integer f4,
  integer f5,
  integer f6
} with {
  variant "EXTENSION_BIT_GROUP(yes, f1, f3)"
  variant "EXTENSION_BIT_GROUP(yes, f5, f6)"
  encode "RAW"
}
const extbitgrouprec cr := { 1, 2, 3, 4, 5, 6 } with {
  erroneous(f1) "before := 1"
  erroneous(f4) "value := 1"
  erroneous(f6) "after := 1" }
// None of the extension bit groups are affected.
// The result will be '0101028301058601'O.
```

3. `LENGTHTO(<param>)` and `LENGTHINDEX(<param>)`

   If any of the fields the length is calculated from or the field the result is stored into are affected

by negative testing attributes a warning will be given and the length calculation will be ignored. In this case the value of the field the result is stored into is undefined, but it's possible to set its value using negative testing attributes.

```
type record lengthtorec1 {
                      integer f1
                    with { variant "" encode "RAW" }
                    type record lengthtorec2 {
                      integer f1 optional,
                      lengthtorec1 f2 optional,
                      charstring f3 optional,
                      charstring f4 optional
                    } with {
                      variant (f2) "LENGTHTO(f3, f4)"
                      variant (f2) "LENGTHINDEX(f1)"
                      encode "RAW"
                    }
                    const lengthtorec2 cr := { 1, { 2 }, "", "one" } with {
                      erroneous(f1) "before := 1" erroneous(f2) "after := 1" }
                    // No conflict, LENGTHTO is calculated normally.
                    // The result will be '010103016F6E65'O.
```

4. `POINTERTO(<param>)`

   If any of the fields between (and including) the pointer and the pointed fields are affected by negative testing attributes (e.g. new fields were added in-between) a warning will be given and the pointer calculation will be ignored. In this case the value of the pointer field will be undefined, but it's possible to set its value using negative testing attributes.

```
type record pointertorec {
                      integer f1,
                      charstring f2,
                      charstring f3
                    } with { variant (f1) "POINTERTO(f3)" encode "RAW" }
                    const pointertorec cr := { 1, "dinner", "bell" } with {
                      erroneous(f1) "before := 1" erroneous(f3) "after := 1" }
                    // No conflict, POINTERTO is calculated normally.
                    // The result will be '010264696E6E657262656C6C01'O.
```

5. `PRESENCE(<param>)` Even if the optional field or any of the fields referenced in the presence indicator list are affected by negative testing attributes a warning will be given and the fields will not be modified according to the PRESENCE RAW encoding attribute, it will be completely ignored.

```
type record presencerec1 {
                       integer f1
                     } with { variant "" encode "RAW" }
                     type record presencerec2 {
                       integer f1,
                       presencerec1 f2,
                       integer f3,
                       integer f4 optional
                     } with { variant (f4) "PRESENCE(f1 = 9, f2.f1 = 99, f3 = 1)"
                       encode "RAW" }
                     const presencerec2 cr := { 1, { 2 }, 3, 4 } with {
                       erroneous(f1) "after := 1" erroneous(f4) "after := 1" }
                     // No conflict.
                     // The result will be '090102030401'O.
```

6. `TAG(<param>)` and `CROSSTAG(<param>)`

   If the field the attribute belongs to or any of the fields referenced in the presence indicator list are affected by negative testing attributes a warning will be given and the fields will not be modified according to the TAG and CROSSTAG RAW encoding attributes, it will be completely ignored.

```
type record tagrec1 {
                      integer f1
                    } with { variant "" encode "RAW" }
                    type record tagrec2 {
                      tagrec1 f1,
                      ragrec1 f2,
                      tagrec1 f3
                    } with { variant "TAG(f1, f1 = 1; f2, f1 = 2)" encode "RAW" }
                    const myrec17 cmr26 := { { 1 }, { 2 }, { 3 } } with {
                      erroneous(f1) "after := 1" erroneous(f2) "after := 1"
                      erroneous(f3) "value := 33" }
                    // No conflict.
                    // The result will be '0101020121'O.
                    type record crosstagrec1 {
                      integer f1,
                      integer f2
                    } with { variant  "" encode "RAW" }
                    type union crosstaguni {
                      crosstagrec1 f1,
                      crosstagrec1 f2
                    } with { variant "" encode "RAW" }
                    type record crosstagrec2 {
                      integer f1,
                      integer f2,
                      crosstaguni f3
                    } with {
                    variant (f3) "CROSSTAG(f1, { f1 = 1, f1 = 11, f2 = 6 };
                      f2, f1 = 3)" encode "RAW" }
                    const crosstagrec2 cr := { 1, 2, { f1 := { 3, 4 } } } with {
                      erroneous(f1) "before := 1" erroneous(f2) "after := 1"
                      erroneous(f3) "after := 9" }
                    // No conflict.
                    // The result will be '01010201030409'O.
```

4.30.8. Special Cosiderations for JSON Encoding

There are a number of particularities related to the negative testing of the JSON encoder.

- **Field names for erroneous values**

  Replaced values in JSON objects (fields of records, sets and unions) keep their field name, even if the replaced value is of a different type.

  Inserted values (in records, sets and unions) receive a field name derived from the name of the value's type. For built-in types (e.g. integer, boolean, universal charstring) the XML name will be according to Table 4 at the end of clause 11.25 in X.680 ([6]), usually the uppercased name of the type (e.g. INTEGER, BOOLEAN, UNIVERSAL_CHARSTRING). For custom types the field name will start with an '@' character, followed by the name of the module the type was defined in, and the name of the type separated by a dot ('.').

Example:

```
module M {
type record R {
  integer i,
  charstring cs
}
type boolean B;
const R c_r := { 3, "a" } with {
  erroneous(i) "before := \"before\"";
  erroneous(i) "value := \"value\"";
  erroneous(cs) "before := B:true";
  erroneous(cs) "after := R.i:10";
}
...
}
// JSON encoding (erroneous values highlighted in yellow):
// {"charstring":"before","i":"value","@M.B":true,"cs":"a","@M.R.i":10}
```

- **Raw values**

  When inserting or replacing with raw values the encoder discards field names (in records, sets and unions) and separating commas (between fields in JSON objects, and between elements of JSON arrays). If a well-formed JSON output is desired, these need to be inserted manually.

  Example:

```
type record R {
  integer i,
  charstring cs
}
type record of integer L;
const R c_r1 := { 1, "a" } with { erroneous(i) "before(raw) := \"abc\"" };
const R c_r2 := { 1, "a" } with { erroneous(i) "value(raw) := \"abc\"" };
const R c_r3 := { 1, "a" } with { erroneous(i) "after(raw) := \"abc\"" };
const L c_l1 := { 1, 2, 3 } with { erroneous([1]) "before(raw) := \"x\"" };
const L c_l2 := { 1, 2, 3 } with { erroneous([1]) "value(raw) := \"x\"" };
const L c_l3 := { 1, 2, 3 } with { erroneous([1]) "after(raw) := \"x\"" };
// JSON encodings (erroneous values highlighted in yellow):
// c_r1: {abc"i":1,"cs":"a"}
// c_r2: {abc"cs":"a"}
// c_r3: {"i":1abc,"cs":"a"}
// c_l1: [1x,2,3]
// c_l2: [1x,3]
// c_l3: [1,2x,3]
```

- **Unsupported types**

  Although the JSON encoder supports anytypes and arrays, these cannot have erroneous

attributes, thus the JSON encoder's negative testing feature is disabled for these types.

4.30.9. Updating erroneous attributes

The erroneous attributes of values and templates can be changed dynamically, using the TITAN-specific statement '@update'.

Its syntax is:

```
UpdateStatement ::= UpdateKeyword "(" ExtendedIdentifier ")" [ WithKeyword
WithAttribList ]

UpdateKeyword ::= "@update"
```

The `@update` statement can be used in functions, altsteps, testcases and control parts. Per the BNF productions in the TTCN-3 standard, the `UpdateStatement' defined here would be in the FunctionStatement and ControlStatement productions.

The `@update` statement replaces the erroneous attributes of the value or template referenced by ExtendedIdentifier with the erroneous attributes specified in WithAttribList. The statement overwrites any erroneous attributes the value or template may have had before. If the `with' attributes are omitted, then the statement removes all the value's or template's erroneous attributes.

Example:

```
type record MyRec {
  integer i,
  boolean b
}
with {
  encode "JSON"
}
const MyRec c_myrec := { i:=1, b:=true }
with {
  erroneous (i) "before := 123"
  erroneous (b) "value := omit"
}
function func() {
  log(encvalue(c_myrec)); // output: {"INTEGER":123,"i":1}

  @update(c_myrec) with { erroneous(i) "value := 3.5" }
  log(encvalue(c_myrec)); // output: {"i":3.500000,"b":true}
  // the erroneous attributes set for c_myrec.b at definition have been
  // overwritten by the @update statement
  @update(c_myrec);
  log(encvalue(c_myrec)); // output: {"i":1,"b":true} // no longer erroneous
}
```

While only literal values and references to constants can be used in the erroneous attribute specs of constant and template definitions, the erroneous specs in an @update statement may contain any value or expression.

Example:

```
function f_sqr(integer p) return integer {
  return p * p;
}
function func2() {
  var integer x := 7;
  @update(c_myrec) with {
    erroneous(i) "value := x + 6";
    erroneous(b) "value := int2str(1 + f_sqr(x - 3)) & \"x\" ";
  }
  log(encvalue(c_myrec)); // output: {"i":13,"b":"17x"}
}
```

Restrictions:

- Only the erroneous attributes of global constants and templates can be updated (including parameterized templates). The reference in the `@update` statement (the ExtendedIdentifier) may not contain field names, array indexes or actual parameters. Only the template identifier shall be used when updating the erroneous attributes of parameterized templates.
- The statement may not contain any attributes other than erroneous attributes.

NOTE	The new erroneous attributes are calculated and attached to the referenced constant or template when the `@update` statement is executed, not when the encoding takes place.

Example:

```
type component MyComp {
  var integer myCompVar;
}
function func3() runs on MyComp {
  myCompVar := 10;
  @update(c_myrec) with {
    erroneous(i) "value := myCompVar"
  } // the erroneous value of c_myrec.i is calculated here

  myCompVar := 3;
  log(encvalue(c_myrec)); // output: {"i":10,"b":true}
  // even though the component variable has changed, the encoder is using the
  // old value stored at the @update statement
}
```

4.31. Testcase Stop Operation

The testcase stop operation defines a user defined immediate termination of a test case with the test verdict **error** and an optional associated reason for the termination. Such an immediate stop of a test case is required for cases where a user defined behavior that does not contribute to the test outcome behaves in an unexpected manner which leads to a situation where the continuation of the test case makes no more sense.

Syntax:

```
testcase "." stop [ "(" { ( FreeText | TemplateInstance ) [ ","] } ")" ]
```

Example:

```
testcase.stop("Unexpected Termination");
// The testcase stops with a Dynamic Testcase Error and the parameter  string is
// written to the log.
```

4.32. Catching Dynamic Test Case Errors

In load testing applications premature termination of test cases due to unforeseen errors is a serious issue because when a dynamic test case error (DTE) occurs there was no way to handle it on TTCN-3 level, thus the test case is stopped with an error verdict. The mechanism of catching DTEs is very similar to exception handling used in other languages: there is a try statement block which contains the guarded code and immediately after it there is a catch statement block which contains the code that is executed if a DTE occurred in the guarded statement block. When a DTE occurs in a statement the rest of the statements in that block are skipped and control is transferred to the 1st statement of the catch block. Two TITAN specific keywords were introduced for this purpose: *@try* and *@catch* , these start with a "@" to avoid backward compatibility issues (new keywords clashing with identifiers used in existing code) and to differentiate them from standard TTCN-3 keywords.

Syntax:

```
function MyFunc() {
  @try { // this is the guarded block, all DTEs are caught here
    <statements>
  }
  @catch(dte_str) { // dte_str will contain the error message of the DTE
    <statements>
  }
}
```

The identifier dte_str becomes an invisible variable definition in the @catch block, the code is semantically equivalent to:

```
@catch {
  var charstring dte_str := <DTE error message>;
  <statements>
}
```

This can be used as a normal charstring type variable whose scope is the @catch statement block.

Example:

```
// The predefined str2int() function causes a DTE if the input is invalid,
// this wrapper function makes it safe to use it on possibly invalid input strings
function safe_str2int(in charstring int_str, in integer defval) return integer {
  @try {
    return str2int(int_str);
  }
  @catch(err) {
    return defval;
  }
}

// Here we check if the DTE was because of a division by zero, if not then
// the DTE with the same message is created again, so that any other DTE will
// stop the execution of the test case with the original error message.
// If it was a division by zero then the verdict is set to fail.
external function throw(in charstring msg);
testcase TC(in integer i) runs on MyComp {
  @try {
    i := 10 / i;
    somefunction(); // this function might cause other DTEs
    setverdict(pass); // this line is reached only if there was no DTE
  }
  @catch(err) {
    if (match(err, pattern "*division by zero*")) {
      log("division by zero detected");
      setverdict(fail); // the verdict is fail instead of error
    } else {
      throw(err); // external function used to re-throw the DTE
    }
  }
}

// external function can be used to re-throw the error in the catch block with a
// modified or original (as in the example above) error message, the {cpp}
// implementation:
void throw_(const CHARSTRING& msg) {
  TTCN_error("%s", (const char*)msg);
}

// @try-@catch blocks can be nested. In this example minor DTEs are ignored and the //
```

```
for loop continues but in case of a major error the DTE is re-thrown an caught by //
the outer catch which also terminates the test case with a fail verdict:
testcase TC() runs on MyComp {
  @try {
    for (var integer i:=0; i<100; i:=i+1) {
      @try {
        <statements that can cause DTEs>
      }
      @catch(dte_str) {
        if (match(err, <some pattern for minor errors>) {
          log("minor error ", dte_str, " ignored, continuing load test…");
        } else {
          throw(dte_str);
        }
      }
    }
    setverdict(pass);
  }
  @catch(dte_msg) {
    log("Something went very wrong: ", dte_msg);
    setverdict(fail);
  }
}
```

4.33. Lazy Parameter Evaluation

This feature was developed for load testing, to speed up function execution by not evaluating the actual parameter of the function when its value is not used inside the function. It speeds up execution when relatively large expressions are used as "in" actual parameters. In the normal case the parameter is always evaluated before the execution of the function starts, even if the parameter is never used. In case of lazy parametrization the parameter is not evaluated if it's never used and it is evaluated exactly once when it is used. It is important to note that the values referenced by the expression may change before the evaluation actually happens. This feature can be used only in case of "in" parameters, in case of "inout" and "out" parameters expressions cannot be used and thus it would be useless. The new titan specific keyword *@lazy* was introduced, this must be placed right before the type in the formal parameter. This can be used for both values and templates of all types.

An example logging function that does not evaluate its message parameter if not used:

```
function MyLog(in @lazy charstring message) runs on MyComp {
  if (logEnabled) {
    log(message);
  }
}
```

calling the function with an expression:

```
MyLog( "MyLog: " & log2str(some_large_data_structure) );
```

If `logEnabled` is false the above actual parameter will not be evaluated. Example for evaluation:

```
type component MyComp { var integer ci := 0; }
function MyFuncDieIfZero() runs on MyComp return integer {
  if (ci==0) { testcase.stop; } // die if the component variable is zero
  return ci;
}
function MyLazyFunc(in @lazy integer pi) runs on MyComp {
  ci := 1;
  log(pi); // first use of pi -> it is evaluated, ci==1, 3*1=3 is logged
  ci := 2;
  log(pi); // second use of pi -> not evaluated, the value 3 is used again
}
```

Calling the function:

```
MyLazyFunc(3*MyFuncDieIfZero()); // the MyFuncDieIfZero() function is not
                                                          // evaluated here, ci==0
here, it would die
```

In the above example we see that MyFuncDieIfZero() can also have a side effect if ci==0. If the actual parameter expression has a side effect it must be used carefully in case of lazy formal parameter, because it is either executed at a later time or never executed at all.

Currently the only limitation is that function reference types cannot have lazy formal parameters, thus functions with lazy parameters cannot be invoked.

4.34. Differences between the Load Test Runtime and the Function Test Runtime

The Function Test runtime sometimes provides extra features that the default Load Test runtime doesn't (due to it being optimized for performance). One of these features, negative testing for encoders, was already discussed here.

4.34.1. Referencing elements of `record of` component variables through function parameters

One or more elements of a `record of`, `set of` or array component variable can be passed as `inout` parameters to a function or altstep, where that component variable is visible. In this case changes made inside the function to either of these parameters or to the component variable can cause changes in the others. Lowering the size of the `record of`/`set of` (inside the function) can cause some of these parameters to become `unbound` (this functionality only works in the Function Test runtime).

Example:

```
type record of integer RoI;

type component CT {
  var RoI cv_roi := { 1, 2, 3, 4, 5 };
}

function f_param_ref(inout integer p_ref) runs on CT
{
  cv_roi := { 10 };
  log(p_ref);      // <unbound>
  p_ref := 20;
  log(cv_roi);     // { 10, <unbound>, <unbound>, 20 }
}
...
// function call:
f_param_ref(cv_roi[3]);
```

This also works if the `record of`/`set of`/array is embedded into another structured component variable, or if the referenced elements are embedded into other structures, and these structures are passed as the function's parameters. It also works if the `record of`/`set of`/array is an `optional` field of a `record` or `set`, and the field is set to `omit` inside the function.

This functionality does not work for templates.

4.34.2. Compatibility of record of types

In the Function Test runtime `record of` and `set of` types of the same element type are compatible with each other. In the Load Test runtime this is only true for the following (TTCN-3) element types:

- boolean
- integer
- float
- bitstring
- hexstring
- octetstring
- charstring
- universal charstring

The `record of`s / set ofs are also compatible for the ASN.1 equivalents of the previously listed element types:

- BOOLEAN
- INTEGER

- REAL
- BIT STRING
- OCTET STRING
- NumericString
- PrintableString
- IA5String
- VisibleString
- Unrestricted Character String
- UTCTime
- GeneralizedTime
- UTF8String*
- TeletexString*
- VideotexString*
- GraphicString*
- GeneralString*
- UniversalString*
- BMPString*
- ObjectDescriptor*

There is one exception to this rule: records/sets of universal charstring (or any of its ASN.1 equivalents, marked with *) are not compatible with other records/sets of universal charstrings in the Load Test runtime, if they or their element type have the XER coding instruction `anyElement`.

Example:

```
ype record of integer RoI1;
type record of integer RoI2;

type record Date {
  integer month,
  integer day,
  integer year
}

type record of Date Dates1;
type record of Date Dates2;

...

var RoI1 roi1 := { 1, 2, 3 };
var RoI2 roi2 := roi1; // works in both runtimes

var Dates1 dates1 := { { 3, 30, 2015 }, { 3, 31, 2015 }};
var Dates2 dates2 := dates1; // works in the Function Test runtime, displays
                             // a compilation error in the Load Test runtime
```

4.34.3. Module parameters in the configuration file

When initializing a module parameter in the configuration file, references to other module parameters can only be used in the Function Test runtime. In the Load Test runtime identifiers on the right hand side are treated as enumerated values.

In the Function Test runtime, the left hand side of an assignment or concatenation (in the configuration file) can refer to a part of a module parameter, instead of the whole module parameter, through field names and array indexes. In the Load Test runtime, field names and array indexes are not supported for module parameters.

4.34.4. Multiple value redirects and redirects of partial values

In the Function Test runtime, TITAN supports the use of multiple value redirects in one value redirect structure. These redirects can also be used to store only a part of the received value, or to store the decoding result of a part of the value (with the help of the `@decoded` modifier).

In the Load Test runtime, the received value can only be redirected to one variable or parameter (only the whole value can be redirected, not parts).

Example:

```
type MyType record {
  Header header,
  octetstring payload
}

...

var MyType v_myVar;
var Header v_myHeaderVar;
var MyType2 v_myVar2;

MyPort.receive(MyType:?) -> value v_myVar; // works in both runtimes,
// the entire record value is stored in variable v_myVar

MyPort.receive(MyType:?) -> value (v_myVar, v_myHeaderVar := header)
// only works in the Function Test runtime, the record is stored in v_myVar
// and its field 'header' is stored in v_myHeaderVar;
// causes a compilation error in the Load Test runtime

MyPort.receive(MyType:?) -> value (v_myVar2 := @decoded payload)
// only works in the Function Test runtime, the field 'payload' from the
// received value is decoded (into a value of type MyType2, with the encoding
// type set for MyType2) and stored in v_myVar2;
// causes a compilation error in the Load Test runtime
```

4.34.5. Deactivating altsteps that are currently running

A default `altstep` can deactivate itself (by deactivating the `default` variable that initiated the `altstep` or by deactivating all defaults).

In the Load Test runtime the `deactivate` operation deletes the in parameters of the default `altstep`, if it is currently running. Accessing these parameters after the `deactivate` call may cause segmentation faults or other unexpected behavior. Because of this a warning is displayed whenever `deactivate` is used within a `function` or `altstep`.

In the Function Test runtime this issue is handled by automatically copying all in parameters in `altsteps`.

Example:

```
type component CT {
  var default ct_def;
  timer ct_tmr[2];
  ...
}
altstep as(in integer x) runs on CT {
  [] ct_tmr[x].timeout {
    deactivate(ct_def); // causes a compiler warning in the Load Test runtime
    log(x); // logs 1 in the Function Test runtime, logs memory garbage or
    // causes a segmentation fault in the Load Test runtime
    ct_tmr[x].stop; // stops ct_tmr[1] in the Function Test runtime, causes a
    // dynamic test case error (invalid index) in the Load Test runtime (if it
    // gets this far)
  }
}
testcase tc() runs on CT {
  ct_def := activate(as(1));
  alt {
    ...
  }
}
```

4.35. Profiling and code coverage

The TTCN-3 Profiler measures the execution time of TTCN-3 code lines and functions, and determines the number of times code lines and functions were executed (including tracking lines/functions that were never executed).

The profiler stores the data it has collected in a database file in JSON format.

When a TTCN-3 project (with profiling enabled) finishes execution, a statistics file is generated, containing the information gathered by the profiler in various representations.

4.35.1. Activating the TTCN-3 Profiler

The profiler can be activated in the desired TTCN-3 modules with the compiler option `-z`, followed by a text file containing the list of TTCN-3 file names to be profiled separated by new line characters (this list can also contain `ttcnpp` files).

The TTCN-3 makefile generator can set this option automatically with its own `-z` option (with the same file argument) or through the TPD file.

When `-z` is set, the compiler generates extra code for profiling and code coverage in the code generated for these modules.

The compiler option `-L` must also be specified.

Usage example (activating the profiler in all modules in the current folder):

```
ls -1 *.ttcn > prof_files.txt
ttcn3_compiler -L -z prof_files.txt *.ttcn
```

Once activated the profiler's behavior can be customized through the [PROFILER] section in the configuration file (for more details see here).

4.35.2. Gathered information

The profiler measures two things: the total execution time of a code line or function (the amount of time the TTCN-3 Executor spent running this code line/function) and the total execution count of a code line or function (the number of times a code line/function was executed).

The profiler measures all times with microsecond precision.

The profiler classifies the following TTCN-3 elements as functions: functions, testcases, altsteps, parameterized templates and the control part (in this case the function's name is `control`). External functions are not considered `functions' by the profiler, they are treated as regular TTCN-3 commands.

The `code lines' contain any line with at least one TTCN-3 command. The first line of a function is always a code line (even if it doesn't contain any commands), and measures the time between the beginning of the function's execution and the beginning of the execution of the first line in the function. The time between the end of the last line's execution and the end of the function's execution is not measured separately; it is simply added to the last line's total time.

In the following example there are 10 code lines and 3 functions:

```
module prof1 {
type component C {}
const integer c1 := 7;   // line 5
function f1(inout integer x) runs on C   // line 7, function 'f1'
{
  x := x + c1;   // line 9
}
testcase tc1() runs on C   // line 12, function 'tc1'
{
  var integer x := 6;   // line 14
  f1(x);   // line 15
  log(x);   // line 16
  x := x + 1;   // line 17
}
control {   // line 20, function 'prof1'
  execute(tc1());   // line 21
}
}
```

Gross and net times

The line times measured by the profiler are gross times, meaning they also contain the execution times of all function calls in that line, not just the execution of the line itself. A setting in the configuration file can change the profiler to measure net line times instead, in which case the execution times of function calls will not be added to lines' total execution times.

The same is true for functions: by default the execution times of function calls inside the function are included in the function's total time. A configuration file setting can change the profiler to measure net function times, in which case the function's total time will not contain the times of embedded function calls.

If a function is defined in a module where profiling is not activated (does not appear in the compiler option's file list), and is called from a module where profiling is activated, then that function call acts as if it were a regular code line (its execution time will be added the caller line's and function's total time in both cases).

4.35.3. Contents of the statistics file

The statistics file contains lists of either code lines or functions with the data gathered for each line or function.

The lists start with a title specifying what data the list contains. The titles are preceded and followed by a line made of dashes (-).

Each element in a list starts with the gathered data (either the total execution time, the execution count, or both) or the average time it took for the code line or function to execute (followed by the gathered data this was calculated from in brackets: total time / execution count). The second part of a list element specifies the code line or function the data belongs to in the following format:

<TTCN-3 file name>:<line number> [<function name>]

The function name is only displayed for functions and for code lines that mark the beginning of a function. The line number for functions is the line the function starts in.

The lists either contain raw data or sorted data. Raw data is sorted ascendingly by line number and is always grouped by module (the modules are separated by lines made of dashes). Sorted data is sorted descendingly by either total time, execution count or average time, and can be global or grouped by module.

Here is an example of the sorted average times in the previously mentioned module `prof1` (where `tc1` is called directly from the configuration file and `f1` is called 2 more times from other modules):

```
------------------------------------------------------------
- Average time / execution of code lines, sorted, per module -
------------------------------------------------------------
0.000362s   (0.000362s / 1) prof1.ttcn:17
0.000222s   (0.000222s / 1) prof1.ttcn:12 [tc1]
0.000050s   (0.000050s / 1) prof1.ttcn:16
0.000007s   (0.000021s / 3) prof1.ttcn:5
0.000004s   (0.000012s / 3) prof1.ttcn:9
0.000001s   (0.000003s / 3) prof1.ttcn:7 [f1]
0.000001s   (0.000001s / 1) prof1.ttcn:14
0.000001s   (0.000001s / 1) prof1.ttcn:15
------------------------------------------------------------
...
```

The statistics file contains the following lists:

- Number of code lines and functions - this list is an exception from the previously mentioned rules; this list contains one element per module and specifies the number of code lines and function in that module, ending with a line made of dashes and the total number of code lines and functions

  Example:

  ```
  --------------------------------------
  - Number of code lines and functions -
  --------------------------------------
  prof1.ttcn: 10 lines,   3 functions
  prof2.ttcn: 13 lines,   4 functions
  prof3.ttcn: 13 lines,   3 functions
  --------------------------------------
  Total:      36 lines,   10 functions
  ```

- Raw data (4 lists) - one list containing both the total time and execution count for each code line and one list containing their average times, followed by one list with the total times and execution counts of functions and one list with their average times
- Sorted total times (4 lists) - one list for code lines grouped by module and one (global) for the code lines in all modules, followed by the same for functions (these lists only contain the total execution time of each code line and function)
- Sorted execution counts (4 lists) - same as the lists of total times
- Sorted average times (4 lists) - same as the previous two
- Top 10 total times, execution counts and average times (6 lists) - these contain the first 10 entries from every sorted global list (in the order mentioned in the previous three)
- Unused code lines and functions (2 lists) - these don't contain any data, only the code line / function specification, they are grouped by module, first the unused lines, then the functions

Any of these lists can be disabled in the configuration file (either one by one or grouped).

4.35.4. Starting and stopping the profiler

The profiler can be stopped using the `@profiler.stop` command. When stopped the profiler does not measure new data. This command has no effect if the profiler is already stopped.

A stopped profiler can be restarted with the `@profiler.start` command. Similarly, this has no effect if the profiler is already running.

The execution count of a function is measured at the start of the function, thus if the profiler is stopped when the function is called, its call will not be measured, even if the profiler is restarted during the function's execution.

Starting and stopping the profiler only affects profiling and code coverage in the current component.

The boolean value `@profiler.running` stores the state of the profiler in the current component (`true` if it's running or `false` if it's stopped).

By default the profiler is automatically started for each component when the program starts, this can be changed in the configuration file.

Usage example (a function that's always profiled and returns the profiler to its original state):

```
function f1(inout integer x) runs on C
{
  var boolean stop_prof := not @profiler.running;
  @profiler.start;
  x := x + c1;
  if (stop_prof) {
    @profiler.stop;
  }
}
```

4.35.5. Profiling with multiple Host Controllers

In parallel mode a separate instance of the TTCN-3 Profiler runs on each process (each component, including the MTC, and each HC). These each generate a database file.

The PTCs' and the MTC's profilers generate temporary database files (their names are created by appending a dot and either `mtc` or the PTC's component reference to the original database file name).

The Host Controller's profiler merges the database files produced by its child processes with its own (and with the data gathered on previous runs, if data aggregation is set) and prints the final database file. The HC's profiler is also responsible for generating the statistics file.

The profilers on multiple Host Controllers do not communicate with each other, thus a profiled test system running on multiple hosts will generate multiple database files and statistics files (one of each for every HC).

If more than one host uses the same working directory, then the files generated by the different profilers will overwrite each other. To avoid this clash, certain metacharacters need to be inserted into the database and statistics file names in the configuration file (e.g.: `%h` inserts the host name, or `%p` inserts the HC's process ID, etc; see here).

The `ttcn3_profmerge` tool can be used to merge the database files of multiple Host Controllers and to generate statistics from them.

4.35.6. The ttcn3_profmerge tool

The `ttcn3_profmerge` utility, which can be found in `$TTCN3_DIR/bin`, merges all profiler database files specified in the command line arguments into one output database file. These database files are generated by the TTCN-3 Profiler during runtime, when profiling is activated in a test system. The utility can also generate statistics from the merged database, the same way as the TTCN-3 Profiler.

Since there are two possible outputs, neither of them are written to the standard output.

This tool is useful for test systems that are run on multiple Host Controllers (in parallel mode). It can merge the databases generated on each host.

The command line syntax is:

```
ttcn3_profmerge [-pc] [-o file] [-s file] [-f filter] db_file1 [db_file2 ⋯]
```

or

```
ttcn3_profmerge -v
```

The command line arguments are the input database files, these must always be listed after the switches, and there must be at least one of them.

The settings available in the [`PROFILER`] section of the configuration file (see here) are also available with this tool. `AggregateData` can be achieved by adding the database of the previous run(s) to the input file list. `StartAutomatically`, `NetLineTimes` and `NetFunctionTimes` only affect the data gathering process, and are not relevant to merging databases and to generating the statistics file.

The rest of the configuration file settings are available through command line switches. These are:

- `-p`

  Discards all profiler data from the merged database. All execution times will be set to zero in the output database file, and all statistics related to execution times will not be generated (both profiler and code coverage data cannot be discarded, since there would be no data left). Has the same effect as the configuration file setting `DisableProfiler`.

- `-c`

  Discards all code coverage data from the merged database. All execution counts will be set to

zero in the output database file, and all statistics related to execution counts will not be generated (both profiler and code coverage data cannot be discarded, since there would be no data left). Has the same effect as the configuration file setting `DisableCoverage`.

- `-o file`

  Prints the output (merged) database to the specified file. If this option is not set, then the merged database will not be saved. At least one of the ouput files (this or the statistics file) must be set. Has a similar effect to the configuration file setting `DatabaseFile` (except metacharacters cannot be used and there is not default value).

- `-s file`

  Prints statistics for the merged database to the specified file. If this option is not set, then statistics will not be generated. At least one of the ouput files (this or the output database file) must be set. Has a similar effect to the configuration file settings `StatisticsFile` (except metacharacters cannot be used and there is not default value) and `DisableStatistics` (if the option is not set).

- `-f filter`

  Specifies which statistics entries are generated. The filter is a hexadecimal number, the last 25 digits each correspond to the 25 entries in the statistics file (the least significant bit represents the first entry, and so on; see Table 27 and Table 28). The filter is ignored if the statistics file is not set. Has a similar effect to the configuration file setting `StatisticsFilter` (except the filter can only be specified with a hex number and cannot be concatenated).

- `-v`

  Prints version and license information and exits.

4.36. Defining enumeration fields with values known at compile time

The standard explicitly says that the enumeration fields can have values that must be given as an integer literal. TITAN relaxes this restriction and allows `bitstring`, `octetstring`, `hexstring` literals. In addition, the fields can only be defined with values known at compile time.

For example:

```
For example:
const integer c_myint := 5;
type enumerated MyEnum {
  e_first (1), // allowed, integer literal
  e_second ('10'B), // allowed bitstring literal
  e_third ('43'O), // allowed octetstring literal
  e_fourth ('55'H), // allowed hexstring literal
  e_fifth (bit2int('1101'B)), // allowed value known at compile time
  e_sixth (oct2int('12'O)), // allowed value known at compile time
  e_seventh (2+3), // allowed value known at compile time
  e_eight (c_myint), // allowed value known at compile time
  e_ninth (f()), // not allowed, functions' return values are not known at
                 // compile time
}

function f() return integer {
  return 3;
}
```

4.37. Ports with translation capability

TTCN-3 has standardized the usage of ports with translation capability. See ES 202 781 clause 5.2 ([21]). In this section the differences and limitations in TITAN will be explained.

Limitations of implementation in TITAN:

- The `connect to` port clause is unsupported
- Translation of `address` types is unsupported
- The elements of the list of `form` and `to` clauses of the `in` and `out` messages are separated with a colon (:) instead of a comma (,)

Restrictions:

- It is not possible to chain ports using the `map to` clause.
- The ports that are referenced in the `map to` clause shall have the `provider` extension. A port with `provider` extension cannot have a `map to` clause.
- Port variables can only be constants, variables, templates and var templates.
- Translation functions shall have the `prototype(fast)` extension.
- Test port parameters from the config file cannot be given with wildcard in the place of the component for ports with translation capability. For example `*.*.param_name:="param_value"` will not be passed to the port but with the following line the port parameter will be passed: `system.*.param_name:="param_value"`
- A difference from the dual faced test ports is that the ports with translation capability can work in two modes. In normal mode and in translation mode. In normal mode, the port behaves as a standard messaging port, while in translation mode it works as it is defined in the ES 202 781 standard clause 5.2 ([21]). A test port skeleton must be generated or a real test port

implementation should be available for the port type with the translation capability, to ensure that the port can work in both modes. The test port skeleton is not needed when the port with translation capability is marked as `internal` port. In this case the port can only work in translation mode.

Known issues:

- `user_map` and `user_unmap` will be called for both endpoints of the map and `unmap` operation.
- If a port is added to the `map to` clause of another port, then its module must also be re-compiled (because its generated C++ code changes in this case). Generated Makefiles won't re-compile it if there were no actual changes in the first (provider) port's module. The same is true if a port is removed from another port's `map to` clause.
- When using central storage, or multiple Eclipse projects that reference each other, all of the port declarations in a translation port's `map to` clause must be in the same project as the translation port.

Additions:

- An option to discard the message is added to the `setstate` operation. This is achieved by setting the state to 4 (`DISCARDED`). Technically this does the same thing as setting the state to 2 (`FRAGMENTED`).

Ports with translation capability mechanism:

When a message is sent on a port with translation capability which is working in translation mode (mapped to a port which is present in the `map to` clause of the port with translation capability), the message will be translated using the translation functions.

For example:

```
function int_to_oct(in integer input, out octetstring
    output) {
if (input <= 100) {
        output := int2oct(input, 2);
        port.setstate(0, "Successful <=100!");
        } else {
            port.setstate(1, "Not successful <=100!");
        }
    } with {
        extension "prototype(fast)"
    }

function int_to_oct2(in integer input, out octetstring
    output) {
        if (input > 100) {
            output := int2oct(input, 4);
            port.setstate(0, "Successful >100!");
        } else {
            port.setstate(1, "Not successful >100!");
        }
    } with {
        extension "prototype(fast)"
    }

function char_to_char(in charstring input, out charstring
    output) {
  port.setstate(4); // charstring messages are discarded
} with {
  extension "prototype(fast)"
}

    type port DataPort message map to TransportPort {
    out integer to octetstring with int_to_oct() :
                   octetstring with int_to_oct2()
    out charstring to charstring with char_to_char()
    } with {
        extension "internal"
    }

    type port TransportPort message {
        out octetstring, charstring
    } with {
        extension "provider";
    }
```

If we send the integer `128` on a `DataPort` port instance, the translation functions of the `DataPort` (which are mapping an integer value) will be called in lexicographic order. When one of the translation functions is successful (`port.setstate(0)` statement is executed), no more translation functions will be called. The output of the successful translation function will be forwarded to the

`TransportPort` to send the translated message.

In this case the `int_to_oct` function will be called first, but this function is only mapping an integer which is less than 100. The translation using the `int_to_oct` function will not be successful. Then the int_to_oct2 function will called. This translation function will be successful, and the result of the function will be forwarded to the `TransPortPort`.

If we try to send a `charstring` on a `DataPort` port instance, the translation function will discard it, and nothing will be sent.

The translation logic of receiving is analogous to the translation logic of sending.

The testing of some protocols requires the necessity to be able to send and receive during the execution of a translation function. This can be achieved using the `port.send()` and `port.receive()` statements in a translation function. This is only possible if the translation function has a port clause. The statements will be executed on the port instance which the execution of the translation function belongs to. The send and receive statements in a translation function should be used with caution as they can easily create an infinite loop.

For example when a `port.send()` statement is executed in a translation function, the same translation function could be called automatically because the same translation procedure will be executed on the parameter of the send statement. The handling of these situations can be resolved using module parameters or port parameters.

4.38. Real-time testing features

TITAN supports the real-time testing features described in chapter 5 of the TTCN-3 standard extension `TTCN-3 Performance and Real Time Testing` (ETSI ES 202 782 V1.3.1, [26]) with the differences and limitations described in this section.

- The real-time testing features are disabled by default. They can be activated with the compiler command line option `-I`. When real-time testing features are disabled, the names `now`, `realtime`, and `timestapm` can be used as identifiers (for backward compatibility).
- The `stepsize` setting is not supported. The symbol `now` always returns the current test system time with microsecond precision (i.e. 6 decimal precision).
- While the compiler allows timestamp redirects (`→ timestamp`) to be used in all situations indicated by the standard, they only store a valid timestamp if one was stored in the user-defined port's C++ code (as specified in the API guide, [16]). Timestamp redirects on connected (internal) ports don't do anything, and the referenced variable remains unchanged (see example below).
- Timestamp redirects also work on ports in translation mode, as long as both the translation port type and the provider port type have the `realtime` clause. Timestamp redirects can also be used inside translation functions (i.e. the `port.send` and `port.receive` statements can also have timestamp redirects).
- The `wait` statement is not supported.

Example:

```
type port PT message realtime {
  inout integer
}

type port PT_Internal message realtime {
  inout integer
}
with {
  extension "internal"
}

type component CT {
  port PT pt;
  port PT_Internal pt_int;
}

// timestamp redirects on a mapped user-defined port
testcase tc() runs on CT {
  map(self:pt, system:pt);

  var float time_start := now, time_send, time_receive;

  pt.send(1) -> timestamp time_send;
  // time_send is set in the user-defined code of port type PT
  if (time_send - time_start > 1.0) {
    setverdict(fail);
  }

  alt {
    [] pt.receive(integer: ?) -> timestamp time_receive {
      // time_receive is set in the user-defined code of port type PT
      if (time_receive - time_send > 10.0) {
        setverdict(fail);
      }
    }
  }
}

// timestamp redirects on a connected internal port
testcase tc2() runs on CT {
  connect(self:pt, self:pt);

  var float time_start := now, time_send, time_receive;

  pt_int.send(1) -> timestamp time_send;
  // time_send is unbound

  alt {
    [] pt.receive(integer: ?) -> timestamp time_receive {
      // time_receive is unbound
    }
```

```
  }
}
```

In order for the timestamp redirects to work on port type `PT`, the following need to be set in the port type's C++ code:

- The value pointed to by parameter `timestamp_redirect` in function `PT::outgoing_send` needs to be set to the current test system time (this sets the timestamps for `send` operations on the port).
- The optional `timestamp` parameter of function `PT_BASE::incoming_message` needs to be set to the current test system time, when it is called (this sets the timestamps for `receive`, `trigger` and `check` operations on the port).

For example:

```
void PT::outgoing_send(const INTEGER& send_par, FLOAT* timestamp_redirect)
{
  if (timestamp_redirect != NULL) {
    *timestamp_redirect = TTCN_Runtime::now();
  }

  // the message is redirected to the port's input
  incoming_message(send_par, TTCN_Runtime::now());
}
```

4.39. Object-oriented features

TITAN supports the object-oriented programming features described in chapter 5 of the TTCN-3 standard extension `Object-Oriented Features` (ETSI ES 203 790 V1.1.1, [27]) with the clarifications and limitations described in this section.

- The object-oriented features are disabled by default. They can be activated with the compiler command line option `-k`. When object-oriented features are disabled, the names `class`, `finally`, `this`, `super` and `object` can be used as identifiers (for backward compatibility).
- The following module declaration format is accepted by the compiler:

```
module <name> "TTCN-3:2018 Object-Oriented" { ... }
```

4.39.1. Classes and objects

- Class members of `port` and `port array` types are not allowed.
- Class member `templates` cannot have parameters.
- The members and methods of the resulting object from a casting operation (i.e. ⇒ operation) cannot be accessed directly, the casting result needs to be saved into a variable first.

Example:

```
type class Node {
  var integer data;
  var Node next;
  public function get_data() return integer { return data; }
  public function get_next() return Node { return next; }
}

...

var object v_obj := Node.create(3, null);

var integer v_int := (v_obj => Node).get_data(); // error

var Node v_node := v_obj => Node;
var integer v_int2 := v_node.get_data(); // OK
```

- The result of logging an instance of a class contains the class name and the logging results of the superclass (if there is one) and supertraits (if there are any) in round brackets, separated by commas.

Logging example:

```
type class MyClass2 {
  ...
}

type class @trait MyTraitClass {
  ...
}

type class MyClass3 extends MyClass2, MyTraitClass {
  ...
}

...

var MyClass3 my_object := MyClass3.create(...);
log(my_object); // result: MyClass3 ( MyClass2, MyTraitClass )
```

NOTE Even though every class automatically extends the built-in class `object`, if it has no superclass specified, this class does not appear in the log.

- Two object references are only equal, if they both refer to the same object, or if they are both `null`. Two separate objects with equal members are not equal with each other.

Equality example:

```
var MyClass3 my_object1 := MyClass3.create(...);
var MyClass3 my_object2 := my_object1;
var MyClass3 my_object3 := MyClass3.create(...);
log(my_object1 == my_object2); // true
log(my_object1 == my_object3); // false
```

- The reference `this` can be used inside a class method, constructor or destructor to access members and methods of the class. The stand-alone `this` refers to the object itself.
- The reference `super` can be used inside a class method, constructor or destructor to access methods of the superclass.
- Function parameters of class type are passed by reference, which means:
  - an `in` parameter of class type can be changed to refer to a different object inside the function, but it will keep referring to the initial object after the function call; any changes made to the initally referred object will remain active after the function call; variables, function parameters, the `null` reference and the result of a class `create` operation are valid `in` actual parameters;
  - an `inout` parameter of class type retains all changes to the object reference and the referred object after the function call; only variables and function parameters are valid `inout` actual parameters;
  - an `out` parameter of class type is set to `null` at the beginning of the function (since objects don't have an `unbound` state), and retains all changes made to the object reference after the function call; only variables and function parameters are valid `out` actual parameters.
- A definition in the subclass can only have the same name as a definition in one of the superclasses or supertraits, if both are methods (incl. external or abstract), and both have the same prototype (i.e. identical return type, identical formal parameter types and direction), or if both are properties of the same type.
- Class members cannot be public, as per the standard. The visibility of overriding methods must be the same or less restrictive than that of the inherited method (i.e. `public` methods can only be overriden by `public` methods, and `protected` methods can only be overriden by `public` or `protected` methods).
- Only one object is created when instanciating an `external` class, there are no separate internal or external objects. The code generated for an `external` class is written into a separate pair of source and header files. This code contains both the internal and external parts of the class. It's up to the user to implement any external functions, initialize the external part of the object in the constructor, add additional members and methods, or change existing ones. It's also up to the user whether the external or the internal part of the object is initialized first in the constructor.
- Internal classes cannot have `external` methods.
- The `external` parameters of a constructor cannot be referenced inside the constructor body in TTCN-3 code. They can only be used in the `external` class implementation in C++.

Titan also supports trait classes and multiple inheritance as described in version 1.3.1 of the standard extension, with the clarification, that trait classes do not extend the built-in class `object`,

only normal classes do. Trait classes also cannot be `external`.

- Default values of class method parameters cannot contain references to class members.
- The second parameter of `string2ttcn` cannot be of class type or of a type containing a class.

4.39.2. Constructors and destructors

- Unhandled exceptions and unhandled dynamic test case errors inside the destructor of a class cause the program to terminate (due to C++ limitations).
- Members of `timer` and `timer array` types cannot be initialized in the constructor, and the implicit constructor doesn't add a formal parameter for the initialization of these members.

Implicit constructor example:

```
type class MyClass {
  private const integer c1 := 0;
  private template MyRecordType t1 := *;
  private const boolean c2;
  private template MySetOfType t2;
  private var MyOtherClass v;
  private var template charstring vt;
  private timer tmr;
  private timer tmr_arr[3];
  // default constructor:
  // create(in integer c2, in template MyRecordType t2, in MyOtherClass v, in template
charstring vt) {
  //   this.c2 := c2;
  //   this.t2 := t2;
  //   this.v := v;
  //   this.vt := vt;
  // }
}
```

The constructor and class member initializers work according to version 1.3.1 of the standard extension, with the following clarifications and limitations:

- The restrictions related to the constructor body from section 5.1.1.5 of the standard extension are ignored. There is only a runtime check that ensures all `const` and `template` members are initialized by the end of the constructor's execution.
- The default values of the constructor's formal parameters cannot contain references to class members with no initial value.
- The implicit constructor takes the initial values of `var` and `var template` class members and uses them as the default values of the formal parameters corresponding to these members (as described in the standard extension). In TITAN, these members are left without an initial value. Because of this, the initializers of `var` and `var template` class members cannot contain references to other `var` or `var template` members, even if these have initial values.

Implicit constructor default parameter example:

```
type record MyRecord {
  integer num,
  charstring str
}

type class MyClassImplicit {
  /* implicit constructor */
  private var template integer vt1 := 1;
  private var template MyRecord vt2 := { vt1, ? };
  // error: Implicit constructor default parameter cannot contain
  // reference to template variable class member `vt1'
}

type class MyClassExplicit {
  private var template integer vt1;
  private var template Rec vt2;
  create(template integer vt1 := 1, template MyRecord vt2 := { this.vt1, ? })
  // error: Constructor default parameter cannot contain reference to
  // template variable class member `vt1', which has no initial value
  {
    this.vt1 := vt1;
    this.vt2 := vt2;
  }
}
```

- The super-constructor call cannot contain references to any of the class' members. In TITAN the super-constructor call occurs before class members are initialized.

Example:

```
type class MySuperClass {
  create(integer p) { ... }
}

type class MySubClass extends MySuperClass {
  private const integer c := 1;
  create(integer p := c): MySuperClass(p) { }
  // error: Super-constructor call cannot contain reference to
  // constant class member `c'
}
```

4.39.3. Properties

- The visibility of overriding getters and setters must be the same or less restrictive than that of the inherited getter/setter (i.e. `public` getters/setters can only be overriden by `public` getters/setters, and `protected` getters/setters can only be overriden by `public` or `protected`

getters/setters), same as class methods.

- If the visibility of a getter or setter is omitted, then it inherits the property's visibility (as per the standard). This means that a getter or setter cannot be given the `protected` visibility directly, but it can inherit this visibility from the property.
- When a property is used as the actual parameter of a `function`, then that parameter will not behave as a property inside the function body (i.e. references to that parameter will not call a getter or setter).
  - For `in` parameters this doesn't change anything, since these are passed by value. The getter is called before the function call and its return value is passed to the formal parameter.
  - In case of `inout` parameters a temporary variable of the property's type is passed to the function instead. This temporary is initialized with the return value of the getter before the function call, and after the function call the setter is called with the new value of the temporary.
  - The `out` parameter case is the same as the `inout` parameter case, except the getter is not called, and the temporary remains unbound when the function is called.
- Predefined functions that have `out` or `inout` parameters function the same way (i.e. `decvalue`, `decvalue_unichar`, `string2ttcn` and `int2enum`). Redirects (e.g. value redirects) are treated as `out` parameters in this case.

Example:

```
type class Rectangle {
  private var integer heightVal;
  public var integer @property height {
    @get => heightVal;
    @set {
      if (value > 0) {
        heightVal := value;
      }
    }
  }
}

function f_inc(inout integer val, in integer inc) {
  val := val + inc;
}

...

var Rectangle v_rect := Rectangle.create(10);
f_inc(v_rect.height, 6);
// is equivalent to:
{
  var integer tmp_1 := v_rect.height; // getter
  f_inc(tmp_1, 6);
  v_rect.height := tmp_1; // setter
}
```

- The implicit constructor only adds a formal parameter for an automatic property if it does not inherit another automatic property. The constructor of the class, where the automatic property first appears in the inheritance chain, is the one responsible for initializing the automatic property. If an automatic property inherits an automatic property, then the default value of the implicit constructor formal parameter with the property name is changed to the overriding automatic property's initial value (if it has one).

Example:

```
type class C1 {
  public var integer @property p4 := 1;
  // implicit constructor:
  // create(integer p4 := 1) { this.p4 = p4; }
}

type class C2 extends C1 {
  public var integer @property p4 := 2;
  // implicit constructor:
  // create(integer p4 := 2): C1(p4) { }
}
```

- The bodies of getters and setters, either in simple form or as statement blocks, can reference the variables in the class' `runs-on`, `mtc` and `system` components, same as class methods.
- Trait classes cannot contain property definitions, abstract or otherwise.

4.39.4. Exception handling

- Unhandled exceptions and unhandled dynamic test case errors inside a `finally` block cause the program to terminate (due to C++ limitations).
- The `finally` clause of an altstep's statement block is executed every time the `altstep` is executed, even if none of the altstep's alt-branches are chosen.
- Exceptions of class types can be caught by `catch` clauses of the same class type or any of its superclasses, including `object`.

Example:

```
type class A {}
type class B extends A {}
type class C extends B {}
...
catch(C x) {} // catches any C exceptions
catch(B x) {} // catches any B or C exceptions
catch(A x) {} // catches any A, B or C exceptions
catch(object x) {} // catches any exceptions of class type
```

- Exceptions of non-class types are only caught if the `catch` block contains the exact same type. Type aliases, subtypes, `record`/`set`/`union`/`anytype` fields and `record of`/`set of`/array element types count as different types than their base types.

Example:

```
type integer IntAlias;
type integer SmallNumber (-100..100);
type record Rec {
  integer num
}
type record of integer RoI;
...
catch(integer x) {} // catches only integers, not IntAlias, SmallNumber, Rec.num or
RoI[-]
catch(RoI[-] x) {} // catches only the element type of RoI and no other types
```

- Exception lists can be added to `function`, `external function` and `altstep` declarations, but these are mostly ignored by the compiler (e.g. raised exceptions are not checked against the containing function/altstep's exception list). The exception lists are still checked for correctness (i.e. no duplicate types or invalid types).

4.39.5. Object templates

- Object templates are not the only matching mechanism that can be assigned to a template of class type. The matching mechanisms `?` (any value), `*` (any value or none), `omit`, as well as template lists, complement lists, conjunction lists, implication matching and dynamic matching can also be assigned to templates of class type. The `ifpresent` modifier can also be used.
- A specific value (i.e. a variable or parameter of class type, or the result of a class `create` operation) cannot be assigned to a template of class type.
- The `isvalue` operation always returns `false`, when used on a template of class type.
- These restrictions also apply to templates of structured types, that contain a class.
- If a property assignment is used in an object template, the getter of the property must be public. The property itself can have any visibility. The getter also does not have to be deterministic.
- If a function assignment is used in an object template, then that function may only have `in` parameters, and the types of these parameters must not be, or contain, a class type. (This is because `out`, `inout` parameters and parameters of class type are passed by reference, and the referenced values may have already been deallocated by the time they are used for matching.)
- The `object` method `toString` can be used in an object template, but the method `equals` cannot be used (because it has a parameter of class type).

4.40. Default alternatives of union types

TITAN supports the default alternatives of union types (i.e. the `@default` modifier) described in the TTCN-3 core language standard with the clarifications and limitations described in this section.

- Module parameter values and templates in the configuration file ignore the `@default` modifier. The union field must be specified explicitly, even if the default alternative is used.
- When encoding or decoding union values with pre-defined functions or built-in operators (e.g. `encvalue`, `decvalue`, `@decoded`, etc.), the union type is always the one being encoded or decoded, and not its default alternative (i.e. the `@default` modifier is ignored), even if the default alternative can be encoded with the specified codec and the union type cannot. Encoding and decoding with external functions do take the `@default` modifier into consideration.

Example:

```
type record Rec {
  integer num,
  charstring str optional
}
with {
  encode "JSON";
}

external function f_enc_rec(in Rec x) return octetstring
with { extension "prototype(convert) encode(JSON)" }

type union UniDefRec {
  @default Rec def,
  boolean bool
}

...

var UniDefRec val := { num := 3, str := "abc" };

var bitstring res1 := encvalue(val);
// produces an error, because type `UniDefRec` does not have any encoding specified

var octetstring res2 := f_enc_rec(val);
// successfully encodes the default alternative of `val`
```

- Sending and receiving operations on union values also ignore the `@default` modifier, even if the default alternative is in the port's incoming/outgoing list and the union type is not.

4.41. Advanced matching

TITAN supports the conjunction, implication and dynamic matching mechanisms described in the TTCN-3 standard extension `Advanced matching` (ETSI ES 203 022 V1.4.1, [28]) with the clarifications and limitations described in this section.

- Dynamic matching templates cannot be assigned to module parameters in the configuration file. They can be assigned as module parameter initial values in TTCN-3 modules.
- A dynamic template's statement block cannot contain the `@update` operation.
- The special keyword `value` within a dynamic template's statement block represents the value being matched. It behaves as a reference to an imaginary `in` formal parameter, whose name is `value` and whose type is the same as the type of the dynamic template. The value of `value` can be changed inside the statement block, which will not change the original value being matched.
- A `length` and/or `ifpresent` attribute at the end of an implication match template belongs to the second operand of the implication (i.e. the implied template) and not to the resulting template. If there are two sets of `length`/`ifpresent` attributes at the end of an implication match template, then the first set belongs to the second operand, and the second set belongs to the resulting template.

For example:

```
template charstring t1 := ? length (1..4) ifpresent implies ? length (2..3) ifpresent;
// the 2nd operand of t1 is '? length (2..3) ifpresent'
// and the resulting template has no 'length' or 'ifpresent' attributes

template charstring t2 := ? length (1..4) ifpresent implies ? length (2..3) ifpresent
length (3) ifpresent;
// the 2nd operand of t2 is '? length (2..3) ifpresent'
// and the resulting template has the attributes 'length (3) ifpresent'

template charstring t3 := (? length (1..4) ifpresent implies ? length (2..3)
ifpresent) length (3) ifpresent;
// t3 is the same as t2, but easier to read

template charstring t4 := ? length (1..4) ifpresent implies (? length (2..3)
ifpresent) length (3) ifpresent;
// the 2nd operand of t4 is interpreted as a value list match with 1 element,
// which cannot have the 'ifpresent' modifier, so this causes a semantic error
```

- If a type indicator precedes an implication match template, then the type indicator belonds to the type of the first operand (i.e. the precondition), not to the resulting template. Two type indicators cannot precede an implication match template.

For example:

```
log(match(3, integer: (1..3) implies integer: 3));
// the 1st operand of the matching template is 'integer: (1..3)'

log(match(3, integer: integer: (1..3) implies integer: 3));
// causes a parsing error
```

4.42. Logging sensitive data

There is a feature in TITAN for hiding sensitive data in the console output and log files. If a TTCN-3 type has the extension attribute `sensitive_data`, then the logs of all values and templates of this type will be replaced by the string "<redacted>".

This also affects the `log2str` function, but does not affect the `ttcn2string` function and the TTCN-3 debugger (both of which also use the logging feature to convert TTCN-3 values and templates into strings).

The run-time configuration file option `LogSensitiveData` in the `[LOGGING]` section can be used to enable or disable this feature. It's set to `No` by default, which means all data marked as sensitive is redacted. If set to `Yes`, then all data marked as sensitive is logged normally.

Example:

TTCN-3:

```
type record UserData {
  charstring   username,
  octetstring  secret_key
}
with {
  extension (secret_key) "sensitive_data"
}

...

var UserData x := { username := "User1", secret_key := '11223344'O }
log(x);
```

Logging result, if `LogSensitiveData` is set to `No` or is unset:

```
{ username := "User1", secret_key := <redacted> }
```

Logging result, if `LogSensitiveData` is set to `Yes`:

```
{ username := "User1", secret_key := '11223344'O }
```

NOTE Redacting sensitive data doesn't always happen automatically when logging from user C++ code, such as test port and external function implementations. See section "Logging sensitive data" in [16] for more details)

NOTE Sensitive data is not redacted from the result of encoding a value of a type that is at least partially sensitive (with any codec). If an `external function` is used for the encoding process, then the output string type can be replaced with a user-defined type that has the `sensitive_data` extension attribute, causing the entire encoding result to be redacted.

Chapter 5. Supported ASN.1 Constructs and Limitations

All kind of comments defined in X.680 clause 11.6 can be used ([6]).

All tagging environment is supported: `IMPLICIT`, `EXPLICIT` and also `AUTOMATIC`.

The type constraints are ignored. The BER (de)coding is not influenced by the constraints, except for the table constraints. For details, see section Using Component Relation Constraints from TTCN-3.

Table 11 summarizes how the different ASN.1 types are supported.

There is a special type: `ANY`. It has the same interface as the `OCTET STRING`, but during the decoding, it accepts any valid encoded message, and its value will be the complete TLV. This type is very useful if the protocol carries an encoded message.

Value sets are not supported as they are closely related to type constraints. Value set assignments in modules are parsed, but silently ignored. However, value set fields of information object classes (both fixed and variable type ones) cannot even be parsed, syntax errors are reported when processing such fields. As a consequence of this the information objects governed by such classes cannot be parsed either.

Variable type value fields of information object classes are not supported. Processing of the class definition results in syntax error.

A missing `IMPORTS` keyword implies an implicit import of all symbols from all modules according to the ASN.1 recommendation X.680 [4]. However, the missing `IMPORTS` keyword is interpreted by TITAN as an empty `IMPORTS`; keyword, thus, no symbols at all will be imported from any of the modules.

Table 11. Supported ASN.1 types

ASN.1 type	**syntactic check**		**semantic analyzing**		**code generation**		**(de)coding**
	type definition	value definition	typechecking	valuechecking	type definition	value assignment	
NULL	•	•	•	•	•	•	•
BOOLEAN	•	•	•	•	•	•	•
INTEGER	•	•	•	•	•	•	•
ENUMERATED	•	•	•	•	•	•	•
REAL	•	•	•	•	•	•	•[4]
BIT STRING	•	•	•	•	•	•	•

ASN.1 type	syntactic check		semantic analyzing		code generation		(de)coding
OCTET STRING	●	●	●	●	●	●	●
OBJECT IDENTIFIER	●	●	●	●	●	●	●
RELATIVE-OID	●	●	●	●	●	●	●
string[1]	●	●	●	●[6]	●	●	●
string[2]	●	●	●	●[6]	●	●	●[7]
string[3]	●	●	●	●[6]	●	●	●
CHOICE	●	●	●	●	●	●	●[5]
SEQUENCE	●	●	●	●	●	●	●[5]
SET	●	●	●	●	●	●	●
SEQUENCE OF	●	●	●	●	●	●	●
SET OF	●	●	●	●	●	●	●

● supported
○ not supported
1 IA5String, NumericString, PrintableString, VisibleString (ISO646String)
2 GeneralString, GraphicString, TeletexString (T61String), VideotexString
3 BMPString, UniversalString, UTF8String
4 only base 10 coding is supported
5 the ellipsis can be a problem during the decoding
6 the character repertoire is not checked
7 the conversion between ISO-10646 and ISO-2022 character stream is not fully implemented but can be overridden to meet the user's needs

Chapter 6. Compiling TTCN-3 and ASN.1 Modules

You can translate your TTCN-3 and ASN.1 modules to C++ source code using the program compiler.

6.1. Command Line Syntax

This section describes the options allowed in the command line of the compiler and the Makefile generator.

6.1.1. Compiler

The program compiler resides in the directory `$TTCN3_DIR/bin`.

The command line syntax of the compiler is the following:

```
compiler [ -abBcdDeEfFghHiIjklLMnNOpqrRstuwxXyY0 ] [ -J file ] [ -K file ] [ -z file ]
[ -N old|new ][ -o dir ] [ -V n ] [ -P toplevel pdu ] [ -Qn ] [ -U none|type|"number"
] [long options] ⋯[ -T ] module.ttcn [ -A ] module.asn ⋯ [ - module.ttcn module.asn
⋯ ]
```

or

```
compiler -v
```

or

```
compiler --ttcn2json [ -jf ] ⋯ [ -T ] module.ttcn [ -A ] module.asn ⋯ [ -
schema.json ]
```

or

```
compiler --asn1enc [ -bjop ] ⋯ module.asn ⋯
```

The compiler takes the name of files containing TTCN-3 and ASN.1 modules as arguments. The usual and recommended suffix is `.ttcn` for TTCN-3 and `.asn` for ASN.1 source files, but it is not stringent[9]. For TTCN-3 and ASN.1 modules, the names of the output files are the same as the name of the modules, except for the suffixes which are `.hh` and `.cc`.

NOTE In the ASN.1 module names hyphens are replaced by underscore character.

WARNING	If you have a modular test suite, you have to translate all modules of the test suite in one step, i.e. you have to specify all TTCN-3 and ASN.1 files in the argument list.

The switches denoted by square brackets are optional. More than one option may be merged for abbreviation. For example, `-r -u` has exactly the same meaning as `-ru`.

The following command line options are available (listed in alphabetical order):

- `-a`

  Enables the generation of Basic XML encoder/decoder functions for ASN.1 types. By default, these encoder/decoder functions are omitted because Basic XER has limited practical use.

- `-A, --force-asn1 file`

  Forces the interpretation of file as an ASN.1 module.

- `-b, --disable-ber`

  Disables the generation of BER encoder/decoder routines for all ASN.1 types.

- `-B`

  This is a legacy switch that allows the selected alternative in a `union` value to be unbound. This is only possible when initializing module parameters in the configuration file, and only if the selected alternative in question is a `record` or `set` (since initializing a record or set module parameter with empty braces {}, causes it to remain unbound).

  A warning is displayed whenever a `union` module parameter is initialized with an unbound selected alternative, and when a `union` with an unbound selected alternative is copied.

  This is only a temporary switch. It will be removed in future versions.

- `-c, --checksums`

  This switch is designed to help identifying compilation failures caused by mismatched versions of TTCN-3 and/or ASN.1 modules. If the compilation fails, the compiler will display the module checksums (and module version information, if available) computed during compilation.

- `-d, --default-omit`

  This switch changes the way fields of ASN.1 SEQUENCE /SET types with DEFAULT values are handled. Without `-d`, the runtime handles the encoding and decoding of default values in a way that is consistent with DER and CER. With `-"d` in effect, the ETS is responsible for the handling of fields with default values. This makes it possible to send encodings which are valid BER but not valid DER/CER and to verify that the SUT has performed the encoding correctly (note that without `-d`, the cases marked * and ** below cannot be distinguished in TTCN code).

Sending	Without -d	With -d
Explicit value	Send value	Send value
Default value	Do not send (omit)	Send default value
omit	N/A	Do not send (omit)

Receiving	Without -d	With -d
Receive explicit value	TTCN sees value	TTCN sees value
Receive default value	TTCN sees default value*	TTCN sees default value
Omitted	TTCN sees default value**	TTCN sees `omit'

For more details, see [14]

WARNING	Existing tests may behave differently when compiled with `-d`. Every behavior without `-d` can be duplicated when compiled with `-d`.

- `-D`

  Instructs the compiler to not generate the user and time information into the header of the generated .cc and .hh files.

- `-e`

  Instructs the compiler to use the legacy method of handling `encode` and `variant` attributes (see section Legacy codec handling).

- `-E`

  Instructs the variant attribute parser to display warnings instead of errors for unrecognized/erroneous encoding variants.

- `-f, --force-overwrite`

  Forces the compiler to overwrite (update) the output files even if they exist or the contents of them will be identical. Without this flag the output C++ header and source files will be overwritten only if their contents change compared to the previous version.

- `-F`

  Forces the compiler to generate the full C++ classes for `records of`/`sets of` basic types (i.e. `boolean`, `integer`, `float`, `bitstring`, `hexstring`, `octetstring`, `charstring` and `unversal charstring`). Otherwise only type aliases to pre-generated classes are generated for these types. This also disables the type compatibility between `records of`/`sets of` basic types in the Load Test Runtime (see Compatibility of record of types).

  As of compiler version /5 R3A (5.3.pl0) the C++ classes for the mentioned `record of`/`set of` types are pre-generated, and are part of the runtime library. Before that they were generated into the C++ code the same as for other structured types. This command line option is mostly a legacy

behavior switch, however the classes generated with this option are not identical to the classes generated with the old behavior (the new classes contain all codecs and the code needed to handle any possible coding instructions, not just the ones specified for the type).

This option is used internally for pre-generating the mentioned C++ classes.

- `-g`

  The compiler error/warning messages will contain the starting line number, and the starting column number if available. This option provides compatibility with the GNU compiler and many tools which are able to interpret its output (including Eclipse).

  If both `-g` and `-i` are specified, `-g` takes precedence.

- `-h, --allow-unsafe-conversion`

  Allow unsafe universal charstring to charstring conversion.

- `-H, --disable-per`

  Disables the generation of PER encoder/decoder routines for all ASN.1 types, and allows the user to implement these routines manually (legacy behavior).

- `-i`

  The compiler error/warning messages will contain only the line numbers, the column numbers will remain hidden. This option provides backward compatibility with the error message format of earlier versions.

- `-I, --enable-realtime`

  Enables the use of real-time testing features, i.e. the test system clock ('now') and timestamp redirects. These features are disabled by default for backward compatibility.

- `-j, --disable-encdec`

  Disables the generation of JSON encoder/decoder routines for all TTCN-3 types.

- `-k, --enable-oop`

  Enables the use of object-oriented features. These features are disabled by default for backward compatibility.

- `-K, --enable-coverage file`

  Enable code coverage for TTCN-3 modules listed in `file`. `file` is an ASCII text file which contains one `file` name per line. The set of files in file needs to be a subset of the TTCN-3 modules listed on the command line.

- `-J, --input-files file`

  The compiler will read the input files form `file` which must contain the input files separated by

spaces. Every file that is in the `file` is treated as if they were passed to the compiler directly. It is possible to use the -A and -T flags to tell the compiler that a file is an ASN.1 or a TTCN-3 file.

Example:

```
compiler Myttcn.ttcn Myasn.asn -J files.txt
```

where the contents of the `files.txt` is the following:

```
First.ttcn Second.asn -T Third.ttcn -A Fourth.asn
```

The command above is equivalent to this command:

```
compiler Myttcn.ttcn Myasn.asn First.ttcn Second.asn -T Third.ttcn -A Fourth.asn
```

Because of the `-T` flag the `Third.ttcn` will be treated as a TTCN-3 file, and because of the `-A` flag the `Fourth.asn` will be treated as an ASN.1 file.

- `-l, --include-source-info`

  Instructs the compiler to generate source file and line information (that is, #line directives) into the output code so that the error messages of the C++ compiler refer back to the lines of the original TTCN-3 input module. This makes finding the reason of C++ error messages easier. This option has effect only in the equivalent C++ code of TTCN-3 functions, test cases and control parts and this feature is not provided in other TTCN-3 definitions such as types, constants or templates.WARNING! This is an experimental feature and the C++ compiler may report misleading error messages that refer to totally wrong (e.g. non-existent) TTCN-3 line numbers. In these cases please turn off this flag, repeat the compilation and analyze the generated code manually. Without this flag, the compiler also inserts the source code information for the better understanding of C++ error messages, but only as C++ comments. This option has no impact on the run-time performance of the generated code. The compiler performs full semantic analysis on its entire input; it normally does not generate erroneous C++ code. So this option became obsolete and will be removed in future versions.

- `-L`

  Instructs the compiler to add source file and line number information into the generated code to be included in the log during execution. This option is only a prerequisite for logging the source code information. The run-time configuration file parameters `OptionsSourceInfoFormat` and `LogEntityName` in `[LOGGING]` have also to be set appropriately. This feature can be useful for finding the cause of dynamic test case errors in fresh TTCN3 code. Using this option enlarges the size of the generated code a bit and reduces execution speed slightly; therefore it is not recommended when the TTCN3 test suite is used for load generation.

- `-M`

  Enforces legacy behavior when matching the value `omit`. Allows the use of the value `omit` in

template lists and complemented template lists, giving the user another way to declare templates that match omitted fields. If set, an omitted field will match a template list, if the value `omit` appears in the list, and it will match a complemented template list, if `omit` is not in the list (the `ifpresent` attribute can still be used for matching omitted fields). This also affects the `ispresent` operation and the `present` template restriction accordingly.

- `-n, --activate-debugger`

  Activates the debugger and generates extra code needed for gathering debug information and for inserting breakpoints into the TTCN-3 program.

- `-N`

  Ignore `UNTAGGED` encoding instruction applied to top level union types when encoding or decoding with XML. Legacy behavior.

- `-o, --output-dir dir`

  The output files (including Test Port skeletons) will be placed into the directory specified by `dir`. Otherwise, the current working directory is the default.

- `-O, --disable-oer`

  Disable the generation of OER encoding and decoding functions.

- `-p, --parse-only`

  Instructs the compiler only to parse the given TTCN-3 and ASN.1 modules. This will detect only the syntax errors in them because semantic checks are not performed. The presence of all imported modules is not necessary, thus, it is allowed (and recommended) to parse the modules one-by-one. All options that influence the code generation are silently ignored when used together with `-p`.

NOTE	This option includes complete syntax checks for TTCN-3 modules, but in ASN.1 there are some special constructs (e.g. the user-defined syntaxes) that cannot be parsed without semantic analysis. So there is no guarantee that an ASN.1 module is free of syntax errors if it was analyzed with compiler using the -p flag.

- `-P, --top-level-pdu top level pdu …`

  Defines a top-level pdu that must have the format `modulename.identifier`. If this switch is used, then only the defined top-level PDU(s) and the referenced assignments are checked and included in code generation, the other ASN.1 assignments are skipped. You can define not only types but any kind of ASN.1 assignments.

- `-q, --quiet`

  Quiet mode. Equivalent to the flag `-V 0`.

- `-Q, --quit-after n`

Quit after n errors (n must be a positive integer). The compiler will abort after encountering the specified number of errors.

NOTE	Errors count is cumulative across all modules. Using this option may cause some modules not to be analyzed at all (if previous modules "used up" all the allowed errors).

- `-r, --disable-raw`

  Disables the generation of RAW encoder/decoder routines for all TTCN-3 types.

- `-R, --function-test-runtime`

  Instructs the compiler to generate code for use with the function test runtime. The size of the generated code is significantly reduced, much of the functionality was migrated to the runtime. The generated C++ code has to be compiled using the TITAN_RUNTIME_2 symbol and has to be linked with the function test version of the runtime library. For example instead of the library file libttcn3.a the alternative libttcn3-rt2.a file must be used. The included c++ header files are the same.

- `-s, --semantic-only`

  Instructs the compiler to parse the given TTCN-3 and ASN.1 modules and perform semantic analysis on them, but not to generate C++ output. The list of given modules shall be complete so it is not allowed to import from a module that is not in the list. All options that influence the code generation are silently ignored when used together with `-s`.

NOTE	The TTCN-3 semantic analyzer of the compiler is still under development, thus, it is not capable of detecting every kind of semantic error.

- `-S, --suppress-context-info`

  Suppress context information. When the compiler reports an error or warning, it also reports context information (italic in the example below):

  ```
  quitter3.ttcn: In TTCN-3 module `quitter3':
   quitter3.ttcn:11.3-23.3: In control part:
    quitter3.ttcn:12.11-30: In variable definition `v_r':
     quitter3.ttcn:12.20-28: error: Reference to non-existent field `z' in record
  value for type `@quitter3.R'
  ```

  The `-S` option causes the compiler to output only the error (last line), without the preceding lines of context.

- `-t, --generate-skeletons`

  Generates Test Port skeleton header and source files for all port types in the input TTCN-3 modules. Existing Test Port files will not be overwritten unless the -f option is used.

NOTE	In versions 1.1 and earlier this was the default behavior of the compiler, but if the existing Test Port source files were stored in a different directory, the generated new skeletons could be confusing.

- `-T, --force-ttcn3 file`

  Forces the interpretation of file as a TTCN-3 module.

- `-u, --duplicate-underscores`

  Forces the compiler to insert duplicated underscore characters in all output file names. This option turns on the compatibility mode with versions 1.1 or earlier.

- `-U, --code-splitting none|type|"number"`

  Selects the code splitting mode for the generated code. This option is position dependent, i.e., the modules/files listed before it will not be split, consequently the modules listed after it will be split. If selective code generation (“-”) is not used, all modules/files will be split. The option "none" means that the traditional code generation method will be used. When using the option "type", TITAN will create separate source files for the implementation code of the following types (for the selected modules): sequence, sequence of, set, set of, union. In this case a common header file and a source file holding everything else will also be created. The option can also be a positive number. In that case each file will be split into “number” smaller files. The compiler tries to create files which have equal size and empty files may be created. The “number” parameter must be chosen carefully to achieve compilation time decrease. The “number” parameter should not be larger than the number of the CPU cores. This splitting mode provides decreased compilation time, if the compilation is parallelized. For example, this can be achieved using the **make** command’s **-j** flag which needs a number argument that controls how many cores the compilation may use. This number should be equal to the “number” parameter. Furthermore, it is also useful to split ASN.1 files, containing a lot of type definitions, into smaller pieces. According to our experiences, in such cases the generated C++ code can grow beyond a point, where GCC starts to consume extra resources, while the compilation time significantly increases. The code splitting option can help to avoid such situations. Selecting the optimal number is the subject of experimentation. An example can be found here about TITAN’s strategy when splitting the files using the “number” parameter.

- `-v, --version`

  Prints version and license key information and exits.

- `-V, --verbosity n`

  Sets the verbosity bit-mask directly to `n` (where n is a decimal value between 0 and 65535). Meaning of the bits:

  1: "NOT SUPPORTED" messages.

  2: WARNING messages.

  4: NOTIFY messages.

32 | 16 | 8: DEBUG messages. The debug-bits act like a 3-bit-length number, so the debug level has a value between 0 and 7. It is useful in case of abnormal program termination.

NOTE	When only parsing (option -p) DEBUG messages for ASN.1 values will appear in TTCN-3 form (e.x.: booleans will appear as `true` or `false`, instead of `TRUE` or `FALSE`).

Example: If you use the option `-V 6`, then all NOTIFY and WARNING messages will be printed, but the "NOT SUPPORTED" messages will be suppressed. To have the most detailed log, use the option `-V 63`. The default is `-V 7`.

- `-w, --suppress-warnings`

  Suppresses all warning messages. Equivalent to `-V 4`.

- `-x, --disable-text`

  Disables the generation of TEXT encoder/decoder routines for all TTCN-3 types.

- `-X, --disable-xer`

  Disables the generation of XER encoder/decoder routines for all TTCN-3 types.

- `-y, --disable-subtype-check`

  Disable subtype checking. Subtype information is parsed but ignored, there is no semantic check of the parsed subtype data.

- `-Y, --legacy-out`

  Enforces legacy behaviour for `out` function parameters (`out` parameters will behave as `inout` parameters).

- `-z, --enable profiling file`

  Enables code coverage and profiling in the TTCN-3 files listed in the `file` argument. The TTCN-3 files in the list must be separated by new lines and must also appear among the compiler's arguments.

- `-0`

  Disables attribute checks for `encvalue` and `decvalue`. Must be used together with option `-s`.

- `-`

  The single dash character as command line argument has a special meaning: it controls the selective code generation. After the list of all TTCN-3 and ASN.1 modules a subset of these files can be given separated by a dash. This option instructs the compiler to parse all modules, perform the semantic analysis on the entire module hierarchy, but generate code only for those modules that are listed after the dash again. It is not allowed to specify a file name after the dash that was not present in the list before the dash. If the single dash is not present in the

command line the compiler will generate code for all modules.

- `--legacy-template-subrefs`

  Enable legacy handling of non-specific template subreferences. Details can be found here.

- `--legacy-out2`

  Enforce legacy behaviour (2nd version) for `out` function parameters (`out` parameters will behave as `inout` parameters, and are set to `unbound` at the start of the function).

- `--ttcn2json`

  This option is obsolete. Use `--generate-json` for more flexibility.

  Changes the purpose of the compiler to generate a JSON schema from the types defined in the specified TTCN-3 and ASN.1 modules. The parsing and semantic check of the input modules is still executed, but a JSON schema is generated instead of the C++ files. This must always be the first compiler option, and the previously listed options don't apply (apart from options `-A` and `-T`), instead the following options are available:

  - `-j`

    Only types that have JSON coding enabled are included in the generated schema.

  - `-f`

    Only types that have JSON encoding or decoding functions (or both) are validated by the schema.

- `--generate-json`

  Changes the purpose of the compiler to generate a JSON schema from the types defined in the specified TTCN-3 and ASN.1 modules. The parsing and semantic check of the input modules is still executed, but a JSON schema is generated instead of the C++ files. Previously listed options don't apply (apart from options `-A` and `-T`), instead the following options are available:

  - `-j`

    Only types that have JSON coding enabled are included in the generated schema.

  - `-f`

    Only types that have JSON encoding or decoding functions (or both) are validated by the schema.

- `- file`

  The single dash character as command line argument can be used to specify the name of the generated JSON schema file (it must be followed by exactly one argument, which is the file name). If it is not present, the schema file name is generated from the name of the first input file (the extension ".ttcn" or ".asn" is replaced by ".json", or ".json" is appended to the end of the

file name if neither extension is present).

- `--asn1enc`

  This option is obsolete. Use `--asn1-only` for more flexibility.

  Activates the standalone ASN.1 encoder/decoder operating mode of the compiler. This generates the minimum code needed to encode/decode ASN.1 values. The parsing and semantic check of the input modules is still executed, but all input files are treated as ASN.1 modules. This must always be the first compiler option, and the previously listed options don't apply, apart from the options for disabling the generation of `BER`, `JSON` and `OER` encoder/decoder functions (`-o` is used for `OER` instead of `-O`). The option for disabling `PER` encoder/decoder functions is `-p`. See [30] for defails about the Standalone ASN.1 Encoder.

  The meaning of options is also included in the manual page of the compiler. If your TTCN-3 installation is correct, the command `man compiler` will show you this manual page. If not, check the `MANPATH` environment variable.

- `--asn1-only`

  Activates the standalone ASN.1 encoder/decoder operating mode of the compiler. This generates the minimum code needed to encode/decode ASN.1 values. The parsing and semantic check of the input modules is still executed, but all input files are treated as ASN.1 modules. Previously listed options don't apply, apart from the options for disabling the generation of `BER`, `JSON` and `OER` encoder/decoder functions (`-o` is used for `OER` instead of `-O`). The option for disabling `PER` encoder/decoder functions is `-p`. See [30] for defails about the Standalone ASN.1 Encoder.

6.1.2. Makefile Generator

You can generate the Makefile skeleton used for building the test suite using the program `ttcn3_makefilegen`, which resides in the directory `$TTCN3_DIR/bin`. See section 2.3.1 of the TITAN User Guide [13] for details. The generated Makefile skeleton will use the parallel mode of the run-time environment by default. This can be overridden by using the option `-s` (see below).

The command line syntax of the makefile generator is the following:

```
usage: makefilegen [-abcdDEfFgGhiklLmMnNprRsStTVwWXZ] [long options] [-K file] [-P
dir]
[-J file] [-U none|type|"number"] [-e ets_name] [-o dir|file] [-z file]
[-t project_descriptor.tpd [-b buildconfig]] [-I path] [-O file] [-x "xsd2ttcn_flags"]
TTCN3_module[.ttcn] ... ASN1_module[.asn] ... XSD_MODULE.xsd ... Testport_name[.cc]
...
```

or

```
makefilegen -v
```

The ttcn3_makefilegen tool is able to process XSD modules along with TTCN3 or ASN1 modules. The

`ttcn3_makefilegen` tool first translates the XSD modules into TTCN3 modules using the `xsd2ttcn` tool. The XSD modules will be parsed and the information which is needed for the Makefile generation will be extracted from them. It is a requirement that the XSD modules MUST be partially syntactically correct (the `schema` tag must be correct). Is is also possible to configure the `xsd2ttcn` tool by adding arbitrary flags using the `-x` option.

The command line switches for generating Makefile(s) from Titan Project Descriptor (TPD) file(s) are not listed here, these are described in the next chapter.

The switches denoted by square brackets are optional. More than one option may be merged for abbreviation. For example, `-g -p` has exactly the same meaning as `-gp`. For a couple of options, a long version is also available for better readability (for example `--enable-oop` instead of `-k`) The following command line options are available (listed in alphabetical order):

- `-a`

  The flag refers to files using absolute path names in the generated Makefile. Makefile uses by default relative path names to access files located outside the current working directory of the compiler. Files from the current working directory are always referenced using only the file name without any directory. The flag generates a Makefile that is capable of using pre-compiled C++ and object files from central directories in order to save disk space and compilation time. WARNING! This feature works only if the generated files of the central directory is kept up-to-date and all directories use exactly the same environment (platform, TTCN-3 Executor and C++ compiler version, etc.).

- `-c`

  Treat files not in the current directory as being in “central storage“. These files are assumed to be already built in their (separate) location.

- `-d`

  Dumps the data used for Makefile generation.

- `-e <ETS name>`

  Sets the name of the target binary program (i.e. the executable test suite) to <ETS name> in the generated Makefile. If this option is omitted, the name of the first TTCN-3 module will be used as default.

- `-E`

  Instructs the variant attribute parser to display warnings instead of errors for unrecognized/erroneous encoding variants.

- `-f`, `--force-overwrite`

  Forces the makefile generator to overwrite the output Makefile. Without this flag the existing one will not be overwritten.

- `-g`, `gnu-make`

Generates a Makefile that can be used with GNU `make` only. The resulting Makefile will be smaller and less redundant. It exploits the pattern substitution features of GNU `make`, which may cause syntax errors with other versions of make. By default, Makefiles generated for GNU `make` use incrementally updated dependency files instead of `makefilegen`.

- `-G`

  Instructs the compiler to use the legacy encoding rules for semantic checking and for generating the code (see compiler option "-e" and its description in Legacy codec handling).

- `-h`

  Allow unsafe universal charstring to charstring conversion.

- `-i`, `--enable-realtime`

  Enables the use of real-time testing features, i.e. the test system clock ('now') and timestamp redirects. These features are disabled by default for backward compatibility.

- `-I path`

  Add path to the list of search paths which are used to search for referred TPD-s. `path` must be an absolute path and multiple `-I` switches can be used. The search paths are used when the `-t` switch is also present and a referred TPD project cannot be found at the location specified by `projectRelativeURI`. In this case the `makefilegen` tool tries to find the TPD file using the paths provided by `path`. If the `tpdName` attribute of a `ReferencedProject` element is present in the TPD, then the value of the `tpdName` attribute will be used as a TPD filename during the search. However if the `tpdName` attribute is missing, then the name attribute's value with the `.tpd` suffix will be used. If there are multiple paths specified then the first `.tpd` file with the correct name found in the list of search paths is taken. See 6.1.3.4 for an example.

- `-J file`

  The `makefilegen` tool will read the input files form `file` which must contain the input files separated by spaces. Every file that is in the `file` is treated as if they were passed to the `makefilegen` tool directly.

- `-k`, `--enable-oop`

  Enables the use of object-oriented features. These features are disabled by default for backward compatibility.

- `-K file`

  Enable code coverage for TTCN-3 modules listed in `file`. `file` is an ASCII text file which contains one file name per line. The set of files in `file` needs to be a subset of the TTCN-3 modules listed on the command line. (This parameter is simply passed to the TTCN-3 compiler through `COMPILER_FLAGS` in the Makefile.)

- `-l`, `--enable-dynamic-linking`

Enable dynamic linking. All files of the project will be compiled with `-fPIC` and for each (static) object, a new shared object will be created. Then, these shared objects will be linked to the final executable instead of the (static) objects. It can be used to speed up the linking phase, in the price of somewhat lower performance and increased memory usage. It's recommended to use this flag only in the development phase of the project. Please note, that both the project's working directory (which holds the TITAN generated `.so` files) and the directory of TITAN libraries, most probably `${TTCN3_DIR}/lib`, should be appended to the `LD_LIBRARY_PATH` environment variable. Otherwise, the dynamic linker (or loader) won't be able to locate the shared objects required by the executable. This option is not supported on Windows (platform string `WIN32`).

- `-L`

  Create the makefile with library as the default target. The name of the generated library archive is <ETS name>_lib.so if the dynamic linking is enabled and <ETS name>.a otherwise.

- `-m`

  Always use `makedepend` for dependencies. By default, for makefiles used by GNU `make`, the compiler (usually GCC) is used to generate dependency information in an incremental fashion. This switch reverts to the process for generic make tools, which use the `makedepend` tool. This switch has no effect if switch `-g` (GNU make) is not set.

- `-M`

  Enforces legacy behavior when matching the value `omit`. Allows the use of the value `omit` in template lists and complemented template lists, giving the user another way to declare templates that match omitted fields. If set, an omitted field will match a template list, if the value omit appears in the list, and it will match a complemented template list, if `omit` is not in the list (the `ifpresent` attribute can still be used for matching omitted fields). This also affects the `ispresent` operation and the `present` template restriction accordingly.

- `-n`

  Activates the debugger and generates extra code needed for gathering debug information and for inserting breakpoints into the TTCN-3 program.

- `-N`

  Ignore UNTAGGED encoding instruction applied to top level union types when encoding or decoding with XML. Legacy behavior.

- `-o <dir> | <file>`

  Writes the Makefile to the given directory or file. If the given argument is an existing directory, the generated Makefile will be placed into that directory. Otherwise, it is assumed to be the name of the generated Makefile. By default the file name is `Makefile`, placed in the current working directory.

- `-O <file>`

Add file to the list of other files in the generated `Makefile` without analyzing the file contents and suffix. This option can be used to temporarily exclude some TTCN-3, ASN.1 modules ASN.1 or C++ files from the build process, but add them to the archive created by the command make archive.

- `-p`

  Generates `Makefile` with TTCN-3 preprocessing. All the TTCN-3 source files with the suffix `.ttcnpp` will be preprocessed using the C preprocessor. For details see Using the TTCN-3 Preprocessing Functionality.

- `-R`, `--function-test-runtime`

  Use function test runtime (TITAN_RUNTIME_2). Generates a Makefile that compiles and links the source code using the function test runtime.

- `-s`

  Generates a `Makefile` skeleton to be used with the single mode of the run-time environment. The only difference between the Makefiles for single and parallel modes is the setting of variable `$TTCN3_DIR` within them.

- `-S`, `--suppress-warnings`

  Suppresses all warning messages generated by the `makefilegen` tool.

- `-U none|type|"number"`

  Generates a `Makefile` skeleton to be used with the chosen code splitting option. For details see the compiler options in 6.1.1.

- `-v`

  Prints version and license key information and exits.

- `-w`, `--suppress-compiler-warnings`

  Suppresses all warning messages generated by TITAN compiler. This flag overrides the `suppressWarning` option in the `.tpd` file.

- `-x "xsd2ttcn_flags"`

  Valid flags for the `xsd2ttcn` tool. If the tool is not used, i.e. no XSD module is specified for the Makefile generator, it does not have any effect.

- `-Y`

  Enforces legacy behaviour for `out` function parameters (`out` parameters will behave as `inout` parameters).

- `-z file`

Enables code coverage and profiling in the TTCN-3 files listed in the `file` argument. The TTCN-3 files in the list must be separated by new lines and must also appear among the makefile generator's arguments (this switch is ignored if the -t option is present).

- `--legacy-per`

  Disables the generation of PER encoder/decoder routines for all ASN.1 types, and allows the user to implement these routines manually (legacy behavior).

- `--legacy-template-subrefs`

  Enable legacy handling of non-specific template subreferences. Details can be found here.

- `--legacy-out2`

  Enforce legacy behaviour (2nd version) for `out` function parameters (`out` parameters will behave as `inout` parameters, and are set to `unbound` at the start of the function).

  If any of the source (TTCN-3, ASN.1, user-written C++) files does not exist or cannot be accessed, `ttcn3_makefilegen` exits with an error.

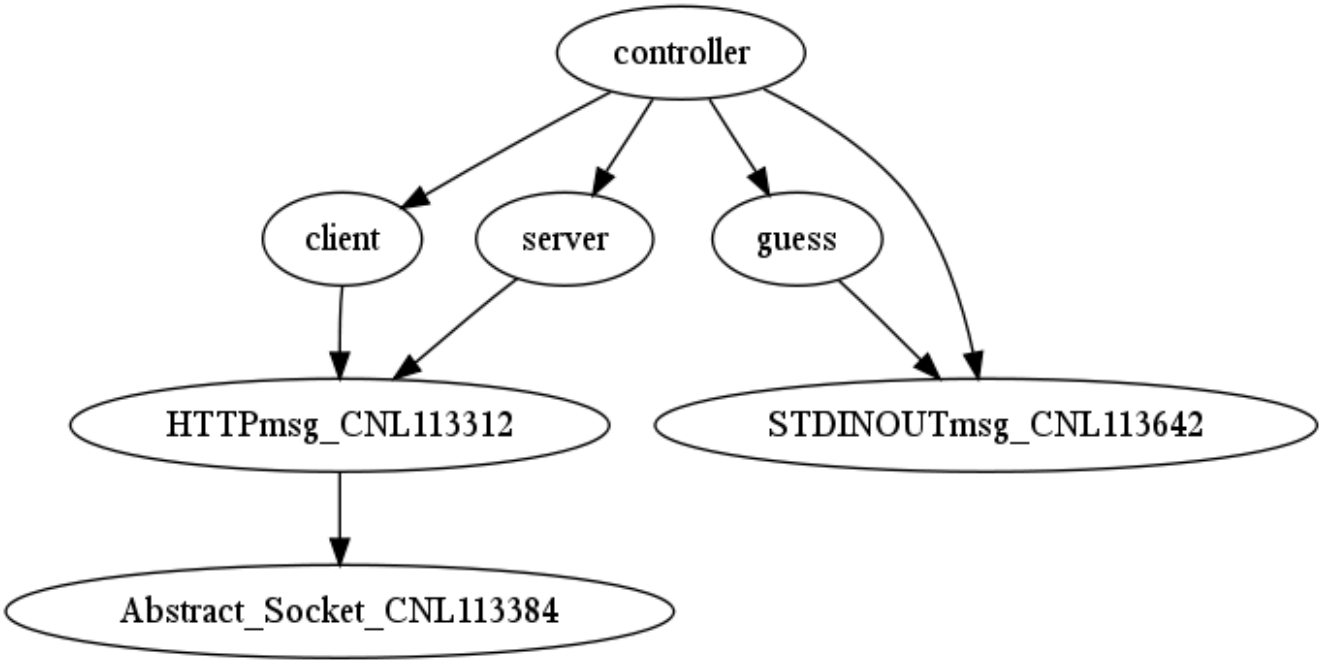
  Other options are discussed in the next chapters.

6.1.3. Using the Makefile Generator to Generate Makefile(s) from Titan Project Descriptor file(s)

It is possible to generate Makefile(s) from command line using the Titan Project Descriptor file(s) (TPD) generated by the Eclipse plugin. The Eclipse plugin generates a TPD file for each project. This file contains all the information needed to generate a Makefile for that project. See reference [17] for details.

The makefile generator validates the TPD file with a schema file which is located at `${TTCN3_DIR}/etc/xsd/TPD.xsd`. If there are validation errors or the xsd file cannot be found some warnings will be displayed, this validation can be disabled with the "-V" option. Validation errors will not prevent the generation of makefiles and symlinks, however if there are such warnings it is strongly recommended to check the TPD files for errors because these errors may cause either other errors during the generation of the makefiles and symlinks or the creation of invalid makefiles and symlinks.

Projects can reference other projects. These dependencies between projects are contained by the generated TPD files. The TPD file is placed in the project root directory. Every project has also a working directory (usually named `bin') which is relative to the project root directory. The working directory will contain symlinks to all the source files contained by the project and the files generated when building the project. The TPD file of the project contains the names and relative paths of all the projects that the project depends on, therefore the relative location of these projects must not be changed or these won't be found. For large projects the TPD files will describe a project hierarchy that is not necessarily a tree structure, for example:



The command line makefile generator can process the TPD file hierarchy generated by Eclipse and generate one or more Makefiles for these. There are three methods to generate Makefiles:

1. Generate a single Makefile that will contain all files from all projects. The following command line options can be used for this: `-t -b -D -P -V -W`. When using this method the -c switch should not be used because in this case all the files are seen as part of one large project.
2. Generate a Makefile hierarchy recursively (-r): for each TPD file generate a Makefile that builds only one project but has a dependency on other projects using the feature called "central storage". This dependency relation in the Makefile means that prior to building a project all the other projects that it depends on must be built. The dependency relation is contained by the top-level project's Makefile. For that to work the central storage (`-c` switch in the makefile generator) feature is used to avoid compiling the source files also in top level projects that have already been compiled in the sub-projects where they belong to. Using this one Makefile all the projects can be built in the proper order. The following command line options can be used for this: `-t -b -D -P -F -T -X -V -W -Z -H`.
3. Generate a Makefile hierarchy with improved linking method (`-Z`): for each TPD file generate a Makefile that builds only one project but has a dependency on other projects. It provides highly flexible way of linking static- and/or dynamic libraries together. The following command line options are obligatory `-t -Z -F -r` and these are optional: `-H -W`.

When generating multiple Makefiles the working directories of each referenced project are determined by the TPD file of the project. The TPD file can contain more than one build configuration, but there's always one active configuration. The active configuration is determined by the TPD file itself by the element `<ActiveConfiguration>`. Different build configurations can have different working directories and other settings. This can be overruled by the referencing project's required configuration setting (via `<requiredConfiguration>` embedded into `<configurationRequirement>`) or in case of a top-level TPD by using the -b command line option. Both the Makefile and the symlinks to source files are generated into the working directory.

If there is no "workingDirectory" specified in the TPD file, default directory will be created with name "bin". If more than one project define the same directory for working directory a collision can

happen. This can be avoided by the command line switch -W (see below).

If you want to generate Makefiles from tpd files with incremental dependency (via .d files), you shall apply switch -g and you must not apply -m, in addition to these the top level project descriptor (tpd) file shall contain the element ordering incremental dependency as follows:

```
<incrementalDependencyRefresh>
true
</incrementalDependencyRefresh>
```

The following TPD related command line options are available:

- `-t filename.tpd [-b buildconfig]`

  Use the supplied Titan Project Descriptor file to generate the Makefile. Information about directories, files and certain options is extracted from the TPD file and used to generate the Makefile(s). Additional files can be specified on the command line. Options from the command line override any option specified in the TPD file. If hierarchical makefilegen is ordered (-Frcg or -FrcgWp) then the immediately referred projects will be generated according to the element <requiredConfiguration> of the ordered top level active configuration. This is applied recursively.

- `-b buildconfig`

  On top level use the specified build config instead of the default active configuration defined in the TPD.

- `-r`

  Generate a Makefile hierarchy recursively from TPD hierarchy.

- `-P <dir>`

  Print out a file list found in a given TPD and optionally in its referenced TPDs to `stdout`, relative to the given directory `<dir>`. It requires the `-t` option and a valid directory as its parameter. If used together with the `-a` option the list will contain absolute paths and the directory parameter will not be taken into account.

- `-V`

  Disable validation of TPD file with the TPD.xsd schema file

- `-F`

  Force the makefile generator to overwrite all output Makefiles. Without this flag the existing files in the Makefile hierarchy will not be overwritten.

- `-T`

  Generate only the top-level Makefile of the hierarchy.

- `-X`

  Generate an XML file that describes the project hierarchy as seen by the makefile generator.

- `-D`

  Use current directory as working directory.

NOTE	This flag overrides the working directory coming from the TPD. In case of Generate Makefile hierarchy recursively (`-r`) flag, `-D` flag is valid only for top level project.

- `-W`

  Prefix working directories with project name. This is needed if more than one TPD file is in the same directory, because this directory will then be the root of more than one project. The working directories (usually `bin') will conflict, by using -W the working directory name will be prefixed by the project name, for example `MyProject_bin'.

NOTE	In case of incorrect TPD files, the errors are displayed multiple times if the faulty TPD is included more than once in the project structure.

- `-Z`

  Use the improved linking method. It generates a flexible hierarchy of static and dynamic libraries. Each project can be set to build a static or dynamic library or an executable too.

- `-H`

  Use hierarchical project generation. Use it with -Z option. It provides makefiles for generating hierarchical binaries without flattening the project hierarchy. make can be called in any working directory, the lower level projects will be handled properly. All project can be regarded as top level project. Execution time of make is higher than in case of applying -Z without -H. (The difference is 50-100% for top level modification, 0-10% for lower level modification.)

Examples:

1. Hierarchical makefile file generation from the directory of the top level project:

   ```
   makefilegen -FrcgWp -t MyTopLevelProj.tpd
   ```

2. Project hierarchy file generation:

   ```
   makefilegen -rcX -t MyTopLevelProj.tpd
   ```

3. Hierarchical makefile file generation from the directory of the top level project:

```
makefilegen -FWrZH -t MyTopLevelProj.tpd
```

4. Generate list of files of all hierarchical projects: Go to the folder of your top level tpd file (or to the root directory of your projects) then use the following bash command:

```
makefilegen -V -P $(pwd) -t TopLevelProj.tpd
```

5. Create archive file of all files in all hierarchical projects: Go to the root directory of your projects then use the following bash command:

```
makefilegen -V -P $(pwd) -t path/to/your/TopLevelProj.tpd | xargs tar cfz
YourArchive.tgz
```

Using the improved linking method (-Z and -H option)

Node <ReferencedProjects> contains the projects whose <defaultTarget> is either a library (static or dynamic) or an executable. See the XML excerpt.

```
<ReferencedProjects>
    <ReferencedProject name="refProj1"
projectLocationURI="../workspace/refProjDir1/refProj1.tpd"/>
    <ReferencedProject name="refProj2"
projectLocationURI="../workspace/refProjDir2/refProj2.tpd"/>
</ReferencedProjects>
<MakefileSettings>
  <GNUMake>true</GNUMake>
  <incrementalDependencyRefresh>true</incrementalDependencyRefresh>
  <dynamicLinking>true</dynamicLinking>
  <defaultTarget>library</defaultTarget>
  <targetExecutable>bin/yourexecutable</targetExecutable>
  <linkerLibraries>
        <listItem>externallib1</listItem>
  </linkerLibraries>
  <linkerLibrarySearchPath>
        <listItem>${externallib}/lib</listItem>
  </linkerLibrarySearchPath>
</MakefileSettings>
```

"refProj1" and "refProj2" are subprojects of the actual one. Other info about these subprojects can only be obtained in their own TPD file. <incrementalDependencyRefresh> is set to true in the project structure. <GNUMake> shall be set to true. In this scope other tools are not supported. The node <dynamicLinking> true sets the dynamic linking method for the actual project. The node <defaultTarget> indicates whether the output is a library. If it is omitted the output is an executable.

<linkerLibrarySearchPath> and <linkerLibraries> provide information about third party (not in the

project hierarchy) libraries.

The solution is based on the following corner stones:

Static and dynamic libraries can only be linked on <defaultTarget> executable build level. This means that a <defaultTarget> library cannot be generated by mixing other static and dynamic libraries.

A <defaultTarget> library with dynamic linking can be generated only from its own project's object file(s) and subprojects dynamic libraries.

Static libraries are so called thin archives. This means that a static library is generated from own projects's object file(s) and contains links to object files of other thin archive(s).

Third party libraries (e.g.: Linux core libs, openssl) can only be linked dynamically.

If the <defaultTarget> is library and <dynamicLinking> is true, the following aspects are to be considered:

- it can be linked together with another dynamic library
- it cannot be linked together with a static library
- it can be linked together with a third party dynamic library (e.g. openssl)
- it cannot have subproject(s) with <defaultTarget> is executable

Position dependent code cannot be linked with position independent code. This is a known limitation of the GNU linker. The third party libraries shall be added to LD_LIBRARY_PATH, or be copied to a directory contained by the LD_LIBRARY_PATH

If the <defaultTarget> is library and <dynamicLinking> is false, the following aspects are to be considered:

- it can be linked together with another static library
- it cannot be linked together with a dynamic library
- it cannot be linked together with a third party static library[10] (e.g. openssl)
- it can have subproject(s) with <defaultTarget> is executable

If the project's <defaultTarget> is an executable, then the static and dynamic libraries can be linked together. If on a lower level project there is reason to link static and dynamic libraries together, then the node <defaultTarget> shall be set to executable, too. If -H option is NOT set then NO executable file will be generated for lower level projects. In this case the Makefile will generate only objects. The top-level project's <defaultTarget> shall be set to executable. This is not checked if the -H option is set, since it causes every node to behave as if it were the top-level node.

Important: within a Project hierarchy if the real top-level project with <defaultTarget> executable is set to <dynamicLinking> true, then every sublevel project with <defaultTarget> executable shall be set to <dynamicLinking> true as well. A top-level project with <defaultTarget> executable and <dynamicLinking> false has no such constraint. If the above requirements are not fulfilled it results in a linker error. The Project hierarchy cannot contain circular references otherwise an error will

be displayed.

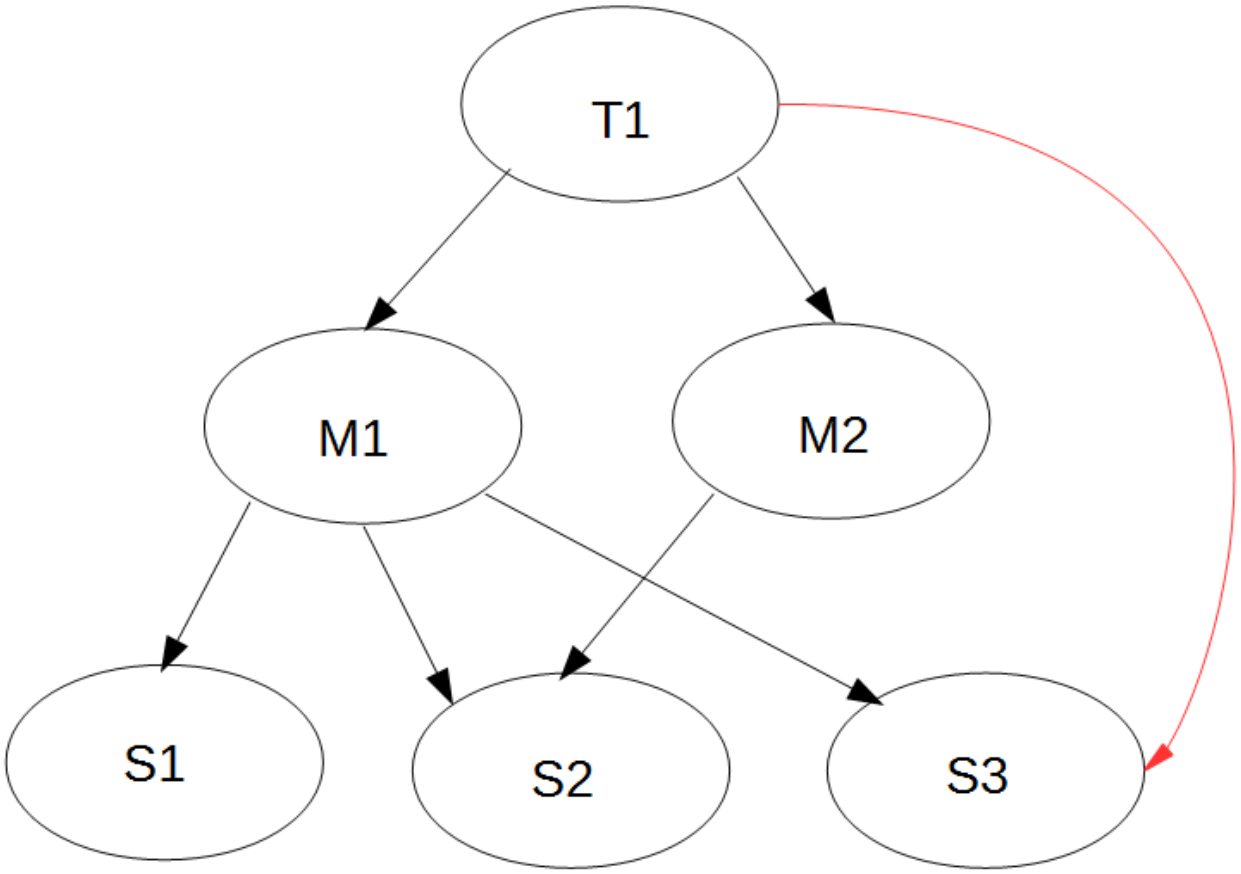
The makefile uses the linker flag -rpath, so there is no need to set the environment variable LD_LIBRARY_PATH for the libraries in the project hierarchy.

If option -H is used: There is a new make command in the makefile that is generated by using the H flag. The call of **make clean-all** cleans the whole hierarchy, whereas the behavior of **make clean** changed, it only cleans the actual project.

If option -H is not used: In a cleaned Makefile structure the **make** shall be called in the working directory of the top-level project.

The optimal TPD for hierarchical structure (-H option)

The following picture shows a simple Project structure:



The arrows show the Project references. T1 has two `<ReferencedProjects>` nodes in the TPD: M1 and M2. M1 has three: S1, S2 and S3, and so on. Since the structure is hierarchical S2 will be iterated twice. M1-S2 dependencies make S2 be compiled and linked. The makefile of M2 only knows about the project S2. If the code for M2 is generated, the iteration of S2 is inevitable, even if the make of M1 had generated the code. This cannot be avoided and increases the run time of T1's make. But relations like T1-S3 (red arrow) should be omitted since they are unnecessary and avoidable. T1 does not need to iterate S3 since M1 did it before and T1 can reach it via M1. Summarized: Try to minimize the loops in the project hierarchy graph. In big hierarchies (50-100 Projects) a well-

organized structure can have an overhead of up to 50%-100%. A poorly organized one can run even 5 times longer than the flat hierarchy (without -H option).

Rewriting an existing hierarchy can lead to linker errors. For example: an error message beginning with "*undefined reference to...*" means that one (or more) project(s) is/are missing from the calling one.

Usage hints:

1. Hierarchical building can be applied by option -Z.
2. Any project can be regarded as top level project if the makefiles are generated with the additional option -H.
3. Remove unnecessary references. It can dramatically decrease the hierarchical build time. The Project hierarchy cannot contain circular references.
4. To optimize the building time, work only on the highest-level project(s). They should be set for executable, all lower level and all unused branches should be set for library, especially for dynamically linked library. Take into account that it is not the best solution for the the final executable because the dynamically linking libraries can decrease the speed of running.

Placing custom rules and new targets into the generated Makefile

Custom rules and new targets can be inserted into the generated Makefile. This feature consists of two parts: calling a script whose output will be inserted into the generated Makefile and specifying new targets in the TPD file which will be inserted into the generated Makefile to the appropriate places. These two parts can be used together to accomplish the desired solution. The script shall print the project's custom Makefile rules to its standard output.

These rules have targets such as:

```
customtarget : dependency1 dependency2
      <command1>
      <command2>
```

The second part of the feature is to add these custom targets to other specified places of the Makefile. Currently these places are: `PHONY`, `TTCN3_MODULES`, `TTCN3_PP_MODULES`, `TTCN3_INCLUDES`, `ASN1_MODULES`, `USER_SOURCES`, `USER_HEADERS`, `USER_OBJECTS`, `OTHER_FILES`. These places usually contain a list of files which will be used in the build process at different stages. By adding a new custom target to one or more of these places it becomes part of the dependency tree which will be processed by the make program and our new custom rule will be automatically invoked when necessary.

An example of how to print some message when make is done with the "all" target:
First make a script that prints the rule itself (here a python script):

```
print """
buildwithmessage: all
	@echo 'Here i built the project for you!!!'
"""
```

Next add the new rule to the appropriate place, in this case to the `PHONY` targets because it's not a real file to be created. The script invocation and the addition of the new target are specified in the TPD file inside the MakefileSettings element (after the `buildLevel` element):

```
<ProjectSpecificRulesGenerator>
  <GeneratorCommand>python MyRulesAdder.py</GeneratorCommand>
  <Targets>
    <Target name=" buildwithmessage" placement="PHONY"/>
  </Targets>
</ProjectSpecificRulesGenerator>
```

To see the message after the build make shall be invoked with the new target: `make buildwithmessage`

Of course in most cases real files are generated and not phony targets. These can be ttcn files generated from some type descriptions written in other notations or languages. Or cc/hh files generated by lexer and parser generators (flex/bison). In these cases the generated file name shall be the custom target and it shall be added to places like `TTCN3_MODULES` or `USER_SOURCES`. This way when the make program encounters a rule that depends on the new target (for example our new custom ttcn-3 source file) it will use our added custom rule and the <command> part of that rule will create/update the ttcn-3 file before it will be used by the TITAN compiler to generate cc/hh files and then object file and finally the executable.

This method of making changes on the generated Makefile is preferred over the legacy makefile modifier scripts method. The makefile modifier scripts are error prone because these contain many assumptions about the exact content of the Makefile which may not be true for future versions of the makefile generator.

External directory usage in tpd

External directory usage is described with OSS_DIR example.

To enable proper OSS usage, some parameters must be set in the tpd file. Lower extractions from tpd file can be seen, details which are useful for setting up OSS usage.

```
<Files>
    <FileResource projectRelativePath="OSS_H323_MESSAGES.c"
relativeURI="src/OSS_H323_MESSAGES.c"/>
    <FileResource projectRelativePath="OSS_H323_MESSAGES.h"
relativeURI="src/OSS_H323_MESSAGES.h"/>
    <FileResource projectRelativePath="OSS_H323_MESSAGES_linux.c"
relativeURI="src/OSS_H323_MESSAGES_linux.c"/>
 </Files>

<ActiveConfiguration>Default</ActiveConfiguration>
  <Configurations>
    <Configuration name="Default">
      <ProjectProperties>
        <MakefileSettings>
          <preprocessorIncludes>
            <listItem>[OSS_DIR]/include</listItem>
          </preprocessorIncludes>
          <linkerLibrarySearchPath>
            <listItem>[OSS_DIR]/lib</listItem>
          </linkerLibrarySearchPath>
        </MakefileSettings>
```

NOTE	OSS_DIR system variable needs to be set properly in your path.

NOTE	Using makefile updater scripts are obsolete.

Referred project usage with -I switch

If there are different TPD projects which often change location, then the -I path switch(es) can be used.

Example TPD structure: `MainTpd.tpd` is the top level TPD and has several project dependencies. `MainTpd` depends on the following projects: `DepTpd1.tpd`, `DepTpd2.tpd` and `DepTpd3.tpd`.

```
MainTpd.tpd is located at /home/titan/main_project/MainTpd.tpd
DepTpd1.tpd is located at /home/titan/dep_project1/DepTpd1.tpd
DepTpd2.tpd is located at /home/titan/dep_project2/DepTpd2.tpd
DepTpd3.tpd is located at /home/titan/random_folder/ dep_project3/DepTpd3.tpd
```

The relevant part of the MainTpd.tpd is the following:

```
<TITAN_Project_File_Information version="1.0">
  <ProjectName>MainTpd</ProjectName>
  <ReferencedProjects>
    <ReferencedProject name="DepTpd1" projectLocationURI="../dep1/DepTpd1.tpd" />
    <ReferencedProject name="DepTpd2X" tpdName="DepTpd2.tpd"
projectLocationURI="../incorrect/path/DepTpd2.tpd" />
    <ReferencedProject name="DepTpd3"
projectLocationURI="../incorrect/path/DepTpd3.tpd" />
  </ReferencedProjects>
```

When executing the `makefilegen` command

```
makefilegen -t MainTpd.Tpd -I /home/titan/foo
-I /home/titan/dep_project2
-I /home/titan/random_folder/dep_project3
```

Then the tool's logic when resolving the paths is the following:

The first referred project's name is `DepTpd1` and the tool will be able to find the `DepTpd1.tpd` in the relative path provided in the `projectLocationURI` attribute. The next referred project's name is DepTpd2X and the tool will NOT be able to find the `DepTpd2.tpd` in the relative path provided in the `projectLocationURI` attribute. In this case the tool looks for the `tpdName` attribute which is now present. The tool takes the value of the `tpdName` attribute and in input order tries to find the `DepTpd2.tpd` at the paths in the -I switches. First try is at `/home/titan/foo` which is not correct. Second try is at `/home/titan/dep_project2` which is correct because the DepTpd2.tpd file is at `/home/titan/dep_project2/DepTpd2.tpd` and the search stops at this point. No further trying will be done.

The last referred project's name is DepTpd3 and the tool will NOT be able to find the `DepTpd3.tpd` in the relative path provided in the projectLocationURI attribute. In this case the tool looks for the `tpdName` attribute which is NOT present now. In this case the tool takes the value of the name attribute and concatenates it with the `.tpd` suffix and this name will be used during the search. The first and second tries are not successful but the third try is correct because the `DepTpd3.tpd` file is at `/home/titan/random_folder/dep_project3/DepTpd3.tpd`.

NOTE	We strongly advise you to not use this feature. Most projects don't need this feature.

Usage of code splitting when generating makefile from TPD

The `makefilegen` tool allows the usage of code splitting mechanisms when generating makefile(s) from a TPD file using the `codeSplitting` tag in the TPD with a few restrictions:

- In the project hierarchy every project shall have the same `codeSplitting` tag set. The `codeSplitting` tag can have the following values: none, type, a positive number, or an empty string which defaults to none. If the `codeSplitting` tag is missing, then the code splitting strategy will set to none.
- Code splitting is not supported when the H or Z flags are used. (see page 228)

TITAN's strategy of the code splitting mechanism when using a "number" (-U "number")

Let "number" be equal to 4 for this example. We want to split the files into four pieces.

Firstly, TITAN finds the TTCN3 module whose C++ generated code will be the largest. In this example, it will be 10000 characters (let's call it MAX). So the largest generated C++ module contains 10000 characters.

Secondly TITAN calculates the splitting threshold by dividing MAX with "number", so it will be 10000 / 4 = 2500 in this case. TITAN will only split the generated C++ files for the selected modules which are larger than 2500 characters.

BUT TITAN will always generate "number" pieces for each file. The reason behind this is the following: The makefilegen tool needs to know what c++ files will be generated.

Let's complete the example.

We have three TTCN3 modules:

- `My_Types.ttcn` (whose generated c++ code contains 10000 characters (MAX))
- `My_Functions.ttcn` (whose generated c++ code contains 6000 characters)
- `My_Constants.ttcn` (whose generated c++ code contains 1000 characters)

If we execute the command: compiler `-U 4 My_Types.ttcn My_Functions.ttcn My_Constants.ttcn` the following c++ source files will be generated:

- `My_Types_part_1.cc` (contains approximately 2500 characters)
- `My_Types_part_2.cc` (contains approximately 2500 characters)
- `My_Types_part_3.cc` (contains approximately 2500 characters)
- `My_Types_part_4.cc` (contains approximately 2500 characters)
- `My_Functions_part_1.cc` (contains approximately 2500 characters)
- `My_Functions_part_2.cc` (contains approximately 2500 characters)
- `My_Functions_part_3.cc` (contains approximately 1000 characters)
- `My_Functions_part_4.cc` (contains approximately 0 effective characters)
- `My_Constants_part_1.cc` (contains approximately 1000 characters)
- `My_Constants_part_2.cc` (contains approximately 0 effective characters)
- `My_Constants_part_3.cc` (contains approximately 0 effective characters)
- `My_Constants_part_4.cc` (contains approximately 0 effective characters)

6.2. The Compilation Process for TTCN-3 and ASN.1 Modules

Before analyzing the input modules the compiler applies some heuristics for each source file to determine whether it contains a TTCN-3 or ASN.1 module. These so-called pre-parsing algorithms

consider only the first few words of the files so it can happen that the compiler is unable to recognize its input and stops immediately with an error message. This is the case, for example, if the beginning of the module is either erroneous or contains strange and misleading comments. In this case using the command-line options `-T` and `-A` you can bypass the pre-parsers and force to interpret the corresponding file as a TTCN-3 or ASN.1 module, respectively.

During its run, the compiler reports its activities on its standard error stream like the following. The level of detail for these messages can be controlled with the flag `-V`.

```
Notify: Parsing TTCN-3 module 'MyModule.ttcn'...
Notify: Parsing ASN.1 module 'MyAsnModule.asn'...
Notify: Checking modules...
Notify: Generating code...
Notify: File 'MyModule.hh' updated.
Notify: File 'MyModule.cc' updated.
Notify: File 'MyAsnModule.hh' updated.
Notify: File 'MyAsnModule.cc' updated.
Notify: 4 files updated.
```

First, the compiler reads the TTCN-3 and ASN.1 input files and performs syntax check according to the BNF of TTCN-3 [1] (including the additions of [3]) or ASN.1 [4], [7], [8], [9]. The syntax errors are reported with the appropriate line number. Whenever it is possible, the compiler tries to recover from syntax errors and continue the analysis in order to detect further errors.

NOTE	Error recovery is not always successful and it might result in additional undesired error messages when the parser gets out of synchronization. Therefore it is recommended to study the first lines on the compiler's error listings because the error messages at the end are not always relevant.

After the syntax check the compiler performs semantic analysis on TTCN-3 /ASN.1 module(s) and verifies whether the various definitions and language elements are used in the appropriate way according to the static semantics of TTCN-3 and ASN.1 languages. In addition to error messages the compiler reports a warning when the corresponding definition is correct, but it might have unwanted effects.

If both syntax and semantic checks were successful, the compiler generates a C++ header and source file that contains the translated module. If the name of the input module is `MyModule` (i.e. it begins with module `MyModule`), the name of the generated header and source file will be `MyModule.hh` and `MyModule.cc`, respectively. Note that the name of the output file does NOT depend on the name of input file. In ASN.1 module names the hyphens are converted to underscore characters (e.g. the C++ code for `My-Asn-Module` will be placed into `My_Asn_Module.hh` and `My_Asn_Module.cc`).

By default, the compiler generates the C++ code for all input modules. This can be unnecessarily time-consuming when doing incremental builds for large projects. The build process can be significantly speed up if the compiler option - (single dash) is used. In this case the C++ code will be generated only for those modules that have changed since last build of the ASN.1 modules. With selective code generation it can be exploited that the make utility can easily tell which source files were changed since the last compilation.

This sophisticated command line syntax is necessary because in general case it is impossible to perform the semantic analysis on a subset of the modules because those may import from modules outside the list. Moreover, to avoid undesirable side-effects of the code optimization techniques implemented in the compiler (e.g. type and value folding) the C++ code is generated not only for the modified modules, but for all modules that import definitions (either directly or indirectly) from the modified ones.

When the compiler translates an ASN.1 module, the different ASN.1 types are mapped to TTCN-3 types as described in the table below.

Table 12. Mapping of ASN.1 types to TTCN-3 types

ASN.1	**TTCN-3**
Simple types	
NULL	- *
BOOLEAN	boolean
INTEGER	integer
ENUMERATED	enumerated
REAL	float
BIT STRING	bitstring
OCTET STRING	octetstring
OBJECT IDENTIFIER	objid
RELATIVE-OID	objid
string †	charstring
string ‡	universal charstring
string §	universal charstring
Compound types	
CHOICE	union
SEQUENCE	record
SET	set
SEQUENCE OF	record of
SET OF	set of

* there is no corresponding TTCN-3 type
† IA5String, NumericString, PrintableString, VisibleString (ISO646String)
‡ GeneralString, GraphicString, TeletexString (T61String), VideotexString
§ BMPString, UniversalString, UTF8String

6.3. Particularities of ASN.1 Modules

This section describes the checks the complier performs on ASN.1 modules.

6.3.1. Type Assignments

In this first phase only basic checks are made: the compiler checks for unresolved and for circular references. The simplest example for circular reference is the following:

```
T1 ::= T2
T2 ::= T1
```

But there are more complex cases, especially related to non-optional fields of compound types. For example, X.680 clause 47.8.1 contains an illegal type definition:

```
A ::= SET {
  a A,
  b CHOICE {
    c C,
    d D,
    ...
  }
}
```

It is easy to see that one can not assign a (finite) value to a variable of type A: there is an endless recursion because of the field a, which is the same type as the parent-type. If this field were optional, then the recursion could be broken at any level.

6.3.2. Value Assignments

The compiler checks the unresolved/circular references also in case of value assignments.

The value is checked according to the type:

- `NULL`: Only the `NULL` value is accepted.
- `BOOLEAN:` `TRUE` or `FALSE` value is accepted.
- `BIT STRING:` You can use `hstring`, `bstring` or (even empty) set of named bits. In the latter case, the compiler checks if there are bits with the given names.
- `OCTET STRING:` Only `hstring` or `bstring` form is accepted.
- `character strings`: The `cstring`, `tuple`, `quadruple` form and the combination of these forms (`CharacterStringList`) are accepted.
- `INTEGER`: Number form and named values (defined in the type with which the value is associated) can be used.
- `REAL`: You can use the special values (`0`, `PLUS-INFINITY`, `MINUS-INFINITY`) as well as the associated `SEQUENCE` type (defined in X.680 clause 20.5) and the real number form (defined in X.680 clause 11.9).
- `OBJECT IDENTIFIER`: All possible notations (i.e. `NameForm`, `NumberForm`, `NameAndNumberForm` and `DefinedValue`) can be used for the components. The predefined names given in Annex A-C of X.660 are recognized for the first two or three components. According to X.680 clause 31 it is

checked whether the Number and `DefinedValue` is (a reference to) a non-negative integer or `OBJECT IDENTIFIER/RELATIVE-OID` value, respectively. If `NameAndNumberForm` is used, only the number is considered for the code generation. A warning message is displayed if in the first two components the given name and number is not in accordance with each other.

- `ENUMERATED`: Only the identifiers defined in the corresponding type can be used.
- `CHOICE`: The compiler checks if there is an alternative with the given name, then checks if the value corresponds to the type of the selected alternative.
- `SEQUENCE` OF and `SET OF`: You can use the "empty" value (`{}`) or list of values separated by commas, enclosed in braces. Each value in the list is checked.
- `SEQUENCE`: The values of the fields shall be in the same order as in the corresponding type definition. Components marked with OPTIONAL or DEFAULT can be skipped. The components are checked recursively.
- `SET`: There shall be one and exactly one value definition for each not `OPTIONAL` or * `DEFAULT` component defined in the corresponding type. They can appear in any order.

Named values (in case of `INTEGER` and `ENUMERATED`) have higher priority than defined values.

6.3.3. Type Definitions

The compiler makes an exhaustive check of the types defined in the module. For the different types, the following checks are executed:

tagged types: As you can use value references in the tags, the compiler checks if the value is a non-negative integer.

- `BIT STRING`: When using named bits, the bit number must be a non-negative integer. Each bit can have only one identifier (duplications are not permitted).
- `INTEGER`: Value references in named numbers (if any) must reference integer values.
- `ENUMERATED`: Value references (if any) must reference integer values. The compiler assigns a number[11] for each item which does not have an associated number. Duplicated values (neither in identifiers, nor in associated number values) are not permitted. Items defined after an ellipsis must have associated numbers that increase monotonously. For details, see X.680 clause 19.
- `CHOICE`: Every alternative must have different tag. Tags in the extension must increase monotonously (X.680 28.4). (The canonical order of tags is defined in X.680 8.4.)
- `SEQUENCE`: The tags in optional groups[12] must have different tags (X.680 24.5).All tags in the extension must be distinct from tags used in the first optional group after the second ellipsis (X.680 24.6).
- `SET`: The types used in a SET type shall all have different tags (X.680 26.3).Tags in the extension must increase monotonously.

  Extension is not always permissible in `CHOICE`, `SEQUENCE` and `SET` (see X.680 47.7). Here is an example:

```
Illegal-type ::= SET {
  a INTEGER,
  b CHOICE {
    c C,
    d D,
    ...,
    ...
  },
  ...,
  e E
}
```

The problem is that the (BER) decoder of a version 1 system cannot attribute an unknown element (received from a version 2 system) unambiguously to a specific insertion point.

6.4. Using Component Relation Constraints from TTCN-3

To handle constructs defined in X.681, X.682 and X.683 is not easy from TTCN-3. There is an ETSI technical report[13] which describes how to transform these constructs to equivalent X.680 constructs. The clause 4.4 of this document is about transforming information objects.

"The transformation rules presented in this clause cannot reproduce the full semantics of the information objects they replace. The transformation rules cannot preserve component relation constraints. These constraints provide the ability to constrain a type or value with reference to a different field within an information object set."

This is not such a great problem, because BER does not "see" the constraints. But there is a situation when the transformations are unusable: when references to information object type fields are constrained by component relation constraints. Let's take the example from X.682, clause 10 (with a little bit of modifications, to enlighten the problem):

```
ERROR-CLASS ::= CLASS
{
  &category PrintableString(SIZE(1)),
  &code INTEGER,
  &Type

}
WITH SYNTAX {&category &code &Type}

ErrorType1 ::= [1] INTEGER

ErrorType2 ::= [1] REAL

ErrorType3 ::= [1] CHARACTER STRING
```

```
ErrorType4 ::= [1] GeneralString

ErrorSet ERROR-CLASS ::=

{
  {"A" 1 ErrorType1} |
  {"A" 2 ErrorType2} |
  {"B" 1 ErrorType3} |
  {"B" 2 ErrorType4}

}

ErrorReturn ::= SEQUENCE

{
  errorCategory ERROR-CLASS.&category ({ErrorSet}) OPTIONAL,
  errors SEQUENCE OF SEQUENCE
  {
    errorCode ERROR-CLASS.&code({ErrorSet}{@errorCategory}),
    errorInfo ERROR-CLASS.&Type({ErrorSet}{@errorCategory,@.errorCode})
  } OPTIONAL
}
After applying the transformation rules described in ETSI technical report, the
equivalent definitions look like this:

ErrorReturn ::= SEQUENCE
  {
    errorCategory PrintableString(SIZE(1)) OPTIONAL,
    errors SEQUENCE OF SEQUENCE
    {
      errorCode INTEGER,
      errorInfo CHOICE
      {
        errorType1 ErrorType1,
        errorType2 ErrorType2,
        errorType3 ErrorType3,
        errorType4 ErrorType4
      }
    } OPTIONAL
  }
```

It is plainly seen that this is not a legal type definition: the alternatives of a `CHOICE` must have distinct tags. The original definition is unambiguous, because the `errorCode` component "tells" the decoder how to interpret the `errorInfo` component.

[9] Other tool vendors may use .mp, .3mp or .asn1 suffixes as well.

[10] Not supported

[11] According to X.680 clause 19.3

[12] Optional group: One or more consecutive occurrences of OPTIONAL or DEFAULT fields, including the first not OPTIONAL or DEFAULT field.

[13] TR 101 295

Chapter 7. The Run-time Configuration File

The behavior of the executable test program is described in the run-time configuration file. This is a simple text file, which contains various sections. The usual suffix of configuration files is `.cfg`.

Each section begins with a section name within square brackets. Different sections use different syntax, thus the section name determines the possible syntax of the members.

The configuration file can contain any white space characters. There are three ways to put comments in the file: you can use the C comment delimiters (i.e. /* and */). Additionally, characters beginning with # or // are treated as comments until the end of line.

Character string values shall be given between quotation marks. The non-printable characters as well as the quotation mark character within string values must be escaped using TTCN-3 or C escape sequences or quadruple notation. All kinds of escape sequences that are available in TTCN-3 and C languages (including octal and hexadecimal notation) are recognized.

All sections are optional, thus an empty file is also a valid (but meaningless) configuration file. The sections are processed in the given order. In case of duplicated sections, all sections will be processed and no warnings will be issued.

In the following we present all possible sections of the configuration file. The majority of sections are applicable in both single and parallel modes with identical meaning. However, some sections or settings in a section are applicable either in single or parallel mode only.

The indication (Parallel mode) following the section title signals that the concerned section is processed in parallel operation mode only. Irrelevant sections or options are ignored in each mode and a warning message is displayed during configuration file processing. For details on running TITAN TTCN-3 test suites in either single or parallel mode using see the TITAN TTCN-3 User Guide (see [13]).

The component name defined by the `create` operation (see here) can contain any characters. This component name can be used without quoted marks but it shall be used with quoted marks (i.e as quoted string) if it contains extra characters e.g. space (" "), hyphen ("-") or dot ("."). See also the examples of this chapter in sections [MODULE_PARAMETERS] and [TESTPORT_PARAMETERS].

7.1. [MODULE_PARAMETERS]

This section may contain the values of any parameters that are defined in your TTCN-3 modules.

The parameters shall be specified after each other in any order. Each parameter must be given in the following order: module name (optional) [24], parameter name, field names (in case of records, sets or unions; optional) or array indexes (in case of arrays, records of or sets of; optional), assignment or concatenation operator (that is, the characters := or &=) and parameter value. An optional terminator semicolon may be used after each parameter value. The concatenation operator can be applied to list types (record of, set of). In this case only the value list notation can be used.

7.1.1. BNF Productions for this Section

In the Function Test runtime:

```
ModuleParametersSection ::= "[MODULE_PARAMETERS]" {ModuleParameter}
ModuleParameter ::= ParameterName ParamOpType ParameterValue [SemiColon]
ParameterName ::= [(ModuleName | "*") "."] ParameterIdentifier
ModuleName ::= Identifier
ParameterIdentifier ::= Identifier | ParameterIdentifier "." Identifier
| ParameterIdentifier "[" ParameterExpression "]"
ParamOpType ::= ":=" | "&="
ParameterValue ::= ParameterExpression [LengthMatch] ["ifpresent"]
ParameterExpression ::= SimpleParameterValue | ParameterIdentifier
    | "(" ParameterExpression ")" | ("+" | "-") ParameterExpression
    | ParameterExpression ("+" | "-" | "*" | "/" | "&") ParameterExpression
LengthMatch ::= "length" "(" LengthBound [".." (LengthBound|"infinity")] ")"
LengthBound ::= ParameterExpression
SimpleParameterValue ::= IntegerValue | FloatValue | BitstringValue | HexstringValue
    | OctetstringValue | StringValue | UniversalCharstringValue | BooleanValue
    | ObjIdValue | VerdictValue | EnumeratedValue | "omit" | "NULL" | "null"
    | "?" | "*" | IntegerRange | FloatRange | StringRange
    | "pattern" PatternChunk
    | BitStringMatch | HexStringMatch | OctetStringMatch
    | "mtc" | "system"
    | CompoundValue
IntegerValue ::= Number
FloatValue ::= FloatDotNotation | FloatENotation
StringValue ::= Cstring
BitstringValue ::= Bstring
HexstringValue ::= Hstring
OctetstringValue ::= Ostring
UniversalCharstringValue ::= Quadruple
Quadruple ::= "char" "(" ParameterExpression "," ParameterExpression ","
    ParameterExpression "," ParameterExpression ")"
ObjIdValue ::= "objid" "{" {ObjIdComponent}+ "}"
ObjIdComponent ::= NumberForm | NameAndNumberForm
NumberForm ::= Number
NameAndNumberForm ::= Identifier "(" Number ")"
EnumeratedValue ::= Identifier
PatternChunk ::= Cstring | Quadruple
IntegerRange ::= "(" ("-" "infinity" | IntegerValue) ".." (IntegerValue | "infinity")
")"
FloatRange ::= "(" ("-" "infinity" | FloatValue) ".." (FloatValue | "infinity") ")"
StringRange ::= "(" StringRangeBound ".." StringRangeBound ")"
StringRangeBound ::= Cstring | Quadruple
CompoundValue ::= "{" "}"
    | "{" FieldValue {"," FieldValue} "}"
    | "{" ArrayItem {"," ArrayItem} "}"
    | "{" IndexItem {"," IndexItem} "}"
    | "(" ParameterValue "," ParameterValue {"," ParameterValue} ")"
```

```
    | ("complement" | "superset" | "subset") "("ParameterValue {","
   ParameterValue} ")"
FieldValue ::= FieldName ":=" ParameterValueOrNotUsedSymbol
FieldName ::= Identifier | ASN1LowerIdentifier
ArrayItem ::= ParameterValueOrNotUsedSymbol | ("permutation" "("ParameterValue {","
ParameterValue} ")")
IndexItem ::= "[" ParameterExpression "]" ":=" ParameterValue
ParameterValueOrNotUsedSymbol ::= ParameterValue | "-"
```

The BNF productions in the Load Test runtime are mostly the same with one difference:

`ParameterIdentifier ::= Identifier`

The parameter value can be one of the following:

- Integer value. A number in decimal notation or an expression composed of numbers using the basic arithmetic operations to allow more flexibility with macro substitution.
- Floating point value. A floating point number in decimal dot notation or exponential notation or an arithmetic expression composed of such values.
- Bitstring value (in the notation of TTCN-3). String fragments can be concatenated to allow more flexible macro substitution.
- Hexstring value (in the notation of TTCN-3). String fragments can be concatenated to allow more flexible macro substitution.
- Octetstring value (in the notation of TTCN-3). String fragments can be concatenated to allow more flexible macro substitution.
- Charstring value (between quotation marks; escape sequences are allowed). String fragments can be concatenated to allow more flexible macro substitution.
- Universal charstring value (sequence of concatenated fragments; each fragment can be either a printable string within quotation marks or a quadruple in TTCN-3 notation).
- Boolean value (`true` or `false`).
- Object identifier (`objid`) value (in the notation of TTCN-3). Only `NumberForm` or `NameAndNumberForm` notations are allowed.
- Verdict value (`none`, `pass`, `inconc`, `fail` or `error`).
- Enumerated value (the symbolic value, i.e. an identifier). Numeric values are not allowed.
- Omit value (i.e. `omit`). Valid for optional record or set fields only.
- `null` value for TTCN-3 component and default references.
- `NULL` value for the ASN.1 `NULL` type.
- "?" value: AnyValue for matching
- "*" value: AnyValueOrNone for matching
- IntegerRange/FloatRange/StringRange: matching an integer/float/charstring range
- Pattern for pattern matching in charstring and universal charstring
- Bit/Hex/Octet -string matching mechanism which are bit/hex/octet strings that contain "?" and

"*" for matching

- "mtc" and "system" values for component references
- Compound value with assignment notation. One or more fields (separated by commas) with field names within brackets for types `record` and `set`.
- Compound value with value list notation. Comma separated list of values between brackets for types `record of`, `set of` and `array`.
- Compound value with indexed list notation. One or more fields with field index and value for types record of and set of
- Compound value containing a template list. Can be a value list, complemented value list, superset and subset list.
- Reference to a module parameter, or a field/element of a module parameter (only in the Function Test runtime). Its syntax is the same as the left hand side of a module parameter assignment/concatenation (except the symbol '*' cannot be used to specify all modules). The reference is substituted with the current value of the specified module parameter (its value prior to the execution of this module parameter assignment/concatenation). References can also appear in the expressions specified above (integer and float references can appear in arithmetic expressions, references to string types can be concatenated with other strings of the same type). A dynamic test case error is displayed if the referenced module parameter is unbound.

Nested compound values are permitted in arbitrary depth. In compound values with assignment and value list notations the "-" symbol can be used to skip an element. In value list notation for record of and set of permutation lists can be used.

Parsing conflict: An asterisk (*) after a module parameter expression could be treated as a multiplication operator or the "all modules" symbol for the next module parameter assignment/concatenation. The configuration parser always treats it as a multiplication operator. In order to use the asterisk as an "all modules" symbol, the previous statement must be closed with a semicolon (;).

Example:

```
# correct:
tsp_IntPar := tsp_IntPar + 1;
*.tsp_FloatPar := 3.0;
# incorrect, causes a parser error:
tsp_IntPar := tsp_IntPar + 1
*.tsp_FloatPar := 3.0;
```

7.1.2. Differences between the TTCN-3 and the Configuration File Module Parameters Section Syntax

Neither the ttcn-3 syntax nor the module parameter section syntax is the subset of the other. Historically some module parameter values that are not legal in ttcn-3 have been accepted, for backward compatibility reasons this behavior has been kept. In most cases the module parameter syntax is a subset of the ttcn-3 syntax but there are important exceptions:

- Field values of records and sets can be referenced multiple times, in this case all field value assignments will be executed in order of appearance. Example: mp_myRecord := { a :=1, b:=3, a:=2 } // a==2
- In an assignment notation used for a union multiple field values can appear, only the last value will be used. Example: mp_myUnion := { a:=1,b:=2,a:=3 } only a:=3 will be used, the other 2 assignments are disregarded without warnings.
- The order of fields in assignment notation for records does not have to be identical to the order of fields specified in the type definition of the record type. Example: type record MYREC { integer a, integer b } mp_myRec := { b:=2, a:=1}
- The "*" matching symbol can be used for mandatory fields (in TTCN-3 this cannot be used directly but indirectly any field of a variable template can be set to "*").

In the module parameters section only constant values and references to module parameters can be used. Function calls are not allowed (not even predefined functions).

Example:

```
[MODULE_PARAMETERS]
par1 := -5
MyModule1.par2 := "This is a string\n"
MyModule1.par3 := {
  ethernet_header := {
    source_address := '000100010005'O,
    destination_address := '0008C7993605'O,
    ether_type := 34525
  },
  ipv6 := {
    header := {
      version := 6,
      traffic_class := 0,
      flow_label := 0,
      payload_length := 0,
      next_header := 58,
      hop_limit := 255,
      source_address := 'FE8000000000000020100FFFE010005'O,
      destination_address := 'FE8000000000000208C7FFFE993605'O
    },
    extension_headers := {
      {
        hop_by_hop_options_header := {
          next_header := 1,
          header_length := 2,
          options := '13'O
        }
      }
    },
    data := {
      f_router_advertisement := {
        icmp_type := 134,
```

```
        code := 0,
        checksum := 0,
        hop_limit := 255,
        m_bit := '0'B,
        o_bit := '0'B,
        reserved := '000000'B,
        lifetime := 600,
        reachable_time := 300000,
        retrans_timer := 1000,
        options := {
          {
            lla := {
              option_type := 1,
              option_length := 1,
              lla_address := '0008C7993605'O
            }
          }
        }
      }
    }
  }
}
MyModule2.par4 := '10010'B
par5 := objid { itu_t(0) identified_organization(4) etsi(0) 12345 6789 }
par6 := "Character " & char(0, 0, 1, 113) & " is a Hungarian letter."
par_record_of_int &= {1,2,3}
par_record_of_int &= {4,5,6}
par_record_of_int[6] := 7
MyModule1.par3.ethernet_header.ether_type := 34526
```

7.2. [LOGGING]

The executable test program produces a log file during its run. The log file contains important test execution events with time stamps[26]. This section explains how to set the log file name and the event classes to be logged. Logging may be directed to file or displayed on console (standard error).

Various options can be set in the section [LOGGING]. They affect the format and appearance of the test execution logs. Any option may be omitted; that is, each has a default value that is used if the option is omitted. If the same option is present several times in a configuration file only the latest value will take effect; the previously assigned values are ignored and a warning message is issued during configuration file processing.

7.2.1. LoggerPlugins

TITAN is equipped with an extensible logging architecture. The test execution events are directed towards the logger plugin interface. This interface can host arbitrary number of logger plugins.

The logging can be statically and dynamically loaded.

The default logging is the statically loaded built in logging.

The dynamically loaded logging can be the built in LegacyLogger, the plugins shipped with TITAN (see here) or user created dynamically linked plugins (for advanced users, see chapter 3 in [16].

NOTE	**When using dynamically loaded logger plugins it is very important to use the dynamic runtime library of TITAN, this can be done by using the -l switch when generating the makefile.**

The desired logger plugins need to be set in the `LoggerPlugins` option. The `LoggerPlugins` option takes a non-empty, comma separated list of logger plugin settings. These settings can be specified in the following ways:

- A logger plugin identifier followed by an assignment and a quoted string containing the plugin location. E.g. `LoggerPlugins := { plugin1 := "/absolute/path/to/plugin1.so" }` or `LoggerPlugins := { plugin1 := "/absolute/path/to/plugin1" }`. The identifier of a logger plugin is determined by its author can be learned from its documentation. The plugin location includes the file name of the plugin with an optional file system path prefix. A logger plugin is a dynamically linked shared object thus the logger plugin typically resides in a `.so` file. When the path prefix is missing or a relative path is used then TITAN attempts to locate the plugin in the path specified in the `LD_LIBRARY_PATH` environment variable. The plugin file name can be provided in two different ways: either by specifying the whole file name (ending with .so) or by specifying only the base of the name (omit the .so ending). The latter method is preferred because a logger plugin usually consists of 4 different shared library files and the proper file must be selected. The 4 different versions correspond to the single/parallel mode and the load/function test TITAN runtimes. If the file name ending is not provided the executable will determine it automatically, but if the whole file name is provided then it must correspond to the runtime which is actually used by the executable.
- A single logger plugin identifier. E.g. `LoggerPlugins := { plugin1 }`. In this case there should be a logger plugin named plugin1.so in one of the paths specified in the `LD_LIBRARY_PATH` environment variable.

NOTE	If TITAN is unable to locate any of the listed logger plugins, it will quit immediately with an error message. So, if the `LoggerPlugins` option is used, take special care to set `LD_LIBRARY_PATH` correctly or use absolute paths when specifying the plugins. There is a built-in logger plugin in TITAN, which provides the usual text-based logging format. This plugin is used when the `LoggerPlugins` option is omitted or it was listed explicitly in the list with the `LegacyLogger` case insensitive identifier. Since it's not possible to specify the path for this special, built-in logger plugin, only the second (with no path) specification mode is applicable here.

The logger plugins are responsible for the "final appearance" of the test execution log. The current TITAN distribution comes with a single logger plugin but it also supports user written logging plugins. The built-in `LegacyLogger` plugin produces log files and console log entries with similar content to elder TITAN revisions.

In case of overlapping plugin settings in multiple `LoggerPlugins` options, all configured plugins are attached to the list of existing plugins and take part in logging (i.e. do not overwrite them).

The configured logger plugins are valid for each test component unless otherwise configured (see subsection below).

It is possible to reference all the listed plugins with * to e.g. assign the same parameters and values for multiple plugins. However, plugin configuration through * will not have an effect on plugin parameters, whose value was set previously by referencing explicitly the exact plugin using its name and optionally the component identifier it is configured for.

Logger plugins can be configured with arbitrary name-value pairs in the configuration file (see here). E.g. `..param1 := "value1"` will set the param1 parameter to a string value1 for all, not explicitly configured logger plugins on all test components. Logger plugins can ignore unknown parameters.

Each logger plugin has 4 .so files. The following table contains the names of the 4 different cases if the base file name (this is not the name of the plugin, library file names start with "lib") of the plugin is "libplugin":

NOTE	The preferred method of specifying the above logger plugin in the configuration file is: `LoggerPlugins := { MyPluginName := "libplugin"}` The name MyPluginName is the name of the plugin and can be different than the library file names. If the full plugin file name is provided ("libplugin.so, libplugin-parallel.so, etc.) then care must be taken to always use the name that corresponds to the mode in which the executable is running.

Component-based Logger Plugin Settings

It is possible to associate an individual set of logger plugins for each test component. The component designation can be:

- the component name as given in the command `create`,
- the component reference (though using component references as identifiers is not recommended as this is a tool dependent identifier),
- the symbol * meaning all valid test components or
- the keyword `mtc` in the case of the Main Test Component.

The component name, if present, precedes the keyword `LoggerPlugins`. They are separated by a dot (.). The absent component reference is equivalent to `*.LoggerPlugins` meaning all valid test components.

7.2.2. Dynamically Loaded Logger Plugins Shipped with TITAN

Anyone can write a logger plugin to use with TITAN, but there are some widely used plugins that were made part of TITAN. These are available in the `$(TTCN3_DIR)/lib` sub-directory. Usually `LD_LIBRARY_PATH` contains this directory, if not then it should either be added to it or the path to the .so file has to be specified.

JUnitLogger Plugin

It outputs XML files in JUnit format. This format is needed by Hudson/Jenkins, a continuous integration test tool. The XML files written by this logger plugin are a subset of the JUnit XML output format.

At first select dynamic linking at Makefile generation according to LoggerPlugins.

To load the plugin the section `[LOGGING]` of the runtime configuration file should contain the following line: `LoggerPlugins := { JUnitLogger := "libjunitlogger" }`

The plugin has 2 parameters:filename_stem: set the output file name, the name will start with the string specified in this parameter and end with "-<process_id>.log". The default value is "junit-xml". testsuite_name: the name of the test suite, this will be written into the XML file.

Example 1: Simplest HelloWorld example.

Source file

```
module hello {
  type component CT {}
  testcase tc1() runs on CT {
    log("Hello Titan!");
    setverdict(pass,"Everything is ok");
  }

  testcase tc2() runs on CT {
    log("Hello Titan!");
    setverdict(fail,"Something was wrong");
  }
control {
  execute(tc1());
  execute(tc2());
}

}
}
```

Configuration file (`cfg.cfg`):

```
[LOGGING]
LogSourceInfo := Yes
SourceInfoFormat := Single
LoggerPlugins := { JUnitLogger := "libjunitlogger" }
*.JUnitLogger.filename_stem := "MyJunitLogFile"
*.JUnitLogger.testsuite_name := "myJUnitTest"

[EXECUTE]
hello.control
```

The makefile was generated by the command `makefilegen -fl hello.ttcn`.

The executable was executed by the command `ttcn3_start hello cfg.cfg`.

After running the log file name was `MyJunitLogFile-6426.log`, its content was:

```
<?xml version="1.0"?>
<testsuite name='myJUnitTest'><!-- logger name="JUnitLogger" version="v1.0" -->
<!-- Testcase tc1 started -->
<!-- Testcase tc1 finished in 0.000399, verdict: pass, reason: Everything is ok -->
  <testcase classname='hello' name='tc1' time='0.000399'>
  </testcase>
<!-- Testcase tc2 started -->
<!-- Testcase tc2 finished in 0.000225, verdict: fail, reason: Something was wrong -->
  <testcase classname='hello' name='tc2' time='0.000225'>
    <failure type='fail-verdict'>Something was wrong

      hello.ttcn:14 hello control part
      hello.ttcn:10 tc2 testcase
    </failure>
  </testcase>
</testsuite>
```

Format of the results

The results are included in testcases (between <testcase> tags) within a testsuite (between <testsuite> tags).The testsuite has only one attribute "name" which contains the name of the testsuite.

Each testcase starts with two xml style comments which is followed by a <testcase> xml tag.

The comments

Each testcase starts with the following two xml comments:

- `<!- Testcase "name of the testcase" started →`
- `<!- Testcase "name of the testcase" finished in "execution time is seconds", verdict:"verdict" →`

This is followed by a <testcase> tag.

The <testcase> tag

The <testcase> tag has the following attributes:

- `classname="name of the module"`
- `name="name of the testcase"`
- `time="execution duration in seconds"`

Depending of the verdict the <testcase> may have element contents which can be the following:

Verdict: pass

No children

Verdict: fail

Has the following element content with the following attribute:

<failure type=`fail-verdict'>

The <failure> tag can have several of the following text contents - each in a separate line (see results log example above):

- control part
- testcase
- altstep
- function
- external function
- template

Each line contains `"filename:linenumber identifier definition"`, where each of the items mean the following:

- filename:linenumber - the file and the line where the test failed
- identifier - the identifier depending on the definition, e.g. the classname in case of "control part" or the name of the test in case of "testcase"
- definition - see the list in <failure> tag text content

This is a simple stacktrace of the failure.

Verdict: none

```
<skipped>no verdict</skipped>
```

Verdict: `inconclusive`

No children

Verdict: `error`

Has the following element content with the following attribute:

```
<error type='DTE'>
```

The <error> tag has the text contents containing the reason of the error.

XSD validation

This is the xsd file to validate the xml generated by the plugin:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="failure">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="skipped" type="xs:string"/>

    <xs:element name="error">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="testcase">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="skipped" minOccurs="0" maxOccurs="1"/>
                <xs:element ref="error" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element ref="failure" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="classname" type="xs:string" use="required"/>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="time" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="testsuite">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="testcase" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

JUnitLogger2 Plugin

It outputs XML files in JUnit format. This format is needed by Hudson/Jenkins, a continuous integration test tool. The XML files written by this logger plugin are a subset of the JUnit XML output format.

The JUnitLogger2 plugin works as the JUnitLogger plugin with the following exceptions:

- Configuration file (`cfg.cfg`):

  In the configuration file when specifying the logger plugin libjunitlogger2 should be written.

  ```
  [LOGGING]
  LogSourceInfo := Yes
  SourceInfoFormat := Single
  LoggerPlugins := { JUnitLogger := "libjunitlogger2" }
  *.JUnitLogger.filename_stem := "MyJunitLogFile"
  *.JUnitLogger.testsuite_name := "myJUnitTest"
  ```

- The results are included in testcases (between <testcase> tags) within a testsuite (between <testsuite> tags) like the JUnitLogger but the testsuite has attributes other than "name" which contains the name of the testsuite.

  New attributes:

  tests - number of all testcases executed

  failures - number of testcases whose final verdict is fail

  errors - number of testcases that encountered an error

  skipped - number of testcases with a none verdict

  time - the time in seconds from the start of the tests until all the testcases executed.

- XML comments are not added to the output.
- XSD:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="failure">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="skipped" type="xs:string"/>

    <xs:element name="error">
        <xs:complexType mixed="true">
            <xs:attribute name="type" type="xs:string" use="optional"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="testcase">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="skipped" minOccurs="0" maxOccurs="1"/>
                <xs:element ref="error" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element ref="failure" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="classname" type="xs:string" use="required"/>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="time" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="testsuite">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="testcase" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string"  use="required"/>
            <xs:attribute name="tests" type="xs:int" use="required"/>
            <xs:attribute name="failures" type="xs:int" use="required"/>
            <xs:attribute name="errors" type="xs:int" use="required"/>
            <xs:attribute name="skipped" type="xs:int" use="required"/>
            <xs:attribute name="time" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

TSTLogger plugin

The `TestStatistics` TITAN Logger plugin sends HTTP messages to the `TestStatistics` web based

tool. `TestStatistics` has a web interface (http://eta-teststatistics.rnd.ki.sw.ericsson.se/ts/login) where the test result data can be examined. Currently the following messages are sent to the `TestStatistics` tool:

- Test suite started (tsstart)
- Test case started (tcstart)
- Test case finished (tcstop)
- Test suite finished (tsstop)
- Test case fail reason (tcfailreason)

The content of these messages is filled based on the log data and data given in the configuration file. Some data needs to be set as logger plugin parameters in the configuration file because it is needed by the `TestStatistics` tool but the TITAN log messages do not contain such information. The plugin parameters:

Name	Default value	Description
`tst_host_name`	"eta-teststatistics.rnd.ki.sw.ericsson.se"	`TestStatistics` web service host name
`tst_service_name`	"http"	`TestStatistics` web service name or port number
`tst_tcstart_url`	"/ts-rip/rip/tcstart"	Path for the tcstart message
`tst_tcstop_url`	"/ts-rip/rip/tcstop"	Path for the tcstop message
`tst_tsstart_url`	"/ts-rip/rip/tsstart"	Path for the tsstart message
`tst_tsstop_url`	"/ts-rip/rip/tsstop"	Path for the tsstop message
`tst_tcfailreason_url`	"/ts-rip/rip/tcfailreason"	Path for the tcfailreason message
`dbsUrl`	"esekilx0007-sql5.rnd.ki.sw.ericsson.se:3314"	database URL
`dbUser`	"demo"	database user
`dbPass`	"demo"	plain text password of the user
`dbName`	"teststatistics_demo"	name of the database
`log_plugin_debug`	"0"	The logger plugin will print some debug information to the console if this value is not "0".
`testConfigName`	"DefaultConfigName"	name of this specific configuration of the test suite
`suiteName`	"DefaultSuiteName"	name of test suite
`executingHost`	the host name of the MTC (determined with gethostname())	host where the test was executed

Name	Default value	Description
sutId	"0.0.0.0"	IP address of SUT
sutName	"DefaultSUTName"	name of SUT
lsvMajor	"1"	major version number of SUT
lsvMinor	"0"	minor version number of SUT
runByUser	Login name of the user running the MTC (determined with getlogin())	name of user running the tests
projectName	"DefaultProjectname"	name of the project
productName	"DefaultProductName"	name of the product
productVersion	"0.0"	version of the product
configType	"configType"	
configVersion	"configVersion"	
testType	"testType"	
logLink	"default_log_location"	absolute location of log files
logEnd	"default_web_log_dir"	log directory relative to web server root
reportEmail	automatically set to < runByUser > +"@ericsson.com"	who is to be notified via email
reportTelnum	"0"	where to send the SMS notification

The parameters starting with "tst_" should be modified only if the TestStatistics tool is moved to another location on the intranet.

There are 4 parameters for setting the database connection, the database can also be selected on the web interface, all information sent by TITAN will be stored there. The default values will use a demo database which can be used for experimentation.

All parameters have default values, thus the plugin can be used even without setting any parameters. However some parameters should be set to a meaningful value, such as the suiteName or the projectName. An example from a configuration file:

```
LoggerPlugins := { TSTLogger := "libtstlogger" }
*.TSTLogger.testConfigName = "Hiper Giga Test"
*.TSTLogger.sutId = "11.22.33.44"
*.TSTLogger.projectName = "MagicProject"
*.TSTLogger.suiteName = "Super Test Suite"
*.TSTLogger.lsvMajor = "3"
*.TSTLogger.lsvMinor = "14"
```

To load the plugin the runtime configuration file should contain the following line:

```
LoggerPlugins := { TSTLogger := "libtstlogger" }
```

LTTngUSTLogger plugin

The LTTng-UST logger plugin emits each logging statement as an LTTng-UST event. LTTng is a low-overhead tracer for Linux which generates CTF traces.

To use the LTTng-UST logger plugin:

1. Make sure LTTng (2.7 or greater) is installed (the LTTng-tools and LTTng-UST components are required).
2. Add the following line to your runtime configuration file:LoggerPlugins := { LTTngUSTLogger := "liblttng-ust-logger" }
3. Create an LTTng tracing session:lttng create
4. Enable TITAN events:lttng enable-event -userspace titan_core:'*'
5. Start tracing:lttng start
6. Run your test script.
7. When you are done, stop tracing:lttng stop
8. Inspect the recorded events with an LTTng trace viewer, for example Trace Compass or Babeltrace.

When the plugin is loaded, it dynamically loads the LTTng-UST tracepoint provider, a shared object installed in the same directory. It is important that the `LD_LIBRARY_PATH` environment variable be set to this directory, otherwise the plugin issues a warning message and does not emit LTTng events.

7.2.3. `LogFile`

Option `LogFile` stands for the name of the log file. A string value is expected, which is interpreted as a skeleton to determine the file name. To make the file name handling more flexible the string may contain special metacharacters, which are substituted dynamically during test execution.

The table below contains the list of available metacharacters in alphabetical order. Any unsupported metacharacter sequence, that is, if the `%` character is followed by any character that is not listed in the table below or a single percent character stays at the end of the string, will remain unchanged.

Table 13. Available metacharacters for setting log file names

Meta-character	Substituted with . . .
`%c`	the name of the TTCN-3 test case that the PTC belongs to.
`%m`	the name of the TTCN-3 module containing the test case
`%e`	the name of the TTCN-3 executable. The .exe suffix (on Windows platforms) and the directory part of the path name (if present) are truncated.

Meta-character	Substituted with . . .
%h	the name of the computer returned by the gethostname(2) system call. This usually does not include the domain name.
%i	the sequence number of the log fragment.
%l	the login name of the current user. If the login name cannot be determined (e.g. the current UNIX user ID has no associated login name) an empty string is returned.
%n	- the name of the test component if the PTC has been given a name with the command create in the TTCN-3 create operation; an empty string otherwise. - the string HC if the logfile is generated by a Host Controller - the string MTC if the component is the Main Test Component (both in parallel and in single mode)
%p	the process ID (pid) of the UNIX process that implements the current test component. The pid is written in decimal notation.
%r	the component reference or component identifier. On PTCs it is the component reference (an integer number greater or equal to 3) in decimal notation. On the Main Test Component, Host Controller or in single mode the strings mtc, hc or single are substituted, respectively.
%s	the default suffix for the log files without the leading dot. Now it always results in string log, but future versions may support log file compression. In this case the suffix will depend on the chosen compression method.
%t	the TTCN-3 component type. Note: The MTC and HC have no dedicated component type since they can implement several component types in different test cases.
%%	a single % character.

The outcome of substitution will result in the name of the log file. It may resolve either to a simple file name or to an absolute or relative path. The relative pathnames are always related to the current working directory of the executable tests in single mode or that of the Host Controller in parallel mode, respectively. If the pathname contains one or more nonexistent directories, those directories (and/or subdirectories) will be automatically created with permissions 0755 before the log file is opened.

If the given string or the result of substitution is empty, no log file will be created and only console logging will be performed regardless the setting of FileMask. Empty log files will not be created when logging to files is completely disabled (i.e. FileMask is set to LOG_NOTHING) even if the value of LogFile would yield a valid file name.

In parallel mode the user must ensure that the resulting log file names are unique for every component. Otherwise, if several components try to write into the same log file, the contents of the log will be unpredictable. The uniqueness is automatically provided if the host name (%h) and the component reference (%r) or the process ID (%p) is included in the file name.

If testcase name (%c) or component name (%t) is included, the log file name may change during runtime. A new log file will be created/opened in this case. If a log file with the new name already exists, it is overwritten by default. Because of this, it is **highly recommended** to set AppendFile (see here) to Yes if LogFile contains %c or %t.

If omitted, the default value for option LogFile is %e-part%i.%s in single mode and %e.%h-%r-part%i.%s in parallel mode, respectively. This ensures backward compatibility with earlier versions in parallel mode and follows the most commonly used file naming convention in single mode.

7.2.4. FileMask

Option FileMask determines which events will be written to the log file and which ones will be filtered out.

Category-based Filtering

Table 14 enumerates the first level groups of events (that is, logging categories) along with a symbolic constant. The selected constants separated by a pipe (|) will determine what will be logged. When FileMask is omitted from the configuration file, its default value (LOG_ALL) is applied.

Some of the first level logging categories may be divided in second level subcategories to get a finer logging granularity. These categories are listed in the tables 11 to 21. First level categories and second level subcategories may be mixed in the option.

First level logging categories may be considered as notational convenience. They are maintained also for backward compatibility: legacy configuration files containing only first level categories will continue to work unmodified. However, the resulting log file may look different, as event categories have been modified; some messages, mostly PARALLEL and FUNCTION, have changed category, usually to EXECUTOR. When a first level logging category is included in the option FileMask, all second level subcategories pertaining to it will also be implicitly included, thus, it is redundant to list one ore more subcategories along with the concerned first level category.

Component and Plugin Based Filtering

It is possible to filter events based on the identity of the component generating them. For component designation it is recommended to use the component name given in the command create or the keyword mtc; latter one in the case of the Main Test Component. Using component numbers as identifiers is not recommended as this is a tool dependent identifier.

The component name, if present, precedes the keyword FileMask. They are separated by a dot (.).

It is also possible to apply the filtering on selected logger plugins of a component. The identifier of the desired logger plugin is appended after the component designation. The component and plugin identifiers are separated by a dot(.).

7.2.5. ConsoleMask

Option ConsoleMask determines which events will be written to the console and which ones will be filtered.

Category-based Filtering

Table 14 enumerates the first level groups of events (that is, logging categories) along with a symbolic constant. The selected constants separated by a pipe (|) will determine what will be logged. When ConsoleMask is omitted from the configuration file, its default value (ERROR|WARNING|ACTION |TESTCASE|STATISTICS) is applied.

Some of the first level logging categories may be divided in second level subcategories to get a finer logging granularity. These categories are listed in the tables 11 to 21. First level categories and second level subcategories may be mixed in the option.

First level logging categories may be considered as notational convenience. They are maintained also for backward compatibility: legacy configuration files containing only first level categories will continue to work unmodified. However, the resulting log file may look different, as event categories have been modified; some messages, mostly PARALLEL and FUNCTION, have changed category, usually to EXECUTOR. When a first level logging category is included in the option ConsoleMask, all second level subcategories pertaining to it will also be implicitly included, thus, it is redundant to list one ore more subcategories along with the concerned first level category.

In single mode the console log messages are written to the standard error of the ETS. For the interpretation of console logging in parallel mode, see section 12.3.3. of the TITAN User Guide ([13]).

WARNING	Please note that neither the timestamp nor the event type, nor location information is printed on the console.

Component and Plugin Based Filtering

It is possible to filter events based on the identity of the component generating them. For component designation it is recommended to use the component name given in the command create or the keyword mtc; latter one in the case of the Main Test Component. Using component numbers as identifiers is not recommended as this is a tool dependent identifier.

The component name, if present, precedes the keyword ConsoleMask. They are separated by a dot (.).

It is also possible to apply the filtering on selected logger plugins of a component. The identifier of the desired logger plugin is appended after the component designation. The component and plugin identifiers are separated by a dot (.).

Table 14. First level (coarse) log filtering

Logging categories	Symbolic constant[14]
TTCN-3 action(...) statements(No subcategory is implemented)	ACTION
Debug messages in Test Ports and external functions(For subcategories see Table 15)	DEBUG
Default operations (activate, deactivate, return)(For subcategories see Table 16)	DEFAULTOP
Dynamic test case errors (e.g. snapshot matching failures)(No subcategory is implemented)	ERROR
Internal TITAN events(For subcategories see Table 17)	EXECUTOR
Random number function calls(For subcategories see Note: EXECUTOR_EXTCOMMAND is always logged, regardless of the user's settings.	FUNCTION
Bitwise OR of all the above except MATCHING and DEBUG	LOG_ALL
Nothing to be logged	LOG_NOTHING
Analysis of template matching failures in receiving port operations(For subcategories see Table 19)	MATCHING
Parallel test execution and test configuration related operations (create, done, connect, map, etc.)(For subcategories see Table 20)	PARALLEL
Port events (send, receive)(For subcategories see Table 22)	PORTEVENT
Statistics of verdicts at the end of execution(For subcategories see Table 23)	STATISTICS
The start, the end and the final verdict of test cases(For subcategories see Table 21)	TESTCASE
Timer operations (start, stop, timeout, read)(For subcategories see Table 24)	TIMEROP
User log(...) statements(No subcategory is implemented)	USER
Verdict operations (setverdict, getverdict)(For subcategories see Table 25)	VERDICTOP
Run-time warnings (e.g. stopping of an inactive timer)(No subcategory is implemented)	WARNING

Table 15. Second level (fine) log filtering for **DEBUG**

Logging subcategories	Symbolic constant
Debug information coming from generated functions of dual faced ports and built-in encoder/decoders.[15]	DEBUG_ENCDEC
	DEBUG_TESTPORT
Other, non categorized log messages of the category	DEBUG_UNQUALIFIED

Table 16. Second level (fine) log filtering for **DEFAULTOP**

Logging subcategories	Symbolic constant
TTCN-3 `activate` statement (activation of a default)	DEFAULTOP_ACTIVATE
Deactivation of a `default`	DEFAULTOP_DEACTIVATE
Leaving an invoked default at the end of a branch (causing leaving the `alt` statement in which it was invoked) or calling `repeat` in an invoked default (causing new snapshot and evaluation of the `alt` statement)	DEFAULTOP_EXIT
Other, non categorized log messages of the category	DEFAULTOP_UNQUALIFIED

Table 17. Second level (fine) log filtering for **EXECUTOR**

Logging subcategories	Symbolic constant
Starting and stopping MTC and HCs	EXECUTOR_COMPONENT
ETS runtime events (user control of execution, control connections between the processes of the ETS, ETS overloaded messages, etc.)	EXECUTOR_RUNTIME
Runtime test configuration data processing	EXECUTOR_CONFIGDATA
When this subcategory is present in the configuration file, logging options are printed in the second line of each log file.	EXECUTOR_LOGOPTIONS
Running of external command	EXECUTOR_EXTCOMMAND
Other, non categorized log messages of the category	EXECUTOR_UNQUALIFIED

NOTE EXECUTOR_EXTCOMMAND is always logged, regardless of the user's settings.

Table 18. Second level (fine) log filtering for **FUNCTION**

Logging subcategories	Symbolic constant
Random number functions in TTCN-3	FUNCTION_RND

Logging subcategories	Symbolic constant
Other, non categorized log messages of the category	`FUNCTION_UNQUALIFIED`

Table 19. Second level (fine) log filtering for **MATCHING**

Logging subcategories	Symbolic constant
Matching a TTCN-3 `done` operation	`MATCHING_DONE`
Timer in timeout operation is not started or not on the list of expired timers	`MATCHING_TIMEOUT`
Procedure-based mapped ports: successful template matching	`MATCHING_PMSUCCESS`
Procedure-based mapped ports: unsuccessful template matching	`MATCHING_PMUNSUCC`
Procedure-based connected ports: successful template matching	`MATCHING_PCSUCCESS`
Procedure-based connected ports: unsuccessful template matching	`MATCHING_PCUNSUCC`
Message-based mapped ports: successful template matching	`MATCHING_MMSUCCESS`
Message-based mapped ports: unsuccessful template matching	`MATCHING_MMUNSUCC`
Message-based connected ports: successful template matching	`MATCHING_MCSUCCESS`
Message-based connected ports: unsuccessful template matching	`MATCHING_MCUNSUCC`
Unsuccessful matching	`MATCHING_PROBLEM`
Other, non categorized log messages of the category	`MATCHING_UNQUALIFIED`

Table 20. Second level (fine) log filtering for **PARALLEL**

Logging subcategories	Symbolic constant
PTC creation and finishing, starting and finishing a function started on a PTC	`PARALLEL_PTC`
Port `connect` and `disconnect` operations	`PARALLEL_PORTCONN`
Port `map` and `unmap` operations	`PARALLEL_PORTMAP`
Other, non categorized log messages of the category	`PARALLEL_UNQUALIFIED`

Table 21. Second level (fine) log filtering for **TESTCASE**

Logging subcategories	Symbolic constant
A testcase is starting	`TESTCASE_START`
A testcase ends; final verdict of the testcase	`TESTCASE_FINISH`
Other, non categorized log messages of the category	`TESTCASE_UNQUALIFIED`

Table 22. Second level (fine) log filtering for **PORTEVENT**

Logging subcategories	Symbolic constant
Procedure-based ports: call, reply or exception enqueued in the queue of the port or extracted from the queue	`PORTEVENT_PQUEUE`
Message-based ports: message enqueued in the queue of the port or extracted from the queue	`PORTEVENT_MQUEUE`
Port state changes[16] (`halt`, `start`, `stop`, port `clear` operation finished)	`PORTEVENT_STATE`
Procedure-based mapped ports: incoming data received (`getcall`, `getreply`, `catch`, `check`)	`PORTEVENT_PMIN`
Procedure-based mapped ports: outgoing data sent (`call`, `reply`, `raise`)	`PORTEVENT_PMOUT`
Procedure-based connected ports: incoming data received (`getcall`, `getreply`, `catch`, `check`)	`PORTEVENT_PCIN`
Procedure-based connected ports: outgoing data sent (`call`, `reply`, `raise`)	`PORTEVENT_PCOUT`
Message-based mapped ports: incoming data received (`receive`, `trigger`, `check`)	`PORTEVENT_MMRECV`
Message-based mapped ports: outgoing data sent (`send`)	`PORTEVENT_MMSEND`
Message-based connected ports: incoming data received (`receive`, `trigger`, `check`)	`PORTEVENT_MCRECV`
Message-based connected ports: outgoing data sent (`send`)	`PORTEVENT_MCSEND`
Mappings of incoming message from the external interface of dual-faced ports to the internal interface (decoding)	`PORTEVENT_DUALRECV`
Mappings of outgoing message from the internal interface of dual-faced ports to the external interface (encoding)	`PORTEVENT_DUALSEND`
Other, non categorized log messages of the category	`PORTEVENT_UNQUALIFIED`

Table 23. Second level (fine) log filtering for **STATISTICS**

Logging subcategories	Symbolic constant
Verdict statistics of executed test cases (% of `none`, `pass`, `inconc`, `fail`, `error`)	`STATISTICS_VERDICT`
Other, non categorized log messages of the category	`STATISTICS_UNQUALIFIED`

Table 24. Second level (fine) log filtering for **TIMEROP**

Logging subcategories	Symbolic constant
TTCN-3 `read timer` operation	`TIMEROP_READ`
TTCN-3 `start timer` operation	`TIMEROP_START`
Log events related to the guard timer used in TTCN-3 `execute` statements	`TIMEROP_GUARD`
TTCN-3 `stop timer` operation	`TIMEROP_STOP`
Successful TTCN-3 `timeout` operation (timer found on the list of expired timers)	`TIMEROP_TIMEOUT`
Other, non categorized log messages of the category	`TIMEROP_UNQUALIFIED`

Table 25. Second level (fine) log filtering for **VERDICTOP**

Logging subcategories	Symbolic constant
TTCN-3 `getverdict` operation	`VERDICTOP_GETVERDICT`
TTCN-3 `setverdict` operation	`VERDICTOP_SETVERDICT`
Final verdict of a test component (MTC or PTC)	`VERDICTOP_FINAL`
Other, non categorized log messages of the category	`VERDICTOP_UNQUALIFIED`

7.2.6. `AppendFile`

Option `AppendFile` controls whether the run-time environment shall keep the contents of existing log files when starting execution. The possible values are `Yes` or `No`. The default is `No`, which means that all events from the previous test execution will be overwritten.

This option can be used in both single and parallel modes. Its usefulness in single mode is obvious. If the executable test suite is started several times, the logs of the successive test sessions will be stored in the same single file after each other.

In parallel mode the naming of log files is automatic and is based on the host name and component references. The option is applicable to all log files: all of them will be either appended or overwritten. If the test execution is repeated several times with different configuration or test case selection, the same file may contain the log of totally different test components. When appending is enabled the log files can be interpreted after using the logmerge utility (see Section 13.1. of the TITAN User Guide, [13]). The option `AppendFile` guarantees only that no logged events will be lost during the entire test campaign.

7.2.7. TimeStampFormat

Option TimeStampFormat configures the formatting of timestamps in the log file. It can have three possible values: Time stands for the format hh:mm:ss.microsec. DateTime results in yyyy/Mon/dd hh:mm:ss.microsec. This is useful for long test durations (for instance, when a stability test runs for a couple of days). Using the third alternative (Seconds) results relative timestamps in format s.microsec. The time is related to the starting of the test component or test execution (i.e. this is the zero time). The default value for TimeStampFormat is Time.

7.2.8. ConsoleTimeStampFormat

Option ConsoleTimeStampFormat configures the formatting of timestamps in console log. It can have the same three values as TimeStampFormat can have: Time, DateTime and Seconds (see here). If it is omitted (default) timestamp will not be inserted in the console log.

7.2.9. LogEventTypes

Option LogEventTypes indicates whether to include the symbolic event type (without the TTCN prefix) in each logged event immediately after the timestamp. This option can be useful for log post-filtering scripts. The possible values for LogEventTypes are Yes, No, Detailed and Subcategories.

The default is No: no events will be logged.

The setting Yes results a logfile containing event categories listed in Table 14.

The setting Subcategories (and the equivalent Detailed) produces a logfile containing both event categories and subcategories. Subcategories are listed in the tables 11 to 21.

In both single and parallel modes some log events are created before processing the configuration data. At this time the logging options (name of the log file, filter settings, timestamp format, etc.) are not known by the run-time environment, thus, these events are collected in a temporary memory buffer and are flushed to the log file when the processing of configuration file is finished. This implies that if the Host Controller is stopped in parallel mode before configuring it, no log file will be created at all.

7.2.10. SourceInfoFormat

Option SourceInfoFormat controls the appearance of the location information for the test events. The location information refers to a position in the TTCN-3 source code. It consists of the name of the TTCN-3 file and the line number separated by a colon character (:). Optionally, it may contain the name of the TTCN-3 entity (function, testcase, etc.) in parentheses that the source line belongs to. See also the option LogEntityName below.

The option can take one of the three possible values: None, Single and Stack. In case of Single, the location information of the TTCN-3 statement is logged that is currently being executed. When Stack is used the entire TTCN-3 call stack is logged. The logged information starts from the outermost control part or testcase and ends with the innermost TTCN-3 statement. An arrow (that is, the character sequence →) is used as separator between the stack frames. The value None disables the printing of location information. The default value for SourceInfoFormat is None.

The location information is placed in each line of the log file between the event type or timestamp and the textual description of the event.

This option works only if the command line option `-L` is passed to the compiler (see here). This feature is useful for debugging new TTCN-3 code or for understanding the traces of complex control constructs. If the location information is not generated into the C++ code the executable tests run faster, which can be more important when doing performance tests.

NOTE The reception of messages or procedure calls can only occur while the run-time environment is taking a new snapshot. A new snapshot is taken when the testcomponent is evaluating a stand-alone receiving operation, an `alt` construct or a standalone `altstep` invocation. Thus, the location information of the incoming messages or calls points to the first line of the above statements. The log event belonging to a TTCN-3 operation can be the extraction of a message from the port queue and not the reception of an incoming message.

If the event has no associated line in the TTCN-3 source code (e.g. because it belongs to test environment startup or termination) and `SourceInfoFormat` is set to either `Single` or `Stack`, a single dash character `(-)` is printed into the log instead of the location information. This makes the automated processing of log files easier.

The obsolete option `LogSourceInfo` is also accepted for backward compatibility with earlier versions. Setting `LogSourceInfo := Yes` is equivalent to `SourceInfoFormat := Single`, and similarly `LogSourceInfo := No` means `SourceInfoFormat := None`.

7.2.11. `LogEntityName`

Option `LogEntityName` controls whether the name of the corresponding TTCN-3 entity (`function`, `testcase`, `altstep, control` part, etc.) shall be included in the location information. If this option is set to `Yes`, the file name and line number is followed by the name of the TTCN-3 entity within parentheses. The default value is `No`. The option has no effect if `SourceInfoFormat` is set to `None`.

7.2.12. `LogFileSize`

Option `LogFileSize` sets the upper limit for the log file size. The limitation prevents load tests or long duration tests from triggering dynamic test case error when the growing log file exceeds file system size limits or available disk space.

When the size limit is reached, the file is closed and a new log file will be created with an increased part number. For example, the first two log files when running `ExampleTestCase` in single mode will be `ExampleTestCase-part1.log` and `ExampleTestCase-part2.log`, respectively provided that the file name skeleton default values have not been modified.

This option must be set together with `LogFileNumber`.

The parameter value, a non-negative integer, is understood in kilobytes. The default value is 0, meaning that the file size is unlimited; or, to be precise, is only limited by the file system.

Component and Plugin Dependent File Size

It is possible to set different file sizes based on the identity of the component generating the log. For component designation it is recommended to use the component name given in the parameter of the command `create` (or the keyword `mtc` for the Main Test Component). Using component numbers as identifiers is not recommended as this is a tool dependent identifier.

The component name, if present, precedes the keyword `LogFileSize`. The name and the keyword are separated by a dot (.).

It is also possible to limit the file size on selected logger plugins of a component. The identifier of the desired logger plugin is appended after the component designation. The component and plugin identifiers are separated by a dot (.).

7.2.13. LogFileNumber

Option `LogFileNumber`, a positive integer, sets the maximum number of log files (fragments) kept. If the log file number limit is reached, the oldest log file of the component will be deleted and logging continues in the next log fragment file.

The default value is 1, meaning that the number of log files equals one.

Component and Plugin Dependent Fragment Number

It is possible to set different fragment limits based on the identity of the component generating the log. For component designation it is recommended to use the component name given in the parameter of the command `create` (or the keyword `mtc` for the Main Test Component). Using component numbers as identifiers is not recommended as this is a tool dependent identifier.

The component name, if present, precedes the keyword `LogFileNumber`. The name and the keyword are separated by a dot (.).

It is also possible to limit the number of log fragments on selected logger plugins of a component. The identifier of the desired logger plugin is appended after the component designation. The component and plugin identifiers are separated by a dot (.).

7.2.14. DiskFullAction

Option `DiskFullAction` determines TITAN behavior when writing to the log file fails.

If this option set to `Stop` test case execution continues without logging when an error occurs.

The setting `Retry` causes test case execution to continue without logging and TITAN attempts to restart logging activity periodically (events in the unlogged periods are lost). The retry period is set by default to 30 seconds and can be changed by a parameter. Example: `Retry(60)` doubles the period.

If the parameter is set to `Delete`, TITAN deletes the oldest log file and continues logging to a new log file fragment. If log writing fails again, the procedure is repeated until one two log files (the actual one and the previous one) are left. Further log writing failure causes a dynamic test case error.

The default behavior is `Error`. With this setting, writing error causes a runtime (dynamic) test case

error.

Component and Plugin Dependent Behavior

It is possible to set different error behavior based on the identity of the component generating the log. For component designation it is recommended to use the component name given in the parameter of the command `create` (or the keyword `mtc` for the Main Test Component). Using component numbers as identifiers is not recommended as this is a tool dependent identifier.

The component name, if present, precedes the keyword `DiskFullAction`. The name and the keyword are separated by a dot (.).

It is also possible configure different error behavior on selected logger plugins of a component. The identifier of the desired logger plugin is appended after the component designation. The component and plugin identifiers are separated by a dot (.).

7.2.15. `MatchingHints`

Option `MatchingHints` controls the amount and format of log messages describing template matching failures. These are written during port receive operations as logging category `MATCHING`, and as a response to TTCN-3 `log(match(…))` statements as logging category `USER`.

There are two possible values: `Compact` and `Detailed`.

When the `Detailed` option is in effect, a field-by-field description of the value and template is logged, followed by additional hints when matching set-of types. Example:

```
{
    {
        field_rr1 := 1,
        field_rr2 := 2
    },
    {
        field_rr1 := 3,
        field_rr2 := 4
    }
} with {
    {
        field_rr1 := 1,
        field_rr2 := 2
    },
    {
        field_rr1 := 3,
        field_rr2 := 5
    }
} unmatched Some hints to find the reason of mismatch: {
    value elements that have no pairs in the template: {
        field_rr1 := 3,
        field_rr2 := 4
    } at index 1,
    template elements that have no pairs in the value: {
        field_rr1 := 3,
        field_rr2 := 5
    } at index 1,
    matching value <-> template index pairs: {
        0 <-> 0
    },
    matching unmatched value <-> template index pairs: {
        1 <-> 1: {
            {
                field_rr1 := 3 with 3 matched,
                field_rr2 := 4 with 5 unmatched
            }
        }
    }
}
```

The printout is similar to the TTCN-3 assignment notation for the entire structure.

When the `Compact` option is in effect, fields and structures that match are omitted in order to pinpoint the reason why the entire match operation failed. Every mismatch is represented as a path from the outermost (containing) type to the innermost simple type that failed to match. This is similar to a mixture of dot notation referencing fields of record/set types and indexed notation referencing elements of record-of/set-of types, as it would be used to reference the innermost member of a structured type:

- Mismatched fields of a record/set are represented by the field name preceded by a dot (a.k.a. full stop).
- Mismatched elements of a record-of are represented by the index in square brackets.
- Mismatched elements of a set-of are represented by the indexes of the mismatching elements in the vale and the template, separated by a two-headed arrow.

Example: The following line is the equivalent of the nested display above when the `Compact` option is in effect instead of `Detailed`.

```
[1 <-> 1].field_rr2 := 4 with 5 unmatched
```

This means that the second element (indexing is 0-based) of the value didn't match the second element of the template because field_rr2 in the value was 4 whereas field_rr2 in the template was 5.

The default value of `MatchingHints` is `Compact`.

7.2.16. `EmergencyLogging`

Titan implements an emergency logging feature. The purpose of this feature is to help diagnose errors by logging events that would normally be suppressed, for example if only a few event types are logged (e.g. to minimize I/O overhead or log file size) and all the other log events are discarded. If something unexpected occurs (e.g. Dynamic Testcase Error), it can be difficult to diagnose the problem, but there is no way to recover the discarded events.

With emergency logging, log events which are not written to the log file can be stored in a ring buffer. In case of an error, the stored events can be recovered from the buffer and written to the log. Because the buffered events are closest in time to the error, they are most likely to be helpful in diagnosing the cause.

The value of the `EmergencyLogging` option is the ring buffer size (the number of log events that are kept). The default value is zero, which turns off the emergency logging feature.

7.2.17. `EmergencyLoggingBehaviour`

Buffering of events can be performed in two ways:

- Buffering only selected messages. This option is selected with the `BufferMasked` value of the `EmergencyLoggingBehaviour` option. This is the default behaviour. Log events are sent to the plugins to be filtered and logged. Additionally log events not included by the `FileMask` and included by the `EmergencyLoggingMask` are buffered. This method cannot guarantee that timestamps of the log events passed to the plugins are always monotonically increasing. Monotonically increasing timestamps are a requirement for ttcn3_logmerge. The LegacyLogger plugin ensures that the requirements of ttcn3_logmerge are satisfied by writing the emergency log messages to a separate log file.
- Buffer all messages. This option is selected with the `BufferAll` value of the `EmergencyLoggingBehaviour` option. The value of the `EmergencyLoggingMask` is ignored. All events are initially placed in the buffer. If the buffer is full, the oldest buffered event is extracted and sent to the logger plugins to be filtered and logged. If an error occurs, all stored events are

extracted and logged without filtering. This method guarantees that all log events passed to the plugins have their timestamps in a monotonically increasing order. In this case there is no separate emergency log file.

7.2.18. EmergencyLoggingMask

Option `EmergencyLoggingMask` determines which events will be saved in the emergency logging buffer when the value of `EmergencyLoggingBehaviour` is `BufferMasked`.

7.2.19. EmergencyLoggingForFailVerdict

Option `EmergencyLoggingForFailVerdict` controls whether `setverdict(fail)` operations trigger emergency logging or not. The possible values are `Yes` or `No`. The default is `No`, which means that emergency logging would not be triggered when the component's verdict is set to `fail`. Emergency logging is always triggered when a dynamic test case error is reached, regardless of this option.

7.2.20. LogSensitiveData

Option `LogSensitiveData` controls whether values and templates marked as sensitive data (i.e. whose types have the `sensitive_data` extension attribute) are redacted or not while logging. The possible values are `Yes` or `No`. The default is `No`, which means sensitive data is redacted. If set to `Yes`, then sensitive data is displayed normally.

7.2.21. BNF productions for this section

```
LoggingSection ::= "[LOGGING]" {LoggingAssignment}
LoggingAssignment ::= [ComponentId "." [PluginId "."]] LoggingParam
    | [ComponentId "."] "LoggerPlugins" AssignmentChar "{" LoggerPluginList "}"
LoggingParam ::= (LogFile | FileMask | ConsoleMask | AppendFile
    | TimeStampFormat | ConsoleTimeStampFormat | LogEventTypes
    | SourceInfoFormat | LogEntityName
    | LogFileSize | LogFileNumber | DiskFullAction | MatchingHints
    | PluginSpecificParameter
    | EmergencyLogging | EmergencyLoggingBehaviour | EmergencyLoggingMask
    | EmergencyLoggingForFailVerdict | LogSensitiveData) [SemiColon]
LoggerPluginList ::= LoggerPluginSetting ["," LoggerPluginSetting ]
LoggerPluginSetting ::= Identifier AssignmentChar PluginLocation | Identifier
PluginId ::= Identifier | "*"
PluginSpecificParameter ::= Identifier AssignmentChar StringValue
PluginLocation ::= StringValue
LogFile ::= ("LogFile" | "FileName") AssignmentChar StringValue
FileMask ::= "FileMask" AssignmentChar LoggingBitmask
ConsoleMask ::= "ConsoleMask" AssignmentChar LoggingBitmask
MatchingHints := "Compact" | "Detailed"
ComponentId ::= Identifier | Number | MTCKeyword | "*"
LoggingBitmask ::= LoggingBit {"|" LoggingBit}
LoggingBit ::= ... /* defined in Table 12 to Table 23 */
AppendFile ::= "AppendFile" AssignmentChar ("Yes" | "No")
TimeStampFormat ::= "TimeStampFormat" AssignmentChar ("Time" | "DateTime"
    | "Seconds")
ConsoleTimeStampFormat ::= "ConsoleTimeStampFormat" AssignmentChar ("Time" |
"DateTime"
    | "Seconds")
LogEventTypes ::= "LogEventTypes" AssignmentChar ("Yes" | "No" | "Detailed"
| "Subcategories")
SourceInfoFormat ::= ("SourceInfoFormat" | "LogSourceInfo") AssignmentChar ("None"
| "Single" | "Stack")
LogEntityName ::= "LogEntityName" AssignmentChar ("Yes" | "No")
LogFileSize ::= "LogFileSize" AssignmentChar Number
LogFileNumber ::= "LogFileNumber" AssignmentChar Number
DiskFullAction ::= "DiskFullAction" AssignmentChar DiskFullActionValue
DiskFullActionValue ::= ( "Error" | "Stop" | "Retry" ["(" Number ")"] | "Delete" )
EmergencyLogging ::= "EmergencyLogging" AssignmentChar Number
EmergencyLoggingBehaviour ::= "EmergencyLoggingBehaviour" AssignmentChar
    ( "BufferAll" | "BufferMasked" )
EmergencyLoggingMask ::= "EmergencyLoggingMask" AssignmentChar LoggingBitMask
EmergencyLoggingForFailVerdict ::= "EmergencyLoggingForFailVerdict" AssignmentChar
("Yes" | "No")
LogSensitiveData ::= "LogSensitiveData" AssignmentChar ("Yes" | "No")
```

7.2.22. *Example 1*

```
[LOGGING]
LogFile := "/usr/local/TTCN3/logs/%l/%e.%h-%t%r-part%i.%s"
"Alma-Ata".FileMask := LOG_ALL
MyComponent.FileMask := MATCHING
mtc.FileMask := LOG_ALL | MATCHING
ConsoleMask := ERROR | WARNING | TESTCASE | TIMEROP_START
AppendFile := No
TimeStampFormat := DateTime
ConsoleTimeStampFormat := Time
LogEventTypes := No
SourceInfoFormat := Single
LogEntityName := Yes
MatchingHints := Detailed
EmergencyLogging := 2000
EmergencyLoggingBehaviour := BufferAll
#EmergencyLoggingMask := LOG_ALL
```

7.2.23. *Example 2*

```
[LOGGING]
LogFile := "logs/%e-%r.%s"
ConsoleMask := LOG_ALL
FileMask := TESTCASE | ERROR | EXECUTOR | VERDICTOP
TimeStampFormat := Time
LogEventTypes := Yes
SourceInfoFormat := Stack
LogEventTypes := Yes
*.EmergencyLogging:=15
*.EmergencyLoggingBehaviour := BufferMasked
*.EmergencyLoggingMask := LOG_ALL | DEBUG
```

7.3. [TESTPORT_PARAMETERS]

In this section you can specify parameters that are passed to Test Ports. Each parameter definition consists of a component name, a port name, a parameter name and a parameter value. The component name can be either an identifier that is assigned to the component in the `create` operation (see here) or an integer value, which is interpreted as component reference[31]. The port and parameter names are identifiers while the parameter value must be always a `charstring` (with quotation marks). Instead of component name or port name (or both of them) the asterisk (*) sign can be used, which means "all components" or "all ports of the component".

If the keyword `system` is used as a component identifier, the parameter is passed to all ports of all components that are mapped to the given port of the test system interface. In this case the port identifier refers to the port of the test system and not the port of a TTCN-3 test component. These parameters are passed to the appropriate Test Port during the execution of map operations because the future mappings are not known at test component initialization. The asterisk ("*") sign can also be used as port name with the component identifier system. This wildcard means, of course, all

ports of the Test System Interface (that is, the parameter will be passed during all `map` operations).

The names and meaning of Test Port parameters depend on the Test Port that you are using; for this information please consult the user documentation of your Test Port. It is the Test Port writer's responsibility to process the parameter names and values. For the details of Test Port API see the section "Parameter setting function" in [16].

7.3.1. BNF Productions for this Section

```
TestPortParametersSection ::= "[TESTPORT_PARAMETERS]" {TestPortParameter}
TestPortParameter ::= ComponentId "." PortName "." PortParameterName AssignmentChar
PortParameterValue [SemiColon]
ComponentId ::= Identifier | Number | MTCKeyword | SystemKeyword | "*"
MTCKeyword ::= "mtc"
SystemKeyword ::= "system"
PortName ::= Identifier {ArrayRef} | "*"
ArrayRef ::= "[" IntegerValue "]"
PortParameterName ::= Identifier
PortParameterValue ::= StringValue
```

7.3.2. *Example*

```
[TESTPORT_PARAMETERS]
mtc.*.LocalIPAddress := "164.48.161.146"
"Alma-Ata". RemoteIPAddress := "164.48.161.147"
mtc.RADIUS[0].LocalUDPPort := "12345"
mtc.RADIUS[1].LocalUDPPort := "12346"
system.MySystemInterface1.RemoteIPAddress := "10.1.1.1"
system.MySystemInterface2.RemoteIPAddress := "10.1.1.2"
```

7.4. [DEFINE]

In this section you can create macro definitions that can be used in other configuration file sections except `[INCLUDE]`. This way if the same value must be given several times in the configuration file, you can make a definition for it and only refer to the definition later on. In case of a change, you wouldn't need to change the values all over the configuration file, but only in the definition.

This section may contain zero, one or more macro definitions (assignments). Each macro definition consists of a macro identifier, which shall be a TTCN-3 identifier, an assignment operator and the macro value. The macro value is either a simple value, a structured value or the result of a multiplication (see the BNF below).

The simple macro value is a sequence of one or more literal values and macro references. The elements of the sequence must not be separated by anything, whitespaces or comments are not allowed.

The structured macro value can be used to define instances of complex TTCN-3 data structures. The

defined values can be assigned to compound module parameters. There are two restrictions regarding the syntax of this value. The first and last character of the value are '{' and '}'. The value must be well-formed regarding the curly brackets. Every value which satisfies these two rules is accepted as a macro definition. NOTE: macro definitions do not have a type. The defined values are copied to the place of the macro references. The semantic correctness is determined by the context of the macro reference (see the examples section).

Simple macro values containing numbers and references to macros can be multiplied arithmetically, by using asterisk (*) characters between the operands. Whitespaces or comments are not allowed between an operand and an asterisk character. The resulting macro will contain the result of the multiplication as a floating point number.

Macro references can refer to previously defined macros. The reference can be provided in the following 3 formats which have the same meaning:

- `$macroname`
- `${macroname}`
- `${macroname,charstring}`

The above 3 different notations can also be used in other sections to refer to the macro with name "macroname".

The literal value can be either a word (a sequence of arbitrary characters except whitespace) or a character string value delimited by quotation marks. The latter form is useful when the macro value is an empty string or contains whitespace characters. Literal values cannot follow each other, only macro references can.

The values of macros as well as environmental variables set in the shell can be expanded in the configuration file using a special syntax described below. If both a macro and an environment variable are defined with the same name the macro of the configuration file has the precedence. If neither exists an error message is reported. It is possible to assign value to the same macro identifier more than once, in this case the last assignment will determine the value of the macro. When assigning a new value to the same macro, it is also possible to use the macro's previous value.

In parallel mode, in order to ensure the consistency of the test system, all macro substitutions are performed in the Main Controller. Hence the settings of environment variables are inherited from the shell that the Main Controller was started from.

Macro definitions of this section do not change the environment space maintained by the operating system in any process. Thus, the macros defined in this section are not visible by the system call `getenv(3)` issued in test ports or external functions.

Macro references can have one of these two formats:

- Simple reference: a dollar character followed immediately by the macro identifier. Example: `$macroName`. In this case the value of the definition will be inserted as a literal charstring value.
- Modified reference: a dollar character followed by a pair of curly brackets containing the macro identifier and a modifier separated by a comma. Example: `${macroName, modifier}`. Whitespaces

are allowed within the pair of brackets, but the opening bracket must follow the dollar character immediately. In this case the type of the substituted token is specified by the modifier. Before substitution it is verified whether the value of the referred macro or environment variable fulfills the requirements for the given modifier.

The following modifiers are available for macro substitution:

- `integer`

  Transforms the value of the macro into an integer value. The macro value may contain decimal numbers only (leading and trailing whitespaces are not allowed).

- `float`

  Transforms the value of the macro into a value of type float. The substitution is possible only if the value is an integer or a floating point number.

- `boolean`

  Transforms the value of the macro into a boolean value. The macro value shall contain the word true or false.

- `bitstring`

  Transforms the value of the macro into a literal bitstring value. Only binary digits are allowed in the macro value.

- `hexstring`

  Transform the value of the macro into a hexstring value. Only hexadecimal digits are allowed in the macro value.

- `octetstring`

  Transforms the value of the macro into an octetstring. The macro value shall contain even (including zero) number of hexadecimal digits.

- `charstring`

  Transforms the value of the macro into a literal value of type charstring. There is no restriction about the contents of the macro value.

NOTE

The reference with this modifier has the same result as a simple reference.

- `binaryoctet`
  Transforms the value of the macro into an octetstring value so that the octets of the resulting string will contain the ASCII character code of the corresponding character from the macro value. The macro value to be substituted may contain any kind of character.
- `identifier`
  Transforms the value of the macro into a TTCN-3 identifier. This modifier is useful, for instance, for specifying values of enumerated types in section [`MODULE_PARAMETERS`]. The macro value shall contain a valid TTCN-3 identifier. Leading and trailing whitespace characters are not allowed in the macro value.
- `hostname`
  Transforms the value of the macro into a host name, DNS name or IPv4 or IPv6 address. The modifier can be used in sections [`GROUPS`], [`COMPONENTS`] and [`MAIN_CONTROLLER`]. The value to be substituted shall contain a valid host name, DNS name or IP address formed from alphanumerical, dash (-), underscore (_), dot (.), colon(:) or percentage (%) characters. Leading and trailing whitespace is not allowed.

7.4.1. BNF Productions for this Section

```
DefineSection ::= "[DEFINE]" {DefinitionAssignment}
DefinitionAssignment ::= Identifier AssignmentChar DefinitionRValue
DefinitionRValue ::= SimpleValue | StructuredValue | Multiplication
SimpleValue ::= {Word | String | IPaddress | MacroReference}
StructuredValue ::= "{" { {SimpleValue} | StructuredValue } "}"
        | "{" "}"
Multiplication ::= SimpleValue { "*" SimpleValue }
```

`Word` may contain numbers, letters and other non-whitespace characters mixed in any way.

7.4.2. *Example*

```
[DEFINE]
Localhost := 127.0.0.1
binary_240 := 11110000
four := 4.0
LongString := "This is a very long string."
x1 = "Connecting to "${Localhost}
x2 = $LongString${Localhost,charstring}" is an IP address"
binary_str := ${binary_240}010101

/* Examples for the structured macro definitions */
// on the left side of the arrow is the definition
```

```
// the substituted value is on the right side
DEF_20 := 1 // 1
DEF_21 := "1"   // 1
DEF_22 := "\"1\""   // "1"
DEF_23 := a // a
DEF_24 := "a"   // a
DEF_25 := "\"a\""   // "a"

DEF_30 := { f1 := ${DEF20}} // => DEF_30 := { f1 := 1}
DEF_31 := { f1 := ${DEF21}} // => DEF_31 := { f1 := 1}
DEF_32 := { f1 := ${DEF22}} // => DEF_32 := { f1 := "1"}
DEF_33 := { f1 := ${DEF23}} // => DEF_33 := { f1 := a}
DEF_34 := { f1 := ${DEF24}}     // => DEF_34 := { f1 := a}
DEF_35 := { f1 := \"${DEF24}\"}     // => DEF_35 := { f1 := "a"}
DEF_36 := { f1 := ${DEF25}}     // => DEF_36 := { f1 := "a"}
DEF_37 := { f1 := a}        // => DEF_37 := { f1 := a}
DEF_38 := { f1 := "a"}      // => DEF_38 := { f1 := "a"}
DEF_39 := { f1 := "${DEF_20}"}  // => DEF_39 := { f1 := "${DEF_20}"}
// DEF_30 and DEF_31 are valid module parameter definitions for tsp_1
// the other definitions are not valid for tsp_1


DEF_40 := { f2 := ${DEF20}}     // => DEF_40 := { f2 := 1}
DEF_41 := { f2 := ${DEF21}} // => DEF_41 := { f2 := 1}
DEF_42 := { f2 := ${DEF22}}     // => DEF_42 := { f2 := "1"}
DEF_43 := { f2 := ${DEF23}}     // => DEF_43 := { f2 := a}
DEF_44 := { f2 := ${DEF24}} // => DEF_44 := { f2 := a}
DEF_45 := { f2 := \"${DEF24}\"}     // => DEF_45 := { f2 := "a"}
DEF_46 := { f2 := ${DEF25}}     // => DEF_46 := { f2 := "a"}
DEF_47 := { f2 := a}        // => DEF_47 := { f2 := a}
DEF_48 := { f2 := "a"}      // => DEF_48 := { f2 := "a"}
DEF_49 := { f2 := "${DEF_20}"}  // => DEF_49 := { f2 := "${DEF_20}"}
// DEF_{42|45|46|48|49} are valid module parameter definitions for tsp_1
// the other definitions are not valid for tsp_1

// complex data structures can also be referenced
DEF_50 := { f1 := ${DEF_42}, f2 := "a"}


// Multiplication example:
TEN := 10
TWO := 2.0

RESULT := ${TEN}*${TWO}*1.5
```

7.4.3. *Use example:*

```
[MODULE_PARAMETERS]
par1 := $Localhost // "127.0.0.1"
par2 := ${binary_240, bitstring} // '11110000'B
par3 := ${binary_240, hexstring} // 'F0'H
par4 := ${four, float} // 4.0
par5 := ${four, binaryoctet} // '342E30'O
par6 := ${LongString, identifier} // ERROR: invalid substitution
par7 := "$myVariable" // substitution is not done
par8 := ${RESULT, float} // 30.0
[MAIN_CONTROLLER]
LocalAddress = ${Localhost, hostname} // 127.0.0.1
```

The tokens substituted are given in comments.

7.4.4. *TTCN file example*

```
// ttcn
module a {
  modulepar Rec tsp_1;
  modulepar Rec2 tsp_1;
  type record Rec {
    integer f1 optional, charstring f2 optional
  }
  type record Rec2 {
    Rec f1 optional, charstring f2 optional
  }
}
```

7.5. [INCLUDE]

It is possible to use configuration settings (module parameters, test port parameters, etc.) given in other configuration files. The configuration files just need to be listed in this section, with their full or relative pathnames. To the host controllers it will look like as if the configuration files would have been merged together into one configuration file.

Each included file shall form a valid configuration file with complete section(s). The `[INCLUDE]` directives of included files are processed recursively. Each referenced configuration file is processed exactly once even if it is included from several places. Relative pathnames are resolved based on the directory of the referring configuration file.

7.5.1. BNF Productions for this Section

```
IncludeSection ::= "[INCLUDE]" {IncludeFile}
IncludeFile ::= Cstring
```

The file's name is a character string, given between quotation marks.

Example

```
[INCLUDE]
"base_definitions.cfg"
"../additional_parameters.cfg"
```

7.6. [ORDERED_INCLUDE]

It is possible to include configuration files to a specific location using the [ORDERED_INCLUDE] section. The included file can be given with the same syntax as in the [INCLUDE] section. The file can be specified with an absolute path, or a path relative to the configuration file in which the [ORDERED_INCLUDE] section takes place. Relative pathnames are resolved based on the directory of the referring configuration file.

Each included file shall form a valid configuration file with complete section(s). Circular imports are not accepted. Compared to the "normal" [INCLUDE], the [ORDERED_INCLUDE] processes an included configuration file as many times as it is specified (even if it is included transitively). This behavior has (intentional) side-effects, i.e. during the merging of the configuration files, the described configuration will be repeated as many times as the file is included. For example, the content of an [EXECUTE] will be executed more times, or if &= assignment is used, the final value will be calculated based on the repeated expression. This behavior gives a higher freedom to the users to precisely configure the test environment and execution, but it can also increase the complexity of managing the configuration.

7.6.1. BNF Productions for this Section

```
OrderdIncludeSection ::= "[ORDERED_INCLUDE]" {IncludeFile}
IncludeFile ::= Cstring
```

The file's name is a character string, given between quotation marks.

Example

```
// main.cfg
[ORDERED_INCLUDE]
"oi.cfg"
"oi2.cfg"
[MODULE_PARAMETERS]
tsp_1 := 3

// oi.cfg
[MODULE_PARAMTERS]
tsp_1 := 1
// oi2.cfg
[MODULE_PARAMETERS]
tsp_1 := 2
```

In this example we have 3 configuration files. The names of the files are included as comments. The ETS will be started with the first one ("main.cfg"). This configuration file includes "oi.cfg" and "oi2.cfg". The included files are processed sequentially. The first included file ("oi.cfg") will set the module parameter "tsp_1" to 1. As the processing continues, the second included file ("oi2.cfg") will set it to 2. Finally when the included files are processed, the main configuration file sets it to 3. In this case, the module parameter named tsp_1 will have the final value of 3.

7.7. [EXTERNAL_COMMANDS]

This section defines external commands (shell scripts) to be executed by the ETS whenever a control part or test case is started or terminated. Using this feature you can control external monitor programs (like `tcpdump` in case of IP testing) automatically during test execution. In case of parallel mode, the external command is executed on the host where the MTC runs. The name of the corresponding module or test case is passed to the external command as argument. For `BeginTestCase` and `EndTestCase` the name of the module and test case separated with a dot is passed as argument; and additionally the test case verdict for `EndTestCase`. For example, this allows you to collect the output of `tcpdump` in separate files for each test case where the file name contains the name of the test case.

All commands are optional and can be set independently. The command name (or full path) must be given within double quotes. Whitespaces and special characters are treated as part of the command name and will not be interpreted by the shell. This means that additional, fixed, arguments can not be passed to the external command. If the command string is empty no command will be executed (it also clears the command that was set previously).

7.7.1. BNF Productions for this Section

```
ExternalCommandsSection ::= "[EXTERNAL_COMMANDS]" {ExternalCommand}
ExternalCommand ::= CommandType AssignmentChar Command [SemiColon]
CommandType ::= "BeginControlPart" | "EndControlPart" | "BeginTestCase" |
    "EndTestCase"
Command ::= StringValue
Example
[EXTERNAL_COMMANDS]
BeginTestCase := "/usr/local/tester/bin/StartTcpdump"
EndTestCase := "/usr/local/tester/bin/StopTcpdump"
BeginControlPart := "this will be overwritten"
EndControlPart := ""
```

7.7.2. Example: Running `tcpdump` during test execution

In case of testing IP based protocols it might be useful to monitor the network during TTCN-3 test execution. The following shell scripts show an example how to start the program `tcpdump` in the background at the beginning of every test case and how to terminate it when a test case is finished.

When `tcpdump` is running, its `pid` is stored in the file `/etc/tcpdump.pid` to inform the stopping script which process to kill. Of course, the command line options for tcpdump may be changed to fit your

needs. The output of `tcpdump` is saved in the file `<testcase name>.dump` in the working directory of the executable test program, which is useful when `repgen` is used after test execution.

To make this working, you should give the names or full pathes of these scripts as `BeginTestCase` and `EndTestCase` in section `[EXTERNAL_COMMANDS]` of the configuration file.

A complete example script for starting `tcpdump`:

```
#!/bin/sh

PIDFILE=/tmp/tcpdump.pid

if [ -e $PIDFILE ]
then
  kill `cat $PIDFILE`
  rm $PIDFILE
fi

/usr/local/sbin/tcpdump -e -n -s 200 -x -v -i eth1 ip6 >$1.dump \
  2>/dev/null &
PID=$!

echo $PID >$PIDFILE
```

The script for stopping `tcpdump`:

```
#!/bin/sh

PIDFILE=/tmp/tcpdump.pid

if [ -e $PIDFILE ]
then
  kill `cat $PIDFILE`
  rm $PIDFILE
fi
```

7.8. [EXECUTE]

In this section you have to specify what parts of your test suite you want to execute. In single mode the configuration file is useless without this section. The section [EXECUTE] is optional in parallel mode. If it is missing, You shall start testcases manually from command line with the command `smtc [module name[.control|.testcase name|.*]]` see UG [17] 4.4.2.1. In this case a parameter after smtc is mandatory. Don't omit this section in case of using `ttcn3_start`, otherwise no testcase will be executed.

You can start TTCN-3 module control parts and test cases individually. There is one limitation: only those test cases having no parameters, or only parameters with default values, can be executed from this section. Other test cases can be started from the module control part with proper actual

parameters.

In this section, a single identifier (or an identifier followed by the optional suffix `.control`) means the control part of that TTCN-3 module. Test case names shall be preceded by the name of module that they can be found in and a dot character. You can use the character asterisk (*) instead of test case name, which means the execution of all test cases of the corresponding module in the same order as they are defined in the TTCN-3 source code.

The control parts and test cases are executed in the same order as you specified them in this section. If you define the same module or test case name more than once, that control part or test case will be executed, of course, many times.

7.8.1. The BNF Specification of this Section

```
ExecuteSection ::= "[EXECUTE]" {ExecuteItem}
ExecuteItem ::= (ControlPart | TestCase) [SemiColon]
ControlPart ::= ModuleName [ "." "control" ]
ModuleName ::= Identifier
TestCase ::= ModuleName "." TestCaseName
TestCaseName ::= Identifier | "*"
```

Example

```
[EXECUTE]
IPv6Demo.send_echo
IPv6Demo.send_echo // run it twice
IPv6BaseSpecification
IPv6NeighborDiscovery.*
```

7.9. [GROUPS] (Parallel mode)

In this section you can specify groups of hosts. These groups can be used inside the [COMPONENTS] section to restrict the creation of certain PTCs to a given set of hosts. See also here.

This section contains any number of group specifications in the following form: group name, assignment operator (:=) and either an asterisk (*) or a comma-separated list of host names (DNS names) or IP addresses in which you should enlist each hosts belonging to that group. The asterisk appearing on the right side denotes all hosts that take part in the test execution.

Groups may overlap, that is, the same hosts can belong to several groups. Group references, however, cannot appear on the right side. It is worth mentioning that group names are case sensitive.

NOTE	The groups defined in this section have nothing to do with TTCN-3 group of definitions construct!

7.9.1. The BNF Specification of this Section

```
GroupsSection ::= "[GROUPS]" {GroupItem}
GroupItem ::= GroupName AssignmentChar (GroupMemberList | "*") [SemiColon]
GroupName ::= Identifier
GroupMemberList ::= GroupMember {"," GroupMember}
GroupMember ::= HostName | IPAddress
```

Example

```
[GROUPS]
HeintelAndPauler := heintel, pauler.eth.ericsson.se
myGroup := 153.44.87.34, test-host.123.com
AllHosts := *
```

7.10. [COMPONENTS] (Parallel mode)

This section consists of rules restricting the location of created PTCs. These constraints are useful when distributed tests are executed in a heterogeneous environment. The participating computers may have different hardware setup, computing capacity or operating system. Thus some physical interfaces or Test Ports might be present only on a part of the hosts[17].

The rules are described in form of assignments. The left side contains a component identifier while the right side names a host or a group of hosts on which the given components are executed. The components can be selected by their component type or name assigned in create operations. The component identifiers are case sensitive. The assigned hosts are taken from the corresponding host group set from the section [GROUPS].

Each component type or component name can appear in at most one rule. The asterisk (*) stands for all component identifiers that do not appear in any rule. The asterisk can show in a single rule only.

When a TTCN-3 parallel test component is being created it is the responsibility of the MC to choose a suitable and available[18] host for it. First a subset of available hosts, the set of so-called candidates, is determined based on the component distribution rules. The MC implements a load balancing algorithm so that the location of the component will be the candidate with the smallest load, that is, the least number of active TTCN-3 test components[19]. Once a component is assigned to a host it, cannot be moved to another one later during its life.

If a newly created PTC matches more than one rule (because both its component type and name is found in the section) all available members of both assigned groups are considered to be candidates.

If section [COMPONENTS] is empty or omitted from the configuration file all available hosts are considered to be candidates. If the calculated set of candidates is an empty set (i.e. there is no available host that is allowed by the rules) the `create` operation will fail and dynamic test case error will occur on the ancestor component.

If the location of the PTC is explicitly specified in the `create` operation (see here for the syntax of this language extension) the rules of this section are ignored. In this case the set of candidates is determined based on the host name or group name that was specified as location.

7.10.1. The BNF Specification of this Section

```
ComponentsSection ::= "[COMPONENTS]" {ComponentItem}
ComponentItem ::= ComponentId AssignmentChar ComponentLocation [SemiColon]
ComponentId ::= Identifier | "*"
ComponentLocation ::= GroupName | HostName | IPAddress
```

Example

```
[COMPONENTS]
MyComponentType := HeintelAndPauler
CPComponentType := 153.44.87.34
* := AllHosts
```

7.11. [MAIN_CONTROLLER] (Parallel mode)

The options herein control the behavior of MC. The section [MAIN_CONTROLLER] includes four options to be set.

Options `LocalAddress` and `TCPPort` determine the IP address and TCP port on which the MC application will listen for incoming HC connections. Setting `LocalAddress` can be useful on computers having multiple local IP addresses (multi-homed hosts). The value of `LocalAddress` can be either an IP address or a DNS name, which must resolve to an address that belongs to a local network interface. If this option is omitted MC will accept connections on all local IP addresses.

The value of option `TCPPort` is an integer number between 0 and 65535. The recommended port number is 9034. Using a TCP port number that is less than 1024 may require super-user (root) privileges. The MC will listen on an ephemeral port chosen by the kernel when `TCPPort` is omitted or set to zero.

The optional variable `NumHCs` provides support for automated (batch) execution of distributed tests. When present, the MC will not give a command prompt, but wait for `NumHCs` HCs to connect. When the specified number of HCs are connected, the MC automatically creates MTC and executes all items of the section [EXECUTE]. When finished, the MTC is terminated and the MC quits automatically. If `NumHCs` was omitted then the MC shall be controlled interactively, that is, you have to issue the commands `cmtc` and `smtc` yourself (see also sections 12.3, 12.3.1 of the TITAN User Guide [13]).

The `KillTimer` option tells the MC to wait some seconds for a busy test component (MTC or PTC) to terminate when it was requested to stop[20]. The MC in co-operation with the local HC kills the UNIX process if the component did not terminate properly before `KillTimer` expiry. The purpose of this function is to prevent the test system from deadlocks.

NOTE	When the UNIX process of MTC is killed all existing PTCs are destroyed at the same time.

The value of KillTimer is measured in seconds and can be given in either integer or floating point notation. Setting KillTimer to zero disables the kill functionality, that is, busy test components will not be killed even if they do not respond within a very long time period. When omitted, the default value of KillTimer is 10 seconds. This value is sufficient in typical test setups, but it needs to be increased on heavily loaded computers (e.g. when running performance tests). Setting a too short KillTimer value may have undesired effects as the final verdict of killed PTCs, which is not known by MC, is always substituted by error.

UnixSocketsEnabled has a default value of "yes". When at default value, Titan will use Unix domain sockets for internal communication on the same machine, and TCP sockets to communicate across the network. When set to "no", TCP sockets will be used both internally and over the network.

7.11.1. The BNF Specification of this Section

```
MainControllerSection ::= "[MAIN_CONTROLLER]" {MainControllerAssignment}
MainControllerAssignment ::= (LocalAddress | TCPPort | NumHCs | KillTimer |
  UnixSocketsEnabled) [SemiColon]
LocalAddress ::= "LocalAddress" AssignmentChar (HostName | IPAddress)
TCPPort ::= "TCPPort" AssignmentChar IntegerValue
NumHCs ::= "NumHCs" AssignmentChar IntegerValue
KillTimer ::= "KillTimer" AssignmentChar (IntegerValue | FloatValue)
UnixSocketsEnabled ::= "UnixSocketsEnabled" AssignmentChar ("Yes" | "No")
```

Example:

```
[MAIN_CONTROLLER]
LocalAddress := 192.168.1.1
TCPPort := 9034
NumHCs := 3
KillTimer := 4.5
UnixSocketsEnabled := Yes
```

7.12. [PROFILER]

The settings in this section control the behavior of the TTCN-3 Profiler. These settings only affect the TTCN-3 modules specified in the file list argument of the compiler option -z. If this compiler option is not set, then the [PROFILER] section is ignored.

7.12.1. Enabling and disabling features

The following features can be enabled or disabled through the configuration file:

- DisableProfiler - if set to true, the measurement of execution times is disabled and data related

to execution times or average times will not be present in the statistics file. Default value: false

- `DisableCoverage` - if set to `true`, the execution count of code lines and functions is not measured and data related to execution counts, average times or unused lines/functions will not be present in the statistics file. Default value: `false`
- If both `DisableProfiler` and `DisableCoverage` are set to `true`, then the profiler acts as if it wasn't activated in any of the modules (as if the compiler flag -z was not set). The database and statistics files are not generated.
- `AggregateData` - if set to `true`, the data gathered in the previous run(s) is added to the current data, otherwise all previous data is discarded, default value: `false`
- `DisableStatistics` - if set to `true`, the statistics file will not be generated at the end of execution, default value: `false`
- `StartAutomatically` - if set to `true`, the profiler will start when the program execution starts, otherwise it will only start at the first @profiler.start command (it needs to be started individually for each component in parallel mode), default value: `true`
- `NetLineTimes` - if set to `true`, the execution times of function calls will not be added to the caller lines' total times, default value: `false`
- `NetFunctionTimes` - if set to `true`, the execution times of function calls will not be added to the caller functions' total times, default value: `false`

7.12.2. Setting output files

The DatabaseFile setting can be used to specify the name and path of the database file (as a string with double quotation marks, like a `charstring`). This is the file imported by the profiler if data aggregation is set (if this setting is changed between runs, the profiler will not find the old database).

Default value: `profiler.db`

Similarly the `StatisticsFile` setting can be used to specify the name and path of the statistics file.

Default value: `profiler.stats`

The names of both files may contain special metacharacters, which are substituted dynamically during test execution. These are helpful when there are multiple Host Controllers in the profiled test system.

The table below contains the list of available metacharacters in alphabetical order. Any unsupported metacharacter sequence, that is, if the % character is followed by any character that is not listed in the table below or a single percent character stays at the end of the string, will remain unchanged.

Table 26. Available metacharacters for setting profiler output file names

Meta-character	Substituted with . . .
`%e`	the name of the TTCN-3 executable. The `.exe` suffix (on Windows platforms) and the directory part of the path name (if present) are truncated.

Meta-character	Substituted with . . .
%h	the name of the computer returned by the gethostname(2) system call. This usually does not include the domain name.
%l	the login name of the current user. If the login name cannot be determined (e.g. the current UNIX user ID has no associated login name) an empty string is returned.
%p	the process ID (pid) of the UNIX process that implements the current test component. The pid is written in decimal notation.
%%	a single % character.

7.12.3. Statistics filters

The StatisticsFilter setting can be used to specify which lists will be calculated and displayed in the statistics file. Its value is a list of filters separated by ampersands (&). Vertical lines (|) can also be used to separate the filters (as if they were bits added together with binary or) to the same effect.

The concatenation mark (&=) can also be used with this setting to specify the filters in multiple commands.

The filters can also be specified with hexadecimal values (similarly to hexstrings, but without the quotation marks and the H at the end).

Table 27. Statistics filters, single lists

Filter	Numeric value	Represented list
NumberOfLines	0x00000001	Number of code lines and functions
LineDataRaw	0x00000002	Total time and execution count of code lines (raw)
FuncDataRaw	0x00000004	Total time and execution count of functions (raw)
LineAvgData	0x00000008	Average time of code lines (raw)
FuncAvgData	0x00000010	Average time of functions (raw)
LineTimesSortedByMod	0x00000020	Total time of code lines (sorted, per module)
FuncTimesSortedByMod	0x00000040	Total time of functions (sorted, per module)
LineTimesSortedTotal	0x00000080	Total time of code lines (sorted, total)

Filter	Numeric value	Represented list
FuncTimesSortedTotal	0x00000100	Total time of functions (sorted, total)
LineCountSortedByMod	0x00000200	Execution count of code lines (sorted, per module)
FuncCountSortedByMod	0x00000400	Execution count of functions (sorted, per module)
LineCountSortedTotal	0x00000800	Execution count of code lines (sorted, total)
FuncCountSortedTotal	0x00001000	Execution count of functions (sorted, total)
LineAvgSortedByMod	0x00002000	Average time of code lines (sorted, per module)
FuncAvgSortedByMod	0x00004000	Average time of functions (sorted, per module)
LineAvgSortedTotal	0x00008000	Average time of code lines (sorted, total)
FuncAvgSortedTotal	0x00010000	Average time of functions (sorted, total)
Top10LineTimes	0x00020000	Total times of code lines (sorted, total, top 10 only)
Top10FuncTimes	0x00040000	Total times of functions (sorted, total, top 10 only)
Top10LineCount	0x00080000	Execution count of code lines (sorted, global, top 10 only)
Top10FuncCount	0x00100000	Execution count of functions (sorted, total, top 10 only)
Top10LineAvg	0x00200000	Average time of code lines (sorted, total, top 10 only)
Top10FuncAvg	0x00400000	Average time of functions (sorted, total, top 10 only)
UnusedLines	0x00800000	Unused code lines
UnusedFunc	0x01000000	Unused functions

Table 28. Statistics filters, grouped lists

`AllRawData`	`0x0000001E`	**Total time, execution count and average time of code lines and functions (raw)**
`LineDataSortedByMod`	`0x00002220`	Total time, execution count and average time of code lines (sorted, per module)
`FuncDataSortedByMod`	`0x00004440`	Total time, execution count and average time of functions (sorted, per module)
`LineDataSortedTotal`	`0x00008880`	Total time, execution count and average time of code lines (sorted, total)
`FuncDataSortedTotal`	`0x00011100`	Total time, execution count and average time of functions (sorted, total)
`LineDataSorted`	`0x0000AAA0`	Total time, execution count and average time of code lines (sorted, total and per module)
`FuncDataSorted`	`0x00015540`	Total time, execution count and average time of functions (sorted, total and per module)
`AllDataSorted`	`0x0001FFE0`	Total time, execution count and average time of code lines and functions (sorted, total and per module)
`Top10LineData`	`0x002A0000`	Total time, execution count and average time of code lines (sorted, total, top 10 only)
`Top10FuncData`	`0x00540000`	Total time, execution count and average time of functions (sorted, total, top 10 only)
`Top10AllData`	`0x007E0000`	Total time, execution count and average time of code lines and functions (sorted, total, top 10 only)
`UnusedData`	`0x01800000`	Unused code lines and functions
`All`	`0x01FFFFFF`	All lists

NOTE	the `DisableProfiler` and `DisableCoverage` settings also influence which lists are displayed in the statistics file (e.g.: if `DisableCoverage` is set to `true` and `StatisticsFilter` is set to `Top10LineData`, then the statistics file will only contain the top 10 total times list).

7.12.4. The BNF Specification of this Section

```
ProfilerSection ::= "[PROFILER]" {ProfilerSetting}
ProfilerSetting ::= (DisableProfilerSetting | DisableCoverageSetting |
  DatabaseFileSetting | AggregateDataSetting | StatisticsFileSetting |
  DisableStatisticsSetting | StatisticsFilterSetting) [SemiColon]
DisableProfilerSetting ::= "DisableProfiler" AssignmentChar BooleanValue
DisableCoverageSetting ::= "DisableCoverage" AssignmentChar BooleanValue
DatabaseFileSetting ::= "DatabaseFile" AssignmentChar CharstringValue
AggregateDataSetting ::= "AggregateData" AssignmentChar BooleanValue
StatisticsFileSetting ::= "StatisticsFile" AssignmentChar CharstringValue
DisableStatisticsSetting ::= "DisableStatistics" AssignmentChar BooleanValue
StatisticsFilterSetting ::= "StatisticsFilter" (AssignmentChar | ConcatChar)
  StatisticsFilter [ { ("&" | "|") StatisticsFilter } ]
StatisticsFilter ::= ("NumberOfLines" | "LineDataRaw" | "FuncDataRaw" |
  "LineAvgRaw" | "FuncAvgRaw" | "LineTimesSortedByMod" | "FuncTimesSortedByMod"
  | "LineTimesSortedTotal" | "FuncTimesSortedTotal" | "LineCountSortedByMod" |
  "FuncCountSortedByMod" | "LineCountSortedTotal" | "FuncCountSortedTotal" |
  "LineAvgSortedByMod" | "FuncAvgSortedByMod" | "LineAvgSortedTotal" |
  "FuncAvgSortedTotal" | "Top10LineTimes" | "Top10FuncTimes" | "Top10LineCount"
  | "Top10FuncCount" | "Top10LineAvg" | "Top10FuncAvg" | "UnusedLines" |
  "UnusedFunc" | {Hex}+)
```

Example:

```
[PROFILER]
DisableProfiler := false
DisableCoverage := false
DatabaseFile := "data.json"
AggregateData := false
StatisticsFile := "prof.stats"
DisableStatistics := false
StatisticsFilter := Top10AllData & UnusedData
StatisticsFilter &= AllRawData
StatisticsFilter &= 88A0
```

7.13. Dynamic Configuration of Logging Options

Some logging options may be altered during the run of the test suite. This allows e.g. to vary the logging verbosity between testcases.

The interface is contained in `$(TTCN3_DIR)/include/TitanLoggerControl.ttcn`; this file needs to be

added to the project. `TitanLoggerControl.ttcn` contains definitions of various external functions which can be called from TTCN-3 code. The implementation of these external functions is built into the Titan runtime library; it will be linked automatically.

The individual logging severities are contained in the "Severity" enumerated type. A logging mask is represented by a record-of:

```
type record of Severity Severities;
```

For each logging bit set, the record-of will contain an element.

The TitanLoggerControl module defines several constants:

```
const Severities log_nothing    := {};
const Severities log_console_default := { ⋯ }
const Severities log_all := { ⋯ } // LOG_ALL, without MATCHING,DEBUG
const Severities log_everything := { ⋯ } // really everything
```

These constants can be used when setting logger options.

Each function has a parameter named `plugin`, to specify which `plugin` is being manipulated. Currently, the value of the plugin parameter must be `"LegacyLogger"`.

7.13.1. Retrieving Logging Masks

The following functions can be used to retrieve the current value of the logger file mask/console mask:

```
get_file_mask(in charstring plugin) return Severities;
get_console_mask(in charstring plugin) return Severities;
```

7.13.2. Setting Logging Masks

The following functions set the file mask or console mask, overwriting the previous value:

```
set_file_mask(in charstring plugin, in Severities s);
set_console_mask(in charstring plugin, in Severities s);
```

Logging severities present in the parameter will be switched on; severities absent from the parameter will be switched off.

The following functions allow adding individual Severity values to the list of events that are logged, without affecting other severities:

```
add_to_file_mask(in charstring plugin, in Severities s);
add_to_console_mask(in charstring plugin, in Severities s);
```

Logging severities present in the parameter will be switched on; severities absent from the parameter will not be modified. Example: to turn on DEBUG_ENCDEC without affecting other severities:

```
var Severities encdec := { DEBUG_ENCDEC }
TitanLoggerControl.add_to_file_mask("LegacyLogger", encdec);
```

NOTE	Each bit is treated individually. To turn on a first level category, one needs to enumerate all subcategories. For example, to turn on all `DEBUG` messages, the value of the Severities variable should be:

```
var Severities debug_all := { DEBUG_ENCDEC, DEBUG_TESTPORT, DEBUG_UNQUALIFIED };
TitanLoggerControl.add_to_file_mask("LegacyLogger", debug_all);
```

The following functions allow removing of individual Severities:

```
remove_from_file_mask(in charstring plugin, in Severities s);
remove_from_console_mask(in charstring plugin, in Severities s);
```

Logging severities present in the parameter will be switched off; severities absent from the parameter will not be modified. When setting file/console masks, redundant severity values are ignored. For example, the two following values have the same effect when passed to set_file_mask.

```
var Severities ptc := { PARALLEL_PTC };
var Severities ptc3:= { PARALLEL_PTC, PARALLEL_PTC, PARALLEL_PTC };
```

7.13.3. Manipulating the Log File Name

The following function allows setting the log file name skeleton. See here for possible values.

```
set_log_file(in charstring plugin, in charstring filename);
```

7.13.4. Enabling/disabling the Logging of TTCN-3 Entity Name

The following functions allow the reading and writing of the `LogEntityName` parameter.

```
set_log_entity_name(in charstring plugin, in boolean b);
get_log_entity_name(in charstring plugin) return boolean;
```

[14] In order to be compatible with older configuration files, the following symbolic constants are also recognized: TTCN_ERROR, TTCN_WARNING, TTCN_PORTEVENT, TTCN_TIMEROP, TTCN_VERDICTOP, TTCN_DEFAULTOP, TTCN_TESTCASE, TTCN_ACTION, TTCN_USER, TTCN_FUNCTION, TTCN_STATISTICS, TTCN_PARALLEL, TTCN_EXECUTOR, TTCN_MATCHING and TTCN_DEBUG. These constants have the same meaning as their counterparts without the prefix TTCN_.

[15] Everyone writing encoder/decoder functions should implement logging in this subcategory.

[16] In mixed ports message and proc. ports cannot be distinguished

[17] On the remaining computers the unsupported Test Ports shall be substituted with empty stubs (i.e. generated and unmodified skeletons).

[18] Only those hosts participate in the component distribution algorithm that have an active HC, which has been started by the user. MC ignores all unavailable group members silently and will not start the HC on them.

[19] This term of load has no direct relation to the load average calculated by UNIX kernels.

[20] The MTC can be terminated from the MC's user interface or from a PTC by executing the mtc.stop operation. The termination of a PTC can be requested either explicitly (using a TTCN-3 component stop or kill operation) or implicitly (at the end of test case).

Chapter 8. The TITAN Project Descriptor File

Concept of the project is defined in chapter 4 of User Guide for the TITAN Designer [17] because it is introduced in TITAN Designer which is the name of one of the Eclipse plugin of TITAN.

The content of a project and all project specific settings are described in the TITAN project descriptor (TPD) file. The name of the TPD file is in the form `<project_name>.tpd` e.g. `HelloTitan.tpd`.

The tpd file contains information about the project name, the contained files, referenced (used) projects, make, configuration and running information etc. (se later) but doesn't contain source files at all.

Tpd files are designed to produce portability for projects. Primarily it is automatically created by the Titan Designer (in Eclipse) when project settings of an existing project are exported (see Chapter 4.9.4 of [17]). The created TPD file can be used to import the project into another workspace (belonging to the same or to a different user) if the files and projects referred by the TPD are available at the new place (see Chapter 4.9.5 of [17]). If the project to be exported contains referenced (used) project, then the TITAN project setting of the referenced projects shall be exported previously.

Tpd files can be used without TITAN Designer for example for generating hierarchical Makefiles from command line by the makefile generator (see here). The files and TPD files referred by the TPD should be available.

TPD files are designed to be created by the Eclipse Designer but it can be created or modified by experts.

The data is stored in XML format. The schema definition of the project descriptor is delivered with the Titan package and can be found in the `${TTCN3_DIR}/etc/xsd/TPD.xsd` file. If the type of an element is a subtype of string, for example a file name or a path, the xsd file doesn't restrict it but uses only string or NormalizedString as type. Although the compliance of a TPD file for TPD.xsd is a necessary but not sufficient statement, it is very useful to verify the tpd file against TPD.xsd for example with this command: `xmllint -noout -schema XSD_FILE XML_FILE`

Each element is designed to be extendable, so for almost all elements there is a default value defined, if the element is missing this value is used.To simplify the reviews the elements are always saved in an ordered fashion thus limiting the effects of minor changes. The lists can contain from 0 to infinite elements. The O/M column in the tables describes if the element is optional or mandatory.

The TPD file contains a top level element <TITAN_Project_File_Information>.

This has an attribute called "version". This attribute describes the version of the TPD.xsd file which the tpd file validates to. e.g.:

`<TITAN_Project_File_Information version="1.0">`. Currently this version is fixed.

The elements contained in the top level element are listed below in Table.

The content of these elements is discussed later in this chapter.

Table 29. The sequence of elements of TITAN_Project_File_Information

Name of the element	Meaning	O/M
`ProjectName`	The name of the project. This name is used as the name of the TPD file used for export. This is also the name of the project created by TITAN Designer at import. See also here	M
`ReferencedProjects`	The list of projects referenced by the actual one.Not present if the actual project does not reference any other. See also here	O
`Folders`	The list of folders present in the project.Not present if the actual project does not contain any folders to be saved. See also here	O
`Files`	The list of files present in the project.Not present if the actual project does not contain any files to be saved. See also here	O
`PathVariables`	The list of eclipse path variables active in the workbench.Not present if the actual workbench did not contain any path variables. See also here	O
`ActiveConfiguration`	The name of the build configuration active whose parameters will be used by TITAN for building the project. See also here	M
`Configurations`	The list of configurations set on the actual project. Please note, that there is always at least one configuration, the "Default", set for every project. See also here	O

Name of the element	Meaning	O/M
`PackedReferencedProjects`	PackedReferencedProjects contains not references for other TPDS but the content of the TPDs belonging to the referenced projects (without element PackedReferencedProjects). Not present if the actual project is not referencing any other, or if the saving of this data is not explicitly requested by the user. See also here	O

8.1. Project Name

The ProjectName element contains the name of the project.

Example:

```
<ProjectName>HelloTitan2</ProjectName>
```

8.2. Referenced Projects

The `ReferencedProjects` element contains a list of `ReferencedProject` elements, each of which describes a single project referencing relation. A referenced project is a project whose content is used by our project, e.g. its files are imported by the file of the current project.

The following tags are supported:

Table 30. Attributes of ReferencedProject

Name of the attributes	Meaning	O/M
`name`	The name of the project referenced. This will create the relation between the actual and the referenced project. The value of this attribute must be equal to the `ProjectName` of the referenced project.	M

Name of the attributes	Meaning	O/M
projectLocationURI	The path of the project descriptor of the referenced project, relative to the actual descriptor. If the project is not already loaded in eclipse, it will be loaded from this path.	M
tpdName	The file name of the TPD file. This attribute is used with the makefilegen -I switch to provide search paths to find the referenced project if it is not found at projectLocationURI.	O

Example:

```
<ReferencedProjects>
       <ReferencedProject name="Hello_world"
projectLocationURI="../Hello_world/Hello_world.tpd"/>
</ReferencedProjects>
```

8.3. Files and Folders

These elements contain some basic information on files and folders present in the project that is needed to recreate the structure of the project anytime later.

The Files element is a list of FileResource elements, the Folders element is a list of FolderResource elements.

Right now the contents of the FileResource and FolderResource elements are the same. All information is stored in attributes.

Table 31. The attributes of FileResource *and* FolderResource.

Name of the attribute	Meaning	O/M
projectRelativePath	The path of the resource inside the resource system of eclipse.	M
relativeURI	This is the path relative to the location where the project descriptor files is being saved, in case the path of the resource does not contain any feature, that needs to be resolved, and it is possible to calculate the relative path.	O

Name of the attribute	Meaning	O/M
rawURI	This is the raw path of the resource as stored by the platform. In this form the path variables, or any other feature, are not yet resolved.	O

If both tags are present the relativeURI should be used. It is an error if neither the relativeURI nor the rawURI attribute is present.

Example:

```
<Folders>
    <FolderResource projectRelativePath="src" relativeURI="file:src"/>
    <FolderResource projectRelativePath="virtual" rawURI="virtual:/virtual"/>
  </Folders>
  <Files>
    <FileResource projectRelativePath=".TITAN_properties"
relativeURI="file:.TITAN_properties"/>
    <FileResource projectRelativePath=".project" relativeURI="file:.project"/>
  </Files>
```

8.4. Path Variables

The PathVariables element stores the list of PathVariable elements, each of which describes a single eclipse path variable. They are not used at Makefile generation from command line.

Table 32. Attributes of PathVariable

Name of the attribute	Meaning	O/M
Name	The name of the path variable.	M
Value	The value of the path variable.	M

Example:

```
<PathVariables>
    <PathVariable name="path_variable1" value="C:/ekrisza"/>
    <PathVariable name="path_variable2" value="C:/Users/ekrisza/doksi-workbench-
workspace/masik/masik.TITAN_Project_Format"/>
  </PathVariables>
```

8.5. ActiveConfiguration

The ActiveConfiguration element stores the name of the active build configuration whose parameters will be used by TITAN for building the project.

NOTE	This can be overwritten from TITAN Designer (from Eclipse) or from command line generating Makefile(s) by ttcn3_makefilegen using the -b flag, see here.

See also chapter Configurations.

Example:

```
<ActiveConfiguration>Default</ActiveConfiguration>
```

8.6. Configurations

The `Configurations` element stores a list of `Configuration` elements, each of which describes a single build configuration. Different build configurations can use a different file set, different makefile settings (e.g single or parallel mode, operational system specific settings etc).

The `Configuration` element has a single tag called `name`, and stores the name of the configuration.

Table 33. Elements of Configuration

Name of the element	Meaning	O/M
`ProjectProperties`	Stores the settings of the project required to create a Makefile, to build the project, and the project level naming conventions. See here.	O
`FolderProperties`	Stores the properties of each folder contained in the project. Not present if there are no folders in the project, or all folders have all attributes on their default values. See chapter here.	O
`FileProperties`	Stores the properties of each file contained in the project. Not present if there are no files in the project, or all files have all attributes on their default values. See chapter here.	O

8.6.1. Project Properties

This element contains all information needed to create a proper Makefile for the project to drive the build process (all information other than the list of files) and for naming convention checking. Compare this chapter with chapter "Setting Project Properties" in TITAN Designer Documentation [17].

It can contain 5 elements: `MakefileSettings`, `LocalBuildSettings`,

`RemoteBuildProperties`, `NamingConventions`, `ConfigurationRequirements`.

Name of the element	Meaning	O/M
`MakefileSettings`	Stores the settings of the project required to create a Makefile.For more information see here	O
`LocalBuildSettings`	Stores the settings of the project required to perform the build. See here	O
`RemoteBuildProperties`	Stores information necessary for remote build. See here	
`NamingConventions`	Stores the project specific naming conventions. See here	O
`ConfigurationRequirements`	Stores the required build configurations of the referenced projects. See here	O

Makefile Settings

The flags for the TTCN3 compiler can be specified in this section. For more information please refer to section Compiler. A supplementary information is placed in brackets.

Useful information can be found in TITAN Designer documentation [17] as well.

Table 34. The elements of MakefileSettings

Name	Makefilegen option	Compiler option	Default value (used when not being present)	O/M
`generateMakefile (meaningful only in Eclipse)`	-	-	true	O
`generateInternalMakefile (meaningful only in Eclipse)`	-	-	false	O
`symboliclinklessBuild`	-	-	false	O
`useAbsolutePath`	-a	-	false	O
`GNUMake`	-g	-	false	O

Name	Makefilegen option	Compiler option	Default value (used when not being present)	O/M
incrementalDependencyRefresh (meaningful not only in Eclipse, necessary to apply incremental dependency via gnu Make and .d files)	-	-	false	O
dynamicLinking	-l	-	false	O
functiontestRuntime (use function test runtime (TITAN_RUNTIME_2)	-R	-R	false	O
singleMode	-s	-	false	O
codeSplitting (select code splitting mode for the generated C++ code)	-U	-U	none	O
defaultTarget ("executable" or "library", if -L applied, see 6.1.2)	-L	-	executable	O
targetExecutable	-e	-	N/A	O
TTCN3preprocessor (the name of the preprocessor meaningful only in Eclipse)	-	-	cpp	O
TTCN3preprocessorIncludes	-	-	empty	O
preprocessorIncludes	-p	-	empty	O
semanticCheckOnly	-	-s	false	O
disableAttributeValidation	-	-0	false	O
disableBER (disable BER encoder/decoder functions)	-	-b	false	O
disableRAW (disable RAW encoder/decoder functions)	-	-r	false	O

Name	Makefilegen option	Compiler option	Default value (used when not being present)	O/M
disableTEXT (disable TEXT encoder/decoder functions)	-	-x	false	O
disableXER	-	-X	false	O
disableOER	-	-O	false	O
forceXERinASN.1 (force XER in ASN.1 files)	-	-a	false	O
defaultasOmit (-d compiler option)	-	-d	false	O
gccMessageFormat (emulate GCC error/warning message format)	-	-g	false	O
lineNumbersOnlyInMessages (use only line numbers in error/warning messages)	-	-i	false	O
includeSourceInfo (include source line info in C++ code)	-	-l	false	O
addSourceLineInfo (add source line info for logging)	-	-L	false	O
suppressWarnings (suppress warnings)	-w	-w	false	O
Quietly (suppress all messages, quiet mode)	-	-q	false	O
namingRules (only in Eclipse)	-	-	unspecified	O
disableSubtypeChecking (disable subtype checking)	-	-y	false	O
forceOldFuncOutParHandling (force old function out parameter handling) Note: overwrites obsolete tag outParamBoundness	-Y	-Y	false	O

Name	Makefilegen option	Compiler option	Default value (used when not being present)	O/M
CxxCompiler (The name of the compiler, only in Eclipse)	-	-	g++	O
optimizationLevel (only in Eclipse)	-	-	"Common optimizations"	O
otherOptimizationFlags (only in Eclipse)	-	-	empty	O
disablePredefinedExternalFolder (OPENSSL_DIR and XMLDIR)	-	-	false	O
enableLegacyEncoding	-G	-e	false	O
disableUserInformation	-	-D	false	O
enableRealtimeTesting	-i	-I	false	O
buildLevel (only in Eclipse, see below and in 6.1.6 The actual building in [17])	-	-	"Level 5 - Creating Executable Test Suite with dependency update"	O
ProjectSpecificRulesGenerator	-	-	Used to place custom rules and new targets into the generated Makefile	O
profiledFileList (enables profiling and code coverage in the specified modules)	-z	-z	empty	O
omitInValueList	-M	-M	false	O
warningsForBadVariants	-E	-E	false	O
activateDebugger	-n	-n	false	O
ignoreUntaggedOnTopLevelUnion	-N	-N	false	O
xsd2ttcnFlags	-x	-	false	O

The supported values of optimizationLevel are:

- "None"
- "Minor optimizations"
- "Common optimizations"
- "Optimize for speed"
- "Optimize for size"
- The optimization flags given as the value of otherOptimizationFlags are passed to the Cxx compiler.

The support values for buildLevel are:

- "Level 0 - Semantic Check"
- "Level 1 - TTCN3 → C++ compilation"
- "Level 2 - Creating object files"
- "Level 2.5 - Creating object files with heuristical dependency update"
- "Level 3 - Creating object files with dependency update"
- "Level 4 - Creating Executable Test Suite"
- "Level 4.5 - Creating Executable Test Suite with heuristical dependency update"
- "Level 5 - Creating Executable Test Suite with dependency update"

NOTE	The targetExecutable path is stored either relative to the root of the project, or with full path.

It is possible to reference environment variables in the following fields with the syntax `"[" VariableName "]"`:

- `TTCN3preprocessorIncludes`
- `preprocessorIncludes`
- `SolarisSpecificLibraries`
- `Solaris8SpecificLibraries`
- `LinuxSpecificLibraries`
- `FreeBSDSpecificLibraries`
- `Win32SpecificLibraries`
- `linkerLibraries`
- `linkerLibrarySearchPath`

The variables referenced with this syntax will be recognized by the Eclipse Designer plugin. If the tpd is used for makefile generation, the `ttcn3_makefilegen` will replace the reference with its command line equivalent in the generated makefile. (e.g. `[VariableName] ⇒ $(VariableName)`).

Contents of the `ProjectSpecificRulesGenerator` element:

Exactly one GeneratorCommand element that specifies the external command to be run

An optional Targets element that contains any number of Target elements, each element having two attributes: name - name of the target, placement - the place of where the target shall be inserted. Possible places are defined in the TPD.xsd file.

The content of the profiledFileList element is the path to a text file, using the same path attributes as FileResource elements. The text file contains the list of files (TTCN-3 modules), that will be profiled, separated by new lines. This file is stored in the variable PROFILED_FILE_LIST in the generated makefile.

TPDs of referenced projects may also contain profiled file lists, these are merged with each other and with the top-level project's file list (a new rule is created that merges the lists). In this case the variable PROFILED_FILE_LIST contains the merged file list, and PROFILED_FILE_LIST_SEGMENTS contains the individual file lists.

NOTE	A new rule is added to make target compile (both rules are switched to double colon rules), if the profiled file list exists, since changing the list of profiled files requires all modules to be recompiled. If the profiled file list is also a make target (in case it is merged from other lists), then a new dependency is added to make targets check and compile-all (also with the use of double colon rules).

Local Build Settings

Table 35. Elements of LocalBuildSettings

Name of the element	Default value	O/M
MakefileFlags	empty	O
MakefileScript	empty	O
workingDirectory	N/A	O

MakefileScript is a script which modifies the Makefile generated by the makefilegen program or by the TITAN Designer internal makefile generator. This kind of script is widely used to automatically insert or remove flags which are not handled by the ttcn3_makefilegen. If the Makefile is generated by the TITAN Designer, this script is generally not necessary because the TITAN Designer can handle (insert or remove) them but even in this case there can be necessary modifications. Duplicated insertion of flags can cause errors.

The MakefileScript shall be a shell script and it must have two parameters. The first parameter is the name of the generated Makefile and the second parameter is the name of the generated Makefile with the .tmp suffix. The MakefileScript should write the contents of the modified Makefile into the .tmp file from the second argument. The TITAN Designer and the makefilegen tool will automatically move the contents of the .tmp file into the Makefile and then remove the .tmp file.

NOTE	The MakefileScript and workingDirectory paths are stored either relative to the root of the project, or with full path.

Remote Build Properties

RemoteBuildProperties contains a sequence of elements type of "RemoteHost" and one optional ParallelExecution element which is a boolean. A RemoteHost contains 3 elements according to Table 36 Elements of `RemoteHost`.

Table 36. Elements of `RemoteHost`

Name of the element	Type	O/M
`Active`	boolean	M
`Name`	string	M
`Command`	string	M

Naming Conventions

The naming conventions are given using Java regular expressions. All of the elements below are optional.

Table 37. Elements of NamingConventions

Name of the element	Default value	O/M
`TTCN3ModuleName`	.*	O
`ASN1ModuleName`	.*	O
`altstep`	as_.*	O
`globalConstant`	cg_.*	O
`externalConstant`	ec_.*	O
`function`	f_.*	O
`externalFunction`	ef_.*	O
`moduleParameter`	m.*	O
`globalPort`	.*_PT	O
`globalTemplate`	t.*	O
`testcase`	tc_.*	O
`globalTimer`	T.*	O
`type`	.*	O
`group`	[A-Z].*	O
`localConstant`	cl.*	O
`localVariable`	vl.*	O
`localTemplate`	t.*	O
`localVariableTemplate`	vt.*	O
`localTimer`	TL_.*	O
`formalParameter`	pl_.*	O

Name of the element	Default value	O/M
componentConstant	c_.*	O
componentVariable	v_.*	O
componentTimer	T_.*	O

Other than the above mentioned on the project level there is one more called: enableProjectSpecificSettings with being empty as the default value. This element makes it possible to override the global settings.

On folder level the extra node is called enableFolderSpecificSettings with being empty as the default value. This element makes it possible to override the global settings.

Configuration Requirements

The ConfigurationRequirements element stores a list of ConfigurationRequirement elements, each of which describes a single configuration requirement for a referenced project. For each referenced project there can be maximally one ConfigurationRequirement element. If there is no requirement against the actual configuration of a referenced project this element is missing.

Table 38. Elements of ConfigurationRequirement

Name of the element	Meaning	O/M
projectName	Stores the name of the project for which the requirement applies.	M
rerquiredConfiguration	Stores the name of the required project configuration as known by the referenced project.	M

8.6.2. Folder Properties

The FolderProperties element contains a list of FolderResource elements each of which contains all information related to the actual settings of the folders in a given build configuration. The FolderResource element contains a FolderPath and a FolderProperties subelement.

Table 39. Elements of FolderResource

Name of the element	Meaning	O/M
FolderPath	The path of the folder in the eclipse resource system.	O
FolderProperties	The actual properties of the folder.	O

Table 40. The optional elements of the FolderProperties sub element

Name of the element	Default value	O/M
ExcludeFromBuild	false	O

Name of the element	Default value	O/M
centralStorage	false	O
NamingCoventions	Missing if all elements are on default value	O

For more information about the NamingConventions element please refer here.

8.6.3. File Properties

The FileProperties element contains a list of FileResource elements each of which contains all information related to the actual settings of the files in a given build configuration. The FilePath and the FileProperties subelements are mandatory.

Table 41. Elements of FileResource

Name of the element	Meaning	O/M
FilePath	The path of the file in the eclipse resource system.	M
FileProperties	The actual properties of the file. see Table 42	M

Table 42. Elements of the FileProperties sub element

Name of the element	Default value	O/M
ExcludeFromBuild	false	O
EnableCodeSplitting	false	O

8.7. Packed Referenced Projects

The PackedReferencedProjects element stores a list of PackedReferencedProject elements, each of them describes a single project reachable from the actual one or from referenced projects via project referencing.

The elements of this list are the same as TITAN_Project_File_Information but they cannot contain PackedReferencesProjects. All referred projects in the project chain are listed in this list.

A single PackedProjectReference element stores the same data in the same manner as it is stored in the referenced project but it cannot store the element PackedReferencedProjects.

Example:

```
<PackedReferencedProjects>
    <PackedReferencedProject>
      <ProjectName>HelloTitan</ProjectName>
      <Folders>
        <FolderResource projectRelativePath="bin" relativeURI="../HelloTitan/bin"/>
        <FolderResource projectRelativePath="src" relativeURI="../HelloTitan/src"/>
```

```
      </Folders>
      <Files>
        <FileResource projectRelativePath=".TITAN_properties"
relativeURI="../HelloTitan/.TITAN_properties"/>
        <FileResource projectRelativePath=".project"
relativeURI="../HelloTitan/.project"/>
        <FileResource projectRelativePath="HelloTitan.tpd"
relativeURI="../HelloTitan/HelloTitan.tpd"/>
      </Files>
      <ActiveConfiguration>Default</ActiveConfiguration>
      <Configurations>
        <Configuration name="Default">
          <ProjectProperties>
            <MakefileSettings>
              <generateMakefile>true</generateMakefile>
              <generateInternalMakefile>true</generateInternalMakefile>
              <symboliclinklessBuild>false</symboliclinklessBuild>
              <useAbsolutePath>false</useAbsolutePath>
              <GNUMake>false</GNUMake>
              <incrementalDependencyRefresh>false</incrementalDependencyRefresh>
              <dynamicLinking>false</dynamicLinking>
              <functiontestRuntime>false</functiontestRuntime>
              <singleMode>false</singleMode>
              <codeSplitting>none</codeSplitting>
              <defaultTarget>executable</defaultTarget>
              <targetExecutable>bin\HelloTitan.exe</targetExecutable>
              <TTCN3preprocessor>cpp</TTCN3preprocessor>
              <TTCN3preprocessorIncludes/>
              <preprocessorIncludes/>
              <disableBER>false</disableBER>
              <disableRAW>false</disableRAW>
              <disableTEXT>false</disableTEXT>
              <disableXER>false</disableXER>
              <disableOER>false</disableOER>
              <forceXERinASN.1>false</forceXERinASN.1>
              <defaultasOmit>false</defaultasOmit>
              <enumHackProperty>false</enumHackProperty>
              <forceOldFuncOutParHandling>false<forceOldFuncOutParHandling>
              <gccMessageFormat>false</gccMessageFormat>
              <lineNumbersOnlyInMessages>false</lineNumbersOnlyInMessages>
              <includeSourceInfo>false</includeSourceInfo>
              <addSourceLineInfo>false</addSourceLineInfo>
              <suppressWarnings>false</suppressWarnings>
              <quietly>false</quietly>
              <namingRules>unspecified</namingRules>
              <disableSubtypeChecking>false</disableSubtypeChecking>
              <CxxCompiler>g++</CxxCompiler>
              <optimizationLevel>Commonoptimizations</optimizationLevel>
              <otherOptimizationFlags></otherOptimizationFlags>
<ignoreUntaggedOnTopLevelUnion>false</ignoreUntaggedOnTopLevelUnion>
              <enableLegacyEncoding>false</enableLegacyEncoding>
```

```
              <disableUserInformation>false</disableUserInformation>
<disablePredefinedExternalFolder>false</disablePredefinedExternalFolder>
              <buildLevel>Level5-
CreatingExecutableTestSuitewithdependencyupdate</buildLevel>
            </MakefileSettings>
            <LocalBuildSettings>
              <MakefileFlags></MakefileFlags>
              <MakefileScript></MakefileScript>
              <workingDirectory>bin</workingDirectory>
            </LocalBuildSettings>
            <NamingCoventions>
              <enableProjectSpecificSettings></enableProjectSpecificSettings>
              <TTCN3ModuleName>.*</TTCN3ModuleName>
              <ASN1ModuleName>.*</ASN1ModuleName>
              <altstep>as_.*</altstep>
              <globalConstant>cg_.*</globalConstant>
              <externalConstant>ec_.*</externalConstant>
              <function>f_.*</function>
              <externalFunction>ef_.*</externalFunction>
              <moduleParameter>m.*</moduleParameter>
              <globalPort>.*_PT</globalPort>
              <globalTemplate>t.*</globalTemplate>
              <testcase>tc_.*</testcase>
              <globalTimer>T.*</globalTimer>
              <type>.*</type>
              <group>[A-Z].*</group>
              <localConstant>cl.*</localConstant>
              <localVariable>vl.*</localVariable>
              <localTemplate>t.*</localTemplate>
              <localVariableTemplate>vt.*</localVariableTemplate>
              <localTimer>TL_.*</localTimer>
              <formalParameter>pl_.*</formalParameter>
              <componentConstant>c_.*</componentConstant>
              <componentVariable>v_.*</componentVariable>
              <componentTimer>T_.*</componentTimer>
            </NamingCoventions>
          </ProjectProperties>
          <FolderProperties>
            <FolderResource>
              <FolderPath>src</FolderPath>
              <FolderProperties>
                <ExcludeFromBuild>false</ExcludeFromBuild>
                <centralStorage>false</centralStorage>
                <NamingCoventions>
                  <enableFolderSpecificSettings></enableFolderSpecificSettings>
                  <TTCN3ModuleName>.*</TTCN3ModuleName>
                  <ASN1ModuleName>.*</ASN1ModuleName>
                  <altstep>as_.*</altstep>
                  <globalConstant>cg_.*</globalConstant>
                  <externalConstant>ec_.*</externalConstant>
                  <function>f_.*</function>
```

```
        <externalFunction>ef_.*</externalFunction>
        <moduleParameter>m.*</moduleParameter>
        <globalPort>.*_PT</globalPort>
        <globalTemplate>t.*</globalTemplate>
        <testcase>tc_.*</testcase>
        <globalTimer>T.*</globalTimer>
        <type>.*</type>
        <group>[A-Z].*</group>
        <localConstant>cl.*</localConstant>
        <localVariable>vl.*</localVariable>
        <localTemplate>t.*</localTemplate>
        <localVariableTemplate>vt.*</localVariableTemplate>
        <localTimer>TL_.*</localTimer>
        <formalParameter>pl_.*</formalParameter>
        <componentConstant>c_.*</componentConstant>
        <componentVariable>v_.*</componentVariable>
        <componentTimer>T_.*</componentTimer>
      </NamingCoventions>
    </FolderProperties>
  </FolderResource>
  <FolderResource>
    <FolderPath>bin</FolderPath>
    <FolderProperties>
      <ExcludeFromBuild>false</ExcludeFromBuild>
      <centralStorage>false</centralStorage>
      <NamingCoventions>
        <enableFolderSpecificSettings></enableFolderSpecificSettings>
        <TTCN3ModuleName>.*</TTCN3ModuleName>
        <ASN1ModuleName>.*</ASN1ModuleName>
        <altstep>as_.*</altstep>
        <globalConstant>cg_.*</globalConstant>
        <externalConstant>ec_.*</externalConstant>
        <function>f_.*</function>
        <externalFunction>ef_.*</externalFunction>
        <moduleParameter>m.*</moduleParameter>
        <globalPort>.*_PT</globalPort>
        <globalTemplate>t.*</globalTemplate>
        <testcase>tc_.*</testcase>
        <globalTimer>T.*</globalTimer>
        <type>.*</type>
        <group>[A-Z].*</group>
        <localConstant>cl.*</localConstant>
        <localVariable>vl.*</localVariable>
        <localTemplate>t.*</localTemplate>
        <localVariableTemplate>vt.*</localVariableTemplate>
        <localTimer>TL_.*</localTimer>
        <formalParameter>pl_.*</formalParameter>
        <componentConstant>c_.*</componentConstant>
        <componentVariable>v_.*</componentVariable>
        <componentTimer>T_.*</componentTimer>
      </NamingCoventions>
```

```
            </FolderProperties>
          </FolderResource>
        </FolderProperties>
        <FileProperties>
          <FileResource>
            <FilePath>HelloTitan.tpd</FilePath>
            <FileProperties>
              <ExcludeFromBuild>false</ExcludeFromBuild>
            </FileProperties>
          </FileResource>
          <FileResource>
            <FilePath>.project</FilePath>
            <FileProperties>
              <ExcludeFromBuild>false</ExcludeFromBuild>
            </FileProperties>
          </FileResource>
          <FileResource>
            <FilePath>.TITAN_properties</FilePath>
            <FileProperties>
              <ExcludeFromBuild>false</ExcludeFromBuild>
            </FileProperties>
          </FileResource>
        </FileProperties>
      </Configuration>
    </Configurations>
  </PackedReferencedProject>
```

8.8. Important Information, Limitations

We can only save settings related to the TITAN Designer plug-in. The settings of other plug-ins, related to the actual project, needs to be handled by the user separately.**From our point of view data loss and corruption of any kind and magnitude is acceptable, as long as it does not involve TITAN Designer specific settings directly.**

The import is recreating the structure and the content of the project, not the project as a whole. For example: if on the source side there is a resource called X.ttcn located in the file system as Y.ttcn, after saving and loading this information at another location the created project will also have a resource called X.ttcn located in the file system as Y.ttcn. However as the 2 projects are located at different locations in the file system, the reference used to point to this file will be different. In case the file was originally located in the source project, and is now linked in the loaded project, the resource will be decorated accordingly by the platform.

We support eclipse path variables, by saving the location info of the resource using them in a form where this information is available. However as this is an eclipse internal feature outside of our control, command line processing might be limited, or limiting the user. If a required path variable is not defined at the loading side, the resource will be created, but later on, when the platform tries to resolve it, the user might get some kind of error message.

It is also important to know, that we save the location information of all files and folders as Eclipse

knows it. By default this means only local file system locations, however Eclipse can be extended with additional plug-ins to support remote operations (like FTP, HTTP, SSH connections) or virtual file systems. If such features are used it is the user's responsibility to make sure, that the needed plug-ins are available on every machine where the project is to be used. Also when doing command line builds, their build scripts must be prepared to support such features.

In order to be able to recreate the whole project structure all referenced projects have to be saved as well. Otherwise we would not be able to load a missing referenced project.

However once loaded these projects can be used intermixed with other kinds of project inside Eclipse. For example it is possible to refer to them from not yet saved projects.

In case of a referenced project we only store the location where it was loaded from, or where it was saved to first. If it is saved to a new location it must be loaded first, otherwise the changes will be lost. However inside eclipse the project will still contain all the changes.

In the command line referenced projects are identified by the location of their descriptor, in eclipse they are identified by the name of the project. This means, that if a project to be referenced is loaded on a different name, the referencing project will not see it, and at load time will load it again. Also if a different project is loaded at an expected name, it will be used instead of the one being referenced.

In eclipse all information is persisted by the platform, as such the closing and re-opening of the workspace will not change any project information, or come with data loss.

This is our own project information; it can not be expected from external tool vendors to support it. As such exporting/importing a project via the ClearCase Remote Client will not be possible using this format, or might not produce the expected results.

Chapter 9. XSD to TTCN-3 Converter

The XSD to TTCN-3 converter is converting XSD components to TTCN-3 modules according to the ETSI standard ES 201 873-9 (part 9 of the TTCN-3 standard).

The **XSD to TTCN-3 converter** takes as input one or more schemas written in XML Schema and produces one or more TTCN-3 modules containing a set of type definitions and encoding instructions to keep the same XML encoding, in such a way that there is one-to-one correspondence between TTCN-3 values and valid XML instances.

9.1. Terminology

For the purposes of the present section the following definitions apply:

9.2. Schema Component

Schema component is the generic XSD term for the building blocks that comprise the abstract data model of the schema.

9.2.1. Schema Document

A schema document contains a collection of schema components, assembled in a *schema* element information item. The target namespace of the schema document may be defined (specified by the *targetNamespace* attribute of the *schema* element) or may be absent (identified by a missing *targetNamespace* attribute of the *schema* element). The latter case is handled in this document as a particular case of the target namespace being defined.

9.2.2. Target TTCN-3 Module

This is the TTCN-3 module, generated during the conversion, to which the TTCN-3 definition produced by the translation of a given XSD declaration or definition is added.

9.2.3. XML Schema

An XML Schema is represented by a set of schema documents forming a complete specification (i.e. all definitions and references are completely defined). The set may be composed of one or more schema documents, and in the latter case identifying one or more target namespaces. More than one schema documents of the set may have the same target namespace.

9.3. Command-line Syntax

The command line syntax of the converter is the following:

```
xsd2ttcn [-ceghmNopstVwx] [-f file] [-J file] schema.xsd [-schema.xsd…]
```

or

```
xsd2ttcn -v
```

The XSD to TTCN-3 converter takes the name of files containing XML schemas as arguments (usually with the extension .xsd). The converter expects input files to be plain text files.

The following command line options are available (listed in alphabetical order):

- `-c`

  Disables the generation of comments derived from top level XML comments and also from XSD annotations and notations. TTCN-3 comments derived from XSD annotations and notations contain the actual character data of the nested XSD 'documentation' or 'appinfo' elements.

- `-e`

  Disables the generation of encoding instructions for TTCN-3 types and modules.

- `-f file`

  The list of the XSD files will be taken from the given file instead of the command line. The file format is a white-space separated list of XSD files. (Other XSD files in the command line will not be taken into account if this option is used.)

- `-J file`

  It has the same effect as the `-f file` flag.

- `-g`

  Instructs the converter to generate TTCN-3 code disallowing element substitution.

- `-h`

  Instructs the converter to generate TTCN-3 code allowing type substitution.

- `-m`

  Instructs the converter to only generate the `UsefulTtcn3Types` and `XSD` predefined modules.

- `-N`

  Instructs the converter to generate the definitions of XML schemas with no target namespaces into separate TTCN-3 modules (one module per XML schema). By default all definitions with no target namespace are generated into a single TTCN-3 module.

- `-o`

  Generates all definitions into one module, called `XSD_Definitions`, instead of generating separate modules for each target namespace. The module contains one group for each target namespace.

- `-p`

Disables the generation of `UsefulTtcn3Types` and `XSD` predefined modules.

- `-s`

  Instructs the converter to parse the given XML schemas and perform only syntactic analysis on them (well-formedness check), but not to generate TTCN-3 output. This option is useful if you do not wish to generate TTCN-3 files when debugging an XML Schema.

- `-t`

  Disables the generation of timing information in TTCN-3 modules. These parts are the changing parts of the generated modules. By using this switch it is possible to generate TTCN-3 modules that will be the same even if time and version change.

- `-v`

  Prints version and license key information and exits.

- `-V`

  Disables status messages, for example, indicates which input file is currently being read, while converting.

- `-w`

  Suppresses warnings.

- `-x`

  Disables schema validation but generates TTCN-3 modules.

9.4. The Compilation Process for XML Schema

The XSD to TTCN-3 converter requires that each schema file used by the specification must be present in the input. From the input schema files, the converter will build one or possibly more independent TTCN-3 modules. The names of the output files (and the names of the TTCN-3 modules within) are set according to the value of the `targetNamespace` attribute defined in the schema element. Suffixes of TTCN-3 modules are .ttcn.

Whenever a schema file contains an `import` element with the `namespace` attribute, all components (elements, types, groups, etc.) from that namespace are imported into the final XML schema.

NOTE	There can be several schema files having one namespace. All components from that namespace are imported.

The following examples demonstrate how the XSD to TTCN-3 converter assembles input schema files to create the XML Schema.

9.4.1. Include

Example 1-1. 'include' with resolvable schemaLocation attribute

A.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd"
    targetNamespace="http://www.example.org/xsd">
<xsd:include schemaLocation="B.xsd"/>
</xsd:schema>
```

B.xsd:

```
<xsd:schema  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd"
    targetNamespace="http://www.example.org/xsd">
     ...
</xsd:schema>
```

Converter command:

```
xsd2ttcn A.xsd B.xsd
```

In Example 1-1, the `schemaLocation` attribute indicates a schema file name that is present in the command line. The referenced schema file must be provided and listed in the command line.

9.4.2. Import

Example 1-2. 'import' with resolvable schemaLocation attribute

A.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd"
    targetNamespace="http://www.example.org/xsd">
        <xsd:import namespace=http://www.example.org/xsd/B
    schemaLocation="B.xsd"/>
     ...
</xsd:schema>
```

B.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd/B"
    targetNamespace="http://www.example.org/xsd/B">
     ...
</xsd:schema>
```

Converter command:

```
xsd2ttcn A.xsd B.xsd
```

Example 1-3 shows the use of `import`. Schema `A.xsd` is importing schema `B.xsd`. The `schemaLocation` attribute in schema `A.xsd` is pointing to a schema file which is present on the command line input.

Example 1-3. 'import' without schemaLocation attribute

A.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd"
    targetNamespace="http://www.example.org/xsd">
<xsd:import namespace="http://www.example.org/xsd/B"/>
     ...
</xsd:schema>
```

B.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd/B"
    targetNamespace="http://www.example.org/xsd/B">
     ...
</xsd:schema>
```

B2.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd/B"
    targetNamespace="http://www.example.org/xsd/B">
        <xsd:include schemaLocation="http://www.example.org/xsd/B.xsd"/>
     ...
</xsd:schema>
```

Converter command:

```
xsd2ttcn A.xsd B.xsd B2.xsd
```

An `import` with only `namespace` attribute, imports all the schemas present on the command line having the same targetNamespace as the value specified by the namespace attribute in the import element. In Example 1-3, `A.xsd` contains an `import` element having specified the `namespace` attribute only; the XSD to TTCN-3 converter will import both `B.xsd` and `B2.xsd`, as they have the same targetNamespace as the one defined in the `namespace` attribute of the import element from the schema `A.xsd`.

Example 1-4. 'import' without namespace attribute

A.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd"
    targetNamespace="http://www.example.org/xsd">
     <xsd:import schemaLocation="B.xsd"/>
     ...
</xsd:schema>
```

B.xsd:

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema>
     ...
</xsd:schema>
```

Converter command:

```
xsd2ttcn A.xsd B.xsd
```

If the import element specifies the `schemaLocation` attribute only, the imported schema (`B.xsd`) should not be associated with any namespace; otherwise the converter reports an error message.

Example 1-5. 'import' with no attributes

A.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd"
    targetNamespace="http://www.example.org/xsd">
     <xsd:import/>
     ...
</xsd:schema>
```

Converter command:

```
xsd2ttcn A.xsd B.xsd C.xsd D.xsd E.xsd F.xsd G.xsd H.xsd
```

The `import` statement with no attributes specified imports all the schema files in the command line input that have no `targetNamespace` specified. In Example 1-5, if `B.``xsd`, `C.xsd`, and `H.xsd` are not associated with any namespace they are imported in the `A.xsd`.

9.5. Restrictions

Some features of XSD have no equivalent in TTCN-3 or make no sense when translated to the TTCN-3 language. Whenever possible, these features are translated into encoding instructions completing the TTCN-3 code. For any further information about unsupported features see [4].

Translation of the following XML schema elements is not supported:

`field`, `key`, `keyref`, `selector`, `unique` (identity-constraint definition schema components)

Translation of the following XML schema attributes is not supported:

`final`, `processContents`

The following XML schema attributes are ignored, when they are used as attributes of schema element:

`finalDefault`, `id`, `version`, `xml:lang`

Numeric types are not allowed to be restricted by patterns.

List types are not allowed to be restricted by enumerations or patterns.

All time types restrict year to 4 digits.

Information in the `appinfo` tags are not translated.

9.6. Extensions

The XSD to TTCN-3 Converted has the following non-standard additions to the Using XML Schema with TTCN-3 [4] .

TITAN allows the usage of constants and module parameters in the value of a `defaultForEmpty` encoding instruction. The `xsd2ttcn` tool generates the `defaultForEmpty` encoding instructions with a constant definition as a value to provide reusability of the `defaultForEmpty` values. Only the conversion of `default` and `fixed` attributes of elements is changed.

For example:

A.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.example.org/xsd"
    targetNamespace="http://www.example.org/xsd">
     <xsd:element name="DefStr" type="xsd:string" default="abc"/>

     <xsd:element name="FixStr" type="xsd:string" fixed="def"/>
</xsd:schema>
```

The `DefStr` and `FixStr` elements are generated into the following type definitions:

```
const XSD.String c_defaultForEmpty_1 := "abc";

const XSD.String c_defaultForEmpty_2 := "def";

type XSD.String DefStr
with {
  variant "defaultForEmpty as c_defaultForEmpty_1";
  variant "element";
};

type XSD.String FixStr (c_defaultForEmpty_2)
with {
  variant "defaultForEmpty as c_defaultForEmpty_2";
  variant "element";
};
```

Chapter 10. Code Coverage of TTCN-3 Modules

NOTE	the feature described here is deprecated; please use instead the coverage tool described in Profiling and code coverage.

The TTCN-3 compiler is able to instrument the generated C/C++ code for a set of TTCN-3 modules (= or files) to generate code coverage information during runtime. To enable this feature the `-K file` option needs to be used. For convenience this option is available for `ttcn3_makefilegen` as well. It's possible to generate code coverage information only for a given set of TTCN-3 modules listed on the command line. In that case the set of files in `file` needs to be a subset of the files listed on the command line. If `file` contains a file which is not listed on the command line an error will be issued.

10.1. Generating Code Coverage

Assuming a project with the following files: `a.ttcn`, `b.ttcn`, `c.ttcn` in parallel mode and some PTCs during runtime were created. The scenario is the following:

1. Install LCOV. It's an external tool necessary to generate HTML output. Available here: http://ltp.sourceforge.net/coverage/lcov.php or can be installed using the package manager of your OS. This step needs to be performed only once.
2. Create an ASCII text file listing all the TTCN-3 modules to generate code coverage data for, one file name in a line. In this case we want to generate code coverage data for `a.ttcn` and `b.ttcn` and we're not interested in `c.ttcn`. Our `tcov_file_list.txt` will look like: `a.ttcn, b.ttcn`
3. Generate a `Makefile` using `tcov_file_list.txt`:

   ```
   ttcn3_makefilegen -K tcov_file_list.txt -g -e mytest *.ttcn
   ```

4. Run: make
5. Then: `ttcn3_start ./mytest`

   (During runtime some .tcd XML files will be generated. Namely a `tcov-<component_id>.tcd` file per PTC and one for MTC.)

6. Collect and merge all .tcd files using `tcov2lcov` with default parameters:

   ```
   tcov2lcov
   ```

   (An `output.info` file will be generated in the current working directory. For more detailed information about `tcov2lcov` see here.)

7. Generate HTML using LCOV's `genhtml` and the generated `output.info` (see here for more information):

```
genhtml -no-branch-coverage -t "Titan Coverage" -legend output.info -o
titan_coverage
```

8. Open `titan_coverage/index.html` in a browser.

10.2. Converting Code Coverage Data from XML to HTML

The `tcov2lcov` tool (binary tool) shipped with Titan and LCOV's `genhtml` tool (Perl script) are provided to generate human readable HTML from the Titan generated .tcd XML code coverage data files. LCOV's `genhtml` is not part of the Titan distribution and needs to be installed separately. The basic process is the following:

1. Execute `tcov2lcov` to collect and merge all the .tcd files generated during test execution. One .tcd file per component. By default an `output.info` file will be generated in the current working directory, which can be processed directly by LCOV's `genhtml`. More detailed information about `tcov2lcov` can be found in 10.3.
2. Execute `genhtml` with `output.info` as its input parameter. The recommended parameters are the following:` genhtml -no-branch-coverage -t "Titan Coverage" -legend output.info -o titan_coverage`. For more detailed introduction to `genhtml` and LCOV in general please read their user manuals at http://ltp.sourceforge.net/coverage/lcov.php.

10.3. Command Line Syntax of tcov2lcov

```
tcov2lcov [-h][-b dir][-d dir][-o file][-x xsd]
```

or

```
tcov2lcov -v
```

The Titan code coverage data to LCOV coverage data converter collects all valid .tcd XML files from a given directory recursively and generates a single ASCII text file suitable to be further processed by LCOV's `genhtml` tool to produce human readable HTML output. Please note that the output format generated by `tcov2lcov` is also human readable ASCII, but its only purpose is to provide an input for LCOV's `genhtml` tool. This format is LCOV specific and not documented here.

The following command line options are available (listed in alphabetical order):

- `-b dir` (optional)

  Set code base directory for source files. `dir` is usually the absolute path to the directory of the source files. This path is used as a common prefix for `genhtml`. The default value is the absolute path of the current working directory.

- `-d dir` (optional)

  `.tcd` files will be collected from `dir` recursively. By default the current working directory will be examined.

- `-h` (optional)

  Display help.

- `-o file` (optional)

  Set the name of the output file. The default file name is `output.info`.

- `-x xsd` (optional)

  Path to an XSD to validate all .tcd XML files found against. By default `$TTCN3_DIR/include/tcov.xsd` is used. This XSD file is part of the Titan distribution. If any of the .tcd files doesn't conform to the XSD an error will be given.

10.4. Limitations

The Titan compiler implements some optimizations which can affect the accuracy of the generated code coverage information. These optimizations cannot be turned off. The compile time evaluation of constant expressions can lead to untracked lines and statements (white in LCOV's HTML output) invisible to the code coverage information generator. These lines and statements are not counted as never-executed lines and statements (orange in LCOV's HTML output), so the statistics are not affected, but the end result can be confusing.

External functions are not yet supported and they're not shown in the statistics.

The Titan code coverage implementation is based on Titan's internal location tracking mechanism enabled by the `-L` compiler flag, which must be used together with `-K`. Sometimes it can lead to a little bit confusing or strange code coverage output. E.g. multiple location objects are generated when multiple variable declarations appear on the same line in the TTCN-3 source code. In this case the code coverage output will show that the given line is executed twice. For more complex statements like `for` three location objects are generated etc.

Chapter 11. The TTCN-3 Debugger

The TTCN-3 debugger is a feature in TITAN, which allows the user to pause (halt) the execution of a TTCN-3 program and print (or in some cases overwrite) information about the current state of the program.

The compiler option `-n` activates the debugger and augments the generated C++ code to store debug information and to allow the addition of breakpoints at runtime. For convenience this option is available for `ttcn3_makefilegen` as well.

11.1. Gathered information

This section details the various types of information gathered by the debugger.

NOTE	The debugger refers to the following TTCN-3 entities as "functions": `functions`, `external functions`, `testcases`, `altsteps`, `control` parts and parameterized templates. The debugger refers to the following TTCN-3 entities as "variables": constants (including those imported from ASN.1 modules), external constants, templates, variables, template variables, module parameters, timers and function parameters.

11.1.1. The call stack

The debugger maintains a stack of the currently called functions. The bottom entry in the stack is the oldest (first) function call, while the top-most entry in the stack is the newest function call.

A separate call stack is maintained for each component.

For each entry in the call stack the debugger stores the function name (or the module name in case of control parts), function type (the TTCN-3 keywords used when defining the function) and the current values of the function's parameters.

11.1.2. Variables

The debugger keeps track of all variables (that are currently alive). Each variable's name, type, current value, module name (where the variable was declared) and scope are stored.

A variable's scope refers to which functions and code blocks the variable is visible from. Variables can be grouped into 3 categories based on their scope:

- **Global variables** are the ones that are declared outside of all functions and component types. These are visible from all functions declared in the same module as the variable and from all functions declared in modules that import the variable's module.
- **Component variables** are the one declared in component types. The variables of a specific component type (including variables of component types that the component in question extends) are visible from all functions that run on that component type.
- **Local variables** are the variables declared within a function, and they are only visible inside that function.

NOTE	The names of variables defined in ASN.1 code are converted to TTCN-3 format before being stored (hyphens are replaced with underscores, and an additional underscore is added to the end of the name if it clashes with a TTCN-3 keyword).

11.1.3. Function call data

The debugger stores data about which functions were called since the beginning of the program's execution. A snapshot is taken of each function call at the beginning and end of its execution. The amount of function calls stored can be limited by the user.

These snapshots contain a time stamp, the function's name (or the module name in case of control parts), its type (the TTCN-3 keywords used when defining the function), the value of its parameters (`in` and `inout` parameters for starting snapshots, `inout` and `out` parameters for ending snapshots) and its return value (only for ending snapshots).

11.2. Breakpoints

The debugger provides several ways to halt a TTCN-3 program:

- user breakpoints - breakpoints can be set by the user to any line or function in the TTCN-3 code;
- automatic breakpoints - certain events can be set to halt the program automatically;
- stepping - the user can step to the next line of code (when the program is already halted), or to a specific line or function;
- manual halt - the user can halt the program manually (regardless of which line is currently being executed).

When the program is halted, no further TTCN-3 code is executed, until the user manually resumes the program's execution.

User breakpoints halt the program before the associated line of code starts executing. If a breakpoint is set at a function, then it would halt the program before the first line in the function (that contains executable code) begins execution.

When the program is halted in parallel mode, the execution of all components is halted, not just the component that triggered the halt. The component that triggers the breakpoint (or reaches the stepping point) is halted immediately. The other components are halted with a slight delay. If the program was halted manually, then all components are halted with a slight delay. Resuming a halted program also has a slight delay on all components.

NOTE	Timers are not paused when the program is halted. They continue ticking, and may time out, if the program is halted long enough.External programs that communicate with the TTCN program (e.g. the SUT) are also not halted, and not receiving any messages from the TTCN program might cause them to behave differently than if the program wasn't halted.

11.3. User interface and list of commands

The user communicates with the debugger through a command line interface. In parallel mode this interface is the Main Controller's user interface (details about the MC can be found in [13]). In single mode a similar interface is displayed whenever the program is halted.

An answer or result is displayed through the user interface for all debugger commands (even erroneous ones). In parallel mode certain debugger commands are sent to multiple components. In this case an answer is displayed from each component the command was sent to.

Both user interfaces support the execution of debugger commands from a text file (batch file). Details about batch files can be found here.

The various commands available through the user interface are grouped into 4 categories, detailed in subsections Settings through Bacth Files.

11.3.1. Debugging with the Main Controller

Debugger commands are given to the MC's interface the same way as regular MC commands (all debugger commands start with 'd').

No special settings are needed for the MC to execute debugger commands (like the `-n` switch for the compiler). The MC always knows all debugger commands, and will redirect the commands to the appropriate HCs, PTCs and/or the MTC.

If debugging was activated on an HC, then it and its test components (PTCs and/or the MTC) will process the received command and return an answer to the MC. The MC displays all answers received from the MTC and PTCs, which means that one debugger command can result in multiple answers (one from each affected test component). The HC is always silent; it never returns textual answers to the MC. The debugger on the HC only stores settings for future PTCs.

If debugging is deactivated on an HC, then it and its test components will ignore all debugger commands from the MC.

Example:

One HC is connected to the MC with debugging activated. The MTC and 2 PTCs are running on the one HC. The MC receives the following command:

```
dsetbp MyModule 123
```

See subsection Settings for details about this command.

Results:

```
MTC@hostname: Breakpoint added in module "MyModule" at line 123 with no batch file.
ptc1(3)@hostname: Breakpoint added in module "MyModule" at line 123 with no batch file.
ptc2(4)@hostname: Breakpoint added in module "MyModule" at line 123 with no batch file.
```

11.3.2. Debugging with the single mode UI

The debugger has its own command line user interface in single mode. This interface only knows debugger commands.

The interface becomes active whenever the debugger halts test execution, displaying the prompt 'DEBUG>'.

NOTE	Two command line options, `-h` and `-b`, are available to initialize the debugger (otherwise it would never cause a halt) when running a TTCN-3 executable in single mode. These are described in the User Guide ([13]).

By default, this user interface is only capable of reading strings one line at a time. This can be upgraded to know command completion and command history, like the Main Controller's user interface, by rebuilding the TITAN runtime libraries with the `ADVANCED_DEBUGGER_UI := yes` setting (in the personal Makefile). This requires the `libcurses` or `libncurses` (depending on platform) library to be added to the TTCN-3 executable's linking command in the Makefile. If the Makefile was generated by the `ttcn3_makefilegen` tool, then regenerating it will update the linking command with this new library.

Another solution to using command completion and command history is to build the TTCN-3 executable in parallel mode and use the Main Controller's UI.

11.3.3. Settings

On/off switch

Command syntax: `debug (on | off)`

Default setting: off

Description: This is a separate on/off switch for the debugger, which can be changed at runtime (and thus does not require the program to be rebuild every time it is changed).

When switched off the debugger does not track local variables, does not store function call data and does not halt the program. It still maintains the current call stack and tracks global and component variables.

Global batch file

Command syntax: `dglobbatch (off | (on <batch file>))`

Default setting: off

Description: Activates or deactivates the global batch file, or changes the file it refers to.

The global batch file is executed automatically whenever the program is halted. If the program was halted by a user or automatic breakpoint that has a batch file associated with it, then the global batch file is not executed, only the breakpoint's own batch file.

Set breakpoint

Command syntax: `dsetbp <module> (<line> | <function>) [<batch file>]`

Default setting: -

Description: Creates a new user breakpoint at the specified location (module name and either line number or function name) with or without a batch file, or changes the batch file setting of an existing breakpoint.

NOTE	The first argument is the name of the module, not the name of the TTCN-3 file.

If the breakpoint has a batch file set, then this batch file is automatically executed when the breakpoint is triggered.

If the breakpoint has no batch file of its own set, and a global batch file is set, then the global batch file is executed when the breakpoint is triggered (i.e. the local batch file overwrites the global batch file).

The validity of the command's parameters is not checked. A breakpoint set at a line, function or module that does not exist or does not contain executable code (e.g. a line containing only '}') will never be triggered.

Remove breakpoint

Command syntax: `drembp (all | (<module> (all | <line> | <function>)))`

Default setting: -

Description: Removes the breakpoint at the specified location (if it exists), or removes all breakpoints in the specified module, or removes all breakpoints in all modules.

Examples:

```
Example 1 - removing one breakpoint, from module MyModule, line 114:
drembp MyModule 114

Example 2 - removing all breakpoints from module MyModule:
drembp MyModule all

Example 3 - removing all breakpoints:
drembp all
```

Automatic breakpoint

Command syntax: `dautobp (error | fail) (off | (on [<batch file>]))`

Default setting: all automatic breakpoints are switched off

Description: Activates or deactivates an automatic breakpoint, or changes the batch file settings of an activated automatic breakpoint.

Automatic breakpoints are breakpoints triggered by specific events. The first argument indicates which automatic breakpoint to change:

- **error** - triggered when the component's verdict is set to `error` by a dynamic test case error (not all dynamic test case errors trigger this breakpoint, only those that actually change the local verdict to `error`);
- **fail** - triggered when the component's verdict is set to `fail` (by a `setverdict` operation.

If the automatic breakpoint has a batch file set, then this batch file is automatically executed when the breakpoint is triggered.

If the breakpoint has no batch file of its own set, and a global batch file is set, then the global batch file is executed when the breakpoint is triggered (i.e. the local batch file overwrites the global batch file).

Set output

Command syntax: `doutput (console | file | both) <file name>`

Default setting: console

Description: Changes the destination of the debugger's output (notifications and results of commands). The debugger's output can be displayed by the user interface (`console'), or it can be written to a text file ('file'). The option `both' writes the debugger's output to both the user interface and the specified file.

The file name may contain special metacharacters, which are substituted dynamically during test execution (these are a subset of the metacharacters usable in log file names).

NOTE	In parallel mode output files are created in the host's working directory (not the MC's).

Table 43. Available metacharacters for setting output file names

Meta-character	Substituted with . . .
`%e`	the name of the TTCN-3 executable. The `.exe` suffix (on Windows platforms) and the directory part of the path name (if present) are truncated.
`%h`	the name of the computer returned by the `gethostname`(2) system call. This usually does not include the domain name.
`%l`	the login name of the current user. If the login name cannot be determined (e.g. the current UNIX user ID has no associated login name) an empty string is returned.

Meta-character	Substituted with . . .
`%n`	• the name of the test component if the PTC has been given a name with the command `create` in the TTCN-3 `create` operation; an empty string otherwise. • the string `MTC` if the component is the Main Test Component (both in parallel and in single mode)
`%p`	the process ID (`pid`) of the UNIX process that implements the current test component. The `pid` is written in decimal notation.
`%r`	the component reference or component identifier. On PTCs it is the component reference (an integer number greater or equal to 3) in decimal notation. On the Main Test Component or in single mode the strings `mtc` or `single` are substituted, respectively.
`%%`	a single % character.

Function call data configuration

Command syntax: `dcallcfg (all | <buffer size> | (file <file name>))`

Default setting: all

Description: Changes the method of storing function call data. The new configuration is specified by the command's first argument:

- 'all' - In this case all function calls are stored by the debugger. This option damages the program's performance the most (specifically its memory consumption), since two long strings are stored every time a function is called and they are not deleted until the program's execution ends.
- *buffer size* - This option sets an upper limit to the amount of function call snapshots stored (equal to `<buffer size>`). The function calls are stored in a ring buffer. If the buffer is full, then storing a new snapshot will cause the oldest stored snapshot to be erased. The buffer size can also be set to zero, in which case no function call data is stored.
- 'file' - In this case the function call data is sent directly to a file and not stored by the debugger. The file is specified by the second argument.

  The file name may contain special metacharacters, which are substituted dynamically during test execution.

NOTE	In parallel mode output files are created in the host's working directory (not the MC's). This commands also erases all previously stored function call data (unless the new setting is exactly the same as the old one).

11.3.4. Commands related to printing and overwriting data

Print settings

Command syntax: `dsettings`

Prerequisites: none

Description: Prints the current states of all settings listed in the previous section.

Set component

Command syntax: `dsetcomp (<component name> | <component reference>)`

Prerequisites: parallel mode

Description: Changes the currently active test component to the one specified by the component name or reference.

The active component is the component that receives all of the debugger commands related to printing and overwriting data. Only components that are currently running code are valid candidates.

The active component is automatically set to the MTC when the MC is started. If the active component was set to a PTC that is no longer running anything, then it is set back to the MTC. Whenever the debugger halts the program, the active component is set to the component that triggered the halt (halting the program manually does not change the active component).

List components

Command syntax: `dlistcomp`

Prerequisites: parallel mode

Description: Lists the name and reference of the MTC and the names and references of all PTCs that are currently executing a function (i.e. all test components that can be the target of the `dsetcomp' command).

The active component is marked with an asterisk (*) in the resulting list.

Set stack level

Command syntax: `dstacklevel <level>`

Prerequisites: call stack is not empty

Description: Sets the current stack level to the specified level. The new level must be a valid index in the call stack (from 1 to the size of the call stack).

The current stack level is the level where variables are listed, printed and overwritten from.

The current stack level is automatically set to 1 whenever the program is halted.

Print call stack

Command syntax: `dprintstack`

Prerequisites: call stack is not empty

Description: Prints the current call stack. The function at the top of the call stack is printed first (with the index of '1'). For each function its index, type, name and current value of parameters are printed (together with the parameters' names and directions).

The function at the current stack level is marked with an asterisk (*).

List variables

Command syntax: `dlistvar [(local | global | comp | all)] [<pattern>]`

Prerequisites: call stack is not empty

Description: Lists the names of all variables visible in the function at the current stack level. The variable names are separated by spaces.

The command's two optional arguments can be used to filter the resulting list.

The first argument can reduce the list to only variables of a certain type (with the values 'local', 'global' and 'comp'; their meanings are explained in section Variables). Setting the argument to 'all' or omitting it does not filter the list, and displays all variables of all types.

The second argument is a pattern string, which can be used to filter the list. It has the same syntax as a TTCN-3 pattern.

NOTE	The names of imported global variables are prefixed with their module name.

Print variable

Command syntax: `dprintvar { (<variable name> | $) }`

Prerequisites: call stack is not empty

Description: Prints the types, names and values of the specified variables. Each printed line contains one variable (or a notification if the variable was not found).

The variables are searched for via their name. The searched names of global variables may also be prefixed with the module name (i.e. <module>.<variable>).

The printed value of a variable has the same format as when logged using the `log` function.

The metacharacter `$' is automatically substituted with the result of the last executed `dlistvar' command (on the active component).

Overwrite variable

Command syntax: `dsetvar <variable name> <new value>`

Prerequisites: call stack is not empty

Description: Overwrites the value of the specified variable.

The variable is searched for via its name. The searched name of a global variable may also be prefixed with the module name (i.e. <module>.<variable>).

The syntax of the new value is the same as the syntax of a module parameter or the `text2ttcn` predefined function (this entire command is essentially a `text2ttcn` call on the specified variable with the specified new value string).

If the new value is incomplete (e.g. the assignment notation is used, and not all fields of the `record`/`set` or not all elements of the `record of`/`set of` are specified), then the rest of the variable is not changed.

Variables that are constant in TTCN-3 (i.e. `consts`, `templates`, `modulepars`, `modulepar templates` and external `consts`) cannot be overwritten by this command either.

Timers, ports and signatures cannot be overwritten.

Print function call data

Command syntax: `dprintcalls [(all | <limit>)]`

Prerequisites: none

Description: Prints the stored function call data (detailed in section 11.1.3).

The first argument (integer number) can be used to limit the amount of snapshots to print. In this case only the last (newest) function calls are printed. Setting the argument to 'all' or omitting it prints all stored function calls.

Each printed line contains the called function's type, whether it's a starting or ending snapshot, the function's name, parameters (including the parameter's direction[21], name and value) and the returned value.

11.3.5. Commands related to the halted state

Halt

Command syntax: `dhalt`

Prerequisites: parallel mode, program is not halted, the MTC's call stack is not empty

Description: Halts the program's execution.

This cannot be used in single mode, since the debugger's user interface only appears if the program is already halted.

Continue

Command syntax: `dcont`

Prerequisites: program is halted

Description: Resumes the program's execution.

Exit

Command syntax: `dexit (test | all)`

Prerequisites: none

Description: Stops the execution of the current program. If the argument is `test', then only the current test case or control part is stopped. If the argument is 'all', then the entire program is terminated.

In parallel mode this does not exit the MC nor does it terminate the MTC. Test execution can be restarted after this with the 'smtc' command.

Step over

Command syntax: `dstepover`

Prerequisites: program is halted

Description: Steps onto the next line of code. This type of stepping does not enter functions. If the current line is the last line in the function, then this steps onto the line that called the function.

NOTE	This command is interrupted if the program is halted for any reason before the next line is reached (this will not cause the program to be halted a second time, when the next line is eventually reached).

Step into

Command syntax: `dstepinto`

Prerequisites: program is halted

Description: Steps onto the next line of code. If there is a function call in the current line, then this steps onto the called function's first line. If the current line is the last line in the function, then this steps onto the line that called the function.

NOTE	This command is interrupted if the program is halted for any reason before the next line is reached (this will not cause the program to be halted a second time, when the next line is eventually reached).

Step out

Command syntax: `dstepout`

Prerequisites: program is halted

Description: Steps out of the current function and onto the line that called the function.

NOTE	This command is interrupted if the program is halted for any reason before the specified line is reached (this will not cause the program to be halted a second time, when the specified line is eventually reached).

Run to cursor

Command syntax: `drunto <module> (<line> | <function>)`

Prerequisites: program is halted

Description: Resumes the program's execution until the specified line or function is reached.

NOTE	This command is interrupted if the program is halted for any reason before the specified location is reached (this will not cause the program to be halted a second time, when the specified location is eventually reached).

11.3.6. Batch files

Both the MC's user interface and the debugger's user interface in single mode support the execution of commands from a text file (batch file).

Each line in the batch file is treated as one command. Empty lines are ignored. Encountering an erroneous command does not stop the batch file's execution.

Execution of batch files can be initiated manually using the `batch' command.

Syntax: `batch <file name>`

Batch files may contain any of the debugger commands listed in this section, including the `batch' command. In parallel mode they may also contain any of the MC's commands.

Certain debugger settings (such as breakpoints) can initiate the execution of a batch file automatically. In this case the program's execution remains halted after the execution of the batch file, unless the batch file contains a command that ends the halted state (e.g. dcont).

The entire debugging process can be automated with the use of batch files that end with the `dcont' command. An initial batch file could initialize the debugger's settings, set all breakpoints to automatically execute a batch file, and/or set a global batch file, and start the program. This way whenever the program would be halted, the automatically executed batch file would resume it.

NOTE	In parallel mode batch files are searched for in the MC's working directory.

11.4. Example

This section contains an example for some of the debugger's features. The example contains one TTCN-3 module with one test case, executed in single mode. Two tests are run. In both cases the debugging process is fully automated with the use of batch files. The executable's -b command line option is used to run the first batch file.

The TTCN-3 module (demo.ttcn):

```
module demo {

type component CT {
  // component variables
  const integer ct_int := 4;
  var charstring ct_str := "abc";
}

type record Rec {
  integer num,
  charstring str
}

type record of integer IntList;

// global variable
template integer t_int := (1..10);

function f_fact(in integer n) return integer {
  if (n == 0 or n == 1) {
    return 1; // line 21
  }
  return n * f_fact(n - 1);
}

testcase tc_demo() runs on CT {
  // local variables
  var Rec v_rec := { num := ct_int, str := ct_str };
  var template IntList vt_list := { [0] := 1, [2] := * };

  if (match(f_fact(v_rec.num), t_int)) {
    v_rec.str := v_rec.str & "!";
  }
  else {
    v_rec.str := v_rec.str & "?";
  }

  var IntList v_list := { 1, 2, 9 };

  if (match(v_list, vt_list)) { // dynamic test case error, line 40
    action("matched");
  }
}

} // end of module
```

The makefile for the example is generated with the following command:

```
ttcn3_makefilegen -sgn demo.ttcn
```

Both tests use the following configuration file (cfg.cfg):

```
[EXECUTE]
demo.tc_demo
```

11.4.1. Test 1

Initializer batch file (start1.bat):

```
debug on
dautobp error on error.bat
dsetbp demo 21 bp21.bat
```

Batch file for the breakpoint at line 21 (bp21.bat):

```
dprintstack
dlistvar all
dstacklevel 5
dlistvar comp
dprintvar ct_int
dlistvar local
dprintvar $
dcont
```

Batch file for the automatic breakpoint for error verdicts (error.bat):

```
dprintcalls
dlistvar local v_*
dprintvar $
dcont
```

Results of running `./demo -b start1.bat cfg.cfg` (debugger output highlighted in yellow, echoed debugger commands highlighted in turquoise):

```
TTCN-3 Test Executor (single mode), version CRL 113 200/5 R5A
Using configuration file: `cfg.cfg'
Test execution halted.
Executing batch file 'start1.bat'.
debug on
Debugger switched on.
dautobp error on error.bat
Automatic breakpoint at error verdict switched on with batch file 'error.bat'.
```

```
dsetbp demo 21 bp21.bat
Breakpoint added in module 'demo' at line 21 with batch file 'bp21.bat'.
Test execution resumed.
Test case tc_demo started.
User breakpoint reached at line 21 in module 'demo'.
Test execution halted.
Executing batch file 'bp21.bat'.
dprintstack
1.  [function]  f_fact([in] n := 1)
2.  [function]  f_fact([in] n := 2)
3.  [function]  f_fact([in] n := 3)
4.  [function]  f_fact([in] n := 4)
5.  [testcase]  tc_demo()
dlistvar all
n t_int
dstacklevel 5
Stack level set to:
5.  [testcase]  tc_demo()
dlistvar comp
ct_int ct_str
dprintvar ct_int
[integer] ct_int := 4
dlistvar local
v_rec vt_list
dprintvar $
[Rec] v_rec := { num := 4, str := "abc" }
[IntList template] vt_list := { 1, <uninitialized template>, * }
dcont
Test execution resumed.
demo.ttcn:40: Dynamic test case error: Matching with an uninitialized/unsupported
integer template.
Automatic breakpoint (error verdict) reached at line 40 in module 'demo'.
Test execution halted.
Executing batch file 'error.bat'.
dprintcalls
[testcase]  started     tc_demo()
[function]  started     f_fact([in] n := 4)
[function]  started     f_fact([in] n := 3)
[function]  started     f_fact([in] n := 2)
[function]  started     f_fact([in] n := 1)
[function]  finished    f_fact([in] n := -) returned 1
[function]  finished    f_fact([in] n := -) returned 2
[function]  finished    f_fact([in] n := -) returned 6
[function]  finished    f_fact([in] n := -) returned 24
dlistvar local v_*
v_rec v_list
dprintvar $
[Rec] v_rec := { num := 4, str := "abc?" }
[IntList] v_list := { 1, 2, 9 }
dcont
Test execution resumed.
```

```
Test case tc_demo finished. Verdict: error
Verdict statistics: 0 none (0.00 %), 0 pass (0.00 %), 0 inconc (0.00 %), 0 fail (0.00
%), 1 error (100.00 %).
Test execution summary: 1 test case was executed. Overall verdict: error
```

11.4.2. Test 2

Initializer batch file (start2.bat):

```
debug on
dsetbp demo 40 bp40.bat
```

Batch file for the breakpoint at line 40 (bp40.bat):

```
dprintvar vt_list
dsetvar vt_list { [1] := 2 }
dcont
```

Results of running ./demo -b start2.bat cfg.cfg (debugger output highlighted in yellow, echoed debugger commands highlighted in turquoise):

```
TTCN-3 Test Executor (single mode), version CRL 113 200/5 R5A
Using configuration file: `cfg.cfg'
Test execution halted.
Executing batch file 'start2.bat'.
debug on
Debugger switched on.
dsetbp demo 40 bp40.bat
Breakpoint added in module 'demo' at line 40 with batch file 'bp40.bat'.
Test execution resumed.
Test case tc_demo started.
User breakpoint reached at line 40 in module 'demo'.
Test execution halted.
Executing batch file 'bp40.bat'.
dprintvar vt_list
[IntList template] vt_list := { 1, <uninitialized template>, * }
dsetvar vt_list { [1] := 2 }
[IntList template] vt_list := { 1, 2, * }
dcont
Test execution resumed.
Action: matched
Test case tc_demo finished. Verdict: none
Verdict statistics: 1 none (100.00 %), 0 pass (0.00 %), 0 inconc (0.00 %), 0 fail
(0.00 %), 0 error (0.00 %).
Test execution summary: 1 test case was executed. Overall verdict: none
```

[21] in, inout or out

Chapter 12. Tips & Troubleshooting

This chapter deals with various topics, which could not have been assigned to any of the previous chapters.

12.1. Type Aliasing

Type aliasing in TTCN-3 means that you can assign an alternative name to an existing type. The syntax is similar to a subtype definition, but the subtype restriction tag (value list or length restriction) is missing.

```
type MyType MyAlternativeName;
```

The type aliasing is implemented in the test executor, but it translates this TTCN-3 definition to a C `typedef` statement.

```
typedef MyType MyAlternativeName;
```

The limitation of the C typedef is that the C++ compiler cannot distinguish between the original and alias name in polymorphism (i.e. the identically named functions with parameter type `MyType` and `MyAlternativeName` are treated as same). That is, if you define a port type that allows the sending or receiving both of the original and aliased type, the generated C++ code cannot be compiled because the Test Port class contains two identical send/receive function.

As a work-around to this problem you can repeat the definition of the original type using the alternative name instead of type aliasing. In this case two differently named, but identical classes will be generated and the polymorphism problem will not occur.

12.2. Reusing Logged Values or Templates in TTCN-3 Code

Writing templates can be time-consuming task. To save some time and work, you can use the logs of the messages already sent or received to write templates.

If you would like to use a logged value in TTCN-3 code, then using the `logformat` utility (see the section 13.3 of the TITAN User Guide [13] about this utility) you have to follow these steps:

1. Start a text editor and open the (formatted) log file and the TTCN-3 source file.
2. Select and copy the desired value from the log file.
3. Paste the value at the corresponding position in the TTCN-3 code.
4. Finally, make the following changes:
    - The enumerated values are followed by their numerical equivalents within parentheses. Delete them including the parentheses.
    - If an octetstring value contains only visible ASCII characters, then the hexadecimal octetstring notation is followed by its character string representation between quotation marks and parentheses. Delete the character string (including the parentheses).

- If a `record`, `set`, `record of` or `set of` value contains no fields or elements, then the logformat utility changes the value from `{}` to `{(empty)}` in the log. Delete the word (empty) (including parentheses).

12.3. Using the TTCN-3 Preprocessing Functionality

NOTE	This feature, as preprocessors in general, should be avoided if not absolutely necessary.

Tips for the `Makefile` generated using the option `-p:`

- All the options for the C precompiler can be specified using the variable `CPPFLAGS_TTCN3`. Do not confuse it with the variable `CPPFLAGS`, which is used on the generated C++ code. If standard TTCN-3 output is needed the flag `-P` has to be added manually to the variable `CPPFLAGS_TTCN3`. The resulting `ttcn` files can be compiled with any TTCN-3 compiler (if other special language extensions are not used). Globally used preprocessor symbols can be defined here with the option `-D`. For example to compile the debug version of a project a `DEBUG` symbol can be specified with `-DDEBUG`.
- Files which are included in the `.ttcnpp` source files (with `#include`) and do not need to be translated can be specified in the `TTCN3_INCLUDES` variable. These files will be checked for modification when the `.ttcnpp` files are processed by `make`; any modification will trigger preprocessing of all the `.ttcnpp` files and the recompilation of the affected modules. If the suffix of a file is `.ttcnin` the Makefile Generator will add it to `TTCN3_INCLUDES`; in all other cases the file has to be added manually.
- Do not use any file name identical to the name of the intermediate file produced by the C preprocessor. The intermediate file name is generated by replacing the suffix `.ttcnpp` with `.ttcn`. Use the naming convention of naming the file as the module name avoiding such name collisions.
- The default C preprocessor used to preprocess TTCN-3 files can be replaced by editing the CPP variable.

There are minor issues when precompiling TTCN-3 code with a C preprocessor, these are resulting from the differences between the C and TTCN-3 languages. Tips for writing the `.ttcnpp` files:

- Do not define the B, O and H macros, these letters are used as part of the bitstring, octetstring and hexstring tokens in TTCN-3, but the C preprocessor will replace them.
- There are some predefined macros in the C preprocessor which will be always replaced, do not use any TTCN-3 identifier identical to these. These macros start with double underscore followed by uppercase letters. Some of the most common macros which might be useful:
  - - **FILE** This macro expands to the name of the current input file, in the form of a C string constant.
  - - **LINE** This macro expands to the current input line number, in the form of a decimal integer constant.
  - - **DATE** This macro expands to a string constant that describes the date on which the preprocessor is being run.

- - **TIME** This macro expands to a string constant that describes the time at which the preprocessor is being run.

When writing preprocessor directives keep in mind that within the directive the C preprocessor syntax is in use, not the TTCN-3. Operators such as `defined` or || can be used.

Watch out for macro pitfalls, some well known are: side effects, misnesting, and operator precedence problems.

12.4. More Efficient Implementation of the Types record of and set of

The new implementation of the mentioned TTCN types and their ASN counterparts was introduced in TITAN version 1.7.pl2 (R7C). The performance of assigning record of/set of typed variables improved significantly since TITAN version 1.7.pl1 (R7B). The new implementation uses reference counting when an assignment is made. The whole data structure is copied only when necessary, for example, the user wants to modify its value. Using temporary variables improves the quality of the code.

12.5. Workflow for Native XML Support

In this very short and simple example we are presenting and explaining the procedure of using the XML encoding / decoding. Through the steps of the workflow you can understand the XML related possibilities of TITAN.

First look at data types. These are the base of every test. If you have data representation in XML format (XSD is the standard for defining data types), you have to convert it into the equivalent TTCN-3 data types using the XSD converter. This is a shortened variant of the commonly used SOAP protocol.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.ericsson.com/cai3g1.2/"
targetNamespace="http://schemas.ericsson.com/cai3g1.2/" elementFormDefault="qualified"
attributeFormDefault="unqualified">

    <xs:element name="Set">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="MOType" type="MoType" />
                <xs:element name="MOId" type="AnyMOIdType" />
                <xs:element name="MOAttributes">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element ref="SetMODefinition" />
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="extension" type="AnySequenceType"
                    minOccurs="0" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="AbstractSetAttributeType" abstract="true"/>

<xs:element name="SetMODefinition"
type="AbstractSetAttributeType" abstract="true"/>

    <xs:complexType name="AnyMOIdType">
        <xs:sequence>
            <xs:any namespace="##any"
processContents="lax" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="AnySequenceType">
        <xs:sequence>
            <xs:any namespace="##any"
processContents="lax" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="MoType">
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-Za-z][_A-Za-z0-9]*@.*"/>
        </xs:restriction>
    </xs:simpleType>

</xs:schema>
```

After conversion you have a TTCN-3 module whose name is derived from the targetNamespace attribute of <schema> element. This module contains only data types. Two other files are generated also with standardized base datatypes:

- UsefulTtcn3Types.ttcn
- XSD.ttcn

The content of the generated TTCN-3 file:

```
module schemas_ericsson_com_cai3g1_2 {

import from XSD all;

type record Set
{
    MoType mOType,
    AnyMOIdType mOId,
    record {
        SetMODefinition setMODefinition
    } mOAttributes,
    AnySequenceType extension_ optional
}
with {
variant (mOType) "name as capitalized";
variant (mOId) "name as capitalized";
variant (mOAttributes) "name as capitalized";
variant (mOAttributes.setMODefinition) "name as capitalized";
variant (extension_) "name as 'extension'";
};

type record AbstractSetAttributeType
{};

type AbstractSetAttributeType SetMODefinition;

type record AnyMOIdType
{
    record length(1 .. infinity) of XSD.String elem_list
}
with {
variant (elem_list) "untagged";
variant (elem_list[-]) "anyElement";
};

type record AnySequenceType
{
    record length(1 .. infinity) of XSD.String elem_list
}
with {
variant (elem_list) "untagged";
```

```
variant (elem_list[-]) "anyElement";
};

type XSD.String MoType /* (pattern "[A-Za-z][_A-Za-z0-9]*@.*") */;

}
with {
encode "XML";
variant "namespace as 'http://schemas.ericsson.com/cai3g1.2/'";
variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
variant "elementFormQualified";
}
```

Also manually created type definitions can be used and combined together. This example shows the next module containing also data types.

```
module SOAP {

import from XSD all;
import from schemas_ericsson_com_cai3g1_2 all;

type record ApplicationHeaderContent
{};

type record ApplicationBodyContent {
    Set setRequest
};

type record SoapEnvelope {
    SoapHeader header optional,
    SoapBody body
}
with {
variant "name as 'Envelope'";
variant (header) "name as capitalized";
variant (body) "name as capitalized";
};

type record of ApplicationHeaderContent SoapHeader;

type union SoapBody {
    XSD.String fault,
    record of ApplicationBodyContent content
}
with {
variant (fault) "name as capitalized";
variant (content) "untagged";
variant (content[-]) "untagged";
};

}
with {
encode "XML";
variant "controlNamespace 'http://www.w3.org/2001/XMLSchema-instance' prefix 'xsi'";
variant "namespace as 'http://schemas.xmlsoap.org/soap/envelope/' prefix 'SOAP-ENV'";
variant "namespace as 'http://schemas.xmlsoap.org/soap/encoding/' prefix 'SOAP-ENC'";
variant "namespace as 'http://schemas.ericsson.com/cai3g1.1/' prefix 'ns3'";
}
```

The XML encoding/decoding can be accessed via external functions. To encode a value of the SoapEnvelope type (the top-level record type in our example) to XML, or to decode XML data into a value of SoapType, we can use external functions like the following:

```
module SOAP_ExternalFunctions {

import from SOAP all;

external function enc_SOAP(in SoapEnvelope pdu) return octetstring
with { extension "prototype (convert) encode(XER:XER_EXTENDED)" }

external function dec_SOAP(in octetstring stream) return SoapEnvelope
with { extension "prototype (convert) decode(XER:XER_EXTENDED)" }

}
```

The "prototype (convert)" attribute instructs the compiler to generate a C++ implementation for each of the external functions (see section 4.22.4 above). This permits the use of the encoding/decoding functions directly from TTCN-3 code.

In case more sophisticated processing is required (or some form of pre/postprocessing), the encoder/decoder functions can be reimplemented in C++. The basic functionality provided by the compiler can be used as a starting point.

NOTE	In this case all the ``with'' attributes in the example above must be removed from the external function declaration (otherwise the compiler will generate the functions again with the same signature and duplicate symbol errors will appear at link time).

For representing the usage of encoding and decoding we created this demo module that contains one template definition and in the testcase we will apply encoding and decoding.

```
module demo {

import from SOAP all;
import from SOAP_ExternalFunctions all;

template SoapEnvelope SoapTemplate :=
{
  header := omit,
  body := {
    content := { {
        setRequest := {
          mOType       := "JB007",
          mOId         := {
            elem_list := {
             "<catalog><books count='3'/></catalog>",
             "<catalog><movies count='1'/></catalog>"
            }
          },
          mOAttributes := {
            setMODefinition := {
```

```
            }
          },
          extension_ := omit
        }
      } }
  }
}


type component SOAP_CT
{
  var octetstring v_encodedPDU, v_decodePDU;
  var SoapEnvelope v_decodedPDU;
}

testcase tc_encdec() runs on SOAP_CT
{
  v_encodedPDU := enc_SOAP(valueof(SoapTemplate));

  v_decodedPDU := dec_SOAP(v_encodedPDU);

  log("Encoded set request (SoapEnvelope): ", v_encodedPDU);
  log("Decoded set request (SoapEnvelope): ", v_decodedPDU);
}

control
{
  execute(tc_encdec());
}

}
```

The complete demo project is now ready. If running the test case a log file will be generated in which we can find the encoded representation of the value and the decoded variant.

The resulting XML encoding:

```
<ns3:Envelope xmlns:ns3='http://schemas.ericsson.com/cai3g1.1/'>
  <ns3:Body>
    <ns3:ApplicationBodyContent>
      <ns3:setRequest>
        <ns3:MOType>JB007</ns3:MOType>
        <ns3:MOId>
          <catalog><books count='3'/></catalog>
          <catalog><movies count='1'/></catalog>
        </ns3:MOId>
        <ns3:MOAttributes>
          <ns3:SetMODefinition/>
        </ns3:MOAttributes>
      </ns3:setRequest>
    </ns3:ApplicationBodyContent>
  </ns3:Body>
</ns3:Envelope>
The decoded format (a TTCN-3 value of type SoapEnvelope)

{
  header := omit,
  body := {
    content := { {
        setRequest := {
          mOType       := "JB007",
          mOId         := {
            elem_list := {
             "<catalog><books count='3'/></catalog>",
             "<catalog><movies count='1'/></catalog>"
            }
          },
          mOAttributes := {
            setMODefinition := {
            }
          },
          extension_ := omit
        }
      } }
  }
}
```

12.6. Debug Memory Use of Record/set of Types

One of the common source of the memory leakage in the TTCN test suite is the ever-growing record/set of's. In order to help the debug of such issue, the test suite should be compiled with `-DTITAN_MEMORY_DEBUG_SET_RECORD_OF` flag added to `CPPFLAGS` in the Makefile.

That flag activates a WARNING log statement, issued after every 1000th element added to the record/set of.

Example:

```
module test {

type component test_CT {}
type record of charstring roc

testcase tc_test() runs on test_CT
{
    var roc r:={}
    var integer k

    for(k:=0;k<10001;k:=k+1){
      r[sizeof(r)]:="a";
    }
}

control
{
  execute(tc_test())
}

}
```

Running of the example test above will produce the following log:

```
MTC@esekilxxen1844: Warning: New size of type @test.roc: 1000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 2000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 3000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 4000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 5000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 6000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 7000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 8000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 9000
MTC@esekilxxen1844: Warning: New size of type @test.roc: 10000
```

12.7. Parsing limitations

The TITAN compiler uses `bison` for parsing TTCN-3 files. The parser cannot process every possible combination of TTCN-3 language elements. Some language elements can cause conflicts in the parser, because they can be interpreted in multiple ways (these are called `shift-reduce` conflicts).

One such conflict was introduced in version 7.1.0. It causes the semicolon (`;`) after an `altstep` call in an `alt` statement to no longer be optional (unless the `altstep` call is in the last `alt` branch). For example:

```
alt {
  [] myAltstep()
  [] myPort.receive(integer: ?) { ... }
}
```

If the semicolon after `myAltstep()` is omitted, then the parser is conflicted in how to interpret the following left square bracket (`[`) character.

- It can be interpreted as the continuation of the reference to `myAltstep`, in which case a square bracket would be the start of an array index applied to the return value of `myAltstep`. This is the `shift` case. (Function, altstep and testcase calls cannot be distinguished during parsing. This happens later during semantic checking.)
- It can be interpreted as the start of the next `alt` branch. This is the `reduce` case.

The `bison` parser is greedy and will always choose the `shift` case in these types of conflicts. In this case it will display an error message, saying that an array index was expected instead of the right square bracket (`]`). This error can be avoided by adding the omitted semicolon after `myAltstep()`.

There are many similar conficts in the compiler's parser. Most of them are caused by semicolons at the end of statements being optional. Many of them very unlikely to occur in actual TTCN-3 code.

Example 2:

```
function g() {
  return
  f();
}
```

In this case the parser interprets the `f()` function call as the return value of function `g` (`shift` case), instead of the next statement after an empty `return` statement (`reduce` case). A semicolon after `return` would resolve the conflict. However, such code would be pointless, since statements after `return` will not be executed.

Example 3:

```
altstep as() runs on CT {
  template integer t := a
  [] tmr.timeout { log("timeout"); }
}
```

Yet another conflict between array indexes and alt-guards: if whatever is at the end of the last local declaration in an `altstep` can in theory be indexed, then the following square brackets will be interpreted as an array index instead of an alt-guard. A semicolon at the end of the local declaration would resolve this conflict.

Chapter 13. References

- [1] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 1: Core Language European Telecommunications Standards Institute ES 201 873-1 Version 4.1.1, July 2009
- [2] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 4: TTCN-3 Operational Semantics European Telecommunications Standards Institute. ES 201 873-4 Version 4.1.1, June 2009
- [3] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 7: Using ASN.1 with TTCN-3 European Telecommunications Standards Institute. ES 201 873-7 Version 4.1.1, July 2009
- [4] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 9: Using XML Schema with TTCN-3 European Telecommunications Standards Institute. ES 201 873-9 Version 4.1.1, June 2009
- [5] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. TTCN-3 Language Extensions: Behaviour Types European Telecommunications Standards Institute. ES 202 785 Version 1.5.1, Aug 2017
- [6] ITU-T, X.680, Information TechnologyAbstract Syntax Notation One (ASN.1): Specification of basic notation International Telecommunication Union, July 2002
- [7] ITU-T, X.681, Information TechnologyAbstract Syntax Notation One (ASN.1): Information object specification International Telecommunication Union, July 2002
- [8] ITU-T, X.682, Information Technology Abstract Syntax Notation One (ASN.1): Constraint specification International Telecommunication Union, July 2002
- [9] ITU-T, X.683, Information Technology Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specification International Telecommunication Union, July 2002
- [10] ITU-T, X.690, Information Technology ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)International Telecommunication Union, July 2002
- [11] ISO/IEC 10646-1, Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane, Second edition, 200009-15
- [12] RFC3629: UTF-8, a transformation format of ISO 10646
- [13] User Guide for TITAN TTCN-3 Test Executor
- [14] Installation guide for TITAN TTCN-3 Test Executor
- [15] Release Notes for TITAN TTCN-3 Test Executor
- [16] API Technical Reference for TITAN TTCN-3 Test Executor

- [17] User Guide for the TITAN Designer for the Eclipse
- [18] ETSI ES 201 373-1 V4.3.1 (2011-06)
- [19] 1092-212 Uen (EN/LZB 101 01/1D) Product Changes
- [20] ITU-T, X.696, Information TechnologyASN.1 encoding rules: Specification of Octet Encoding Rules (OER) International Telecommunication Union, August 2015
- [21] ETSI ES 202 781 V1.4.1. (2015-06 Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support)
- [22] RFC7049: Concise Binary Object Representation (CBOR) (October 2013)
- [23] BSON specification version 1.1
- [24] MongoDB Extended JSON document
- [25] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 11: Using JSON with TTCN-3 European Telecommunications Standards Institute. ES 201 873-11 Version 4.8.1, May 2018
- [26] ETSI ES 202 782 V1.3.1. (2015-06 Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing)
- [27] ETSI ES 203 790 V1.1.1. (2015-06 Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Object-Oriented Features)
- [28] ETSI ES 203 790 V1.1.1. (2015-06 Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extension: Advanced Matching)
- [29] ITU-T, X.691, Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)
- [30] User Guide for TITAN Standalone ASN.1 Encoder

Chapter 14. Abbreviations

API

Application Programming Interface

ASCII

American Standard Code for Information Interchange

ASN.1

Abstract Syntax Notation One

ATS

Abstract Test Suite

BER

Basic Encoding Rules (of ASN.1)

BNF

Backus-Naur Formalism

CER

Canonical Encoding Rules (of ASN.1)

CPP

Cello Packet Platform

CR

Change Request

DER

Distinguished Encoding Rules (of ASN.1)

DNS

Domain Name Server

DTD

Document Type Description

ETS

Executable Test Suite

ETSI

European Telecommunications Standards Institute

FIFO

First In, First Out

GCC
: GNU Compiler Collection

GUI
: Graphical User Interface

HC
: Host Controller

HTML
: Hypertext Markup Language

HTTP
: HyperText Transfer Protocol

IDL
: Interface Description Language

IE
: Information Element

IP
: Internet Protocol

ISO
: International Organization for Standardization

JSON
: JavaScript Object Notation

LCOV
: A graphical front-end for GCC's coverage testing tool

LSB
: Least Significant Bit

MC
: Main Controller

MSB
: Most Significant Bit

MTC
: Main (or Master) Test Component

OSE
: Open System Environment

PDU

Protocol Data Unit

pl

Patch Level

PTC

Parallel Test Component

PT

Port Type

SOAP

Simple Object Access Protocol

SUT

System Under Test

TC

Test Component (either MTC or PTC)

TCC

Test Competence Center

TCP

Transmission Control Protocol

TLV

Tag, length, value

TPD

Titan Project Descriptor

TR

Trouble Report

TTCN

Testing and Test Control Notation

TTCN-2

Tree and Tabular Combined Notation version 2

TTCN-3

Tree and Tabular Combined Notation version 3 (formerly)Testing and Test Control Notation (new resolution)

UDP

User Datagram Protocol

URL

Universal Resource Locator

URI

Uniform Resource Identifier

W3C

World Wide Web Consortium

XML

W3C Extensible Markup Language

XSD

W3C XML Schema Definition