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

Kristóf Szabados

Version 10.1.1, 2024-06-05

Table of Contents

Abstract

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

Copyright

Disclaimer

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, for the Java side of the TITAN TTCN-3 Toolset.

1.2. Target Groups

This document is intended for programmers of TTCN–3 test suites, using the prototype Java code generator provided in the plugins, with information in addition to that provided in the TITAN User Guide, API Technical Reference and Programmers' Technical Reference Guide. It is recommended that the programmer reads the TITAN User Guide before reading this document.

1.3. Naming Convention

This document uses the expressions "C side" and "Java side" in relation to the TITAN TTCN-3 Toolset and Test Executor.

C side is used to reference the "original" part of the TITAN TTCN-3 Toolset available from command line. The compiler, makefile generator, the libraries users need to link their executables to during build time.

Java side is used to reference the part of the TITAN TTCN-3 Toolset supporting compiling TTCN-3 and ASN.1 code into Java classes via Java source code and the runtime libraries needed for this form of building.

1.4. 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 TTCN–3 language constructs detailed in [27] are not supported in the current version of the Test Executor on both the C and the Java side. The following list extend that list, with the TTCN–3 language constructs that are not supported, in addition, in the current version of the Java side 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**.

- The `update`, `interleave`, `label`, `goto` statements are not yet supported. (19.7, 19.8, 20.4 and 22.3.1 in [1])
- The `hostId` predefined function is not yet supported.
- Additionally the `@profiler.start`, `@profiler.stop`, `string2ttcn` TITAN extensions are also not yet supported on the Java side.
- The `@profiler.running` TITAN extension is also not supported.
- Concatenating template strings is not yet supported on the Java side.

WARNING	The current version of the Java side of the Test Executor is just a prototype version. Please note that there might still be some changes in some of its APIs.

Chapter 3. TTCN–3 Language Extensions

The Test Executor supports several non-standard additions to TTCN–3 Core Language, as detailed in [27], in order to improve its usability or provide backward compatibility with older versions.

The following list contains the TTCN–3 language extensions that are not yet supported by the Java side of the Test Executor. The sections/features not listed here are supported.

3.1. TTCN–3 Preprocessing

Preprocessing of the TTCN-3 files with a C style preprocessor is supported by the Java side.

Contrary to the C side, on the Java side preprocessing is supported by an internal pre-processor. That is the generated Java files will already have the pre-processable content pre-processed.

Parameterized macros are not supported on the Java side.

3.2. Implicit Message Encoding

Compared to the description in section 3.22 of [27] the Java side has 2 major differences: Only RAW encoding is supported for now. The syntax to be used in Java differs slightly from the one used in C++:

The TTCN–3 attribute `errorbehavior(INCOMPL_ANY:ERROR)`, for example, instead of being mapped to the following C++ statement

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

is mapped to the following Java statement

```
TTCN_EncDec.set_error_behavior(TTCN_EncDec.error_type.ET_INCOMPL_ANY,
  TTCN_EncDec.error_behavior_type.EB_ERROR);
```

3.3. RAW Encoder and Decoder

The Java side supports the same RAW Encoder and Decoder features as the C side.

3.4. TEXT Encoder and Decoder

The TEXT Encoder and Decoder is not yet supported on the Java side.

3.5. XML Encoder and Decoder

The XML Encoder and Decoder is not yet supported on the Java side.

3.6. JSON Encoder and Decoder

The Java side supports the same RAW Encoder and Decoder features as the C side.

3.7. OER Encoder and Decoder

The OER Encoder and Decoder is not yet supported on the Java side.

3.8. 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.

3.8.1. 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.

The Java side of the toolset provides the same features as the C side for TTCN-3 level checking of consistency.

3.8.2. Consistency Check in the Generated Code

The java side offers different consistency checks compared to the C side, for the generated code.

When connecting to the Main Controller in parallel mode, TITAN verifies that the Main Controller and the Java side binaries are of the exact same version of TITAN. This is done to ensure that, that both sides use the same communication protocol.

What is not checked on the Java side or checked differently:

- There is no platform check as Java is platform independent.
- The Java runtime will check if it can execute the compiled code. Generally a Java runtime should be able to execute any Java code built using an earlier Java version.
- During the compilation of the Java code, the Java compiler will check if it is able to compile all parts of the code.

3.9. Negative Testing

Negative Testing is not yet supported on the Java side.

3.10. Differences between the Java side runtime, the C side Load Test Runtime and the C side Function Test Runtime

The Java side was based on the Load Test runtime of the C side. For now it has the same features and limitations.

Please note, that based on the differences between Java and C++, the Java runtime should be treated as its own version of the runtime, when preparing for future developments.

3.11. Profiling and code coverage

The Java side does not yet support profiling and code coverage measuring support directly.

For the time being we recommend using the tools built into Eclipse on the Java generated code (The Java side projects also behave as normal Java projects for Eclipse tooling), or other tools provided for Java.

Chapter 4. Supported ASN.1 Constructs and Limitations

The following list contains the ASN.1 features that are not supported on the Java side, above the limitations listed in [27] for the C side:

- BER Encoding and Decoding are not supported.
- subtypes are not checked.
- Charsymbols are not parsed

Chapter 5. Compiling TTCN–3 and ASN.1 Modules

You can translate your TTCN–3 and ASN.1 modules, located in TITAN Java projects, to Java source code using the builder built into the Designer plugin.

This builder is automatically invoked, when the eclipse's build command is selected on a project. When the **Build Automatically** option is selected in the **Project** menu, eclipse automatically builds the project, in the background, when a file is changed.

The TITAN provided builder will use all TTCN-3 and ASN.1 files from all folders that are not excluded. The .java files are generated into the **java_src** folder of the project into a package generated from the name of the project in this format: "org.eclipse.titan." + projectname + ".generated".

The usual and recommended suffix is `.ttcn` for TTCN–3 and `.asn` for ASN.1 source files, but it is not stringent[1]. 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 `.java`.

NOTE	In the ASN.1 module names hyphens are replaced by underscore character.

WARNING	If you have a modular test suite (the code located in several projects that reference each other), to build a particular project you have to first build all projects it references. This should be done automatically by eclipse.

5.1. Build Options

The options governing how a project is built can be set via right clicking on the project and selecting **Properties** / **TITAN Java Project Properties** and in the window that appears on the **TITAN** / **Flags** sub-page.

The following options are supported:

- `Disable RAW encoding (-r)`

  Disables the generation of RAW encoder/decoder routines for all TTCN–3 types.

- `Disable attribute validation (-0)`

  Disables the validation of "with" attributes.

WARNING	This option should only be used temporarily and only by people transferring projects from other TTCN-3 tool vendors. As the attribute validation is turned off, users will not be notified of invalid attributes, or errors within attributes.

- `Add source line info for logging (-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.

- `Allow 'omit' in template value lists (legacy behavior) (-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.

- `Force the generation of Seof types (-F)`

  Forces the code generator to generate the full classes for record of and set of types. When turned off, and the of type of the set of/record of type is a basic type, the generated code will only refer to pre-generated classes in the runtime library, saving compilation time.

- `Enable object oriented programming - OOP (-k)`

  Enable object oriented programming language elements. It is not working yet on Titan Java Projects. Syntactic and semantic analyzer and java compiler do not support OOP yet.

5.2. Makefile Generator

The Java side of TITAN does not generate a Makefile as the build is governed by the built in tools of Eclipse.

5.3. The Compilation Process for TTCN–3 and ASN.1 Modules

The Java side compilation is integrated into the Designer plug-in using it’s syntactic and semantic checking features.

During their run both the Designer’s analysis and Java code generator’s progress can eb followed in the Progress view of eclipse.

During its run, the Designer might also report some of its activities on the TITAN Debug Console like the following.

```
On-the-fly analyzation of project bughunt started
  **The project bughunt does not seem to need syntax check.
  ** Had to start checking at 0 modules.
  **On-the-fly semantic checking of projects (4 modules) took 1.04777E-4 seconds
Generating code for module `common'
Generating code for module `Bug'
re-Generated code for module `Bug'
Generating code for module `single_test'
Generating code for module `parallel_test'
Generated 4 Java files.
Generating code for single main
The whole analysis block took 0.0022510720000000002 seconds to complete
```

The activities leading to the compilation of the project can be grouped to 3 sets.

5.3.1. The initial analysis

First, the Designer 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 in the Problems view with the appropriate location information. Whenever it is possible, the Designer 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 Designer 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 Designer reports a warning when the corresponding definition is correct, but it might have unwanted effects.

5.3.2. Subsequent analysis after change

Instead of repeating the analysis of the whole project always, the Designer is able to offer incremental analysis. This means that after the first analysis, the semantic information gained from the TTCN-3 and ASN.1 files is not deleted, but kept in the memory. So when users edit something in the same project, the Designer only has to re-read that file, and repeat the semantic analysis on the smallest set of semantic entities, that might be affected by the change. Reducing the length of subsequent analysis duration times.

5.3.3. Actual Java code generation and Java compilation

After at least one analysis was done on a project, the Designer can generate a Java file, for each module without errors, 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 Java file will be

`MyModule.java`. 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 Java code for `My-Asn-Module` will be placed into `My_Asn_Module.java`). The Java files are generated into the "java_src" folder of the project into a package generated from the name of the project in this format: "org.eclipse.titan." + projectname + ".generated".

By default, the compiler generates the Java code for the input modules:

- that do not have any errors inside them
- and were not yet analyzed or the last change might have affected them
- and either do not already have a Java file generated for them, or the content of the file needs to be updated.

This sophisticated methods allows to reduce the length of the build after a change, by minimizing the amount of code re-analyzed, re-generated and re-compiled by Java.

Once the Designer's built in Java code generator finishes, the Java compiler of Eclipse takes the generated Java code and compiles them into .class files. Which can be used for execution inside eclipse, or can be exported as jar files, to be executed from the command line.

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

ASN.1	TTCN–3
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

5.4. Particularities of ASN.1 Modules

The Designer performs the same checks on ASN.1 modules as the compiler, but does not yet have support for BER encoding/decoding.

5.5. Using Component Relation Constraints from TTCN–3

The Designer performs the same checks on ASN.1 modules as the compiler, but does not yet have support for BER encoding/decoding.

[1] .ttcn3, or .asn1 suffixes are supported as well.

Chapter 6. The Run-time Configuration File

In general the Java side supports the exact same configuration file format and options in the same way as the C side does, described in chapter 7 of [27]. There are some features, that are not yet supported on the Java side:

- LoggerPlugins within the LOGGING section are not yet supported. The section is read correctly, but such plugins are not loaded during runtime.
- EXTERNAL_COMMANDS section is not yet supported. The section is read correctly, but the scripts set there will not be executed during runtime.
- In MAIN_CONTROLLER section the `UnixSocketsEnabled` feature is not supported. Java does not seem to offer support for this feature.
- It is also not yet possible to configure the logging options dynamically.

On the C side, in the configuration file it is possible to use the `%e` Meta-character in the log file's name, to insert into it the name of the binary generating the log files. On the Java side this `%e` Meta-character will represent the name of the project. This is because on the Java side the easiest and fastest way to execute TITAN Java projects does not involve the generation of a "binary" to be executed. As such in these situations the concept of the "name of the binary" does not exist.

Chapter 7. Code Coverage of TTCN-3 Modules

Measuring Code Coverage directly from TITAN is not yet supported on the Java side.

Chapter 8. The TTCN-3 Debugger

Debugging TTCN-3 directly from TITAN is not yet supported on the Java side.

Chapter 9. Test Ports

The Java source code generated by the Java code generator is protocol independent, that is, it does not contain any device specific operations. To provide the connection between the executable test suite and SUT, that is, the physical interface of the test equipment[2], a so-called Test Port is needed.

The Test Port is a software library written in Java language, which is a part of the executable test program. It maps the device specific operations to function calls specified in an API. This chapter describes the Test Port API in details.

9.1. Generating the Skeleton

The functions of Test Ports must be written by the user who knows the interface between the executable test suite and the test equipment. In order to make this development easier, Eclipse features can be used to generate and update Test Port skeletons. A Test Port belongs to one certain TTCN–3 port type, so the skeleton is generated based on port type definitions.

A Test Port consists of two parts. One part is generated automatically by the Java code generator, and it is put into the generated Java code. The user has nothing to do with this part.

The other part is a Java class, which is written mainly by the user. This class can be found in a separate Java file (their suffixes are `.java`). It is recommended to store this file in a folder separate from the generated java files (for example called `user_provided`), so as it should not be deleted when clearing the project. The name of the source files and the Java class have to be identical to the name of the port type. And the Java class has to be located in the Java package whos name is generated as `org.eclipse.titan.` + projectname + `.user_provided`. Please note that the name mapping rules described in Mapping of Names and Identifiers also apply to these class and file names.

During the compilation, when the Java compiler encounters the usage of a Test Port that does not yet has a user generated implementation, it will report an error in the generated code for missing its import. Also offering Quick Fixes either by simply bringing the mouse cursor over the error location, or by right clicking and selecting Quick Fix from the menu. Using the action that starts like `Create class 'MyMessagePort' in package ⋯` eclipse will automatically generate the class the user needs. Once the class is create one should set its base class and right click in its body part selecting the `Source/Override\Implement Methods⋯` to automatically generate a skeleton for the needed functions.

If the list of message types/signatures of a TTCN-3 port type changes, the list of the Test Port class member functions also needs to change. Java will report build error like "The typeXY must implement the inherited abstract method...". In this case, the `Override\Implement Methods⋯` action should be invoked again, to create the skeletons of the newly required functions.

If you have defined a TTCN–3 port type that you intend to use for internal communication only (that is, for sending and receiving messages between TTCN–3 test components), you do not need to generate and compile an empty Test Port skeleton for that port type. Adding the attribute with `{extension "internal"}` to the port type definition in the TTCN–3 module disables the generation and use of a Test Port for the port type.

In the following we introduce two port type definitions: one for a message based and another one for a procedure based port. In our further examples we will refer to the test port skeletons generated according to these definitions given within the project called `MyProject` and module called `MyModule`.

9.2. Message-based Example

The definition of `MyMessagePort`:

```
type port MyMessagePort message
{
  in octetstring;
  out integer;
  inout charstring;
};
```

That is, the types integer and charstring can be sent, and octetstring and charstring can be received on port `MyMessagePort`.

The initial Test Port file (that is, `MyMessagePort.java`) will look as follows:

```
package org.eclipse.titan.MyProject.user_provided;

import org.eclipse.titan.MyProject.generated.MyModule.MyMessagePort_BASE;
import org.eclipse.titan.runtime.core.TitanCharString;
import org.eclipse.titan.runtime.core.TitanInteger;

public class MyMessagePort extends MyMessagePort_BASE {

    public MyMessagePort(final String name) {
        super(name);
    }

    @Override
    protected void outgoing_send(TitanInteger send_par) {
        // TODO Auto-generated method stub
    }

    @Override
    protected void outgoing_send(TitanCharString send_par) {
        // TODO Auto-generated method stub
    }
}
```

9.3. Procedure-based Example

The definition of `MyProcedurePort` in module `MyModule`:

```
type port MyProcedurePort procedure
{
  in inProc;
  out outProc;
  inout inoutProc;
};
```

The signature definitions are imported from a module called `MyModule2`, `noblock` is not used and exceptions are used so that every member function of the port class is generated for this example. If the keyword `noblock` is used the compiler will optimize code generation by not generating outgoing reply, incoming reply member functions and their argument types. If the signature has no exception outgoing raise, incoming exception member functions and related types will not be generated.

The port type `MyProcedurePort` can handle `call`, `getreply` and `catch` operations referencing the signatures `outProc` and `inoutProc`, and it can handle `getcall`, `reply` and `raise` operations referencing the signatures `inProc` and `inoutProc`.

The initial Test Port file (that is, `MyProcedurePort.java`) will look as follows:

```
package org.eclipse.titan.MyProject.user_provided;

import org.eclipse.titan.MyProject.generated.MyModule.MyProcedurePort_BASE;
import org.eclipse.titan.MyProject.generated.MyModule2.inProc_reply;
import org.eclipse.titan.MyProject.generated.MyModule2.inoutProc_call;
import org.eclipse.titan.MyProject.generated.MyModule2.inoutProc_reply;
import org.eclipse.titan.MyProject.generated.MyModule2.outProc_call;

public class MyProcedurePort extends MyProcedurePort_BASE {

    public MyProcedurePort(final String name) {
        super(name);
    }

    @Override
    public void outgoing_call(outProc_call call_par) {
        // TODO Auto-generated method stub
    }

    @Override
    public void outgoing_call(inoutProc_call call_par) {
        // TODO Auto-generated method stub
    }

    @Override
    public void outgoing_reply(inProc_reply reply_par) {
        // TODO Auto-generated method stub
    }

    @Override
    public void outgoing_reply(inoutProc_reply reply_par) {
        // TODO Auto-generated method stub
    }
}
```

9.4. Test Port Functions

This section summarizes all possible member functions of the Test Port class. These functions have an empty implementation in the base class of the Test Port.

The identical functions of both port types are:

- the constructor
- the parameter setting function
- the map and unmap function
- the start and stop function
- descriptor event and timeout handler(s)

- some additional functions and attributes

The functions above will be described using an example of message based ports (`MyMessagePort`, also introducing the functions specific to message based port types). Using these functions is identical (or very similar) in procedure based Test Ports.

Functions specific to message based ports:

- send functions: outgoing send
- incoming functions: incoming message
- Functions specific to procedure based ports:
- outgoing functions: outgoing_call, outgoing_reply, outgoing_raise
- incoming functions: incoming_call, incoming_reply, incoming_exception

Both test port types can use the same logging and error handling mechanism, and the handling of incoming operations on port `MyProcedurePort` is similar to receiving messages on port `MyMessagePort` (regarding the event handler).

NOTE	The easiest way to discover what functions can be overwritten and to generate their skeleton is by using the earlier described `Override\Implement Methods···` functionality of eclipse. That functionality automatically list all functions from the class generated for the given testport and the its parent classes, that can be overwritten.
NOTE	Please note, that in Java functions by default inherit the documentation/comments from the function they overwrite. So while the functions just inserted to overwrite functions from the base class might not appear to have a comment, in eclipse moving the cursor over their name will reveal their actual comment.

9.4.1. Constructor

NOTE	On the Java side Test Ports do not have destructors.

The Test Port class belongs to a TTCN–3 port type, and its instances implement the functions of the port instances. That is, each Test Port instance belongs to the port of a TTCN–3 test component. The number of TTCN–3 component types, port types and port instances is not limited; you may have several Test Port classes and several instances of a given Test Port class in one test suite.

The Test Port instances are global and static objects from the point of view of the Java code. This means, their constructor is called before the test execution (that is, before the main function starts). They are also stored as threadlocal to be only accessible by the thread (Parallel Test Component) they belong to. The name of a Test Port object is composed of the name of the corresponding component type and the name of the port instance within the component type.

In case of parallel test execution, each TTCN–3 test component thread has its own Test Port instances. Of course, only the Test Ports of the active component type are used, the member functions of other inactive Test Port instances (except constructor) shall never be called. All Test Port instances should be handled as being static, their constructor is called only once, at the time

their component is created. The test component threads (that is, the child threads of Host Controller) will have to create/initialize their own Test Port instances.

The Test Port class is derived from an abstract base class which can be found in the generated code. The base class implements, for instance, the queue of incoming messages.

The constructor takes one parameter containing the name of the port instance in a String. This string shall be passed further to the constructor of the base class as it can be found in the skeleton code. The default argument for the test port name is a null pointer, which is used when the test port object is a member of a port array.

WARNING	In case of port arrays the name of the test port is set after the constructor is completed. So the name of the test port should not be used in the constructor. The port name is always set correctly when any other member function is called.

9.4.2. Parameter Setting Function

Test Port parameters shall contain information which is independent from the TTCN-3 test suite. These values shall not be used in the test suite at all. You can define them as TTCN–3 constants or module parameters, but these definitions are useless and redundant, and they must always be present when the Test Port is used.

For instance, using Test Port parameters can be used to convey configuration data (that is, some options or extra information that is necessary for correct operation) or lower protocol layer addresses (for example, IP addresses).

Test Port parameters shall be specified by the user of executable tests in the `[TESTPORT_PARAMETERS]` section of the run-time configuration file (see section `[TESTPORT_PARAMETERS]` in Programmer's Technical Reference). The parameters are maintained for each test port instance separately; wildcards can be used as well. In the latter case the parameter is passed to all Test Port matching the wildcard.

Each Test Port parameter must have a name, which must be unique within the Test Port only. The name must be a valid identifier, that is, it must begin with a letter and must contain alphanumerical characters only.

All Test Port parameter values are interpreted by the test executor as character strings. Quotation marks must be used when specifying the parameter values in the configuration file. The interpretation of parameter values is up to you: you can use some of them as symbolic values, numbers, IP addresses or anything that you want.

Before the test execution begins, all parameters belonging to the Test Port are passed to the Test Port by the runtime environment of the test executor using the function `set_parameter`. The default implementation of this function does nothing and ignores all parameters.

Each parameter is passed to the Test Port one-by-one separately[3], the two arguments of `set_parameter` contain the name and value of the corresponding parameter, respectively, in Strings.

It is warmly recommended that the Test Port parameter handling functions be fool-proof. For

instance, the Test Port should produce a proper error message (for example by calling `TtcnError`) if a mandatory parameter is missing instead of causing unreliable behavior later. Repeated setting of the same parameter should produce warnings for the user (for example by using the function `TtcnError.TtcnWarning`) and not memory leaks.

NOTE

On the MTC, in both single and parallel modes, the handling of Test Port parameters is a bit different from that on PTCs. The parameters are passed only to active ports, but the component type of MTC (thus the set of active ports) depends on the `runs on` clause of the test case that is currently being executed. It would be difficult for the runtime environment to check at the beginning of each test case whether the corresponding MTC component type has already been active during a previous test case run. Therefore all Test Port parameters belonging to the active ports of the MTC are passed to the `set_parameter` function at the beginning of every test case. The Test Ports of MTC shall be prepared to receive the same parameters several times (with the same values, of course) if more than one test case is being executed.

If system related Test Port parameters are used in the run-time configuration file (that is, the keyword `system` is used as component identifier), the parameters are passed to your Test Port during the execution of TTCN–3 `map` operations, but before calling your `user_map` function. Please note that in this case the port identifier of the configuration file refers to the port of the test system interface that your port is mapped to and not the name of your TTCN–3 port.

The name and exact meaning of all supported parameters must be specified in the user documentation of the Test Port.

9.4.3. Map and Unmap Functions

The run-time environment of the TTCN–3 executor knows nothing about the communication towards SUT, thus, it is the user's responsibility to establish and terminate the connection with SUT. The TTCN–3 language uses two operations to control these connections, `map` and `unmap`.

For this purpose, the Test Port class provides two member functions, `user_map` and `user_unmap`. These functions are called by the test executor environment when performing TTCN–3 `map` and `unmap` operations, respectively.

The `map` and `unmap` operations take two pairs of component references and ports as arguments. These operations are correct only if one of the arguments refer to a port of a TTCN–3 test component while the other port corresponds to SUT. This aspect of correctness is verified by the run-time environment, but the existence of a system port is not checked.

The port names of the system are converted to Strings and passed to functions `user_map` and `user_unmap` as parameters. Unlike other identifiers, the underscore characters in these port names are not translated.

NOTE	in TTCN–3 it is not allowed to map a test component port to several system ports at the same time. The run-time environment, however, is not so strict and allows this to handle transient states during configuration changes. In this case messages can not be sent to SUT even with explicit addressing, but the reception of messages is permitted. When putting messages into the input queue of the port, it is not important for the test executor (even for the TTCN–3 language) which port of the system the message is received from.

The execution of TTCN–3 test component that requested the mapping or unmapping is suspended until your `user_map` or `user_unmap` functions finish. Therefore it is not allowed to block unnecessarily the test execution within these functions.

When the Test Port detects an error situation during the establishment or termination of the physical connection towards the SUT, the function `TTCN_error` shall be used to indicate the failure. If the error occurs within `user_map` the run-time environment will assume that the connection with SUT is not established thus it will not call `user_unmap` to destroy the mapping during the error recovery procedure. If `user_map` fails, it is the Test Port writer's responsibility to release all allocated resources and bring the object variables into a stable state before calling `TtcnError`. Within `user_unmap` the errors should be handled in a more robust way. After a minor failure it is better to issue a warning and continue the connection termination instead of panicking. `TtcnError` shall be called only to indicate critical errors. If `user_unmap` is interrupted with an error the run-time environment assumes that the mapping has been terminated, that is, `user_unmap` will not be called again.

NOTE	if either `user_map` or `user_unmap` fails, the error is indicated on the initiator test component as well; that is, the respective map or `unmap` operation will also fail and error recovery procedure will start on that component.

Parameters of the Map and Unmap Functions

Parameters can be sent to the `user_map` and `user_unmap` functions from TTCN-3 code using the `param` clause of the `map` and `unmap` operations.

The `user_map` and `user_unmap` functions have a parameter of type `Map_Params`, which contains the string representations of the `in` and `inout` parameters of the `map`/`unmap` operation. The string representations of `out` parameters are empty strings (as these are considered as being `unbound` at the beginning of the `map`/`unmap` operation). After the `user_map` or `user_unmap` function ends and the mapping/unmapping is concluded, the final values (string representations) of `out` and `inout` parameters in the `Map_Params` object are sent back to the mapping/unmapping requestor.

The following member functions can be used to obtain or set data in the `Map_Params` object:

```
public int get_nof_params()
```

Returns the number of parameters in the object. This will either be zero (if the `map` or `unmap` operation had no `param` clause) or the number of parameters specified in the system port type definition's `map param` or `unmap param` clause.

```
public TitanCharString get_param(final int index)
```

Returns the string representation of the parameter at index `p_index`. This method shall be used to retrieve the values of `in` and `inout` parameters. The parameter indices start at 0. The order of the parameters is the same as their order of declaration. Default values of parameters are automatically set by the runtime environment before the `user_map`/`user_unmap` call. The string representations retrieved with this function can be converted back to the parameter's TTCN-3 type with the predefined function `string_to_ttcn`.

```
public void set_param(final int index, final TitanCharString param)
```

Sets the string representation of the parameter at index `p_index` to the string `p_param`. This method shall be used to set the final values of `out` and `inout` parameters. The string representation of a TTCN-3 value can be obtained using the predefined function `ttcn_to_string`. If the final value of an `out` or `inout` parameter is an empty string, then the variable used as parameter will remain unchanged. Otherwise its new value will be calculated by applying `string_to_ttcn` on the string value set in the `user_map` or `user_unmap` function (this could cause dynamic test case errors if the string representation is invalid).

Usage example:

Port type:

```
type port MyPort message {
  ...
  map param(in MyInParType in_par, inout MyInOutParType inout_par, out MyOutParType
out_par)
}
```

`user_map` function in port implementation:

```
@Override
protected void user_map(final String system_port, final Map_Params params) {
  if (params.get_nof_params() == 0) {
    // there were no map parameters

    // do mapping
    ...
  } else {
    // there were map parameters

    // extract 'in' and 'out' parameters
    MyInParType in_par = new MyInParType();
    TitanCharString.string_to_ttcn(params.get_param(0), in_par);
    MyInOutParType inout_par = new MyInOutParType();
    TitanCharString.string_to_ttcn(params.get_param(1), inout_par);
    MyOutParType out_par = new MyOutParType(); // remains unbound

    // do mapping
    ...

    // update 'out' and 'inout' parameters
    params.set_param(1, TitanCharString.ttcn_to_string(inout_par));
    params.set_param(2, TitanCharString.ttcn_to_string(out_par));
  }
}
```

9.4.4. Start and Stop Functions

The Test Port class has two member functions: `user_start` and `user_stop`. These functions are called when executing `port start` and `port stop` operations, respectively. The functions have no parameters and return types.

These functions are called through a stub in the base class, which registers the current state of the port (whether it is started or not). So `user_start` will never be called twice without calling `user_stop` or vice versa.

All ports of test components are started implicitly immediately after creation. Operations put in a `user_start` function must not be blocking the execution for a longer period. This not only hangs the new PTC but the also component that performed the `create` operation (usually the MTC). All ports are stopped at the end of test cases or at PTC termination, even if `stop` statements are missing.

In functions `user_start` and `user_stop` the device should be initialized or shut down towards SUT (that is, the communications socket). Also the event handler should be installed or uninstalled (see later).

9.4.5. Outgoing Operations

Outgoing operations are `send` (specific to message based ports); `call`, `reply`, and `raise` (specific to procedure based ports).

Send Functions

The Test Port class has an overloaded function called `outgoing_send` for each outgoing message type. This function will be called when a message is sent on the port and it should be routed to the system (that is, SUT) according to the addressing semantics[4] of TTCN–3. The messages (implicitly or explicitly) addressed to other test components are handled inside the test executor; the Test Ports have nothing to do with them. The function `outgoing_send` will be also called if the port has neither connections nor mappings, but a message is sent on it.

The only parameter of `outgoing_send` contains a read-only reference to the message in the internal data representation format of the test executor. The access methods for internal data types are described in Predefined TTCN–3 Data Types. The test port writer should encode and send the message towards SUT. For information on how to use the standard encoding functions like RAW, please consult the earlier chapters of this document. Sending a message on a not started port causes a dynamic test case error. In this case `outgoing_send` will not be called.

Call, Reply and Raise Functions

The procedure based Test Port class has overloaded functions called `outgoing_call`, `outgoing_reply` and `outgoing_raise` for each `call`, `reply` and `raise` operations, respectively. One of these functions will be called when a port-operation is addressing the system (that is, SUT using the to `system` statement).

The only parameter of these functions is an internal representation of the signature parameters (and possibly its return value) or the exceptions it may raise. The signature classes are described in Using the Signature Classes.

9.4.6. Incoming Operations

Incoming operations are `receive` for incoming messages (specific to message based ports); `call`, `reply` and `raise` for signatures (specific to procedure based ports).

Descriptor Event and Timeout Handlers

The handling of incoming messages (or operations) is more difficult than sending. The executable test program has two states. In the first state, it executes the operations one by one as specified in the test suite (for example, it evaluates expressions, calls functions, sends messages, etc.). In the other state it waits for the response from SUT or for a timer to expire. This happens when the execution reaches a blocking statement, that is, one of a stand-alone `receive`, `done`, `timeout` statements or an `alt` construct.

After reaching a blocking statement, the test executor evaluates the current snapshot of its timer and port queues and tries to match it with the reached statements and templates. If the matching fails, the executor sleeps until something happens to its timers or ports. After waking up, it re-evaluates its snapshot and tries to match it again. The last two steps are repeated until the executor finds the first matching statement. If the test executor realizes that its snapshot can never match the reached TTCN–3 statements, it causes a dynamic test case error. This mechanism prevents it from infinite blocking.

The test executor handles its timers itself, but it does not know anything about the communication

with SUT. So each Test Port instance should inform the snapshot handler of the executor what kind of event the Test Port is waiting for. The event can be either the reception of data on one or more socket channels or a timeout (when polling is used) or both of them.

When the test executor reaches a blocking statement and any condition – for which the Test Port waits – is fulfilled, the event handler will be called. First one has to get the incoming message or operation from the operating system. After that, one has to decode it (and possibly decide its type). Finally, if the internal data structure is built, one has to put it into the queue of the port. This can be done using the member function `incoming_message` if it is a message, and using `incoming_call`, `incoming_reply` or `incoming_exception` if it is an operation.

The execution must not be blocked in event handler functions; these must return immediately when the message or operation processing is ready. In other words, always use non-blocking calls. In the case when the messages are fragmented (for instance, when testing TCP based application layer protocols, such as HTTP), intermediate buffering should be performed in the Test Port class.

Event and timeout handling interface

To be notified about available events the `Handle_Event` function has to be implemented.

```
public void Handle_Event(final SelectableChannel channel, final boolean is_readable,
final boolean is_writeable);
```

Using `Handle_Event` allows receiving all events of a descriptor in one function call.

The first parameter in all of these functions is the selectable channel. The second is true if the channel is readable. The third is true if it is writeable.

You can install or uninstall the event handler by calling the following inherited member functions:

```
protected void Install_Handler(final Set<SelectableChannel> read_channels, final
Set<SelectableChannel> write_channels, final double call_interval) throws IOException;
protected void Uninstall_Handler() throws IOException;
```

`Install_Handler` installs the event handler according to its parameters. It takes three arguments, two sets of SelectableChannels and a timeout value. Some of the parameters can be ignored, but ignoring all at the same time is not permitted.

`read_channels` is the set of SelectabeChannel to register the handler for reading. If null the handler is not registered for any channel to handle reading. `write_channels` is the set of SelectabeChannel to register the handler for writing. If null the handler is not registered for any channel to handle writing.

The call interval value is measured in seconds. It means that the event handler function will be called when the time elapsed since its last call reaches the given value. This parameter is ignored when its value is set to zero or negative.

If you want to change your event handling parameters, you may simply call the function

`Install_Handler` again (calling of `Uninstall_Handler` is not necessary).

`Uninstall_Handler` will uninstall your previously installed event handler. The `stop` port operation also uninstalls the event handler automatically. The event handler may be installed or uninstalled in any Test Port member function, even in the event handler itself.

Receiving messages

The member function `incoming_message` of message based ports can be used to put an incoming message in the queue of the port. There are different functions for each incoming message type. These functions are inherited from the base class. The received messages are logged when they are put into the queue and not when they are processed by the test suite[5].

In our example the class `MyMessagePort_BASE` has the following member functions:

```
protected void incoming_message(final TitanOctetString incoming_par);
protected void incoming_message(final TitanCharString incoming_par);
```

Receiving calls, replies and exceptions

Receiving operations on procedure based ports is similar to receiving messages on message based ports. The difference is that there are different overloaded incoming functions for call, reply and raise operations called `incoming_call`, `incoming_reply` and `incoming_exception`, respectively. The event handler (when called) must recognize the type of operation on receiving and call one of these functions accordingly with one of the internal representations of the signature (see Additional Non-Standard Functions).

In the example[6] the class `MyProcedurePort_BASE` has the following member functions for incoming operations:

```
protected void incoming_call(final MyModule2.inProc_call incoming_par);
protected void incoming_call(final MyModule2.inoutProc_call incoming_par);
protected void incoming_reply(final MyModule2.outProc_reply incoming_par);
protected void incoming_reply(final MyModule2.inoutProc_reply incoming_par);
protected void incoming_exception(final MyModule2.outProc_exception incoming_par);
protected void incoming_exception(final MyModule2.inoutProc_exception incoming_par);
```

For example, if the event handler receives a call operation that refers to the signature called `inoutProc`, it has to fill the parameters of an instance of the class `inoutProc_call` with the received data. Then it has to call the function `incoming_call` with this object to place the operation into the queue of the port.

The following table shows the relation between the direction of the message type or signature in the port type definition and the incoming/outgoing functions that can be used. `MyPort` in the table header refers to `MyMessagePort` or `MyProcedurePort` in the example depending on the type of the port (message based or procedure based).

Table 1. Outgoing and incoming operations

		MyPort.outgoing_				MyPort_BASE.incoming_			
		send	call	reply	raise	message	call	reply	exception
message type	in	○	○	○	○	●	○	○	○
	out	●	○	○	○	○	○	○	○
	inout	●	○	○	○	●	○	○	○
signature	in	○	○	●	●	○	●	○	○
	out	○	●	○	○	○	○	●	●
	inout	○	●	●	●	○	●	●	●

● supported

○ not supported

9.4.7. Additional Functions and Attributes

Any kind of attributes or member functions may be added to the Test Port. A selectable channel, which you communicate on, is almost always necessary. Names not interfering with the identifiers generated by the Java code generator can be used in the java file (for example, the names containing one underscore character). Avoid using static variables because it can be very confusing when more than one instances of the Test Port run simultaneously. Any kind of software libraries may be used in the Test Port as well.

In addition, the following `protected` attributes of ancestor classes are available:

Table 2. Protected attributes

Name	Type	Meaning
`is_active`	boolean	Indicates whether the Test Port is active.
`is_started`	boolean	Indicates whether the Test Port is started.
`is_halted`	boolean	Indicates whether the Test Port is halted.
`port_name`	String	Contains the name of the Test Port instance.

Underscore characters are not duplicated in port_name. In case of port array member instances the name string looks like this: `"Myport_array[5]"`.

9.5. Support of `address` Type

The special user-defined TTCN–3 type `address` can be used for addressing entities inside the SUT on ports mapped to the `system` component. Since the majority of Test Ports does not need TTCN–3 addressing and in order to keep the Test Port API backward compatible the support of `address` type

is disabled by default. To enable addressing on a particular port type the extension attribute `"address"` must be added to the TTCN–3 port type definition. In addition to component references this extension will allow the usage of `address` values or variables in the `to` or `from` clauses and `sender` redirects of port operations.

In order to use addressing, a type named `address` shall be defined in the same TTCN–3 module as the corresponding port type. Address types defined in other modules of the test suite do not affect the operation of the port type. It is possible to link several Test Ports that use different types for addressing SUT into the same executable test suite.

Test Ports that support SUT addressing have a slightly different API, which is considered when generating Test Port skeleton. This section summarizes only the differences from the normal API.

In the communication operations the test port author is responsible for handling the address information associated with the message or the operation. In case of an incoming message or operation the value of the received address will be stored in the port queue together with the received message or operation.

The generated code for the port skeleton of message based ports will be the same, except `outgoing_send` member function, which has an extra parameter pointing to an `TitanAddress` value. With the example given in Test Port Functions:

```
void outgoing_send(final TitanInteger send_par, final TitanAddress
destination_address);
void outgoing_send(final TitanCharString send_par, final TitanAddress
destination_address);
```

NOTE	when the type named `address` is defined as a synonym of an other type, these functions could also report that type to be the type of the `destination_address` formal parameter.

If an `address` value was specified in the `to` clause of the corresponding TTCN–3 `send` operation the second argument of `outgoing_send` points to that value. Otherwise it is set to the `NULL` pointer. The Test Port code shall be prepared to handle both cases.

The outgoing operations of procedure based ports are also generated in the same way if the `address` extension is specified. These functions will also have an extra parameter. Based on our example, these will have the following form:

```
void outgoing_call(final MyModule2.outProc_call call_par, final TitanAddress
destination_address);
void outgoing_call(final MyModule2.inoutProc_call call_par, final TitanAddress
destination_address);
void outgoing_reply(final MyModule2.inProc_reply reply_par, final TitanAddress
destination_address);
void outgoing_reply(final MyModule2.inoutProc_reply reply_par, final TitanAddress
destination_address);
void outgoing_raise(final MyModule2.inProc_exception raise_exception, final
TitanAddress destination_address);
void outgoing_raise(final MyModule2.inoutProc_exception raise_exception, final
TitanAddress destination_address);
```

The other difference is in the `incoming_message` member function of class `MyMessagePort_BASE`, and in the incoming member functions of class `MyProcedurePort_BASE`. These have an extra parameter, which is a pointer to an `TitanAddress` value. The version of the function that does not have this formal parameter, will call this function with a null value passed as the sender_address. In our example of `MyMessagePort_BASE`:

```
void incoming_call(final MyModule2.inProc_call incoming_par, final int
sender_component, final TitanAddress sender_address);
void incoming_call(final MyModule2.inoutProc_call incoming_par, final int
sender_component, final TitanAddress sender_address);
void incoming_reply(final MyModule2.outProc_reply incoming_par, final int
sender_component, final TitanAddress sender_address)
void incoming_reply(final MyModule2.inoutProc_reply incoming_par, final int
sender_component, final TitanAddress sender_address)
void incoming_exception(final MyModule2.outProc_exception incoming_par, final int
sender_component, final TitanAddress sender_address)
void incoming_exception(final MyModule2.inoutProc_exception incoming_par, final int
sender_component, final TitanAddress sender_address)
```

If the event handler of the Test Port can determine the source address where the message or the operation is coming from, it shall pass a pointer to the incoming function, which points to a variable that stores the `address` value. The given address value is not modified by the run-time environment and a copy of it is created when the message or the operation is appended to the port queue. If the event handler is unable to determine the sender address the default null value shall be passed as the argument.

The address value stored in the port queue is used in `receive`, `trigger`, `getcall`, `getreply`, `catch` and `check` port operations: it is matched with the `from` clause and/or stored into the variable given in the `sender` redirect. If the receiving operation wants to use the address information of the first element in the port queue, but the Test Port has not supplied it a dynamic testcase error will occur.

9.6. Provider Port Types

Test Ports that belong to port types marked with `extension` attribute `"provider"` have a slightly

different API. Such port types are used to realize dual-faced ports, the details of which can be found in section "Dual-faced ports" in the Programmer's Technical Reference.

The purpose of this API is to allow the re-use of the Test Port class with other port types marked with attribute `user` or with ports with translation capability (Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3 Language Extensions: Configuration and Deployment Support). The user port types may have different lists of incoming and outgoing message types. The transformations between incoming and outgoing messages, which are specified entirely by the attribute of the user port type, are done independently of the Test Port. The Test Port needs to support the sending and reception of message types that are listed in the provider port type.

The provider port can be accessed through the port which maps to the port with provider attribute. The `get_provider_port()` is a member function of the TitanPort class:

```
TitanPort get_provider_port();
```

This function is useful when a reference to the provider type is needed. It returns the provider port type for user ports and ports with translation capability. Otherwise returns null. The function causes dynamic testcase error when the port has more than one mapping, or the port has both mappings and connections. The function's return value must be manually cast to the correct provider port type.

This section summarizes only the differences from the normal Test Port API:

- The name of the Test Port class is suffixed with the string `_PROVIDER` (for example `MyMessagePort_PROVIDER` instead of `MyMessagePort`).
- The base class of the Test Port is class `TitanPort`, which is part of the Base Library. Please note that normal Test Ports are also derived from class TitanPort, but indirectly through an intermediate class with suffix `_BASE`.
- The member functions that handle incoming messages and procedure-based operations (that is `incoming_message`, `incoming_call`, `incoming_reply` and `incoming_exception`) must be defined as override-able functions. These functions will be implemented in various descendant classes differently.
- The member functions of the Test Port may refer to Java classes that are generated from user-defined message types and signatures.

The following example shows the skeleton of a provider port type Test Port.

Port type definition in TTCN–3 :

```
type port MyProviderPort mixed {
  inout MyMessage, MySignature;
} with { extension "provider" }
```

Source file `MyProviderPort_PROVIDER.java`:

```
package org.eclipse.titan.MyProject.user_provided;

import java.nio.channels.SelectableChannel;

import org.eclipse.titan.MyProject.generated.MyModule.MyMessage;
import org.eclipse.titan.MyProject.generated.MyModule.MySignature_call;
import org.eclipse.titan.MyProject.generated.MyModule.MySignature_exception;
import org.eclipse.titan.MyProject.generated.MyModule.MySignature_reply;
import org.eclipse.titan.runtime.core.TitanPort;

public class MyProviderPort_PROVIDER extends TitanPort {

    public MyProviderPort_PROVIDER() {
        super();
    }

    public MyProviderPort_PROVIDER(final String name) {
        super(name);
    }

    @Override
    public void set_parameter(final String parameter_name, final String
parameter_value) {
    }

    @Override
    public void Handle_Event(final SelectableChannel channel, final boolean
is_readable,
            final boolean is_writeable) {
    }

    @Override
    protected void user_map(final String system_port, final Map_Params params) {
    }

    @Override
    protected void user_unmap(final String system_port, final Map_Params params) {
    }

    @Override
    protected void user_start() {
    }

    @Override
    protected void user_stop() {
    }

    public void outgoing_send(final MyMessage send_par) {
    }
    public void outgoing_call(final MySignature_call call_par) {
    }
```

```
    public void outgoing_reply(final MySignature_reply reply_par) {
    }
    public void outgoing_raise(final MySignature_exception raise_Exception) {
    }
}
```

9.7. Tips and Tricks

The following sections deal with logging and error handling in Test Ports.

9.7.1. Logging

Test Ports may record important events in the Test Executor log during sending/receiving or encoding/decoding messages. Such log messages are also good for debugging fresh code.

The Test Port member functions may call the functions of class `TTCN_Logger`. These functions are detailed in Logging in Test Ports or External Functions.

If there are many points in the Test Port code that want to log something, it can be a good practice to write a common log function in the Test Port class. We show here an example where the calling of `log` uses Java's MessageFormat.format to create a custom message, and inside the `log` function TTCN_Logger.log_event demonstrates logging using the standard C function `printf` style and forwards the message to the Test Executor's logger:

```
private void value_logging(final TitanInteger i) {
  log(MessageFormat.format("The value of i : {0}.", i.get_int()));
}

private void log(final String content) {
  TTCN_Logger.begin_event(Severity.DEBUG_USER);
  TTCN_Logger.log_event("Example Test Port (%s): ", get_name());
  TTCN_Logger.log_event_str(content);
  TTCN_Logger.end_event();
}
```

9.7.2. Error Handling

None of the Test Port member functions have return value like a status code. If a function returns normally, the run-time environment assumes that it has performed its task successfully. The handling of run-time errors is done using Java exceptions. This simplifies the program code because the return values do not have to be checked everywhere and dynamically created complex error messages can be used if necessary.

If any kind of fatal error is encountered anywhere in the Test Port, an exception of type `TtcnError` should be thrown:

```
throw new TtcnError(errorMessage);
```

Its parameter should contain the description of the error in a String. The exception is usually caught at the end of the test case or PTC function that is being executed. In case of error, the verdict of the component is set to `error` and the execution of the test case or PTC function terminates immediately.

The error string is written into the log file by `TtcnError` immediately. Such type of exception should never be caught or thrown directly. If you want to implement your own error handling and error recovery routines you had better use your own classes as exceptions.

If you write your own error reporting function you can add automatically the name of the port instance to all of your error messages. This makes the fault analysis for the end-users easier. In the following example the error message will occupy two consecutive lines in the log since we can pass only one format string to `TtcnError`.

```
private void error(final String content) {
        TTCN_Logger.begin_event(Severity.ERROR_UNQUALIFIED);
        TTCN_Logger.log_event("Example Test Port (%s): ", get_name());
        TTCN_Logger.log_event_str(content);
        TTCN_Logger.end_event();
        throw new TtcnError(MessageFormat.format("Fatal error in Example Test Port {0}
(see above).", get_name()));
    }
```

There is another function for denoting warnings (that is, events that are not so critical) with the same parameter list as TtcnError:

```
void TtcnError.TtcnWarning(warningMessage);
```

This function puts an entry in the executor's log with severity `TTCN_WARNING`. In contrast to `TtcnError`, after logging the given message `TtcnWarning` returns and your test port can continue running.

9.8. Setting timestamps

In order to use the timestamp redirects (`→ timestamp`) described in chapter 5 of the TTCN-3 standard extension `TTCN-3 Performance and Real Time Testing` (ETSI ES 202 782 V1.3.1, [16]) the test port writer needs to add extra code to set the timestamps for the incoming and outgoing port operations of each port with the `realtime` clause.

9.8.1. Incoming operations

The timestamps of incoming port operations (`receive`, `trigger`, `getcall`, `getreply`, `catch` and `check`) need to be set when the incoming message or procedure is added to the queue.

The member functions `incoming_message`, `incoming_call`, `incoming_reply` and `incoming_exception`

(which add the message/procedure to the queue) have an optional `TitanFloat` parameter called `timestamp`, if the test port was declared with the `realtime` clause.

The value given to this parameter will be the one stored in the variable referenced in the timestamp redirect, if the operation has a timestamp redirect (otherwise the value is ignored).

It is recommended that this parameter be set to the current test system time, which can be queried with `TTCN_Runtime.now();`, or to a float variable that was set to the current test system time earlier in the function.

Examples:

```
incoming_message(my_message, TTCN_Runtime.now());
```

```
TitanFloat reply_time = TTCN_Runtime.now();

...

incoming_reply(my_reply, reply_time);
```

9.8.2. Outgoing operations

The timestamps of outgoing port operations (`send`, `call`, `reply`, `raise`) need to be set in the member functions `outgoing_send`, `outgoing_call`, `outgoing_reply` and `outgoing_raise`.

These functions have a `TitanFloat` pointer parameter called `timestamp_redirect`, if the test port was declared with the `realtime` clause.

The value pointed to by this parameter will be the one stored in the variable referenced in the timestamp redirect, if the operation has a timestamp redirect.

If it does not have a timestamp redirect, then this value parameter will be null. Because of this, the parameter must always have a null check before it is assigned a value.

It is recommended that the value pointed to by the parameter be set to the current test system time, which can be queried with `TTCN_Runtime.now()`.

Example:

```
if (timestamp_redirect != null) {
    timestamp_redirect.operator_assign(TTCN_Runtime.now());
}
```

NOTE

Because of this extra parameter, adding or removing the `realtime` clause from a port will cause already-written Java code for the port to no longer compile. In these cases the new parameters must be manually added or removed from the mentioned functions.

[2] The test equipment not necessarily requires a special hardware; it can even be a simple PC with an Ethernet interface.

[3] If the same parameter of the same port instance is specified several times in the configuration file, the function `set_parameter` will also be called several times.

[4] That is, the port has exactly one mapping and either the port has no connections or the message is explicitly addressed by a `send (⋯) to system` statement.

[5] Note that if the port has connections as well, the messages coming from other test components will also be inserted into the same queue independently from the event handler.

[6] In the example the signatures were defined in a different TTCN–3 module named MyModule2, as a consequence all types defined in that module must be prefixed with the Java name of that module and its class be imported.

Chapter 10. Logger Plug-ins

The Logger Plug-ins feature is not yet supported on the Java side.

Chapter 11. Encoding and Decoding

TITAN is equipped with several standard encoding/decoding mechanisms. A part of these functions reside in the core library, but the type-dependent part must be generated by the Java code generator. In order to reduce the code size and compilation time, the code generation for encoding functions (separately for different encoders) can be switched off if they are not needed as described in Build Options.

To make it easier to use the encoding features, a unified common API was developed. With help of this API the behaviour of the test executor in different error situations can be set during coding. There is also a common buffer class. The details of the above mentioned API as well as the specific features of the certain encoders are explained in the following sections.

11.1. The Common API

The common API for encoders consists of three main parts:

- A dummy class named `TTCN_EncDec` which encapsulates functions regarding error handling.
- A buffer class named `TTCN_Buffer` which is used by the encoders to put data in, decoders to get data from.
- The functions needed to encode and decode values.

11.1.1. TTCN_EncDec

`TTCN_EncDec` implements error handling functions.

Setting Error Behavior

There are lot of error situations during encoding and decoding. The coding functions can be told what to do if an error arises. To set the behaviour of test executor in a certain error situation the following function is to be invoked from the `TTCN_EncDec` class:

```
static void set_error_behavior(final error_type p_et, final error_behavior_type p_eb);
```

WARNING	As error_type and error_behavior_type are enums defined in TTCN_EncDec class, they have to prefixed with the class name (that is `TTCN_EncDec.`). An example usage:

```
TTCN_EncDec.set_error_behavior(TTCN_EncDec.error_type.ET_ALL,
TTCN_EncDec.error_behavior_type.EB_DEFAULT);
```

The possible values of `error_type` are detailed in the sections describing the different codings. Some common error types are shown in the table below:

Table 3. Common error types

ET_UNDEF	**Undefined/unknown error.**
ET_UNBOUND	Encoding of an unbound value.
ET_REPR	Representation error (for example, internal representation of integral numbers).
ET_ENC_ENUM	Encoding of an unknown enumerated value.
ET_DEC_ENUM	Decoding of an unknown enumerated value.
ET_INCOMPL_MSG	Decode error: incomplete message.
ET_INVAL MSG	Decode error: invalid message.
ET_CONSTRAINT	The value breaks some constraint.
ET_INTERNAL	Internal error. Error behaviour cannot be set for this.
ET_ALL	All error type. Usable only when setting error behaviour.
ET_NONE	No error.

The possible values of `error_behavior_type` are shown in the table below:

Table 4. Possible values of `error_behavior_t`

EB_DEFAULT	Sets the default error behaviour for the selected error type.
EB_ERROR	Raises an error if the selected error type occurs.
EB_WARNING	Gives a warning message but tries to continue the operation.
EB_IGNORE	Like warning but without the message.

Getting Error Behavior

There are two functions: one for getting the current setting and one for getting the default setting for a particular error situation.

```
static error_behavior_type get_error_behavior(final error_type p_et)
static error_behavior_type get_default_error_behavior(final error_type p_et)
```

The using of these functions are straightforward: giving a particular `error_type` the function returns the current or default `error_behavior_type` for that error situation, respectively.

Checking if an Error Occurred

The last coding-related error and its textual description can be retrieved anytime. Before using a coding function, it is advisable to clear the "last error". This can be achieved by the following method:

```
static void clear_error();
```

After using some coding functions, it can be checked if an error occurred with this function:

```
static error_type get_last_error_type();
```

This returns the last error, or `ET_NONE` if there was no error. The string representation of the error can be requested with the help of this:

```
static String get_error_str();
```

WARNING	The above two functions do not clear the "last error" flag.

11.1.2. TTCN_Buffer

TTCN Buffer objects are used to store encoded values and to communicate with the coding functions. If encoding a value, the result will be put in a buffer, from which can be get. In the other hand, to decode a value, the encoded octet string must be put in a TTCN_Buffer object, and the decoding functions get their input from that.

```
void clear();
```

Resets the buffer, cleaning up its content, setting the pointers to the beginning of buffer.

```
void rewind();
```

Rewinds the buffer, that is, sets its reading pointer to the beginning of the buffer.

```
int get_pos();
```

Returns the (reading) position of the buffer.

```
void set_pos(final int new_pos);
```

Sets the (reading) position to pos, or to the end of buffer, if `pos > get_len()`.

```
int get_len();
```

Returns the amount of bytes in the buffer.

```
char[] get_data();
```

Returns a copy of the buffer starting from its start. You can read out `count` bytes beginning from this address, where `count` is the value returned by the `get_len()` member function.

```
int get_read_len();
```

Returns how many bytes are in the buffer to read.

```
char[] get_read_data();
```

Returns a copy of the buffer starting from the read position of data. `count` bytes can be read out beginning from this address, where count is the value returned by the `get_read_len()` member function.

```
void put_c(final char c);
```

Appends the byte `c` to the end of buffer.

```
void put_s(final char[] cstr);
```

Writes a string of bytes to the end of buffer.

```
void put_os(final TitanOctetString p_os);
```

Appends the content of the octet string to the buffer.

```
void increase_length(final int size_incr);
```

Increases the size of the buffer.

```
void cut();
```

Cuts (removes) the bytes between the beginning of the buffer and the read position. After calling this, the read position will be the beginning of buffer. As this function manipulates the internal data, pointers referencing to data inside the buffer will be invalid.

```
void cut_end();
```

Cuts (removes) the bytes between the read position and the end of the buffer. After calling this, the read position remains unchanged (that is, it will point to the end of the truncated buffer). As this function manipulates the internal data, pointers referencing to data inside the buffer will be invalid.

11.1.3. Invoking the Coding Functions

Every type class has members like these:

```
public void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf,
         final coding_type p_coding, final int flavour);
public void decode(final TTCN_Typedescriptor p_td,
        final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour);
```

Parameter `p_td` is a special type descriptor. Each type has its own descriptor, which contains the name of the type, and a lot of information used by the different encoding mechanisms. The names of the descriptors come from the name of the types: the appropriate type descriptor for type `XXX` is `XXX_descr_`. This descriptor can be found in the generated code:

- complex types (record, set, record of, set of, arrays, etc..): generated into the body of the class representing the type.
- types for which no code is generated (for example record of integer): generated into the body of the class representing the module of the type.
- built in types (for example integer): in the body of the classes representing the type (for example TitanInteger in the runtime).

Parameter `p_buf` contains the encoded value. For details about using it, please consult the previous subsection.

Parameter `p_coding` is the desired coding mechanism. As `coding_type` is defined in `TTCN_EncDec`, its value must be prefixed with `TTCN_EncDec.`. For the time being, this parameter may have one of the following values[7]:

- CT_RAW - RAW coding;
- CT_JSON - JSON coding;

The `flavour` parameter is depending on the chosen coding.

11.2. BER

BER encoding and decoding is not yet supported on the Java side.

11.3. RAW

You can use the encoding rules defined in the section "RAW encoder and decoder" in the Programmer's Technical Reference to encode and decode the following TTCN–3 types:

- boolean
- integer
- float
- bitstring

- octetstring
- charstring
- hexstring
- enumerated
- record
- set
- union
- record of
- set of

The compiler will produce code capable of RAW encoding/decoding for compound types if they 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.
When a type can be RAW encoded/decoded but with default specification then the empty variant specification can be used: `variant ""`.

11.3.1. Error Situations

Table 5. RAW-coding errors

ET_LEN_ERR	During encoding: Not enough length specified in FIELDLENGTH to encode the value. During decoding: the received message is shorter than expected.
ET_SIGN_ERR	Unsigned encoding of a negative number.
ET_FLOAT_NAN	Not a Number float value has been received.
ET_FLOAT_TR	The float value will be truncated during double to single precision conversion.

11.3.2. API

The Java Application Programming Interface for RAW encoding and decoding is described in the following. It can be used for example in test port implementation, in external function implementation.

Encoding

```
public void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf,
        final coding_type p_coding, final int flavour);
```

The parameter `p_coding` must be set to `TTCN_EncDec.CT_RAW`.

Decoding

```
public void decode(final TTCN_Typedescriptor p_td,
        final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour);
```

The parameter `p_coding` must be set to `TTCN_EncDec.CT_RAW`.

11.3.3. Example

Let us assume that we have a TTCN–3 module which contains a type named `ProtocolPdu`, and this module contains also two ports:

```
type port MyPort1 message
{
  out ProtocolPdu;
  in octetstring;
}

type port MyPort2 message
{
  out octetstring;
  in ProtocolPdu;
}
```

Then we can complete the port skeleton generated by the compiler as follows:

```
protected void outgoing_send(final ProtocolPdu send_par) {
    final TTCN_Buffer buffer = new TTCN_Buffer();
    send_par.encode(MyModule.ProtocolPdu.ProtocolPdu_descr_, buffer,
TTCN_EncDec.coding_type.CT_RAW, 0);
    final TitanOctetString encodedData = new TitanOctetString();
    buffer.get_string(encodedData);
    incoming_message(encodedData);
}

protected void outgoing_send(final TitanOctetString send_par) {
        TTCN_EncDec.set_error_behavior(TTCN_EncDec.error_type.ET_ALL,
TTCN_EncDec.error_behavior_type.EB_WARNING);
        final TTCN_Buffer buffer = new TTCN_Buffer();
        buffer.put_os(send_par);
        final ProtocolPdu pdu = new ProtocolPdu();
        pdu.decode(MyModule.ProtocolPdu.ProtocolPdu_descr_, buffer,
TTCN_EncDec.coding_type.CT_RAW, 0);
        incoming_message(pdu);
    }
```

11.4. TEXT

TEXT encoding and decoding is not yet supported on the Java side.

11.5. XML Encoding (XER)

XML encoding and decoding is not yet supported on the Java side.

11.6. JSON

The encoding rules defined in the section "JSON Encoder and Decoder" of the Programmer's Technical Reference can be used 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

The compiler will produce code capable of JSON encoding/decoding for compound types if they have at least one JSON variant attribute or the `encode "JSON"` attribute (and, for compound types, all fields and elements of compound types also have a JSON variant attribute or the `encode "JSON"` attribute).

The encoder and the decoder work with JSON data encoded in UTF-8 (described in UTF-8, a transformation format of ISO 10646), stored in an object of type `TTCN_Buffer`. Although the contents of this object can be retrieved (using the overloads of the `get_string` function) as an instance of `TitanOctetString`, `TitanCharString` or `TitanUniversalCharString`, it is recommended to use only the `TitanOctetString` representation. `TitanCharString` is not recommended, because UTF-8 is an 8-bit encoding so the buffer may contain bytes with values over 127, which are not valid characters for a TTCN-3 `charstring` (which is implemented by `TitanCharString`, see Charstring). `TitanUniversalCharString` must not be used because its internal representation is not UTF-8.

11.6.1. Error Situations

There are no extra error situations apart from the ones in The Common API.

11.6.2. API

The Application Programming Interface for JSON encoding and decoding is described in the following.

Encoding

```
void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf,
            final coding_type p_coding, final int flavour) const;
```

The parameter `p_coding` must be set to `TTCN_EncDec.CT_JSON`.

Decoding

```
void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf,
            final coding_type p_coding, final int flavour);
```

The parameter `p_coding` must be set to `TTCN_EncDec.CT_JSON`.

11.6.3. Example

Let us assume that we have a TTCN–3 module which contains a type named `ProtocolPdu`, and this module also contains two ports:

```
type port MyPort1 message
{
  out ProtocolPdu;
  in octetstring;
}

type port MyPort2 message
{
  out octetstring;
  in ProtocolPdu;
}
```

Then we can complete the port skeleton generated by the compiler:

```
void MyPort1.outgoing_send(final ProtocolPdu send_par)
{
  final TTCN_Buffer buffer = new TTCN_Buffer();
    send_par.encode(MyModule.ProtocolPdu.ProtocolPdu_descr_, buffer,
TTCN_EncDec.coding_type.CT_JSON, 0);
    final TitanOctetString encodedData = new TitanOctetString();
    buffer.get_string(encodedData);
    incoming_message(encodedData);
}

void MyPort2.outgoing_send(final TitanOctetString send_par)
{
    TTCN_EncDec.set_error_behavior(TTCN_EncDec.error_type.ET_ALL,
TTCN_EncDec.error_behavior_type.EB_WARNING);
    final TTCN_Buffer buffer = new TTCN_Buffer();
    buffer.put_os(send_par);
    final ProtocolPdu pdu = new ProtocolPdu();
    pdu.decode(MyModule.ProtocolPdu.ProtocolPdu_descr_, buffer,
TTCN_EncDec.coding_type.CT_JSON, 0);
    incoming_message(pdu);
}
```

11.7. OER

OER encoding and decoding is not yet supported on the Java side.

[7] BER, TEXT, XER and OER coding is not yet supported

Chapter 12. Mapping TTCN–3 Data Types to Java Constructs

On the Java side the TTCN–3 language elements of the test suite are individually mapped into more or less equivalent Java constructs. The data types are mapped to Java classes, the test cases become Java functions, and so on. In order to write a Test Port, it is inevitable to be familiar with the internal representation format of TTCN–3 data types and values. This section gives an overview about the data types and their equivalent Java constructs.

12.1. Mapping of Names and Identifiers

In order to identify the TTCN–3 language elements in the generated Java program properly, the names of test suite are translated to Java identifiers according to the following simple rules.

If the TTCN–3 identifier does not contain any underscore (_) character, its equivalent Java identifier will be the same. For example, the TTCN–3 variable `MyVar` will be translated to a Java variable called `MyVar`.

If the TTCN–3 identifier contains one or more underscore characters, each underscore character will be duplicated in the Java identifier. So the TTCN–3 identifier `My_Long_Name` will be mapped to a Java identifier called `My__Long__Name`.

The idea behind this name mapping is that we may freely use the Java identifiers containing one underscore character in the generated code and in the Test Ports as well. Otherwise name clashes might happen (and to keep in line with the C++ side of the toolset and its already existing large amount of code). Furthermore, the generated Java language elements fulfill the condition that the scope of a translated Java identifier is identical as the scope of the original TTCN–3 identifier.

The identifiers that are keywords of Java but not keywords in TTCN–3 are mapped to themselves, but a single underscore character is appended at the end (for example `for` becomes `for_`). The same rule applies to the all-uppercase identifiers that are used in the Base Library: identifier `TitanInteger` in TTCN–3 becomes `TitanInteger_` in Java, `TRUE` [8] is mapped to `TRUE_`, etc.

FIXME update list of words Here is the complete list (in alphabetical order) of the identifiers that are handled in such special way:asm, auto, bitand, bitor, bool, break, case, class, compl, continue, delete, double, enum, explicit, export, friend, inline, int, ischosen, long, main, mutable, namespace, new, operator, private, protected, public, register, short, signed, static, stderr, stdin, stdout, struct, switch, this, throw, try, typedef, typeid, typename, unsigned, using, virtual, void, volatile, ADDRESS, BITSTRING, BOOLEAN, CHAR, CHARSTRING, COMPONENT, DEFAULT, ERROR, FAIL, FALSE, FLOAT, HEXSTRING, INCONC, INTEGER, NONE, OBJID, OCTETSTRING, PASS, PORT, TIMER, TRUE, VERDICTTYPE.

The identifiers that are the names of common classes of the Java library (such as `System`, `Map`,) should be avoided in TTCN–3 modules. The name clashes clashes might create problems during the implementation of external functions and testports.

Note that these name mapping rules apply to **all** TTCN–3 identifiers, including module, Test Port,

type, field, variable and function names.

12.2. Modules

The Java code generator generates a Java class for every TTCN–3 and ASN.1 module. All Java definitions that belong to the module (including Test Port classes and external functions) are placed in that class. The name of the class is derived from the module identifier according to the rules described in Mapping of Names and Identifiers.

When accessing a Java entity that belongs to a different module than the referring Test Port or external function is in the reference has to be prefixed with the class of the referenced module and the class of the referenced module being imported. For example, to access the Java class that realizes type `MyType` defined in `MyModule1` from a Test Port that belongs to module `MyModule2` the reference shall be written as `MyModule1.MyType`.

12.3. Predefined TTCN–3 Data Types

in the TTCN–3 Base Library all basic data types of TTCN–3 were implemented as Java classes. This is because: * The TTCN–3 executor must know whether a variable has a valid value or not because sending an unbound value must result in a dynamic test case error. * Complex types (like a record or set) have no equivalents in Java. * Encoding and decoding of types in not present in Java types. * etc.

This section describes the member functions of these classes.

WARNING	The toString of the built in and generated types is not considered part of the public API for Test Port development. Its implementation might be subject to change without notice. Please do not use it.

12.3.1. `Integer`

The TTCN–3 type `integer` is implemented in class `TitanInteger`.
The class `TitanInteger` has the following public member functions:

Table 8. Public member functions of the class `TitanInteger`

Member functions	Notes

Constructors	`TitanInteger()`	Initializes to unbound value.
	`TitanInteger(final int otherValue)`	Initializes to a given value.
	`TitanInteger(final BigInteger otherValue)`	Initializes to a given value.
	`TitanInteger(final TitanInteger otherValue)`	Copy constructor.
	`TitanInteger(final String otherValue)`	Initializes with the String representation of an integer.
Assignment operators	`TitanInteger operator_assign(final int otherValue)`	Sets to given value.
	`TitanInteger operator_assign(final BigInteger otherValue)`	Sets to given value.
	`TitanInteger operator_assign(final TitanInteger otherValue)`	Sets to given value.
	`TitanInteger operator_assign(final Base_Type otherValue)`	Sets to given value.
Comparison operators	boolean operator_equals(final int otherValue)	Returns true if equals.
	boolean operator_equals(final BigInteger otherValue)	and false otherwise.
	boolean operator_equals(final TitanInteger otherValue)	
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final int otherValue)	
	boolean operator_not_equals(final BigInteger otherValue)	
	boolean operator_not_equals(final TitanInteger otherValue)	

Comparison operators	boolean is_less_than(final int otherValue)	
	boolean is_less_than(final BigInteger otherValue)	
	boolean is_less_than(final TitanInteger otherValue)	
	boolean is_less_than_or_equal(final int otherValue)	
	boolean is_less_than_or_equal(final BigInteger otherValue)	
	boolean is_less_than_or_equal(final TitanInteger otherValue)	
	boolean is_greater_than(final int otherValue)	
	boolean is_greater_than(final BigInteger otherValue)	
	boolean is_greater_than(final TitanInteger otherValue)	
	boolean is_greater_than_or_equal(final int otherValue)	
	boolean is_greater_than_or_equal(final BigInteger otherValue)	
	boolean is_greater_than_or_equal(final TitanInteger otherValue)	

Arithmetic operators	TitanInteger add()	Unary plus.
	TitanInteger sub()	Unary minus.
	TitanInteger add(final int other_value)	Addition.
	TitanInteger add(final BigInteger other_value)	
	TitanInteger add(final TitanInteger other_value)	
	TitanInteger sub(final int other_value)	Subtraction.
	TitanInteger sub(final BigInteger other_value)	
	TitanInteger sub(final TitanInteger other_value)	
	TitanInteger mul(final int other_value)	Multiplication.
	TitanInteger mul(final BigInteger other_value)	
	TitanInteger mul(final TitanInteger other_value)	
	TitanInteger div(final int other_value)	Integer division.
	TitanInteger div(final BigInteger other_value)	
	TitanInteger div(final TitanInteger other_value)	
	TitanInteger rem(final int other_value)	remainder of the division.
	TitanInteger rem(final BigInteger other_value)	
	TitanInteger rem(final TitanInteger other_value)	
	TitanInteger mod(final int other_value)	modulo of the division.
	TitanInteger mod(final BigInteger other_value)	
	TitanInteger mod(final TitanInteger other_value)	
Casting operator	int get_int()	Returns the value.
	long get_long()	Returns the value.
	BigInteger get_BigInteger()	Returns the value.

Other member functions	`boolean is_native()`	is the value native int.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

The comparison, arithmetic and shifting operators are also available as global functions for that case when the left side is `int` and the right side is `TitanInteger`. Using the value of an unbound variable for anything will cause dynamic test case error.

The `get_int()` is applicable only to `TitanInteger` objects holding a signed value with at most 31 useful bits, since in Java the native `int` type is 32-bit large including the sign bit. Being used on an `TitanInteger` object holding a bigger (for example a 32-bit unsigned) value will result in run-time error.

Please note that if the value stored in a `TitanInteger` object is too big (that is, it cannot be represented as a `int`) the value returned by `get_long()` will contain only the lowest 64 bits of the original value.

In addition, the following static functions are available for modulo division. These functions return the result of `mod` and `rem` operations according to TTCN–3 semantics.

```
TitanInteger mod(final TitanInteger left_value, final TitanInteger right_value);
TitanInteger mod(final TitanInteger left_value, final int right_value);
TitanInteger mod(final int left_value, final TitanInteger right_value);
TitanInteger mod(final int left_value, int right_value);

TitanInteger rem(final TitanInteger left_value, final TitanInteger right_value);
TitanInteger rem(final TitanInteger left_value, final int right_value);
TitanInteger rem(final int left_value, final TitanInteger right_value);
TitanInteger rem(final int left_value, final int right_value);
```

Other operators (static functions):

```
TitanInteger add(final int int_value, final TitanInteger other_value);  // Add
TitanInteger sub(final int int_value, final TitanInteger other_value);  // Subtract
TitanInteger mul(final int int_value, final TitanInteger other_value);  // Multiply
TitanInteger div(final int int_value, final TitanInteger other_value);  // Divide
boolean operator_equals(final int intValue, final TitanInteger otherValue); // Equal
boolean operator_not_equals(final int intValue, final TitanInteger otherValue); // Not
equal
boolean is_less_than(final int intValue, final TitanInteger otherValue);  // Less than
boolean is_greater_than(final int intValue, final TitanInteger otherValue);  // More
than
```

12.3.2. `Float`

The TTCN–3 type `float` is implemented in class `TitanFloat`.
The class `TitanFloat` has the following public member functions:

Table 9. Public member functions of the class `TitanFloat`

Member functions		Notes
Constructors	`TitanFloat()`	Initializes to unbound value.
	`TitanFloat(final double otherValue)`	Initializes to a given value.
	`TitanFloat(final Ttcn3Float otherValue)`	
	`TitanFloat(final TitanFloat otherValue)`	Copy constructor.
Assignment operators	`TitanFloat operator_assign(final double otherValue)`	Assigns the given value
	`TitanFloat operator_assign(final Ttcn3Float otherValue)`	and sets the bound flag.
	`TitanFloat operator_assign(final TitanFloat otherValue)`	
	`TitanFloat operator_assign(final Base_Type otherValue)`	

Comparison operators	boolean operator_equals(final double otherValue)	Returns true if equals
	boolean operator_equals(final Ttcn3Float otherValue)	and false otherwise.
	boolean operator_equals(final TitanFloat otherValue)	
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final double otherValue)	
	boolean operator_not_equals(final Ttcn3Float otherValue)	
	boolean operator_not_equals(final TitanFloat otherValue)	
	boolean is_less_than(final double otherValue)	
	boolean is_less_than(final Ttcn3Float otherValue)	
	boolean is_less_than(final TitanFloat otherValue)	
	boolean is_less_than_or_equal(final double otherValue)	
	boolean is_less_than_or_equal(final Ttcn3Float otherValue)	
	boolean is_less_than_or_equal(final TitanFloat otherValue)	
	boolean is_greater_than(final double otherValue)	
	boolean is_greater_than(final Ttcn3Float otherValue)	
	boolean is_greater_than(final TitanFloat otherValue)	
	boolean is_greater_than_or_equal(final double otherValue)	
	boolean is_greater_than_or_equal(final Ttcn3Float otherValue)	
	boolean is_greater_than_or_equal(final TitanFloat otherValue)	

Arithmetic operators	TitanFloat add()	Unary plus.
	TitanFloat sub()	Unary minus.
	TitanFloat add(final double other_value)	Addition.
	TitanFloat add(final Ttcn3Float other_value)	
	TitanFloat add(final TitanFloat other_value)	
	TitanFloat sub(final double other_value)	Subtraction.
	TitanFloat sub(final Ttcn3Float other_value)	
	TitanFloat sub(final TitanFloat other_value)	
	TitanFloat mul(final double other_value)	Multiplication.
	TitanFloat mul(final Ttcn3Float other_value)	
	TitanFloat mul(final TitanFloat other_value)	
	TitanFloat div(final double other_value)	Division.
	TitanFloat div(final Ttcn3Float other_value)	
	TitanFloat div(final TitanFloat other_value)	
Casting operator	Double get_value()	Returns the value.
Other member functions	`boolean is_native()`	is the value native int.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

The comparison and arithmetic operators are also available as static functions for that case when the left side is `double` and the right side is `TitanFloat`. Using the value of an unbound variable for anything will cause dynamic test case error.

Other operators (static functions):

```
TitanFloat add(final double double_value, final TitanFloat other_value);    // Add
TitanFloat sub(final double double_value, final TitanFloat other_value);    //
Subtract
TitanFloat mul(final double double_value, final TitanFloat other_value);    //
Multiply
TitanFloat div(final double double_value, final TitanFloat other_value);    // Divide
boolean operator_equals(final double doubleValue, final TitanFloat otherValue); //
Equal
boolean operator_not_equals(final double doubleValue, final TitanFloat otherValue); //
Not equal
boolean is_less_than(final double doubleValue, final TitanFloat otherValue);  // Less
than
boolean is_greater_than(final double doubleValue, final TitanFloat otherValue);  //
More than
```

12.3.3. Boolean

The TTCN–3 type `boolean` is implemented in class `TitanBoolean`.
The class `TitanBoolean` has the following public member functions:

Table 10. Public member functions of the class `TitanBoolean`

Member functions		Notes
Constructors	`TitanBoolean()`	Initializes to unbound value.
	`TitanBoolean(final Boolean otherValue)`	Initializes to a given value.
	`TitanBoolean(final TitanBoolean otherValue)`	Copy constructor.
Assignment operators	`TitanBoolean operator_assign(final boolean otherValue)`	Assigns the given value
	`TitanBoolean operator_assign(final TitanBoolean otherValue)`	and sets the bound flag.
	`TitanBoolean operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final boolean otherValue)	Returns true if equals
	boolean operator_equals(final TitanBoolean otherValue)	and false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final boolean otherValue)	Same as XOR.
	boolean operator_not_equals(final TitanBoolean otherValue)	

Logical operators	boolean not()	Negation (NOT).
	boolean and(final boolean other_value)	Logical AND.
	boolean and(final TitanBoolean other_value)	
	boolean or(final boolean other_value)	Logical OR.
	boolean or(final TitanBoolean other_value)	
	boolean xor(final boolean other_value)	Exclusive or (XOR).
	boolean xor(final TitanBoolean other_value)	
Casting operator	Boolean get_value()	Returns the value.
Other member functions	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log. Like "true" or "false".
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

The comparison and logical operators are also available as static functions for that case when the left side is `boolean` and the right side is `TitanBoolean`. Using the value of an unbound variable for anything will cause dynamic test case error.

Other operators (static functions):

```
boolean and(final boolean bool_value, final TitanBoolean other_value); // And
boolean xor(final boolean bool_value, final TitanBoolean other_value);  // Xor
boolean or(final boolean bool_value, final TitanBoolean other_value); // Or
boolean operator_equals(final boolean boolValue, final TitanBoolean otherValue); //
Equal
boolean operator_not_equals(final boolean boolValue, final TitanBoolean otherValue);//
Not equal
```

12.3.4. `Verdicttype`

The TTCN–3 type `verdicttype` is implemented in class `TitanVerdictType`.
The class `TitanVerdictType` has the following public member functions:

Table 11. Public member functions of the class `TitanVerdictType`

Member functions		Notes
Constructors	`TitanVerdictType()`	Initializes to unbound value.
	`TitanVerdictType(final VerdictTypeEnum otherValue)`	Initializes to a given value.
	`TitanVerdictType(final TitanVerdictType otherValue)`	Copy constructor.
Assignment operators	`TitanVerdictType operator_assign(final VerdictTypeEnum otherValue)`	Assigns the given value
	`TitanVerdictType operator_assign(final TitanVerdictType otherValue)`	and sets the bound flag.
	`TitanVerdictType operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final VerdictTypeEnum otherValue)	Returns true if equals
	boolean operator_equals(final TitanVerdictType otherValue)	and false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final VerdictTypeEnum otherValue)	
	boolean operator_not_equals(final TitanVerdictType otherValue)	
Casting operator	VerdictTypeEnum get_value()	Returns the value.

Other member functions	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log. Like "pass" or "fail".
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

The comparison operators are also available as static functions for that case when the left side is `VerdictTypeEnum` and the right side is `TitanVerdictType`. Using the value of an unbound `TitanVerdictType` variable for anything will cause dynamic test case error.

Other operators (static functions):

```
boolean operator_equals(final VerdictTypeEnum par_value, final TitanVerdictType
other_value); // Equal
boolean operator_not_equals(final VerdictTypeEnum par_value, final TitanVerdictType
other_value); // Not equal
```

There are the following three static member functions in class `TTCN_Runtime` defined in the Base Library for getting or modifying the local verdict of the current test components:

```
void setverdict(final TitanVerdictType.VerdictTypeEnum newValue);
void setverdict(final TitanVerdictType newValue);
void setverdict(final TitanVerdictType.VerdictTypeEnum newValue, final String reason);
setverdict(final TitanVerdictType newValue, final String reason);
TitanVerdictType get_verdict();
```

These functions are the Java equivalents of TTCN–3 `setverdict` and `getverdict` operations. Use them only if your Test Port or Java function encounters a low-level failure, but it can continue its normal

operation (that is, error recovery is not necessary).

12.3.5. `Bitstring`

The equivalent Java class of TTCN–3 type `bitstring` is called `TitanBitString`. The bits of the bit string are stored in an array of ints. In order to reduce the wasted memory space the bits are packed together, so each int contains eight bits. The first int contains the first eight bits of the bit string; the second int contains the bits from the 9th up to the 16th, and so on. The first bit of the bit string is the LSB of the first character; the second bit is the second least significant bit of the first character, and so on. If the length of the bit string is not a multiple of eight, the unused bits of the last character can contain any value. So the length of the bit string must be always given.

The class `TitanBitString` has the following public member functions:

Table 12. Public member functions of the class `TitanBitString`

Member functions		Notes
Constructors	`TitanBitString()`	Initializes to unbound value.
	`TitanBitString(final int other_value[], final int nof_bits)`	Initializes from a given length and int array.
	`TitanBitString(final TitanBitString otherValue)`	Copy constructor.
	`TitanBitString(final TitanBitString_Element otherValue)`	Initializes from a single bitstring element.
Assignment operators	`TitanBitString operator_assign(final TitanBitString otherValue)`	Assigns the given value and sets the bound flag.
	`TitanBitString operator_assign(final TitanBitString_Element otherValue)`	Assigns the given single bitstring element.
	`TitanBitString operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final TitanBitString otherValue)	Returns true if equals
	boolean operator_equals(final TitanBitString_Element otherValue)	and false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final TitanBitString otherValue)	
	boolean operator_not_equals(final TitanBitString_Element otherValue)	

Concatenation operator	TitanBitString operator_concatenate(final TitanBitString other_value)	Concatenates two bitstrings.
	TitanBitString operator_concatenate(final TitanBitString_Element other_value)	Concatenates a bitstring and a bitstring element.
Index operator	TitanBitString_Element get_at(final int index_value)	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	TitanBitString_Element get_at(final TitanInteger index_value)	
	TitanBitString_Element constGet_at(final int index_value)	Gives read-only access to the given element.
	TitanBitString_Element constGet_at(final TitanInteger index_value)	
Bitwise operators	TitanBitString not4b()	not4b. (bitwise negation)
	TitanBitString and4b(final TitanBitString otherValue)	and4b. (bitwise and)
	TitanBitString and4b(final TitanBitString_Element otherValue)	
	TitanBitString or4b(final TitanBitString otherValue)	or4b. (bitwise or)
	TitanBitString or4b(final TitanBitString_Element otherValue)	
	TitanBitString xor4b(final TitanBitString otherValue)	xor4b. (bitwise xor)
	TitanBitString xor4b(final TitanBitString_Element otherValue)	

Shifting and rotating operators	TitanBitString shift_left(int shift_count)	Java equivalent of operator
	TitanBitString shift_left(final TitanInteger shift_count)	<<.(shift left)
	TitanBitString shift_right(int shift_count)	Java equivalent of operator
	TitanBitString shift_right(final TitanInteger shift_count)	>>. (shift right)
	TitanBitString rotate_left(int rotate_count)	Java equivalent of operator
	TitanBitString rotate_left(final TitanInteger rotate_count)	<@. (rotate left)
	TitanBitString rotate_right(int rotate_count)	Java equivalent of operator
	TitanBitString rotate_right(final TitanInteger rotate_count)	@>. (rotate right)
Casting operator	int[] get_value()	Returns a pointer to the int array.
Other member functions	`int lengthof() const`	Returns the length measured in bits.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log. Example: ’100011’B.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

Using the value of an unbound `TitanBitString` variable for anything will cause dynamic test case error.

`Bitstring element`

The Java class `TitanBitString_Element` is the equivalent of the TTCN-3 `bitstring`’s element type (the

result of indexing a `bitstring` value). The class does not store the actual bit, only a reference to the original `TitanBitString` object, an index value and a bound flag.

NOTE	changing the value of the `TitanBitString_Element` (through the assignment operator) changes the referenced bit in the original `bitstring` object.

The class `TitanBitString_Element` has the following public member functions:

Table 13. Public member functions of the class `TitanBitString_Element`

Member functions		Notes
Constructor	`TitanBitString_Element(final boolean par_bound_flag, final TitanBitString par_str_val, final int par_bit_pos)`	Initializes the object with an unbound value or a reference to a bit in an existring TitanBitString object.
Assignment operators	`TitanBitString_Element operator_assign(final TitanBitString otherValue)`	Sets the referenced bit to the given bitstring of length 1.
	`TitanBitString_Element operator_assign(final TitanBitString_Element otherValue)`	Sets the referenced bit to the given bitstring element.
Comparison operators	boolean operator_equals(final TitanBitString otherValue)	Comparison with a bitstring or a bitstring element (the value of the referenced bits is compared, not the references and indexes).
	boolean operator_equals(final TitanBitString_Element otherValue)	
	boolean operator_not_equals(final TitanBitString otherValue)	
	boolean operator_not_equals(final TitanBitString_Element otherValue)	

Concatenation operator	TitanBitString operator_concatenate(final TitanBitString other_value)	Concatenates a bitstring element with a bitstring, or two bitstring elements.
	TitanBitString operator_concatenate(final TitanBitString_Element other_value)	
Bitwise operators	TitanBitString not4b()	not4b. (bitwise negation)
	TitanBitString and4b(final TitanBitString otherValue)	and4b. (bitwise and)
	TitanBitString and4b(final TitanBitString_Element otherValue)	
	TitanBitString or4b(final TitanBitString otherValue)	or4b. (bitwise or)
	TitanBitString or4b(final TitanBitString_Element otherValue)	
	TitanBitString xor4b(final TitanBitString otherValue)	xor4b. (bitwise xor)
	TitanBitString xor4b(final TitanBitString_Element otherValue)	
Other member functions	`boolean get_bit()`	Returns the referenced bit.
	`void log()`	Puts the value into log. Example: '1'B.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.

Using the value of an unbound `TitanBitString_Element` variable for anything will cause dynamic test case error.

12.3.6. `Hexstring`

The equivalent Java class of TTCN–3 type `hexstring` is called `TitanHexString`. The hexadecimal digits (nibbles) are stored in an array of unsigned bytes. In order to reduce the wasted memory space two nibbles are packed into one byte. The first byte contains the first two nibbles of the `hexstring`, the second byte contains the third and fourth nibbles, and so on. The hexadecimal digits at odd (first, third, fifth, etc.) positions occupy the lower 4 bits in the characters; the even ones use the upper 4 bits. The length must be always given with the pointer. If the `hexstring` has odd length the unused

upper 4 bits of the last character may contain any value.

The class `TitanHexString` has the following public member functions:

Table 14. Public member functions of the class `TitanHexString`

Member functions		Notes
Constructors	`TitanHexString()`	Initializes to unbound value.
	`TitanHexString(final byte otherValue[])`	Initializes from a given byte array.
	`TitanHexString(final TitanHexString otherValue)`	
	`TitanHexString(final TitanHexString_Element otherValue)`	
	`TitanHexString(final byte aValue)`	
	`TitanHexString(final String aValue)`	
Assignment operators	`TitanHexString operator_assign(final TitanHexString otherValue)`	Assigns the given value
	`TitanHexString operator_assign(final TitanHexString_Element otherValue)`	
	`TitanHexString operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final TitanHexString otherValue)	Returns true if equals and false otherwise.
	boolean operator_equals(final TitanHexString_Element otherValue)	
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final TitanHexString otherValue)	
	boolean operator_not_equals(final TitanHexString_Element otherValue)	
Concatenation operator	TitanHexString operator_concatenate(final TitanHexString other_value)	Concatenates two hexstrings.
	TitanHexString operator_concatenate(final TitanHexString_Element other_value)	Concatenates a hexstring and a hexstring element.

Member functions		Notes
Index operator	TitanHexString_Element get_at(final int index_value)	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	TitanHexString_Element get_at(final TitanInteger index_value)	
	TitanHexString_Element constGet_at(final int index_value)	
	TitanHexString_Element constGet_at(final TitanInteger index_value)	
Bitwise operators	TitanHexString not4b()	not4b. (bitwise negation)
	TitanHexString and4b(final TitanHexString otherValue)	and4b. (bitwise and)
	TitanHexString and4b(final TitanHexString_Element otherValue)	
	TitanHexString or4b(final TitanHexString otherValue)	or4b. (bitwise or)
	TitanHexString or4b(final TitanHexString_Element otherValue)	
	TitanHexString xor4b(final TitanHexString otherValue)	xor4b. (bitwise xor)
	HTitanHexString xor4b(final TitanHexString_Element otherValue)	
Shifting and rotating operators	TitanHexString shift_left(int shift_count)	Java equivalent of operator
	TitanHexString shift_left(final TitanInteger shift_count)	<<.(shift left)
	TitanHexString shift_right(int shift_count)	Java equivalent of operator
	TitanHexString shift_right(final TitanInteger shift_count)	>>. (shift right)
	TitanHexString rotate_left(int rotate_count)	Java equivalent of operator
	TitanHexString rotate_left(final TitanInteger rotate_count)	<@. (rotate left)
	TitanHexString rotate_right(int rotateCount)	Javaequivalent of operator
	TitanHexString rotate_right(final TitanInteger rotateCount)	@>. (rotate right)

Member functions		Notes
Casting operator	byte[] get_value()	Returns a pointer to the character array. The pointer might be NULL if the length is 0.
Other member functions	`int lengthof() const`	Returns the length measured in bits.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
`void log()`	Puts the value into log. Example: '5A7'H.	`void clean_up()`
Deletes the value, setting it to unbound.	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.

Using the value of an unbound `TitanHexString` variable for anything will cause a dynamic test case error.

`Hexstring` element

The Java class `TitanHexString_Element` is the equivalent of the TTCN-3 `hexstring`'s element type (the result of indexing a `hexstring` value). The class does not store the actual hexadecimal digit (nibble), only a reference to the original TitanHexString object, an index value and a bound flag.

NOTE changing the value of the `TitanHexString_Element` (through the assignment operator) changes the referenced nibble in the original `hexstring` object.

The class `TitanHexString_Element` has the following public member functions:

Table 15. Public member functions of the class `TitanHexString_Element`

Member functions	Notes

Constructor	`TitanHexString_Element(final boolean par_bound_flag, final TitanHexString par_str_val, final int par_nibble_pos)`	Initializes the object with an unbound value or a reference to a nibble in an existring TitanHexString object.
Assignment operators	`TitanHexString_Element operator_assign(final TitanHexString otherValue)`	Sets the referenced nibble to the given hexstring of length 1.
	`TitanHexString_Element operator_assign(final TitanHexString_Element otherValue)`	Sets the referenced nibble to the given hexstring element.
Comparison operators	boolean operator_equals(final TitanHexString otherValue)	Comparison with a hexstring or a hexstring element (the value of the referenced nibbles is compared, not the references and indexes).
	boolean operator_equals(final TitanHexString_Element otherValue)	
	boolean operator_not_equals(final TitanHexString otherValue)	
	boolean operator_not_equals(final TitanHexString_Element otherValue)	
Concatenation operator	TitanHexString operator_concatenate(final TitanHexString other_value)	Concatenates a hexstring element with a hexstring, or two hexstring elements.
	TitanHexString operator_concatenate(final TitanHexString_Element other_value)	

Bitwise operators	TitanHexString not4b()	Java equivalent of operator not4b. (bitwise negation)
	TitanHexString and4b(final TitanHexString other_value)	and4b. (bitwise and)
	TitanHexString and4b(final TitanHexString_Element other_value)	
	TitanHexString or4b(final TitanHexString other_value)	or4b. (bitwise or)
	TitanHexString or4b(final TitanHexString_Element other_value)	
	TitanHexString xor4b(final TitanHexString other_value)	xor4b. (bitwise xor)
	TitanHexString xor4b(final TitanHexString_Element other_value)	
Other member functions	`char get_nibble()`	Returns the referenced nibble (stored in the lower 4 bits of the returned character).
	`void log()`	Puts the value into log. Example: '8'H.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.

Using the value of an unbound `TitanHexString_Element` variable for anything will cause dynamic test case error.

12.3.7. `Octetstring`

The equivalent Java class of TTCN–3 type `octetstring` is called `TitanOctetString`. The octets are stored in an array of unsigned characters. Each character contains one octet; the first character is the first octet of the string. The length of the octet string must be always given.

The class `TitanOctetString` has the following public member functions:

Table 16. Public member functions of the class `TitanOctetString`

Member functions		Notes
Constructors	`TitanOctetString()`	Initializes to unbound value.
	`TitanOctetString(final char otherValue[])`	Initializes from a given character array.
	`TitanOctetString(final TitanOctetString otherValue)`	Copy constructor.
	`TitanOctetString(final TitanOctetString_Element otherValue)`	Initializes from a single octetstring element.
Assignment operators	`TitanOctetString operator_assign(final TitanOctetString otherValue)`	Assigns the given value and sets the bound flag.
	`TitanOctetString operator_assign(final TitanOctetString_Element otherValue)`	Assigns the given octetstring element.
	`TitanOctetString operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final TitanOctetString otherValue)	Returns true if equals
	boolean operator_equals(final TitanOctetString_Element otherValue)	and false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final TitanOctetString otherValue)	
	boolean operator_not_equals(final TitanOctetString_Element otherValue)	
Concatenation operator	TitanOctetString operator_concatenate(final TitanOctetString other_value)	Concatenates two octetstrings.
	TitanOctetString operator_concatenate(final TitanOctetString_Element other_value)	Concatenates an octetstring and an octetstring element.

Member functions		Notes
Index operator	TitanOctetString_Element get_at(final int index_value)	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	TitanOctetString_Element get_at(final TitanInteger index_value)	
	TitanOctetString_Element constGet_at(final int index_value)	Gives read-only access to the given element.
	TitanOctetString_Element constGet_at(final TitanInteger index_value)	
Bitwise operators	TitanOctetString not4b()	not4b.(bitwise negation)
	TitanOctetString and4b(final TitanOctetString otherValue)	and4b.(bitwise and)
	TitanOctetString and4b(final TitanOctetString_Element otherValue)	
	TitanOctetString or4b(final TitanOctetString otherValue)	or4b.(bitwise or)
	TitanOctetString or4b(final TitanOctetString_Element otherValue)	
	TitanOctetString xor4b(final TitanOctetString otherValue)	xor4b. (bitwise xor)
	TitanOctetString xor4b(final TitanOctetString_Element otherValue)	
Shifting and rotating operators	TitanOctetString shift_left(final int shift_count)	operator <<.
	TitanOctetString shift_left(final TitanInteger shift_count)	(shift left)
	TitanOctetString shift_right(final int shift_count)	operator >>.
	TitanOctetString shift_right(final TitanInteger shift_count)	(shift right)
	TitanOctetString rotate_left(final int rotate_count)	operator <@.
	TitanOctetString rotate_left(final TitanInteger rotate_count)	(rotate left)
	TitanOctetString rotate_right(final int rotate_count)	operator @>.
	TitanOctetString rotate_right(final TitanInteger rotate_count)	(rotate right)

Member functions		Notes
Casting operator	char[] get_value()	Returns a pointer to the character array. The pointer might be NULL if the length is 0.
Other member functions	`int lengthof() const`	Returns the length measured in bits.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log. Like ’073CF0’O.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

Using the value of an unbound `TitanOctetString` variable for anything will cause dynamic test case error.

`Octetstring` element

The Java class `TitanOctetString_Element` is the equivalent of the TTCN-3 `octetstring`’s element type (the result of indexing an `octetstring` value). The class does not store the actual octet, only a reference to the original TitanOctetString object, an index value and a bound flag.

NOTE changing the value of the TitanOctetString_Element (through the assignment operator) changes the referenced octet in the original `octetstring` object.

The class `TitanOctetString_Element` has the following public member functions:

Table 17. Public member functions of the class `TitanOctetString_Element`

Member functions		Notes
Constructor	`TitanOctetString_Element(final boolean par_bound_flag, final TitanOctetString par_str_val, final int par_nibble_pos)`	Initializes the object with an unbound value or a reference to an octet in an existing TitanOctetString object.
Assignment operators	`TitanOctetString_Element operator_assign(final TitanOctetString otherValue)`	Sets the referenced octet to the given octetstring of length 1.
	`TitanOctetString_Element operator_assign(final TitanOctetString_Element otherValue)`	Sets the referenced octet to the given octetstring element.
Comparison operators	TitanOctetString_Element operator_equals(final TitanOctetString otherValue)	Comparison with an octetstring or an octetstring element (the value of the referenced octets is compared, not the references and indexes).
	TitanOctetString_Element operator_equals(final TitanOctetString_Element otherValue)	
	boolean operator_not_equals(final TitanOctetString otherValue)	
	boolean operator_not_equals(final TitanOctetString_Element otherValue)	
Concatenation operator	TitanOctetString operator_concatenate(final TitanOctetString other_value)	Concatenates an octetstring element with an octetstring, or two octetstring elements.
	TitanOctetString operator_concatenate(final TitanOctetString_Element other_value)	

Member functions		Notes
Bitwise operators	TitanOctetString not4b()	bitwise negation
	TitanOctetString and4b(final TitanOctetString other_value)	and4b. (bitwise and)
	TitanOctetString and4b(final TitanOctetString_Element other_value)	
	TitanOctetString or4b(final TitanOctetString other_value)	or4b. (bitwise or)
	TitanOctetString or4b(final TitanOctetString_Element other_value)	
	TitanOctetString xor4b(final TitanOctetString other_value)	xor4b. (bitwise xor)
	TitanOctetString xor4b(final TitanOctetString_Element other_value)	
Other member functions	`char get_nibble()`	Returns the referenced octet.
	`void log()`	Puts the value into log. Example: '3C'O.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.

Using the value of an unbound `TitanOctetString_Element` variable for anything will cause dynamic test case error.

12.3.8. `Char`

The `char` type, which has been removed from the TTCN–3 standard, is no longer supported by the run-time environment. The compiler substitutes all occurrences of `char` type with type `charstring` automatically.

12.3.9. `Charstring`

The equivalent Java class of TTCN–3 type `charstring` is called `TitanCharString`. The characters are stored in a StringBuilder..

The class `TitanCharString` has the following public member functions:

Table 18. Public member functions of the class `TitanCharString`

Member functions	Notes

Constructors	`TitanCharString()`	Initializes to unbound value.
	`TitanCharString(final String otherValue)`	Initializes from a String.
	`TitanCharString(final StringBuilder otherValue)`	Initializes from the StringBuilder.
	`TitanCharString(final TitanCharString otherValue)`	Copy constructor.
	`TitanCharString(final TitanCharString_Element otherValue)`	Initializes from a charstring element.
	`TitanCharString(final TitanUniversalCharString otherValue)`	Initializs from the universal charstring.
Assignment operators	`TitanCharString operator_assign(final String otherValue)`	Assigns the given value and sets the bound flag.
	`TitanCharString operator_assign(final TitanCharString otherValue)`	
	`TitanCharString operator_assign(final Base_Type otherValue)`	
	`TitanCharString operator_assign(final TitanCharString_Element otherValue)`	
	`TitanCharString operator_assign(final TitanUniversalCharString otherValue)`	
Comparison operators	boolean operator_equals(final TitanCharString otherValue)	Returns true if equals and false otherwise.
	boolean operator_equals(final TitanUniversalCharString otherValue)	
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_equals(final String otherValue)	
	boolean operator_equals(final TitanCharString_Element otherValue)	
	boolean operator_equals(final TitanUniversalCharString_Element otherValue)	
	boolean operator_not_equals(final TitanCharString otherValue)	
	boolean operator_not_equals(final TitanCharString_Element otherValue)	
	boolean operator_not_equals(final String otherValue)	

Concatenation operator	TitanCharString operator_concatenate(final TitanCharString other_value)	Concatenates two charstrings.
	TitanCharString operator_concatenate(final String other_value)	
	TitanCharString operator_concatenate(final TitanCharString_Element other_value)	
	TitanUniversalCharString operator_concatenate(final TitanUniversalCharString other_value)	Concatenates with a universal charstring.
	TitanCharString append(final String aOtherValue)	Appends a String.
	TitanCharString append(final TitanCharString_Element aOtherValue)	
	TitanCharString append(final TitanCharString aOtherValue)	Appends a charstring.
Index operator	TitanCharString_Element get_at(final int index_value)	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	TitanCharString_Element get_at(final TitanInteger index_value)	
	TitanCharString_Element constGet_at(final int index_value)	Gives read-only access to the given element.
	TitanCharString_Element constGet_at(final TitanInteger index_value)	
Rotating operators	TitanCharString rotate_left(final int rotate_count)	Java equivalent of operator <@.(rotate left)
	TitanCharString rotate_left(final TitanInteger rotate_count)	
	TitanCharString rotate_right(final int rotate_count)	@>. (rotate right)
	TitanCharString rotate_right(final TitanInteger rotate_count)	
Casting operator	StringBuilder get_value()	Returns the StringBuilder.

Other member functions	`int lengthof() const`	Returns the length measured in bits.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log. Like "abc".
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

The comparison, concatenation and rotating operators are also available as static functions for that case when the left side is `String` and the right side is `TitanCharString`.

The log() member function uses single character output for regular characters, but special characters (such as the quotation mark, backslash or newline characters) are printed using the escape sequences of the C language. Non-printable control characters are printed in TTCN–3 quadruple notation, where the first three octets are always zero. The concatenation operator (`&`) is used between the fragments when necessary. Note that the output does not always conform to TTCN–3 Core Language syntax, but it is always recognized by both our compiler and the configuration file parser.

Using the value of an unbound `TitanCharString` variable for anything will cause dynamic test case error.

Other operators (static functions):

```
boolean operator_equals(final String stringValue, final TitanCharString otherValue);
// Equal
boolean operator_equals(final String stringValue, final TitanCharString_Element
otherValue);    // Equal
boolean operator_not_equals(final String stringValue, final TitanCharString
otherValue);            // Not equal
boolean operator_not_equals(final String stringValue, final TitanCharString_Element
otherValue);    // Not equal
TitanCharString operator_concatenate(final String stringValue, final TitanCharString
other_value);          // Concatenation
TitanCharString operator_concatenate(final String stringValue, final
TitanCharString_Element other_value);  // Concatenation
```

`Charstring` element

The Java class `TitanCharString_Element` is the equivalent of the TTCN-3 `charstring`'s element type (the result of indexing a `charstring` value). The class does not store the actual character, only a reference to the original TitanCharString object, an index value and a bound flag.

NOTE	changing the value of the `TitanCharString_Element` (through the assignment operator) changes the referenced character in the original `charstring` object.

The class `TitanCharString_Element` has the following public member functions:

Table 19. Public member functions of the class `TitanCharString_Element`

Member functions		Notes
Constructor	`TitanCharString_Element(final boolean par_bound_flag, final TitanCharString par_str_val, final int par_char_pos)`	Initializes the object with an unbound value or a reference to a character in an existing TitanCharString object.

Assignment operators	`TitanCharString_Element operator_assign(final String otherValue)`	Sets the referenced character to the given String of length 1.
	`TitanCharString_Element operator_assign(final TitanCharString otherValue)`	Sets the referenced character to the given charstring of length 1.
	`TitanCharString_Element operator_assign(final TitanCharString_Element otherValue)`	Sets the referenced character to the given charstring element.
Comparison operators	boolean operator_equals(final String otherValue)	Comparison with a String.
	boolean operator_equals(final TitanCharString otherValue)	
	boolean operator_equals(final TitanCharString_Element otherValue)	
	boolean operator_equals(final TitanUniversalCharString otherValue)	
	boolean operator_equals(final TitanUniversalCharString_Element otherValue)	
	boolean operator_not_equals(final String otherValue)	
	boolean operator_not_equals(final TitanUniversalCharString otherValue)	
	boolean operator_not_equals(final TitanUniversalCharString_Element otherValue)	
Concatenation operator	TitanCharString operator_concatenate(final String other_value)	Concatenates this object with a String.
	TitanCharString operator_concatenate(final TitanCharString other_value)	
	TitanCharString operator_concatenate(final TitanCharString_Element other_value)	
	TitanUniversalCharString operator_concatenate(final TitanUniversalCharString other_value)	
	TitanUniversalCharString operator_concatenate(final TitanUniversalCharString_Element other_value)	

Other member functions	`char get_char()`	Returns the referenced character.
	`void log()`	Puts the value into log. Example: “a”.
	`boolean is_bound()`	Returns whether the value is bound.

Using the value of an unbound `TitanCharString_Element` variable for anything will cause dynamic test case error.

12.3.10. `Universal char`

This obsolete TTCN–3 type is converted automatically to `universal charstring` in the parser.

12.3.11. `Universal charstring`

Each character of a `universal charstring` value is represented in the following C structure defined in the Base Library:

```
public class TitanUniversalChar {
    private char uc_group;
    private char uc_plane;
    private char uc_row;
    private char uc_cell;
  ...
```

The four components of the quadruple (that is, group, plane, row and cell) are stored in fields `uc_group`, `uc_plane`, `uc_row` and `uc_cell`, respectively. All fields are 8bit unsigned numeric values with the possible value range 0 .. 255.

In case of single-octet characters, which can be also given in TTCN–3 charstring notation (between quotation marks), the fields `uc_group`, `uc_plane`, `uc_row` are set to zero. If tuple notation was used for an ASN.1 string value fields `uc_row` and `uc_cell` carry the tuple and the others are set to zero.

Except when performing encoding or decoding, the run-time environment does not check whether the quadruples used in the following API represent valid character positions according to [8]. Moreover, if ASN.1 multi-octet character string values are used, it is not verified whether the elements of such strings are permitted characters of the corresponding string type.

The Java equivalent of TTCN–3 type `universal charstring` is implemented in class `TitanUniversalCharString`. The characters of the string are stored in an array of structure `TitanUniversalChar`. The array returned by the casting operator is not terminated with a special character, thus, the length of the string must be always considered when doing operations with the array. The length of the string, which can be obtained by using member function `lengthof()`, is measured in characters (quadruples) and not bytes.

For the more convenient usage the strings containing only single-octet characters can also be used with class `TitanUniversalCharString`. Therefore some polymorphic member functions and operators have variants that take `String` as argument. In these member functions the characters of the String are implicitly converted to quadruples with group, plane and row fields set to zero.

The class `TitanUniversalCharString` has the following public member functions:

Table 20. Public member functions of the class `TitanUniversalCharString`

Member functions		Notes
Constructors	`TitanUniversalCharString()`	Initializes to unbound value.
	`TitanUniversalCharString(final char uc_group, final char uc_plane, final char uc_row, final char uc_cell)`	Constructs a string containing one character formed from the given quadruple.
	`TitanUniversalCharString(final TitanUniversalChar otherValue)`	Constructs a string containing the given single character.
	`TitanUniversalCharString(final List<TitanUniversalChar> otherValue)`	Constructs a string from an array by taking the given number of single-octet characters.
	`TitanUniversalCharString(final TitanUniversalChar[] otherValue)`	
	`TitanUniversalCharString(final String otherValue)`	
	`TitanUniversalCharString(final StringBuilder otherValue)`	

Constructors	`TitanUniversalCharString(final TitanCharString otherValue)`	Constructs a universal charstring from a charstring value.
	`TitanUniversalCharString(final TitanCharString_Element otherValue)`	Constructs a string containing the given singe charstring element.
	`TitanUniversalCharString(final TitanUniversalCharString otherValue)`	Copy constructor.
	`TitanUniversalCharString(final TitanUniversalCharString_Element otherValue)`	Constructs a string containing the given singe universal charstring element.
Assignment operators	`TitanUniversalCharString operator_assign(final TitanUniversalCharString otherValue)`	Assigns another string.
	`TitanUniversalCharString operator_assign(final TitanUniversalChar otherValue)`	Assigns a single character.
	`TitanUniversalCharString operator_assign(final char[] otherValue)`	Assigns an array single-octet characters.
	`TitanUniversalCharString operator_assign(final String otherValue)`	
	`TitanUniversalCharString operator_assign(final TitanCharString otherValue)`	Assigns a charstring.
	`TitanUniversalCharString operator_assign(final TitanCharString_Element otherValue)`	Assigns a single charstring element.
	`TitanUniversalCharString operator_assign(final TitanUniversalCharString_Element otherValue)`	Assigns a single universal charstring element.
	`TitanUniversalCharString operator_assign(final Base_Type otherValue)`	

Comparison operators	boolean operator_equals(final TitanUniversalCharString otherValue)	Returns true if the strings are identical or false otherwise.
	boolean operator_equals(final TitanUniversalChar otherValue)	Compares to a single character.
	boolean operator_equals(final String otherValue)	Compares to a String.
	boolean operator_equals(final TitanCharString otherValue)	Compares to a charstring.
	boolean operator_equals(final TitanCharString_Element otherValue)	Compares to a charstring element.
	boolean operator_equals(final TitanUniversalCharString_Element otherValue)	Compares to a universal charstring element.
	boolean operator_equals(final Base_Type otherValue)	
Comparison operators	boolean operator_not_equals(final TitanUniversalCharString otherValue)	
	boolean operator_not_equals(final TitanUniversalChar otherValue)	
	boolean operator_not_equals(final String otherValue)	
	boolean operator_not_equals(final TitanCharString otherValue)	
	boolean operator_not_equals(final TitanCharString_Element otherValue)	
	boolean operator_not_equals(final TitanUniversalCharString_Element otherValue)	
	boolean operator_not_equals(final Base_Type otherValue)	

Concatenation operator	TitanUniversalCharString operator_concatenate(final TitanUniversalCharString other_value)	Concatenates two strings.
	TitanUniversalCharString operator_concatenate(final TitanUniversalChar other_value)	Concatenates a single character.
	TitanUniversalCharString operator_concatenate(final String other_value)	Concatenates a single-octet string.
	TitanUniversalCharString operator_concatenate(final TitanCharString other_value)	Concatenates a charstring.
	TitanUniversalCharString operator_concatenate(final TitanCharString_Element other_value)	Concatenates a charstring element.
	TitanUniversalCharString operator_concatenate(final TitanUniversalCharString_Element other_value)	Concatenates a universal charstring element.
Index operator	TitanUniversalCharString_Element get_at(final int index_value)	Gives access to the given element. Indexing begins from zero. Index overflow causes dynamic test case error.
	TitanUniversalCharString_Element get_at(final TitanInteger index_value)	
	TitanUniversalCharString_Element constGet_at(final int index_value)	Gives read-only access to the given element.
	TitanUniversalCharString_Element constGet_at(final TitanInteger index_value)	
Rotating operators	TitanUniversalCharString rotate_left(final int rotate_count)	<@(rotate left).
	TitanUniversalCharString rotate_left(final TitanInteger rotate_count)	
	TitanUniversalCharString rotate_right(final int rotate_count)	@>(rotate right).
	TitanUniversalCharString rotate_right(final TitanInteger rotate_count)	

Casting operator	List<TitanUniversalChar> get_value()	Returns a pointer to the array of characters. There is no terminator character at the end.
UTF-8 encoding and decoding	void encode_utf8(final TTCN_Buffer buf)	Appends the UTF-8 representation of the string to the given buffer
	void encode_utf8(final TTCN_Buffer buf, final boolean addBOM)	
	void decode_utf8(final char[] valueStr, final CharCoding code, final boolean checkBOM)	
	void encode_utf16(final TTCN_Buffer buf, final CharCoding expected_coding)	
	void decode_utf16(final int n_octets, final char[] octets_ptr, final CharCoding expected_coding)	
	void encode_utf32(final TTCN_Buffer buf, final CharCoding expected_coding)	
	void decode_utf32(final int n_octets, final char[] octets_ptr, final CharCoding expected_coding)	

Other member functions	`int lengthof() const`	Returns the length measured in characters.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.
	`void log()`	Puts the value into log. See below.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	encodes the value.
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	decodes the value.

The comparison and concatenation operators are also available as static functions for that case when the left operand is a single-octet string (`String`) or a single character (`TitanUniversalChar`) and the right side is `TitanUniversalCharString` value. Using the value of an unbound `TitanUniversalCharString` variable for anything causes dynamic test case error.

The `TitanUniversalCharString` variable used with the `decode_utf8()` method must be newly constructed (unbound) or `clean_up()` must have been called, otherwise a memory leak will occur.

The logged printout of universal charstring values is compatible with the TTCN–3 notation for such strings. The format to be used depends on the contents of the string. Each character (quadruple) is classified whether it is directly printable or not. The string is fragmented based on this classification. Each fragment consists of either a single non-printable character or a maximal length contiguous sequence of printable characters. The fragments are logged one after another separated by an `&` character (concatenation operator). The printable fragments use the normal charstring notation; the non-printable characters are logged in the TTCN–3 quadruple notation. An empty universal charstring value is represented by a pair of quotation marks (like in case of empty charstring values).

An example printout in the log can be the following. The string consists of two fragments of printable characters and a non-printable quadruple, which stands for Hungarian letter "ű":

```
"Character " & char(0, 0, 1, 113) & " is a letter of Hungarian alphabet"
```

Other operators (static functions):

```
boolean operator_equals(final TitanUniversalChar left_value, final TitanUniversalChar
right_value); //Equal
boolean operator_equals(final TitanUniversalChar ucharValue, final
TitanUniversalCharString otherValue);  // Equal
boolean operator_equals(final String otherValue, final TitanUniversalCharString
rightValue));  // Equal
boolean operator_not_equals(final TitanUniversalChar left_value, final
TitanUniversalChar right_value); //Not equal
boolean operator_not_equals(final TitanUniversalChar ucharValue, final
TitanUniversalCharString otherValue);  // Not equal
boolean operator_not_equals(final String otherValue, final TitanUniversalCharString
rightValue));  // Not equal
TitanUniversalCharString operator_concatenate(final TitanUniversalChar ucharValue,
final TitanUniversalCharString other_value);  // Concatenation
TitanUniversalCharString operator_concatenate(final String stringValue, final
TitanUniversalCharString other_value); // Concatenation
```

`Universal charstring` element

The Java class `TitanUniversalCharString_Element` is the equivalent of the TTCN-3 `universal charstring`'s element type (the result of indexing a `universal charstring` value). The class does not store the actual character, only a reference to the original `TitanUniversalCharString` object, an index value and a bound flag.

NOTE	changing the value of the `TitanUniversalCharString_Element` (through the assignment operator) changes the referenced character in the original `universal charstring` object.

The class `TitanUniversalCharString_Element` has the following public member functions:

Table 21. Public member functions of the class `TitanUniversalCharString_Element`

Member functions		Notes
Constructor	`TitanUniversalCharString_Element(final boolean par_bound_flag, final TitanUniversalCharString par_str_val, final int par_char_pos)`	Initializes the object with an unbound value or a reference to a character in an existing TitanUniversalCharString object.

Assignment operators	`TitanUniversalCharString_Element operator_assign(final TitanUniversalChar otherValue)`	Sets the referenced character to the given universal character.
	`TitanUniversalCharString_Element operator_assign(final String otherValue)`	
	`TitanUniversalCharString_Element operator_assign(final TitanCharString otherValue)`	
	`TitanUniversalCharString_Element operator_assign(final TitanCharString_Element otherValue)`	
	`TitanUniversalCharString_Element operator_assign(final TitanUniversalCharString otherValue)`	
	`TitanUniversalCharString_Element operator_assign(final TitanUniversalCharString_Element otherValue)`	
Comparison operators	boolean operator_equals(final TitanUniversalChar otherValue)	Comparison with a universal character.
	boolean operator_equals(final String otherValue)	
	boolean operator_equals(final TitanCharString otherValue)	
	boolean operator_equals(final TitanCharString_Element otherValue)	
	boolean operator_equals(final TitanUniversalCharString otherValue)	
	boolean operator_equals(final TitanUniversalCharString_Element otherValue)	
Comparison operators	boolean operator_not_equals(final TitanUniversalChar otherValue)	
	boolean operator_not_equals(final String otherValue)	
	boolean operator_not_equals(final TitanCharString otherValue)	
	boolean operator_not_equals(final TitanCharString_Element otherValue)	
	boolean operator_not_equals(final TitanUniversalCharString otherValue)	
	boolean operator_not_equals(final TitanUniversalCharString_Element otherValue)	

Concatenation operator	TitanUniversalCharString operator_concatenate(final TitanUniversalChar other_value)	Concatenates this object with a universal character.
	TitanUniversalCharString operator_concatenate(final String other_value)	
	TitanUniversalCharString operator_concatenate(final TitanCharString other_value)	
	TitanUniversalCharString operator_concatenate(final TitanCharString_Element other_value)	
	TitanUniversalCharString operator_concatenate(final TitanUniversalCharString other_value)	
	TitanUniversalCharString operator_concatenate(final TitanUniversalCharString_Element other_value)	
Other member functions	`TitanUniversalChar get_char()`	Returns the referenced character.
	`void log()`	Puts the value into log. Example: “a” or char(0, 0, 1, 113).
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_value()`	Returns whether the value is a value.

Using the value of an unbound `TitanUniversalCharString_Element` variable for anything will cause dynamic test case error.

12.3.12. Object Identifier Type

The object identifier type of TTCN–3 (`objid`) is implemented in class TitanObjectid. In the run-time environment the components of object identifier values are represented in NumberForm, that is, in integer values. The values of components are stored in an array with a given length. The type of the components is specified with a `TitanInteger`. Class `TitanObjectid` has the following member functions.

Table 22. Public member functions of the class `TitanObjectid`

Member functions		Notes
Constructors	`TitanObjectid()`	Initializes to unbound value.
	`TitanObjectid(final int init_n_components, final TitanInteger… values)`	Initializes the number of components to n_components. The components themselves shall be given as additional integer arguments after each other, starting with the first one.
	TitanObjectid(final TitanObjectid otherValue)	Copy constructor.
Assignment operator	`TitanObjectid operator_assign(final TitanObjectid otherValue)`	Assigns the given value and sets the bound flag.
	`Base_Type operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final TitanObjectid otherValue)	Returns true if the two values are equal and false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final TitanObjectid otherValue)	
Indexing operators	TitanInteger get_at(final int index_value)	Returns a reference to the *i th* component.
	TitanInteger get_at(final TitanInteger index_value)	
	TitanInteger constGet_at(final int index_value)	Returns a read-only reference to the i th component.
	TitanInteger constGet_at(final TitanInteger index_value)	

Member functions		Notes
Other member functions	`TitanInteger lengthof()`	Returns the number of components.
	`void log()`	Puts the value into log in NumberForm. Like this: “objid 0 4 0 ”.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_present()`	Returns whether the value is present.
	`void clean_up()`	Deletes the value, setting it to unbound.

NOTE	The constructor with variable number of arguments is useful in situations when the number of components is constant and known at compile time.

Using the value of an unbound `TitanObjectid` variable for anything will cause dynamic test case error.

12.3.13. Component References

TTCN–3 variables of component types are used for storing component references to PTCs. The internal representation of component references are test tool dependent, our test executor handles them as small integer numbers.

All TTCN–3 component types are mapped to the same Java class, which is called TitanComponent.

There are some predefined constants of component references in TTCN–3. These are public static final members of the TitanComponent class defined in the following way:

Table 23. Predefined component references

TTCN–3 constant	TitanComponent member name	Numeric value
null	NULL	COMPREF 0
mtc	MTC	COMPREF 1
system	SYSTEM	COMPREF 2

The class `TitanComponent` has the following public member functions:

Table 24. Public member functions of the class `TitanComponent`

Member functions		Notes
Constructors	`TitanComponent()`	Initializes to unbound value.
	`TitanComponent(final int otherValue)`	Initializes to a given value.
	`TitanComponent(final TitanComponent otherValue)`	Copy constructor.
Assignment operators	`TitanComponent operator_assign(final int otherValue)`	Assigns the given value
	`TitanComponent operator_assign(final TitanComponent otherValue)`	and sets the bound flag.
	`TitanComponent operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final int otherValue)	Returns true if equals
	boolean operator_equals(final TitanComponent otherValue)	and false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final int otherValue)	
	boolean operator_not_equals(final TitanComponent otherValue)	
Casting operator	int get_component()	Returns the value.
Other member functions	`void log()`	Puts the value into log in decimal form or in symbolic format for special constants. Like 3 or mtc.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.
	`void clean_up()`	Deletes the value, setting it to unbound.

Component references are managed by MC. All new test components are given a unique reference

that was never used in the test campaign before (not even in a previous test case). The new numbers are increasing monotonously. The reference of the firstly created component is 3; the next one will be 4, and so on.

Using the value of an unbound component reference for anything will cause dynamic test case error.

Other operators (static functions):

```
boolean operator_equals(final int left_value, final TitanComponent right_value); //
Equal
boolean operator_not_equals(final int left_value, final TitanComponent right_value);
// Not equal
```

12.3.14. Empty Types

Empty `record` and `set` types are not real built-in types in TTCN–3, but the Java realization of these types also differs from regular records or sets. The empty types are almost identical to each other, only their names are different.

Each empty type is defined in a Java class, which is generated by the Java code generator. Using separate classes enables us to differentiate among them in Java. For example, several empty types can be defined as incoming or outgoing types on the same TTCN–3 port type.

Let us consider the following TTCN–3 type definition as an example:

```
type record Dummy {};
```

The generated class will rely on an enumerated Java type TitanNull_Type, which is defined as follows:

```
public enum TitanNull_Type {
    NULL_VALUE
}
```

The only possible value stands for the TTCN–3 empty record or array value (that is for "{}"), which is the only possible value of TTCN–3 type `Dummy`. Note that this type and value is also used in the definition of `record` of and `set of` type construct.

The generated Java class `Dummy` will have the following member functions:

Table 25. Public member functions of the class `Dummy`

Member functions		Notes
Constructors	`Dummy()`	Initializes to unbound value.
	`Dummy( final TitanNull_Type otherValue )`	Initializes to the only possible value.
	`Dummy( final Dummy otherValue )`	Copy constructor.
Assignment operators	`Dummy operator_assign( final TitanNull_Type otherValue )`	Assigns the only possible value and sets the bound flag.
	`Dummy operator_assign( final Dummy otherValue )`	
	`Dummy operator_assign( final Base_Type otherValue )`	
Comparison operators	boolean operator_equals(final TitanNull_Type otherValue)	Returns true if both arguments are bound.
	boolean operator_equals(final Dummy otherValue)	
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final TitanNull_Type otherValue)	Returns false if both arguments are bound.
	boolean operator_not_equals(final Base_Type otherValue)	
Other member functions	`void log()`	Puts the value, that is, {}, into log.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	

Setting the only possible value is important, because using the value of an unbound variable for anything will cause dynamic test case error.

12.4. Compound Data Types

The user-defined compound data types are implemented in Java classes. These classes are generated by the Java code generator according to type definitions. In contrast with the basic types, these classes can be found in the generated code.

12.4.1. Record and Set Type Constructs

The TTCN–3 type constructs `record` and `set` are mapped in an identical way to Java. There will be a Java class for each record type in the generated code. This class builds up the record from its fields. [9] The fields can be either basic or compound types.

Let us consider the following example type definition. The types `t1` and `t2` can be arbitrary.

```
type record t3 {
  t1 f1,
  t2 f2
}
```

The generated class `t3` will have the following public member functions:

Table 26. Public member functions of the class `t3`

Member functions		Notes
Constructors	`t3()`	Initializes all fields to unbound value.
	`t3(final t1 f1, final t2 f2 )`	Initializes from given field values. The number of arguments equals to the number of fields.
	`t3( final t3 otherValue)`	Copy constructor.
Assignment operator	`t3 operator_assign(final t3 otherValue )`	Assigns the given value and sets the bound flag for each field.
	`t3 operator_assign(final Base_Type otherValue)`	

Comparison operators	boolean operator_equals(final t3 other_value)	Returns true if all fields are equal and false otherwise.
	boolean operator_equals(final Base_Type other_value)	
	boolean operator_not_equals(final t3 other_value)	
Field access functions	t1 get_field_f1(); t2 get_field_f2()	Gives access to the first/second field.
	t1 constGet_field_f1(); t2 constGet_field_f2();	The same, but it gives read-only access.
Other member functions	`TitanInteger size_of()`	Returns the size (number of fields).
	`void log()`	Puts the value into log. Like { f1 := 5, f2 := ”abc”}.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	

The record value is unbound if one or more fields of it are unbound. Using the value of an unbound variable for anything (even for comparison) will cause dynamic test case error.

Optional Fields in Records and Sets

TTCN–3 permits optional fields in record and set type definitions. An optional field does not have to be always present, it can be omitted. But the omission must be explicitly denoted. Let us change our last example to this.

```
type record t3 {
  t1 f1,
  t2 f2 optional
}
```

The optional fields are implemented using a Java generic class called `Optional` that creates an optional value from any type. In the definition of the generated class `t3`, the type `t2` will be replaced by `Optional<t2>` everywhere and anything else will not be changed.

The class `Optional<TYPE extends Base_Type>` has the following member functions:

Table 27. Table Public member functions of the class `Optional<TYPE extends Base_Type>`

Member functions		Notes
Constructors	`Optional(final Class<TYPE> clazz)`	Initializes to unbound value, with a class.
	`Optional(final Class<TYPE> clazz, final template_sel otherValue)`	Initializes to omit value, if the argument is OMIT VALUE.
	`Optional(final Optional<TYPE> otherValue)`	Copy constructor.
Assignment operators	`Optional<TYPE> operator_assign(final template_sel otherValue)`	Assigns omit value, if the right value is OMIT VALUE.
	`Optional<TYPE> operator_assign(final Optional<TYPE> otherValue)`	Assigns the given optional value.
	`Optional<TYPE> operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final template_sel otherValue)	Returns true if the value is omit and the right side is OMIT VALUE or false otherwise.
	boolean operator_equals(final Optional<TYPE> otherValue)	Returns true if the two values are equal or false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final template_sel otherValue)	
	boolean operator_not_equals(final Optional<TYPE> otherValue)	

Casting operators	TYPE get()	Gives read-write access to the value. If the value was not previously present, sets the bound flag true and the value will be initialized to unbound.
	TYPE constGet()	Gives read-only access to the value. If the value is not present, causes a dynamic test case error.
Other member functions	`boolean ispresent()`	Returns true if the value is present, false if the value is omit or causes dynamic test case error if the value is unbound.
	`boolean is_present()`	Returns true if the value is present, false otherwise.
	`boolean is_value()`	Returns true if the value is present and is a value, false otherwise.
	`boolean is_bound()`	Returns true if the value is present or omit, false otherwise.
	`boolean is_optional()`	return true;
	`void log()`	Puts the optional value into log. Either ”omit” or the value of t2.
	`void clean_up()`	Deletes the value, setting it to unbound.

In some member functions of the generic class `Optional` the enumerated Java type `template_sel` is used. It has many possible values, but in the optional class only `OMIT_VALUE` can be used, which

stands for the TTCN–3 omit. Usage of other predefined values of `template_sel` will cause dynamic test case error.

Using the value of an unbound optional field for anything will also cause dynamic test case error.

12.4.2. Union Type Construct

The TTCN–3 type construct union is implemented in a Java class for each union type in the generated code. This class may contain any, but exactly one of its fields. The fields can be either basic or compound types or even identical types.

Let us consider the following example type definition. The types `t1` and `t2` can be arbitrary.

```
type union t3 {
  t1 f1,
  t2 f2
}
```

An ancillary enumerated type is created in the generated class `t3`, which represents the selection:

```
enum union_selection_type { UNBOUND_VALUE,  ALT_f1,  ALT_f2 };
```

The type `t3.union_selection_type` is used to distinguish the fields of the union. The predefined constant values are generated as `t3.ALT_`<field name>.

The generated class `t3` will have the following public member functions:

Table 28. Public member functions of the class `t3`

Member functions		Notes
Constructors	`t3()`	Initializes to unbound value.
	`t3(final t3 otherValue)`	Copy constructor.
Assignment operator	`t3 operator_assign( final t3 otherValue )`	Assigns the given value.
	`t3 operator_assign( final Base_Type otherValue )`	
Comparison operators	boolean operator_equals(final t3 otherValue)	Returns true if the selections and field values are equal and false otherwise.
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final t3 otherValue)	

Member functions		Notes
Field access functions	t1 constGet_field_f1()	Gives read-only access to the first field. If other field is selected, this function will cause a dynamic test case error. So use get_selection() first.
	t1 get_field_f1()	Selects and gives access to the first field. If other field was previously selected, its value will be destroyed.
	t2 constGet_field_f2()	
	t2 get_field_f2()	

Member functions		Notes
Other member functions	`union_selection_type get_selection()`	Returns the current selection. It will return t3.UNBOUND VALUE if the value is unbound, t3.ALT_f1 if the first field was selected, and so on.
	`boolean ischosen(final union_selection_type checked_selection)`	Checks if the provided field is selected or not.
	`void log()`	Puts the value into log. Example: { f1 := 5 } or { f2 := "abc" }.
	`boolean is_present()`	Returns whether the value is present.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	

Using the value of an unbound `union` variable for anything will cause dynamic test case error.

The anytype

The TTCN-3 anytype is implemented as a Java class named anytype. It has the same interface as any other Java class generated for a union, with a few differences:

If a field is a built-in type or the address type, the name used in `union_selection_type` is the name of the runtime class implementing the type.

If a field is a user-defined type, the mapping rules in Mapping of Names and Identifiers above apply.

The names of field accessor functions are prefixed with `get_field_` or `constGet_field_` as with other unions.

For example, for the following module

```
module anyuser {
  type record myrec {}

  control {
    var anytype v_at;
  }
}
with {
  extension "anytype integer, myrec, charstring"
}
```

The generated class name will be "anytype". The union_selection_type enumerated type will be:

```
enum union_selection_type { UNBOUND_VALUE,  ALT_TitanInteger,  ALT_myrec,
ALT_TitanCharString };
```

The field accessor methods will be:

```
TitanInteger get_field_TitanInteger();
myrec get_field_myrec();
TitanCharString get_field_TitanCharString();
```

12.4.3. Record of Type Construct

The TTCN–3 type construct `record` of makes a variable length sequence from one given type. This construct is implemented as a Java class.

Let us consider the following example type definition. The type t1 can be arbitrary.

```
type record of t1 t2;
```

This definition will be translated to a Java class that will be called t2.

There is an `enum` type called `TitanNull_Type` defined in the Base Library that has only one possible value. `NULL_VALUE` stands for the empty `"record of"` value, that is, for {}.

Class `t2` will have the following public member functions:

Table 29. Public member functions of the class `t2`

Member functions		Notes
Constructors	`t2()`	Initializes to unbound value.
	`t2(final TitanNull_Type nullValue)`	Initializes to the empty value.
	`t2( final t2 otherValue )`	Copy constructor.
Assignment operator	`t2 operator_assign(final TitanNull_Type nullValue)`	Assigns the empty value.
	`t2 operator_assign( final t2 otherValue )`	Assigns the given value.
	`t2 operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final TitanNull_Type nullValue)	Returns true if the two values are equal and false otherwise.
	boolean operator_equals(final t2 otherValue)	
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final TitanNull_Type nullValue)	
	boolean operator_not_equals(final t2 otherValue)	
Index operators	t1 get_at(final int index_value)	Gives access to the given element. Indexing begins from zero. If this element of the variable was never used before, new (unbound) elements will be allocated up to (and including) this index.
	t1 get_at(final TitanInteger index_value)	
	t1 constGet_at(final int index_value)	Gives read-only access to the given element. Index overflow causes dynamic test case error.
	t1 constGet_at(final TitanInteger index_value)	

Rotating operators	t2 rotate_left(final int rotate_count)	Java equivalent of operator <@. (rotate left)
	t2 rotate_left(final TitanInteger rotate_count)	
	t2 rotate_right(final int rotate_count)	Java equivalent of operator @>. (rotate right)
	t2 rotate_right(final TitanInteger rotate_count)	
Concatenation operator	t2 operator_concatenate(final t2 other_value)	Concatenates two arrays.
	t2 operator_concatenate(final TitanNull_Type null_value)	

Other member functions	`TitanInteger size_of()`	Returns the number of elements, that is, the largest used index plus one and zero for the empty value.
	`void set_size(final int newSize)`	Sets the number of elements to the given value. If the value has fewer elements new (unbound) elements are allocated at the end. The excess elements at the end are erased if the value has more elements than necessary.
	`t2 substr(final int index, final int returncount)`	Returns the section of the array specified by the given start index and length.
	`t2 replace(final int index, final int len, final t2 repl)`	Returns a copy of the array, where the section indicated by the given start index and length is replaced by the given array.
	`void log()`	Puts the value into log. Like {1, 2, 3 }.

Other member functions	`boolean is_present()`	Returns whether the value is present.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	

A `record of` value is unbound if no value has been assigned to it or it has at least one unbound element. Using the value of an unbound `record of` variable for anything will cause dynamic test case error.

Starting with the largest index improves performance when filling a `record of value`.

Pre-generated `record of` and `set of` constructs

The Java classes for the `record of` and `set of` constructs of most predefined TTCN-3 types are pre-generated and part of the TITAN runtime. For instances of these types declared in TTCN-3 and ASN.1 modules only their pre-generated names are used (and a comment telling which type definition it is representing). There is a class with regular memory allocation and one with optimized memory allocation pre-generated for each type. These classes are located in the `PreGenRecordOf` class inside the runtime.

Table 30. Pre-generated classes for `record of`/`set of` predefined types

C++ class name	Equivalent type in TTCN-3
`PREGEN__RECORD__OF__BOOLEAN`	`record of boolean`
`PREGEN__RECORD__OF__INTEGER`	`record of integer`
`PREGEN__RECORD__OF__FLOAT`	`record of float`
`PREGEN__RECORD__OF__BITSTRING`	`record of bitstring`
`PREGEN__RECORD__OF__HEXSTRING`	`record of hexstring`
`PREGEN__RECORD__OF__OCTETSTRING`	`record of octetstring`
`PREGEN__RECORD__OF__CHARSTRING`	`record of charstring`
`PREGEN__RECORD__OF__UNIVERSAL__CHARSTRING`	`record of universal charstring`
`PREGEN__RECORD__OF__BOOLEAN__OPTIMIZED`	`record of boolean with { extension "optimize:memalloc" }`

C++ class name	Equivalent type in TTCN-3
PREGEN__RECORD__OF__INTEGER__OPTIMIZED	record of integer with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__FLOAT__OPTIMIZED	record of float with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__BITSTRING__OPTIMIZED	record of bitstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__HEXSTRING__OPTIMIZED	record of hexstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__OCTETSTRING__OPTIMIZED	record of octetstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__CHARSTRING__OPTIMIZED	record of charstring with { extension "optimize:memalloc" }
PREGEN__RECORD__OF__UNIVERSAL__CHARSTRING__OPTIMIZED	record of universal charstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__BOOLEAN	set of boolean
PREGEN__SET__OF__INTEGER	set of integer
PREGEN__SET__OF__FLOAT	set of float
PREGEN__SET__OF__BITSTRING	set of bitstring
PREGEN__SET__OF__HEXSTRING	set of hexstring
PREGEN__SET__OF__OCTETSTRING	set of octetstring
PREGEN__SET__OF__CHARSTRING	set of charstring
PREGEN__SET__OF__UNIVERSAL__CHARSTRING	set of universal charstring
PREGEN__SET__OF__BOOLEAN__OPTIMIZED	set of boolean with { extension "optimize:memalloc" }
PREGEN__SET__OF__INTEGER__OPTIMIZED	set of integer with { extension "optimize:memalloc" }
PREGEN__SET__OF__FLOAT__OPTIMIZED	set of float with { extension "optimize:memalloc" }
PREGEN__SET__OF__BITSTRING__OPTIMIZED	set of bitstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__HEXSTRING__OPTIMIZED	set of hexstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__OCTETSTRING__OPTIMIZED	set of octetstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__CHARSTRING__OPTIMIZED	set of charstring with { extension "optimize:memalloc" }
PREGEN__SET__OF__UNIVERSAL__CHARSTRING__OPTIMIZED	set OF\ universal charstring with { extension "optimize:memalloc" }

12.4.4. `Set of` Type Construct

The `set of` construct of TTCN–3 is implemented similarly to `record of`. The external interface of this class is exactly the same as in case of `record of`. For more details please see the previous section.

In the internal implementation only the equality operator differs. Unlike in `record of`, it considers

the unordered property of the `set of` type construct, that is, it returns `true` if it is able to find exactly one pair for each element.

The index is a unique identifier for a `set of` element because the Java class does not reorder the elements when a new element is added or an element is modified. The copy constructor also keeps the original order of elements.

12.4.5. Enumerated Types

The TTCN–3 `enumerated` type construct is implemented as a Java class with an embedded enum type.

```
type enumerated Day { Monday (1), Tuesday, Wednesday (3) };
```

The example above will result in the following, very similar Java `enum` type definition which is embedded in the Java class `Day`:

```
public enum enum_type {
            Monday (1),
            Tuesday (0),
            Wednesday (3),
            UNKNOWN_VALUE(2),
            UNBOUND_VALUE(4);
      ...
```

The automatic assignment of numeric values is done according to the standard. Note that there are two extra enumerated values in Java, which stand for the unknown and unbound values. They are used in the conversion functions described below. The Java code generator assigns the smallest two non-negative integer numbers that are not used by the user-defined enumerated values to the unknown and unbound values.

When using the Java `enum` type and its values from user code the names must be prefixed with the Java class name. The `enum` type in the above example can be referenced with `Day.enum_type`, its values can be accessed as `Day.enum_type.Monday`, `Day.enum_type.Tuesday`, and so on.

The class `Day` will have the following public member functions:

Table 31. Public member functions of the class `Day`

Member functions	Notes

Constructors	`Day()`	Initializes to unbound value.
	`Day(final int otherValue)`	Converts the given numeric value to Day.enum_type and initializes to it. Only valid values are accepted.
	`Day(final Day.enum_type otherValue )`	Initializes to a given value.
	`Day(final Day otherValue)`	Copy constructor.
Assignment operator	`Day operator_assign(final int otherValue)`	Converts the given numeric value to Day.enum_type and assigns it. Only valid values are accepted.
	`Day operator_assign(final Day.enum_type otherValue)`	Assigns the given value.
	`Day operator_assign(final Day otherValue)`	
	`Day operator_assign(final Base_Type otherValue)`	
Comparison operators	boolean operator_equals(final Day.enum_type otherValue)	Returns true if the two values are equal and false otherwise.
	boolean operator_equals(final Day otherValue)	
	boolean operator_equals(final Base_Type otherValue)	
	boolean operator_not_equals(final Day.enum_type otherValue)	
	boolean operator_not_equals(final Day otherValue)	
	boolean operator_not_equals(final Base_Type otherValue)	

Comparison operators	boolean is_less_than(final Day.enum_type otherValue)	
	boolean is_less_than(final Day otherValue)	
	boolean is_less_than_or_equal(final Day.enum_type otherValue)	
	boolean is_less_than_or_equal(final Day otherValue)	
	boolean is_greater_than(final Day.enum_type otherValue)	
	boolean is_greater_than(final Day otherValue)	
	boolean is_greater_than_or_equal(final Day.enum_type otherValue)	
	boolean is_greater_than_or_equal(final Day otherValue)	
Static conversion functions	static String enum_to_str(final enum_type enumPar)	See below.
	static enum_type str_to_enum(final String strPar)	
	static boolean is_valid_enum(final int other_value)	
	boolean is_valid_enum(final enum_type other_value)	
	static int enum2int(final Day enumPar)	
	static int enum2int(final Day.enum_type enumPar)	
Non-static conversion functions	int as_int();	See below
	void from_int(final int intValue);	
	void int2enum(final int intValue);	
	void int2enum(final TitanInteger intValue);	

Other member functions	`void log()`	Puts the value into log. Like this: Monday
	`boolean is_preset()`	Returns whether the value is present.
	`boolean is_bound()`	Returns whether the value is bound.
	`boolean is_value()`	Returns whether the value is a value.
	`void clean_up()`	Deletes the value, setting it to unbound.
	`void encode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	
	`void decode(final TTCN_Typedescriptor p_td, final TTCN_Buffer p_buf, final coding_type p_coding, final int flavour)`	

The static member function `Day.enum_to_str` converts the given parameter of type `Day.enum_type` to a Java String. It returns the string "<unknown>", if the input is not a valid value of the TTCN–3 enumerated type. The returned string is read-only.

The function `Day.str_to_enum` does the conversion in the reverse direction. It converts the symbolic enumerated identifier represented by a Java String back to the `Day.enum_type` equivalent. It returns the value `Day.UNKNOWN_VALUE` if the input string is not the equivalent of any of the possible values in the enumerated type.

In the above two functions the strings are treated case sensitive and they shall not contain any whitespace or other characters that are not part of the enumerated value. In case of ASN.1 `ENUMERATED` types the strings used by `enum_to_str`, `str_to_enum` and log represent the TTCN–3 view of the enumerated value, that is, the hyphenation characters are mapped to a single underscore character. For example, if an ASN.1 enumerated type has a value with name `my-enum-value` and numeric value 2, the function `enum_to_str` will return the string `"my_enum_value"` if the input parameter equals to 2. Of course, its Java equivalent will be `my_enum_value` with numeric value 2.

Static member function `Day.is_valid_enum` returns the boolean value `true` if there is a defined enumerated value having numeric value equal to the `int` parameter and `false` otherwise.

The static member function `Day.enum_to_int` converts the given parameter of type Day or `Day.enum_type` to its numeric value. The member function `as_int` does the same thing for the enumerated instance.

The member function `int_to_enum` initializes the enumerated instance with the enumerated value having numeric value equal to the given `int` parameter. A dynamic test case error is displayed if

there is no such enumerated value. The member function `from_int` does the same thing.

If a value of type `int` is passed to the constructor or assignment operator the value is accepted only if it is a numerical representation of a valid enumerated value, that is, the function `is_valid_enum` returns `true`. A dynamic test case error occurs otherwise.

To avoid run-time errors at the decoding of invalid messages the Test Port writer should use the constructor or assignment operator in this way:

```
Day myDayVar;
int myIntVar = buffer[position];
if (Day.is_valid_enum(myIntVar)) {
  myDayVar = new Day(myIntVar);
} else {
  myDayVar = new Day(Day.enum_type.UNKNOWN_VALUE);
}
```

Using the value of an unbound enumerated variable for anything will cause dynamic test case error.

12.4.6. The `address` Type

The special TTCN–3 data type `address` is represented in Java as if it was a regular data type. The name of the equivalent Java class is `ADDRESS`.

12.5. Predefined Functions

Annex C of Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 1: Core Language European Telecommunications Standards and Annex B of Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 7: Using ASN.1 with TTCN–3 European Telecommunications define a couple of predefined functions. Most of them perform conversion between the built-in types of TTCN–3. In our test executor these functions are implemented in the Base Library in Java language. They are available not only in TTCN–3 , but they can be called directly from Test Ports as well.

The implementation of these functions can be found in the class `AdditionalFunctions` in the runtime library, but for easier navigation we list them also in the present document.

The majority of these functions have more than one polymorphic version: when appropriate, one of them takes literal (built-in) Java types as arguments instead of the objects of equivalent Java classes. For instance, if the incoming argument is stored in an `int` variable in your Java code, you should not construct a temporary object of class `TitanInteger` because passing an `int` is faster and produces smaller binary code. Similarly, the returned type is also literal when it is possible.

12.5.1. `Integer` to character

```
TitanCharString int2char(final int value);
TitanCharString int2char(final TitanInteger value);
```

12.5.2. Character to integer

```
TitanInteger char2int(final char value);
TitanInteger char2int(final String value);
TitanInteger char2int(final TitanCharString value);
TitanInteger char2int(final TitanCharString_Element value);
```

12.5.3. Integer to universal character

```
TitanUniversalCharString int2unichar(final int value);
TitanUniversalCharString int2unichar(final TitanInteger value);
```

12.5.4. Universal character to integer

```
TitanInteger unichar2int(final TitanUniversalChar value);
TitanInteger unichar2int(final TitanUniversalCharString value);
TitanInteger unichar2int(final TitanUniversalCharString_Element value);
```

12.5.5. Bitstring to integer

```
TitanInteger bit2int(final TitanBitString value);
TitanInteger bit2int(final TitanBitString_Element value);
```

12.5.6. Hexstring to integer

```
TitanInteger hex2int(final TitanHexString value);
TitanInteger hex2int(final TitanHexString_Element value);
```

12.5.7. Octetstring to integer

```
TitanInteger oct2int(final TitanOctetString value);
TitanInteger oct2int(final TitanOctetString_Element value);
```

12.5.8. Charstring to integer

```
TitanInteger str2int(final String value);
TitanInteger str2int(final TitanCharString value);
TitanInteger str2int(final TitanCharString_Element value);
```

12.5.9. `Integer` to `bitstring`

```
TitanBitString int2bit(final int value, final int length);
TitanBitString int2bit(final int value, final TitanInteger length);
TitanBitString int2bit(final TitanInteger value, final int length);
TitanBitString int2bit(final TitanInteger value, final TitanInteger length);
```

12.5.10. `Integer` to `hexstring`

```
TitanHexString int2hex(final int value, final int length);
TitanHexString int2hex(final int value, final TitanInteger length);
TitanHexString int2hex(final TitanInteger value, final int length);
TitanHexString int2hex(final TitanInteger value, final TitanInteger length)
```

12.5.11. `Integer` to `octetstring`

```
TitanOctetString int2oct(final int value, final int length);
TitanOctetString int2oct(final int value, final TitanInteger length);
TitanOctetString int2oct(final TitanInteger value, final int length);
TitanOctetString int2oct(final TitanInteger value, final TitanInteger length);
```

12.5.12. `Integer` to `charstring`

```
TitanCharString int2str(final int value);
TitanCharString int2str(final TitanInteger value);
```

12.5.13. Length of string Type

This function is built into the equivalent Java classes of all TTCN–3 string types:

```
TitanInteger <any_string_type>.lengthof() const;
```

12.5.14. Number of elements in a structured type

This function is built into the Java template classes of `record of` and `set of` types:

```
TitanInteger <any_record_of_or_set_of_type>.size_of() const;
```

This function is currently not implemented for `record` and `set` types.

12.5.15. The `IsPresent` Function

This function is built into the wrapper Java generic class `Optional`:

```
boolean <any_optional_field>.ispresent() const;
```

12.5.16. The `IsChosen` Function

These functions are built into the equivalent Java classes of TTCN–3 union types:

```
boolean <union_type>.ischosen(
<union_type>.union_selection_type checked_selection) const;
```

12.5.17. The `regexp` Function

```
TitanCharString regexp(final TitanCharString instr, final TitanCharString expression,
final TitanInteger groupno, final boolean nocase);
TitanUniversalCharString regexp(final TitanUniversalCharString instr, final
TitanUniversalCharString expression, final TitanInteger groupno, final boolean
nocase);
```

12.5.18. `Bitstring` to `charstring`

```
TitanCharString bit2str(final TitanBitString value);
TitanCharString bit2str(final TitanBitString_Element value);
```

12.5.19. `Hexstring` to `charstring`

```
TitanCharString hex2str(final TitanHexString value);
TitanCharString hex2str(final TitanHexString_Element value);
```

12.5.20. `Octetstring` to `charstring`

```
TitanCharString oct2str(final TitanOctetString value);
TitanCharString oct2str(final TitanOctetString_Element value);
```

12.5.21. Character string to octetstring

```
TitanOctetString str2oct(final String value);
TitanOctetString str2oct(final TitanCharString value);
```

12.5.22. Bitstring to hexstring

```
TitanHexString bit2hex(final TitanBitString value);
TitanHexString bit2hex(final TitanBitString_Element value);
```

12.5.23. Hexstring to octetstring

```
TitanOctetString hex2oct(final TitanHexString value);
TitanOctetString hex2oct(final TitanHexString_Element value)
```

12.5.24. Bitstring to octetstring

```
TitanOctetString bit2oct(final TitanBitString value);
TitanOctetString bit2oct(final TitanBitString_Element value)
```

12.5.25. Hexstring to bitstring

```
TitanBitString hex2bit(final TitanHexString value);
TitanBitString hex2bit(final TitanHexString_Element value);
```

12.5.26. Octetstring to hexstring

```
TitanHexString oct2hex(final TitanOctetString value);
TitanHexString oct2hex(final TitanOctetString_Element value);
```

12.5.27. Octetstring to bitstring

```
TitanBitString oct2bit(final TitanOctetString value);
TitanBitString oct2bit(final TitanOctetString_Element value);
```

12.5.28. Integer to float

```
TitanFloat int2float(final int value);
TitanFloat int2float(final TitanInteger value);
```

12.5.29. `Float` to `integer`

```
TitanInteger float2int(final double value);
TitanInteger float2int(final TitanFloat value);
```

12.5.30. The Random Number Generator Function

The implementation is based on `java.util.Random`.

```
TitanFloat rnd();
TitanFloat rnd(final double seed);
TitanFloat rnd(final TitanFloat seed);
```

12.5.31. The Substring Function

Implemented for all string types.

```
TitanBitString substr(final TitanBitString value, final int idx, final int
returncount);
TitanBitString substr(final TitanBitString value, final int idx, final TitanInteger
returncount);
TitanBitString substr(final TitanBitString value, final TitanInteger idx, final int
returncount);
TitanBitString substr(final TitanBitString value, final TitanInteger idx, final
TitanInteger returncount);
...
TitanHexString substr(final TitanHexString value, final TitanInteger idx, final
TitanInteger returncount);
TitanOctetString substr(final TitanOctetString value, final TitanInteger idx, final
TitanInteger returncount);
TitanCharString substr(final TitanCharString value, final TitanInteger idx, final
TitanInteger returncount);
TitanUniversalCharString substr(final TitanUniversalCharString value, final
TitanInteger idx, final TitanInteger returncount);
```

And its versions for the _Element types.

12.5.32. Character string to `float`

```
TitanFloat str2float(final String value);
TitanFloat str2float(final TitanCharString value);
```

12.5.33. The Replace Function

Implemented for all string types.

```
TitanBitString replace(final TitanBitString value, final int idx, final int len, final
TitanBitString repl);
TitanBitString replace(final TitanBitString value, final int idx, final TitanInteger
len, final TitanBitString repl);
TitanBitString replace(final TitanBitString value, final TitanInteger idx, final int
len, final TitanBitString repl);
TitanBitString replace(final TitanBitString value, final TitanInteger idx, final
TitanInteger len, final TitanBitString repl);
...
TitanHexString replace(final TitanHexString value, final TitanInteger idx, final
TitanInteger len, final TitanHexString repl);
TitanOctetString replace(final TitanOctetString value, final TitanInteger idx, final
TitanInteger len, final TitanOctetString repl);
TitanCharString replace(final TitanCharString value, final TitanInteger idx, final
TitanInteger len, final TitanCharString repl);
TitanUniversalCharString replace(final TitanUniversalCharString value, final
TitanInteger idx, final TitanInteger len, final TitanUniversalCharString repl);
```

12.5.34. Octetstring to character string

```
TitanCharString oct2char(final TitanOctetString value);
TitanCharString oct2char(final TitanOctetString_Element value);
```

12.5.35. Character string to octetstring

```
TitanOctetString char2oct(final String value);
TitanOctetString char2oct(final TitanCharString value);
TitanOctetString char2oct(final TitanCharString_Element value);
```

12.5.36. The `Decompose` Function

Not implemented yet.

12.5.37. Additional Non-Standard Functions

```
TitanBitString str2bit(final String value);
TitanBitString str2bit(final TitanCharString value);
TitanBitString str2bit(final TitanCharString_Element value);
TitanHexString str2hex(final String value);
TitanHexString str2hex(final TitanCharString value);
TitanHexString str2hex(final TitanCharString_Element value);
TitanCharString float2str(final double value);
TitanCharString float2str(final TitanFloat value);

TitanCharString TitanCharString.ttcn_to_string(final Base_Type ttcn_data)

void TitanCharString.string_to_ttcn(final TitanCharString ttcn_string, final Base_Type
ttcn_value)

TitanUniversalCharString oct2unichar(final TitanOctetString value);
TitanUniversalCharString oct2unichar(final TitanOctetString value, final String
encodeStr);
TitanUniversalCharString oct2unichar(final TitanOctetString value, final
TitanCharString encodeStr);

TitanOctetString unichar2oct(final TitanUniversalCharString value);
TitanOctetString unichar2oct(final TitanUniversalCharString value, final
TitanCharString stringEncoding);
TitanOctetString unichar2oct(final TitanUniversalCharString value, final String
stringEncoding);

TitanCharString get_stringencoding(final TitanOctetString encoded_value);
TitanOctetString remove_bom(final TitanOctetString encoded_value);

TitanCharString encode_base64(final TitanOctetString msg, final TitanBoolean
use_linebreaks);
TitanCharString encode_base64(final TitanOctetString msg);
TitanOctetString decode_base64(final TitanCharString b64);
```

See the section "Additional predefined functions" in the Programmer"s Technical Reference for more details.

12.6. Using the Signature Classes

A Test Port has three outgoing and three incoming types of operation that require the usage of signatures. These are `call` (`getcall`), `reply` (`getreply`) and `raise` (`catch`). Because of this, there are three representation formats (classes generated by the Java code generator) of a signature the Test Port writer should be familiar with. This section describes these classes using an example.

Let us suppose the following signature definition:

```
signature MyProc(in integer inPar, out float outPar,
    inout bitstring inoutPar)
        return hexstring
        exception(charstring, integer, boolean);
```

The classes generated and needed to write a Test Port using this signature are `MyProc_call`, `MyProc_reply` and `MyProc_exception`. These represent the parameters, the return value and the exception type and value of the signature needed by a call, reply or raise.

For example, if a port uses the signature `MyProc` as an output remote procedure, the Test Port gets the outgoing parameters for a call operation towards the system in an instance of the class `MyProc_call`. In this case the classes `MyProc_reply` and `MyProc_exception` are used for placing an incoming reply or raise operation in the queue of the port (using the functions `incoming_reply` and `incoming_exception` of the port class).

12.6.1. The Representation of the Input Parameters

The class `MyProc_call` (using the above example) represents all incoming parameters of the signature `MyProc`. It temporary stores the parameters inPar and inoutPar.

The generated class `MyProc_call` will have the following public member functions:

Table 32. Public member functions of the class `MyProc_call`

Member functions		**Notes**
Parameter access functions	TitanInteger get_field_inPar()	Gives access to parameter inPar.
	TitanInteger constGet_field_inPar()	
	TitanBitString get_field_inoutPar()	The same, but it gives read-only access.
	TitanBitString constGet_field_inoutPar()	
Other member functions	`void log()`	Puts the parameters into log.

The parameters can be accessed via their access functions that have the same names as the parameters (name mapping also applies to these functions).

12.6.2. The Output Parameters and Return Value

The output parameters and return value (if defined) are represented by the class `MyProc_reply` that has the following public member functions:

Table 33. Public member functions of the class `MyProc_reply`

Member functions	**Notes**

Parameter access functions	TitanFloat get_field_outPar()	Gives access to parameter outPar.
	TitanFloat constGet_field_outPar()	The same, but it gives read-only access.
	TitanBitString get_field_inoutPar()	
	TitanBitString constGet_field_inoutPar()	
Access function for return value	TitanHexString get_return_value()	Gives access to the return value.
	TitanHexString constGet_return_value()	
Other member functions	`void log()`	Puts the parameters into log.

The parameters can be accessed by their access functions, and the return value can be accessed via the function `get_return_value()`.

12.6.3. Representation of Signature Exceptions

The class representing the exceptions of a signature (remote procedure) is similar to the representation of the union data type. Using the above example this class is called `MyProc_exception`. This class is generated only if the signature has at least one exception type.

Table 34. Public member functions of the class `MyProc_exception`

Member functions		**Notes**
Constructors	`MyProc_exception()`	Initializes to unbound value.
	`MyProc_exception( final TitanCharString otherValue)`	
	`MyProc_exception( final TitanInteger otherValue)`	
	`MyProc_exception( final TitanBoolean otherValue)`	
	`MyProc_exception(final MyProc_exception otherValue)`	Copy constructor.
Assignment operator	`MyProc_exception operator_assign( final MyProc_exception otherValue )`	Assigns the given value.

Field access functions	TitanCharString get_field_TitanCharString()	Selects and gives access to the CHARSTRING field. If other field was previously selected, its value will be destroyed.
	TitanCharString constGet_field_TitanCharString( )	Gives read-only access to the CHARSTRING field. If other field is selected, this function will cause dynamic test case error. So use get selection() first.
	TitanInteger get_field_TitanInteger()	
	TitanInteger constGet_field_TitanInteger()	
	TitanBoolean get_field_TitanBoolean()	
	TitanBoolean constGet_field_TitanBoolean()	
Other member functions	`exception_selection_type get_selection()`	Returns the current selection. It will return MyProc exception.UNBOUND VALUE if the exception is unbound, MyProc_exception.ALT_TitanCharString if a charstring value is present in the exception, and so on.
	`void log()`	Puts the contents of the exception into the log.

If an exception type is a user-defined type the field name will be constructed from the Java name of the module that the exception type resides in and the name of the Java class that realizes the exception type. The two identifiers are glued together using a single underscore character. Please note that the module name is always present in the identifiers, even if the exception type is defined in the same module as the signature.

For example, if exception type `My_Record` is defined in module `My_Module` the respective field access functions will be named as `My__Module_My__Record_field` and the associated enum value will be `MyProc_exception.ALT_MyModule_MyRecord`.

[8] The built-in `verdict` and `boolean` constants in TTCN–3 shall be written with all lowercase letters, such as true or pass. Although previous compiler versions have accepted `TRUE` or `PASS` as well, these words are treated by the compiler as regular identifiers as specified in the standard.

[9] This section deals with the record and set types that have at least one field. See Empty Types for the Java mapping of empty record and set types.

Chapter 13. Tips & Troubleshooting

Information not fitting in any of the previous chapters is given in this chapter.

13.1. 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, and it translates this TTCN–3 definition to a Java class extension.

```
public static class MyAlternativeName extends MyType { }

public static class MyAlternativeName_template extends MyType_template { }
```

To keep in line with the C side, a semantic error will be reported when a port allows sending or receiving both of the types.

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.

13.2. Using External Java Functions in TTCN–3 Test Suites

Sometimes standard library functions[10] are called in the test suite or there is a need for efficiently implemented "bit-crunching" functions in the TTCN–3 ATS. In these cases functions to be called from the test suite can be developed in Java.

There are the standard library functions as well as other libraries in the Java functions. The logging and error handling facilities of the run-time environment are also available as in case of Test Ports.

For example, the following definitions makes two Java functions accessible from TTCN–3 module `MyModule` and from any other module that imports `MyModule`.

13.2.1. Example TTCN–3 Module (MyModule.ttcn)

```
module MyModule {
[...]
  external function MyFunction(integer par1, in octetstring par2)
    return bitstring;
  external function MyAnotherFunction(inout My_Type par1,
    out MyAnotherType par2);
[...]
}
```

The compiler will translate calls those external function definitions to calls to Java functions in the generated files.

NOTE	The Java side does not generate a function prototype for external functions. Only translates the calls of these functions into calls of Java functions.

Call of these function on TTCN-3:

```
[...]
MyFunction(1, ''O);
MyAnotherFunction(myVar, myAnotherVar);
[...]
```

Would be translated to call in Java:

```
[...]
MyModule_externalfunctions.MyFunction( new TitanInteger(1), new TitanOctetString("")
);
MyModule_externalfunctions.MyAnotherFunction( myVar, myAnotherVar );
[...]
```

The implementation of these function has to be placed in a class whose name is generated from the module's name by appending the "_externalfunctions" postfix, and this class has to be located in the user_provided package.

NOTE	Please note to locate the hand written Test Port and external function implementations in a folder different, than were the Java code is generated. When the project is cleaned, those folders are cleared.

An example implementation of the external functions in Java:

```
package org.eclipse.titan.MyProject.user_provided;

import org.eclipse.titan.MyProject.generated.MyModule.MyAnotherType;
import org.eclipse.titan.MyProject.generated.MyModule.My__Type;
import org.eclipse.titan.runtime.core.TitanInteger;
import org.eclipse.titan.runtime.core.TitanOctetString;

public class MyModule_externalfunctions {

    public static void MyFunction(final TitanInteger titanInteger, final
TitanOctetString titanOctetString) {
    ...
    }

    public static void MyAnotherFunction(final My__Type myVar, final MyAnotherType
myAnotherVar) {
    ...
    }
}
```

Both pre-defined and user-defined TTCN–3 data types can be used as parameters and/or return types of the Java functions. The detailed description of the equivalent Java classes as well as the name mapping rules are described in chapter Mapping of Names and Identifiers.

Using templates as formal parameters in external functions is possible, but not recommended because the API of the classes realizing templates is not documented and subject to change without notice.

The formal parameters of external TTCN–3 functions are mapped to Java function parameters according to the following table:

Table 35. TTCN–3 formal parameters and their Java equivalents

TTCN–3 formal parameter	Its Java equivalent
`[in] MyType myPar`	`MyType myPar`
`out MyType myPar`	`MyType myPar`
`inout MyType myPar`	`MyType myPar`
`[in] template MyType myPar`	*Not recommended.*

Due to the strictness of the TTCN–3 semantic analyzer one cannot use Java data types with external functions as formal parameters or return types, only TTCN–3 and ASN.1 data types are allowed.

The name, return type and the parameters of the implemented Java functions must match exactly the expected function signature or the compilation will fail.

13.3. Logging in Test Ports or External Functions

When developing Test Ports or external functions the need may arise for debug messages. Instead

of using `System.out.println`, there is a simple way to put these messages into the log file of test executor. This feature can be also useful in case when an error or warning situation is encountered in the Test Port, especially when decoding an incoming message.

There is a class called `TTCN_Logger` in the Base Library, which takes care of logging. Since all member functions of `TTCN_Logger` are static, they can be and should be called without instantiating a logger object.

The class `TTCN_Logger` provides some public member functions. Using them any kind of message can be put into the log file. There are two ways to log a single message, the unbuffered and the buffered mode.

13.3.1. Unbuffered Mode

In unbuffered mode the message will be put into log immediately as a separate line together with a time stamp. Thus, the entire message must be passed to the logger class at one function call. The log member function of the logger class should be used. Its prototype is:

```
log(final Severity msg_severity, final String formatString, final Object... args );
```

The parameter severity is used for filtering the log messages. The allowed values of the parameter are listed in table "First level (coarse) log filtering" in the Programmer's Technical Reference. We recommend using in Test Ports only `TTCN_WARNING`, `TTCN_ERROR` and `TTCN_DEBUG`. The parameter `formatString` is a format string, which is interpreted as in the `String.format` function. The dots represent the optional additional parameters that are referred in format string. There is no need to put a newline character at the end of format string; otherwise the log file will contain an empty line after your entry.

Here is an example, which logs an integer value:

```
int myVar = 5;
TTCN_Logger.log(Severity.WARNING_UNQUALIFIED, "myVar = %d", myVar);;
```

Sometimes the string to be logged is static. In such cases there is no need for `printf`-style argument processing, which may introduce extra risks if the string contains the character `%`. The logger class offers a function for logging a static (or previously assembled) string:

```
void log_str(final Severity msg_severity, final String string );
```

The function `log_str` runs significantly faster than log because it bypasses the interpretation of the argument string.

13.3.2. Buffered Mode

As opposite to the unbuffered operation, in buffered mode the logger class stores the message fragments in a temporary buffer. New fragments can be added after the existing ones. When

finished, the fragments can be flushed after each other to the log file as a simple message. This mode is useful when assembling the message in many functions since the buffer management of logger class is more efficient than passing the fragments as parameters between the functions.

In buffered mode, the following member functions are available.

begin_event

`begin_event` creates a new empty event buffer within the logger. You have to pass the severity value, which will be valid for all fragments (the list of possible values can be found in the table "First level (coarse) log filtering" in the Technical Reference. If the logger already has an unfinished event when begin event is called the pending event will be pushed onto an internal stack of the logger. That event can be continued and completed after finishing the newly created event.

```
void begin_event(final Severity msg_severity);
```

log_event

`log_event` appends a new fragment at the end of current buffer. The parameter `fmt` contains a `printf` format string like in unbuffered mode. If you try to add a fragment without initializing the buffer by calling begin event, your fragment will be discarded and a warning message will be logged.

```
void log_event( final String formatString, final Object... args )
```

log_char

`log_char` appends the character c at the end of current buffer. Its operation is very fast compared to `log_event`.

```
void log_char(final char c);
```

log_event_str and log_event_va_list

The functions `log_str` and `log_va_list` also have the buffered versions called `log_event_str` and `log_event_va_list`, respectively. Those interpret the parameters as described in case of unbuffered mode.

```
void log_event_str(final String string);
void log_event_va_list(final String formatString, final Object... args);
```

log

The Java classes of predefined and compound data types are equipped with a member function called `log`. This function puts the actual value of the variable at the end of current buffer. Unbound

variables and fields are denoted by the symbol `<unbound>`. The contents of TTCN–3 value objects can be logged only in buffered mode.

```
void <any TTCN-3 type>.log();
```

end_event

The function `end_event` flushes the current buffer into the log file as a simple message, then it destroys the current buffer. If the stack of pending events is not empty the topmost event is popped from the stack and becomes active. The time stamp of each log entry is generated at the end and not at the beginning. If there is no active buffer when `end_event` is called, a warning message will be logged.

```
void end_event();
```

If an unbuffered message is sent to the logger while the buffer contains a pending event the unbuffered message will be printed to the log immediately and the buffer remains unchanged.

13.3.3. Logging Format of TTCN-3 Values and Templates

TTCN-3 values and templates can be logged in the following formats:

TITAN legacy logger format: this is the default format which has always been used in TITAN

TTCN-3 format: this format has ttcn-3 syntax, thus it can be copied into TTCN-3 source files.

Differences between the formats:

Value/template	Legacy format output	TTCN-3 format output
Unbound value	"<unbound>"	"-"
Uninitialized template	"<uninitialized template>"	"-"
Enumerated value	name (number)	name

The "-" symbol is the NotUsedSymbol which can be used inside compound values, but when logging an unbound value which is not inside a record or record of the TTCN-3 output format of the logger is actually not a legal TTCN-3 value/template because a value or template cannot be set to be unbound. Thus this output format can be copy-pasted from a log file into a ttcn-3 file or to a module parameter value in a configuration file only if it semantically makes sense.

The Java API extensions to change the logging format:
A new enum type for the format in TTCN_Logger class:+ `enum data_log_format_t { LF_LEGACY, LF_TTCN };`;
Static functions to get/set the format globally:
`data_log_format_t get_log_format(); void set_log_format(final data_log_format_t p_data_log_format)`.

NOTE	Please note that Logger_Format_Scope is not yet support by the Java side of the Test Executor.

13.3.4. Examples

The example below demonstrates the combined usage of buffered and unbuffered modes as well as the working mechanism of the event stack:

```
TTCN_Logger.begin_event(Severity.DEBUG_UNQUALIFIED);
TTCN_Logger.log_event_str("first ");
TTCN_Logger.begin_event(Severity.DEBUG_UNQUALIFIED);
TTCN_Logger.log_event_str("second ");
TTCN_Logger.log_str(Severity.DEBUG_UNQUALIFIED, "third message");
TTCN_Logger.log_event_str("message");
TTCN_Logger.end_event();
TTCN_Logger.log_event_str("message");
TTCN_Logger.end_event();
```

The above code fragment will produce three lines in the log in the following order:

```
third message second message first message
```

If the code calls a Java function that might throw an exception while the logger has an active event buffer care must be taken that event is properly finished during stack unwinding. Otherwise the stack of the logger and the call stack of the program will get out of sync. The following example illustrates the proper usage of buffered mode with exceptions:

```
TTCN_Logger.begin_event(Severity.DEBUG_UNQUALIFIED);
        try {
          TTCN_Logger.log_event_str("something");
          // a function is called from here
          // that might throw an exception (for example TtcnError)
          TTCN_Logger.log_event_str("something else");
          TTCN_Logger.end_event();
        } finally {
          // don't forget about the pending event
          TTCN_Logger.end_event();
        }
```

13.4. 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).

13.5. Using the TTCN-3 Preprocessing Functionality

NOTE	This feature, as preprocessors in general, should be avoided if not absolutely necessary.

The Designer has some support for preprocessing preprocessable files according to the rules of the C preprocessor.

The options governing how preprocessable files inside a project are preprocessed can be set via right clicking on the project and selecting "Properties"/"TITAN Java Project Properties" and in the window that appears on the "TTCN-3 Preprocessor" page and its sub-pages.

- On the `Symbols (define, undefine)` page it is possible to define or undefine symbols that will be available for the preprocessor.
- On the `Include directories` page it is possible to set a list of folders which will be used to find `#includ` -ed files, during preprocessing.

Tips for using the preprocessor:

- Don’t. The preprocessor feature should only be used when absolutely necessary.
  - Several preprocessor features are used to generate or hide parts of the source code. This can make it harder for people to understand the code. Makes the use of advanced refactoring features unsafe.
  - The extra cost of preprocesing adds to the duration of the build process.
  - As several preprocessing feature are used to hide information from the tools, and external factors (like environmental variables, files included from outside) can have an effect on the result … any modification will trigger a preprocessing of all the `.ttcnpp` files, the semantic checking of all modules directly or indirectly importing them, and probably the re-generation of the affected modules.

- On the Java side there is no intermediate file generated as all of the processing steps are done in-memory for performance reasons.

There are minor issues when precompiling TTCN-3 code with the 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 preprocessor will replace them.
- There are some predefined macros in the 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.

13.6. Error Recovery during Test Execution

If a fatal error is encountered in the Test Port, you should throw a `TtcnError` exception to do the error handling. It has the following prototype in the Base Library:

```
TtcnError( final String errorMessage );
```

The error handling in the executable test program is implemented using Java exceptions. This exception is normally caught at the end of each test case and module control part. Finally, the verdict is set to error and the test executor performs an error recovery, so it continues the execution with the next test case.

It is not recommended to use own error recovery combined with the default method (that is, catching this exception).

[10] Java language functions cannot be called directly from TTCN–3; you need at least a wrapper function for them.

Chapter 14. 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 TechnologyAbstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specification International Telecommunication Union, July 2002
- [10] ITU-T, X.690, Information TechnologyASN.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 Technology ASN.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 4.7.1, June 2017
- [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] Programmers' Technical Reference Guide for the TITAN TTCN-3 Toolset

Chapter 15. 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